Solidsoft Enterprise Service Bus

Policy-Driven Service Mediation

Component Design

Charles Young

June 2015

# Table of Contents

[Table of Contents 2](#_Toc434778708)

[Table of Figures 4](#_Toc434778709)

[1. Introduction 5](#_Toc434778710)

[2. The ESB Libraries 7](#_Toc434778711)

[2.1 Policy-Driven Service Mediation 7](#_Toc434778712)

[2.2 Dynamic Models in BizTalk Server 7](#_Toc434778713)

[2.3 The Wider View 8](#_Toc434778714)

[2.4 The Anatomy and Role of a Policy 8](#_Toc434778715)

[2.5 Caching 10](#_Toc434778716)

[3. Installation & Configuration 11](#_Toc434778717)

[3.1 Installation Guide 11](#_Toc434778718)

[3.1.1 Install the Rule Engine Vocabularies. 17](#_Toc434778719)

[3.1.2 Start the Resolution Service. 17](#_Toc434778720)

[3.1.3 Configure BizTalk Server or other clients to consume the Resolution Service. 17](#_Toc434778721)

[3.2 Resolution Web Service 19](#_Toc434778722)

[3.3 Other Assemblies 20](#_Toc434778723)

[3.4 Rule Engine Vocabularies 20](#_Toc434778724)

[3.5 Property Schema 20](#_Toc434778725)

[3.6 Installation MSI 21](#_Toc434778726)

[4. Configurable App Settings 22](#_Toc434778727)

[5. Facts 24](#_Toc434778728)

[6. Directives and Instructions 25](#_Toc434778729)

[6.1.1 Endpoint Resolution 27](#_Toc434778730)

[6.1.2 Transformation 30](#_Toc434778731)

[6.1.3 BAM Steps 32](#_Toc434778732)

[6.1.4 Retries 37](#_Toc434778733)

[6.1.5 Service Windows 38](#_Toc434778734)

[6.1.6 Validation 40](#_Toc434778735)

[6.2 Policy Properties 43](#_Toc434778736)

[7. Implementation Design 46](#_Toc434778737)

[8. Property-Handing for BizTalk Server 52](#_Toc434778738)

[8.1 Creating new properties 54](#_Toc434778739)

[8.2 Routing policy 54](#_Toc434778740)

[8.3 Retry policy 55](#_Toc434778741)

[8.4 Validation policy 55](#_Toc434778742)

[9. Defining BAM Policy 55](#_Toc434778743)

[9.1 Motivation 56](#_Toc434778744)

[9.2 Terminology 57](#_Toc434778745)

[9.2.1 Business Activity 57](#_Toc434778746)

[9.2.2 Milestone 57](#_Toc434778747)

[9.2.3 Step 57](#_Toc434778748)

[9.2.4 Interception Point 57](#_Toc434778749)

[9.2.5 Trace Fragment 58](#_Toc434778750)

[9.2.6 Root Fragment 58](#_Toc434778751)

[9.2.7 Activity ID 58](#_Toc434778752)

[9.2.8 Continuation ID 58](#_Toc434778753)

[9.2.9 Trackpoint 59](#_Toc434778754)

[9.2.10 Continuation 59](#_Toc434778755)

[9.2.11 Extension 60](#_Toc434778756)

[9.2.12 Relationship 60](#_Toc434778757)

[9.3 Advantages of using the ESB Libraries BAM Interception 61](#_Toc434778758)

[9.4 BAM Strategies 62](#_Toc434778759)

[9.4.1 Pre-define steps for future extension 63](#_Toc434778760)

[9.4.2 Address overlapped tracking needs 63](#_Toc434778761)

[9.4.3 Extend through activity versioning 63](#_Toc434778762)

[9.4.4 Mix dynamic and static approaches 64](#_Toc434778763)

[9.4.5 Support the extended service bus 65](#_Toc434778764)

[9.5 Configuring Service Mediation Policy for BAM 65](#_Toc434778765)

[9.6 Configuring BAM Steps and Trackpoints 66](#_Toc434778766)

[9.7 Extraction Strings 71](#_Toc434778767)

[9.8 Modelling Step Extension 73](#_Toc434778768)

[9.9 Using the Extended BAM API 75](#_Toc434778769)

[9.9.1 Interception Points 75](#_Toc434778770)

[9.9.2 Trackpoint-Aware Event Observation 75](#_Toc434778771)

[9.9.3 Standard Event Observation 76](#_Toc434778772)

[9.10 Trackpoint Event Streams 77](#_Toc434778773)

[9.10.1 BAM Step Data 79](#_Toc434778774)

[9.10.2 Orchestration Support for BAM 80](#_Toc434778775)

[9.10.3 Examples of Orchestration BAM Code 82](#_Toc434778776)

[10. Defining Validation Policy 84](#_Toc434778777)

[11. UDDI Support 89](#_Toc434778778)

[12. Pipeline Components 93](#_Toc434778779)

# Table of Figures

[Figure 1: Assert-Infer-Enforce Cycle for Service Mediation 9](#_Toc434763280)

[Figure 2: Installer Welcome Screen 12](#_Toc434763281)

[Figure 3: Installer End-User License Agreement Screen 12](#_Toc434763282)

[Figure 4: Installer Choose Setup Type Screen 13](#_Toc434763283)

[Figure 5: Installer Custom Setup Screen 13](#_Toc434763284)

[Figure 6: Installer IIS Settings Screen 14](#_Toc434763285)

[Figure 7: Installer Application Pool Settings Screen 15](#_Toc434763286)

[Figure 8: Installer Ready to Install Screen 15](#_Toc434763287)

[Figure 9: Installer Progress Screen 16](#_Toc434763288)

[Figure 10: Installer Completed Screen 16](#_Toc434763289)

[Figure 11: Installer Ended Prematurely Screen 17](#_Toc434763290)

[Figure 12: Component Architecture 46](#_Toc434763291)

[Figure 13: Resolution Data Properties 53](#_Toc434763292)

[Figure 14: BAM Terminology in ESB Libraries 61](#_Toc434763293)

[Figure 15: BAM Event Stream Class Hierarchy 77](#_Toc434763294)

# Introduction

The Solidsoft Reply ESB Libraries provide functionality for implementing enterprise service bus patterns on the .NET platform. The core libraries are broadly agnostic with regard to specific service bus technologies, but provide specific support for BizTalk Server features such as dynamic maps and Business Activity Monitoring. The Resolver Service, included as a core part of the technology, has a specific dependency on the Microsoft BizTalk Server rules engine.

The Libraries provide a framework for dynamic, policy-driven, service mediation. They can be used as an alternative to the resolution framework included in the BizTalk Server ESB Toolkit. As well as service endpoint (URL) resolution, the libraries handle additional mediation including dynamic transformation using BizTalk Server maps or XSLT, dynamic BAM interception, arbitrarily complex retry and service window policies and validation policies. The libraries are extensible and can be used, unchanged, to handle any addition policy-driven resolution approach.

In order to support service bus implementations that extend beyond BizTalk Server, the libraries implement a service that provides resolution policies on demand, together with local policy caching as part of a .NET-based API. To use the libraries, you must implement the provided SOAP Resolution service on a BizTalk Server box. This is required because of the dependency on the Microsoft Business Rules Framework which can only legally run on a licenced BizTalk Server box. Each process that runs services on the bus can be configured to use the Resolution service, even if the process runs on a non-BizTalk Server box. Note, however, that this version of the libraries supports .NET only.

As well as an API and local cache, the libraries provide pre-built BizTalk Server pipeline components, allowing a configuration-only approach to service mediation in BizTalk Server. There is also a UDDI library that supports policy based approaches using data stored in one or more UDDI directories accessed via rules executed by the Microsoft BRE (Business Rules Engine).

Service mediation policies are constructed using the Microsoft Business Rules Composer. The libraries provide pre-defined vocabularies to aid the representation of mediation policies using natural language terms. The current version supports English vocabularies, only.

Policy-driven service mediation in a service bus environment allows administrators to express and enforce various types of constraint on services and service interchange. Policy can easily be changed and amended in the production environment without the need for re-engineering.

This version of the ESB Libraries explicitly supports the following policy types:

* **Routing Policy**Message-based interchange between services is supported through endpoint resolution and message metadata.
* **Transformation Policy**Dynamic message transformation via XSLT-based maps and BizTalk Server maps.
* **BAM Policy**Dynamic definition and configuration of Microsoft BAM (Business Activity Monitoring) steps for event observation within service bus interchange.
* **Retry Policy**Definition of multi-level retry strategies to support robust interchange.
* **Service Window Policy**Definition of service availability windows, including multiple windows within a 24 hr. period.
* **Validation Policy**Definition of validations for documents and messages.
* **Custom Policy**Custom policies based on name-value pairs.

In addition to the above, the ESB Libraries support general-purpose features that allow administrators to enforce additional policy types through the framework.

# The ESB Libraries

This section introduces and describes the ESB Libraries and their role in implementing policy-driven service mediation. It describes the notion of policies, directives and instructions. It provides a high-level overview of the process of policy enforcement and describes certain ESB Library behaviours such as caching.

## Policy-Driven Service Mediation

The primary purpose of the ESB Libraries is to enable service mediation within service bus environments through the enforcement of policies. Service mediation is a defining characteristic of service bus design. It facilitates dynamic control of the service bus topology in accordance with operational and business requirements. It is one aspect of run-time governance.

Both WCF and BizTalk Server support the notion of ‘bindings’. Bindings control interchange between services through the selection of appropriate protocols, schemas and contracts. They separate service-orientated business logic from the mechanisms that enable and control communication between services and systems. They define a fixed service bus topology through configuration.

Policy-driven service mediation brings an additional dimension to bindings. Bindings are static. All decisions are made at the time a binding is applied and hold until such time the binding is changed. By contrast, a policy is dynamic. Decisions are made on the fly at run-time and depend on evaluation of current state. Because decision making is dynamic, the policies that define how decisions are made can be varied in the run-time environment without the need for re-configuration. Policy-driven service mediation allows services to be de-coupled to a much greater degree. This minimises cost and disruption by allowing the service bus to adapt more flexibly to changing business requirements.

## Dynamic Models in BizTalk Server

BizTalk Server provides extensive support for static bindings through is administration console and APIs. In essence, a BizTalk binding is a serialisation of the configuration settings supported by different components and artefacts within a BizTalk application. In addition to this, it provides support for dynamic models that can be controlled through policy. However, at this time, BizTalk Server does not implement an integrated and comprehensive set of tools to define dynamic policies for service mediation.

The ESB Libraries go some way to rectifying this. The current version lacks tools of its own. Instead, it allows the BizTalk Rules Composer to be exploited for the purpose of defining and managing policies for service mediation. It provides a general-purpose framework for enforcing policies in the run-time environment and extends this with predefined pipeline components for BizTalk Server. The ESB library makes it easier to exploit existing features of BizTalk Server to build dynamic, flexible solutions that lower the cost and disruption associated with the managements of evolving business requirements.

## The Wider View

An enterprise service bus is a conceptual construct that supports the logical association and collaboration of services. Its physical implementation may involve different technologies and platforms. For example, a service bus may incorporate BizTalk Server and WCF on the Windows platform together with additional service bus and messaging technologies on other platforms. Although the ESB Libraries provide pre-defined support for BizTalk Server, they are designed for wider application. For example, they may be used in the implementation of WCF-based services which, conceptually, reside on the bus. The ESB Toolkit provided by BizTalk Server provides limited support for the wider service bus and is more deeply dependent on BizTalk Server constructs. Although some capabilities are published by service interfaces, its primary purpose is to control service-orientated interchange within the context of BizTalk Server.

The ESB Libraries are written in C# and are therefore tied, in their implementation, to the .NET Framework. However, policies are published via a web service using Basic Profile 1.1 bindings. Policy can therefore be consumed easily by code in other run-time environments such as Java. The current version does not, however, provide any pre-built Java components.

## The Anatomy and Role of a Policy

A policy is composed of rules. Each rule contain two parts:

* A set of conditions that can be evaluated against appropriate run-time state (e.g., properties associated with a message or interchange)
* A set of actions that build directives.

Several rules may contribute to the construction of any one directive. The evaluation of one or more rules may result in the construction of several directives. There is, in effect, a many-to-many relationship between rules and directives.

A typical policy rule might look similar to the following:

IF  
 Message Type is equal to <http://solidsoftreply.com/purchaseorderservice/2014/02#SubmitPurchaseOrder>  
**THEN**  
 [Directive 1] Set the SOAP Action header to <http://solidsoftreply.com/purchaseorderservice/2014/02/IPurchaseOrderProcessingSubmitPurchaseOrder>  
 [Directive 1] Send the message to <http://somehost/PurchaseOrders/PurchaseOrderProcessing.svc> using transport type WCF-BasicHttp  
 [Directive 1] On failure, perform 3 level 0 retries at 1 minute intervals  
 [Directive 2] On failure, perform 3 level 1 retries at 60 minute intervals

A directive can be conceptualised as a kind of flexible data record. It contains a set of instructions that, together, are used to direct the run-time environment to behave in a certain fashion and/or perform a set of actions as a single unit of work. For example, a single directive may instruct the runtime environment to transform a message, collect specified data from the message via BAM, route the message to a given endpoint and perform a number of retries in the event of a failure.

In the above example, the rule creates two directives. The first directive provides instructions to route a message to a service endpoint using a specified transport type (in this case, a BizTalk Server WCF transport). It specifies the SOAP action and also the ‘level 0’ retries to be performed on failure. It defines a second directive which specifies an additional ‘level 1’ retry policy.

The following diagram illustrates the relationship between policies, rules and directives and relates this an ‘assert-infer-enforce’ cycle used to resolve policy at runtime.



Figure 1: Assert-Infer-Enforce Cycle for Service Mediation

* **Assert**At runtime, the ESB Libraries are used to assert information about the current state of a service or interchange. State information is asserted to the ESB Libraries via an API. The API checks a local cache for directives that match the asserted state information. If no directives are found, the ESB Library resolves the asserted state information by handing it off to a resolution service (not shown).
* **Infer**The resolution service evaluates the asserted state information against the rules contained in a selected policy. Each policy is a managed and versioned XML document stored in a central repository. The evaluation results in the logical inference of a set of directives. Each directive specifies a set of instructions that will be used to enforce service mediation policy in the runtime environment.
* **Enforce**The directives are passed back to the runtime environment. Note that, at the level of the core ESB Libraries, these directives simply provide information. Enforcement required code in the runtime environment (not shown) that acts on the instructions contained in the directives. For example, the ESB Libraries provide pre-built BizTalk Server pipeline components to enforce service mediation policy in the context of a BizTalk messaging port. The API can be used directly to enforce directives in the context of a BizTalk orchestration or a custom service implementation.[[1]](#footnote-1)

## Caching

In production environments, the ESB Libraries will generally be configured to cache directives locally. This is vital for performance reasons. The overhead of policy resolution on every instance of an interchange would be too great in many scenarios. Unfortunately, caching undermines the immediacy of service mediation policy enforcement. The ESB libraries will expire local caches on a regular basis, as defined by a configuration setting.[[2]](#footnote-2)

In scenarios which require more immediate policy enforcement, it is currently necessary to recycle services and other relevant processes in order to drop the cache. This includes any process that exploits the ESB Libraries directly.[[3]](#footnote-3)

# Installation & Configuration

To install the ESB Libraries and Resolution Service, use the Solidsoft Reply ESB Libraries Setup installer. This is provided in a file named SolidsoftReply.Esb.Libraries.Setup.msi which can be compiled using the Setup project in the source code.

The installer does the following tasks:

Optional

* Installs the Resolver API library to the GAC and a local Reference Assemblies folder.
* Installs the BizTalk.Pipeline library to the GAC and a local Reference Assemblies folder.
* Installs the BizTalk.Pipeline library to the Pipeline Components folder in the BizTalk Server installation.
* Installs the BizTalk.Orchestration library to the GAC and a local Reference Assemblies folder.
* Installs the UDDI Rule Helper library to the GAC and a local Reference Assemblies folder.
* Installs the Resolver Service to a web site called Solidsoft Reply ESB Web Site. The installer creates a web application and associated Application Pool.

Mandatory

* Creates an event log called SolidsoftReplyESBLibraries with two event sources called ESBResolutionService and ESBLibraries.

## Installation Guide

To install, run the MSI by double-clicking it or using msiexec.exe. The following Welcome screen is displayed.



Figure 2: Installer Welcome Screen

Click Next. The End-User Licence Agreement screen is displayed:

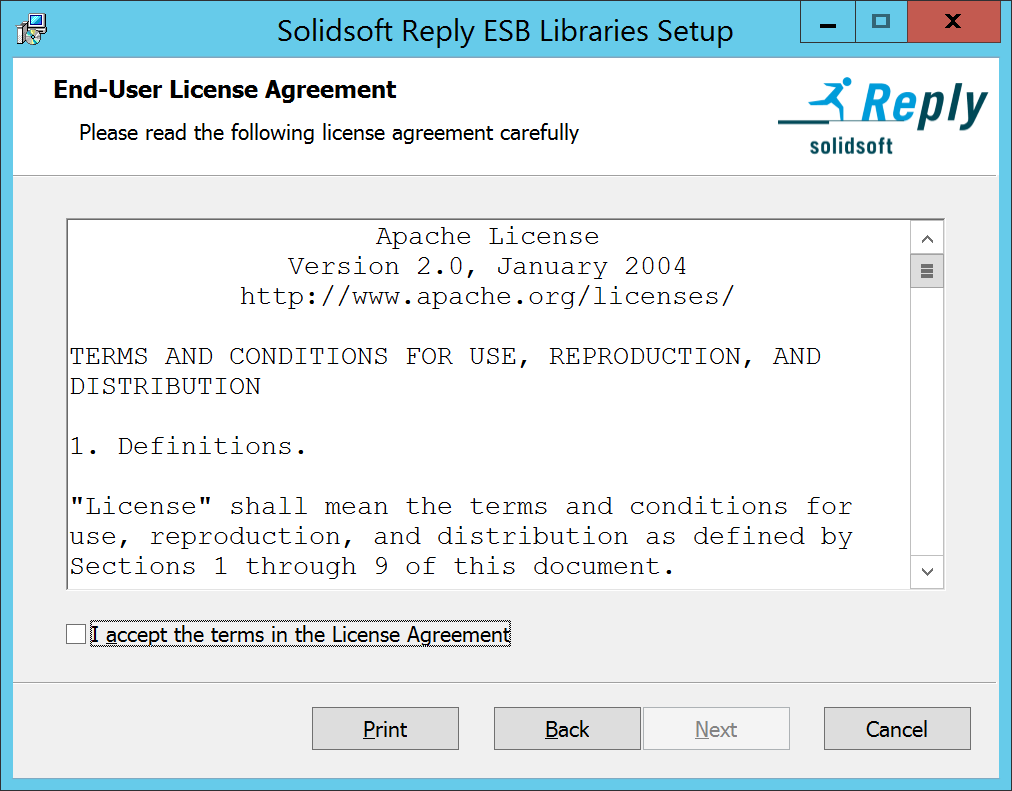


Figure 3: Installer End-User License Agreement Screen

Click the ‘I accept the terms in the License Agreement’ box to indicate acceptance of the Apache 2 licence terms and then click Next. The ‘Choose Setup Type’ screen is displayed:

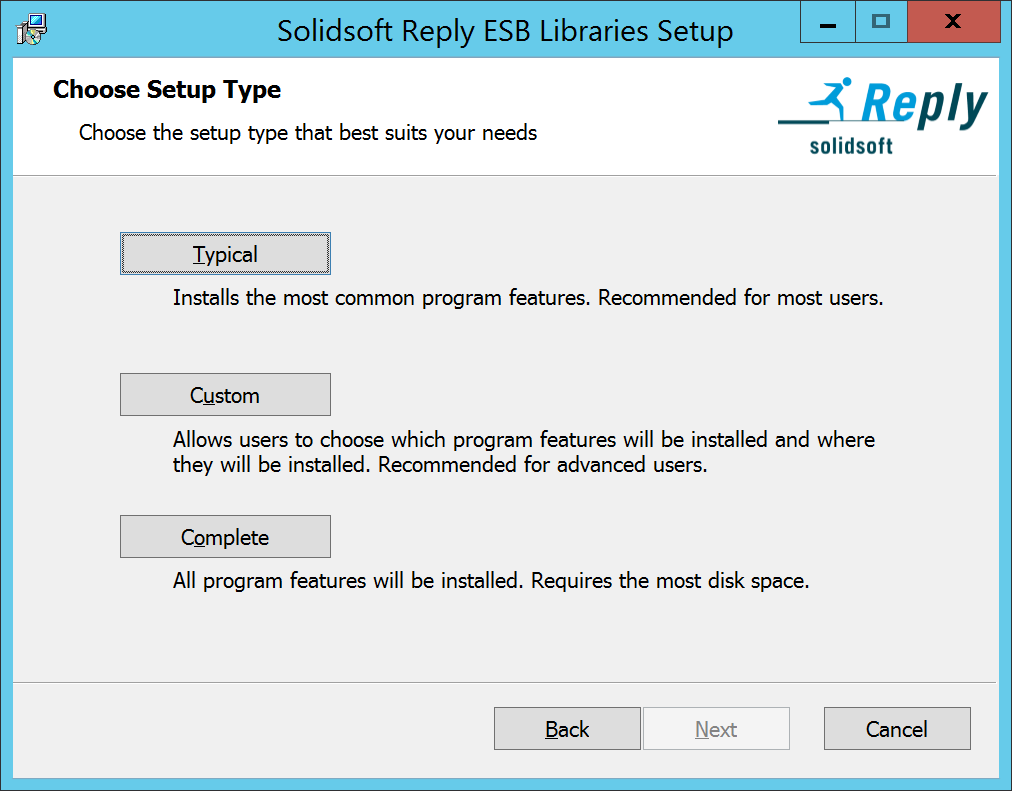


Figure 4: Installer Choose Setup Type Screen

Click one of the three options. Clicking the Typical and Complete buttons will result in the display of the IIS Settings screen (see below). Clicking Custom will display the ‘Custom Setup’ screen:

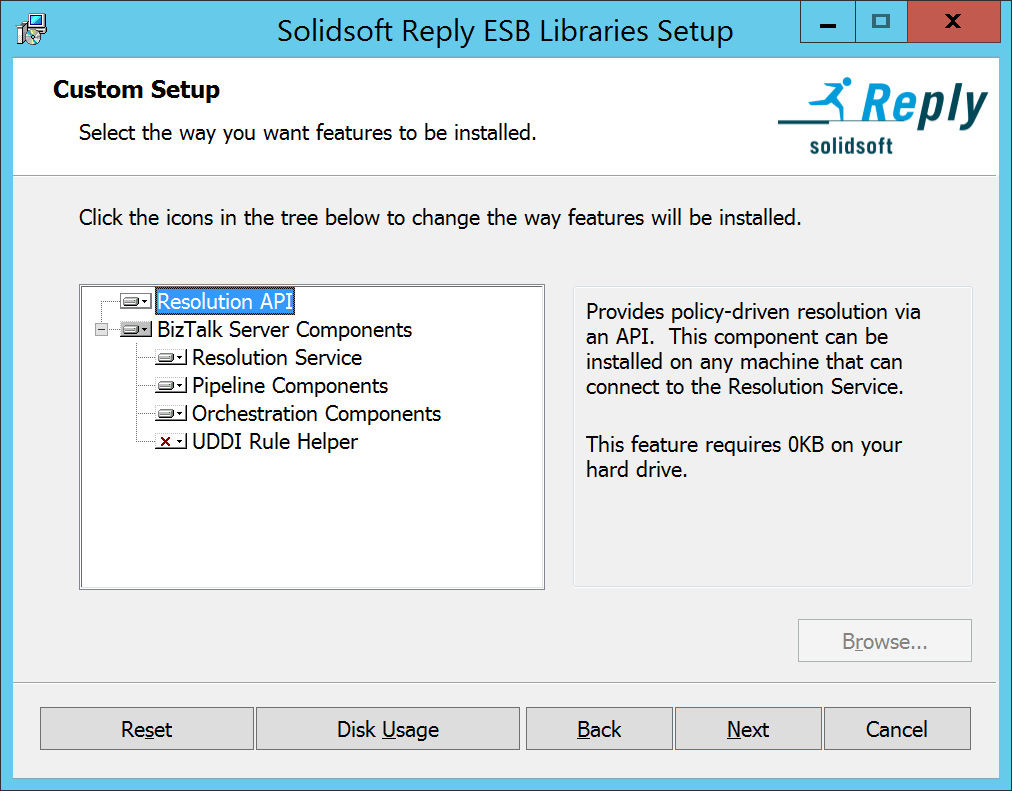


Figure 5: Installer Custom Setup Screen

By default, all components are installed except for the UDDI Rule Helper. This is the same selection used when the Typical button is clicked. You can select or un-select different components as required. Consider the following typical scenarios:

**Installing onto a first BizTalk Server**

Install the default selection, or optionally also select the UDDI Rule Helper library, if UDDI directories will be used. NB., when installing the Pipeline and Orchestration components, you should always install the Resolution API library on which these components depend.

**Installing onto a second BizTalk Server**

You may wish to un-select the Resolution Service. In this case, the Resolver Service will run on a single BizTalk Server. Alternatively, install a second instance of the Resolver Service and configure load balancing (e.g., NLB or router-based balancing) over the instances of the service to provide high availability. NB., when installing the Pipeline and Orchestration components, you should always install the Resolution API library on which these components depend.

**Installing onto a non-BizTalk Server machine**

Un-select all components except the Resolver API library.

When the correct selection has been made, click Next. If the Resolution Service was un-selected, the ‘Ready to install’ screen is displayed. Otherwise the ‘IIS Settings’ screen is displayed:

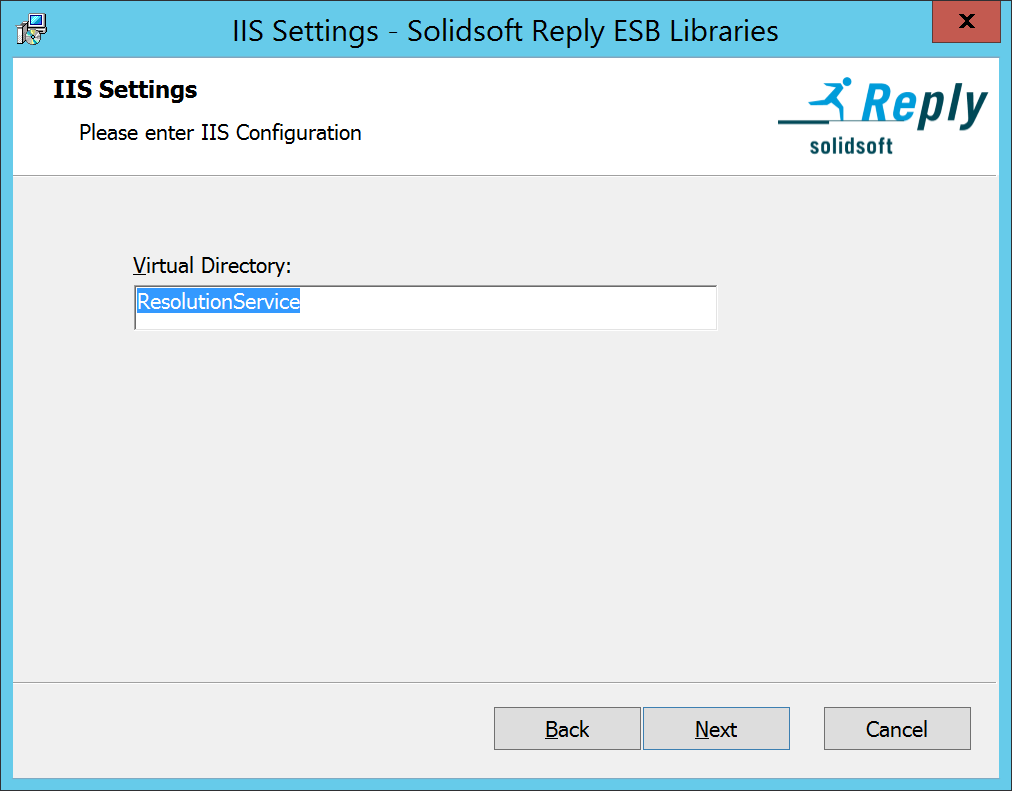


Figure 6: Installer IIS Settings Screen

The default name for the virtual directory is ‘ResolutionService’. Change this if required and then click Next. The ‘Application Pool Settings’ screen is displayed:

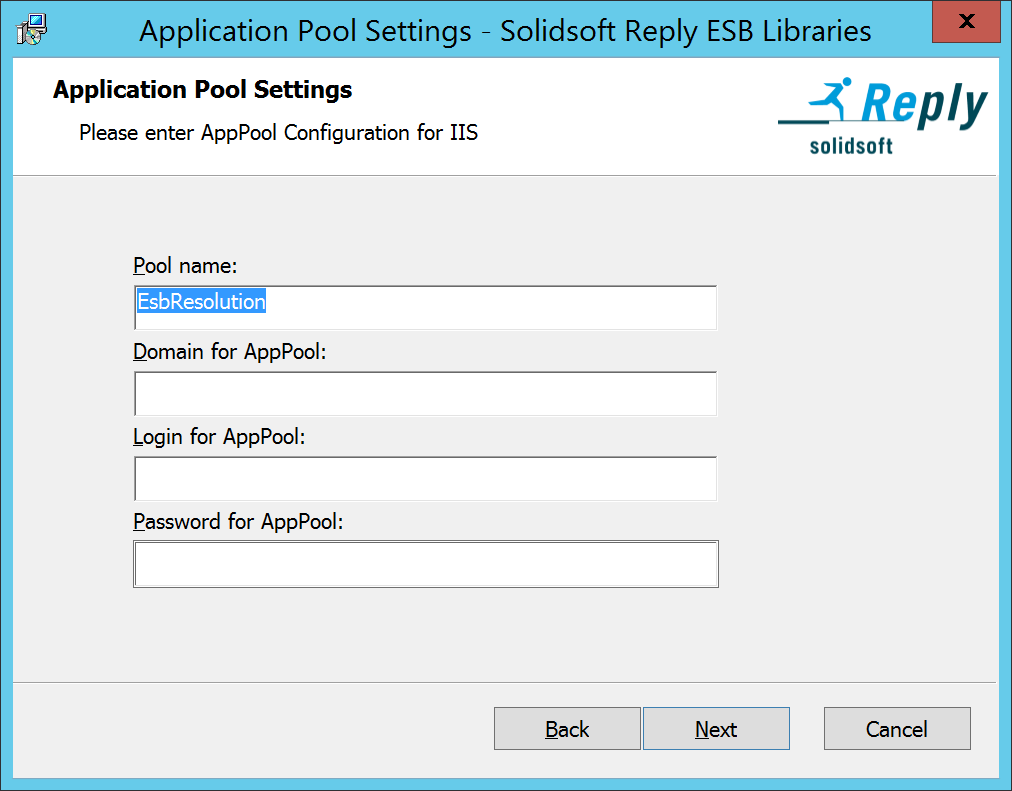


Figure 7: Installer Application Pool Settings Screen

The default application pool name is ‘EsbResolution’. Change this if required. Then enter the credentials for the security principal that will be configured for the application pool. The domain name is optional when using local accounts. Ensure that the user name and password are correct. This version of the installer does not verify that the account is valid. Click Next to display the ‘Ready to install’ screen.

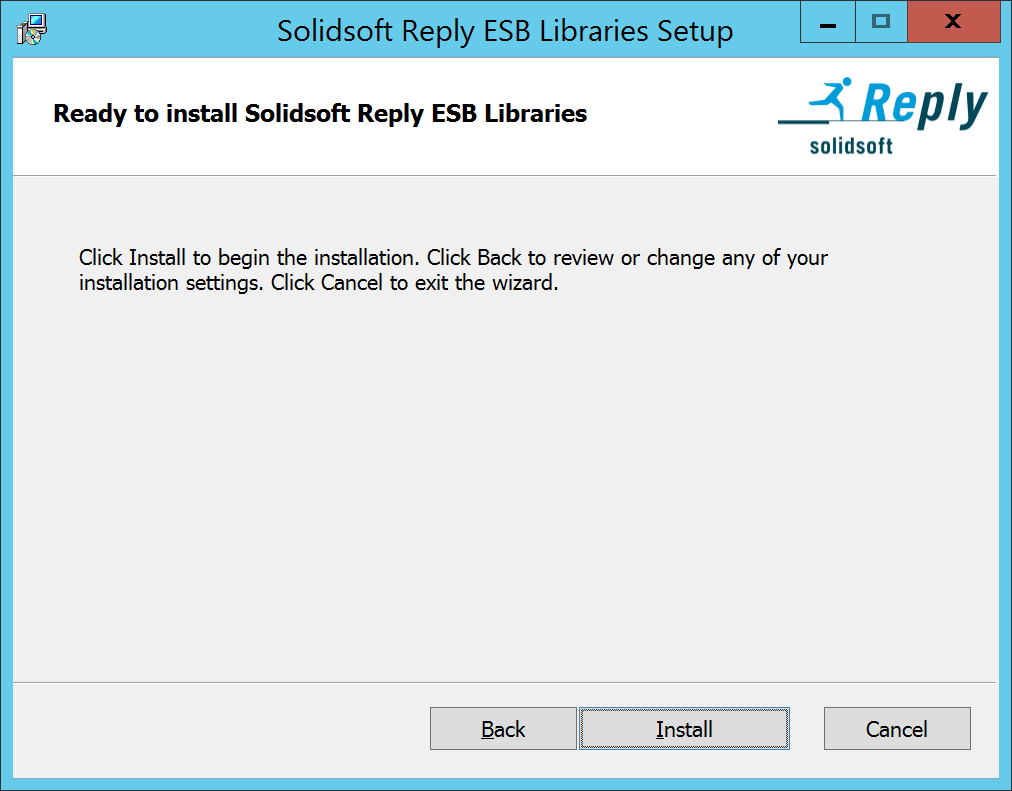


Figure 8: Installer Ready to Install Screen

Click Install to begin installation. The installer will display a progress screen:

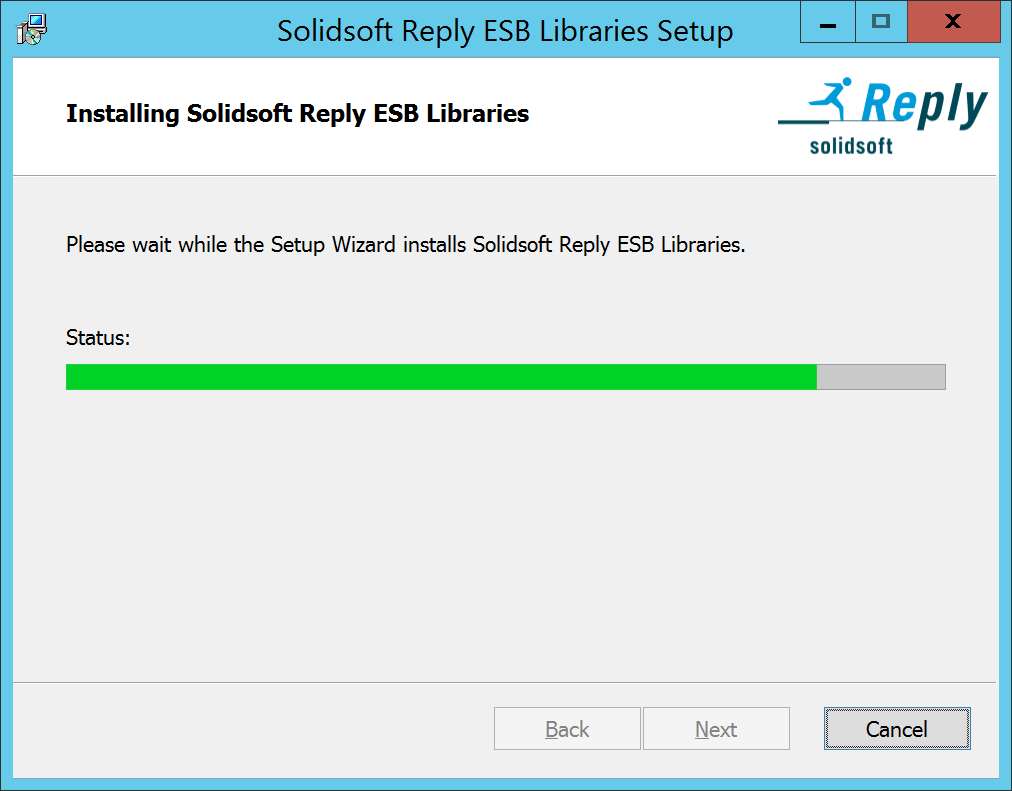


Figure 9: Installer Progress Screen

When installation is complete, the installer will display the ‘Completed’ screen:

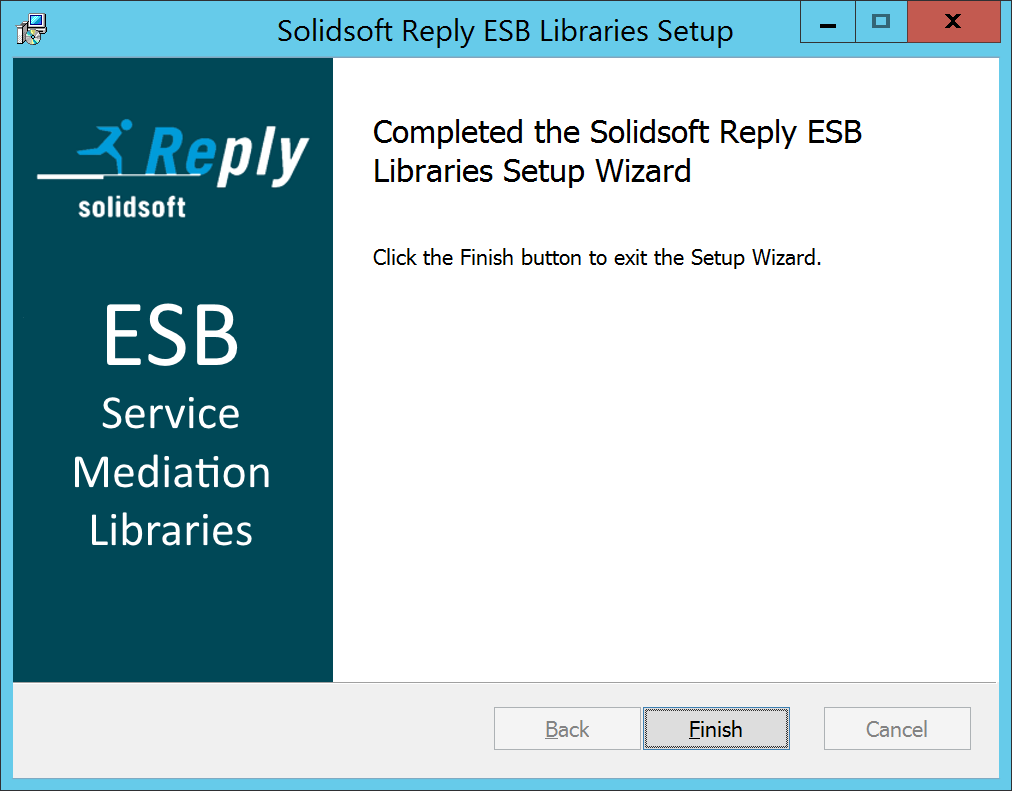


Figure 10: Installer Completed Screen

If installation fails, the installer will display the ‘Ended prematurely’ screen:

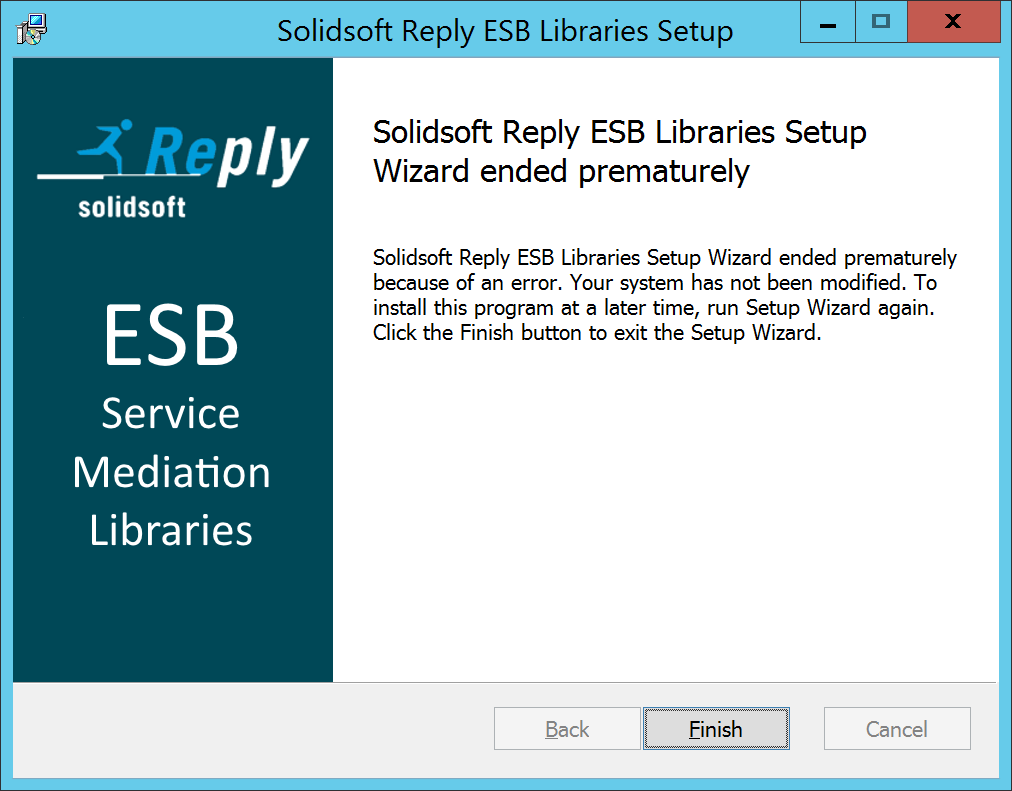


Figure 11: Installer Ended Prematurely Screen

Once installation is complete, you may need to carry out the following post0installation tasks:

### Install the Rule Engine Vocabularies.

### Start the Resolution Service.

### Configure BizTalk Server or other clients to consume the Resolution Service.

The following table provides a manifest of all the artefacts included in the ESB Libraries:

Table 1: Artefacts

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| MSI | SolidsoftReply.Esb.Libraries.Setup.msi | Installs the ESB libraries and Resolution Service. |
| Assembly | Solidsoft.Esb.BizTalk.Orchestration.dll | Provides helper code for using the Resolution API in the context of a BizTalk orchestration. |
| Assembly | Solidsoft.Esb.BizTalk.PipelineComponents.dll | Implements XML and flat file ESB disassembler components together with a general-purpose ESB Governance component. |
| Assembly | Solidsoft.Esb.Facts.dll | Implements fact types and helpers used in the context of the Microsoft Business Rule Engine. |
| Assembly | Solidsoft.Esb.Resolution.dll | The Resolution API library. Provides the core API. |
| Assembly | Solidsoft.Esb.ResolutionService.dll | A WCF service used by the Resolution API to retrieve ESB directives and BAM trackpoint configuration. |
| Assembly | Solidsoft.Esb.Uddi.dll | A helper for use in the context of the Microsoft Business Rules Engine. Allows rules to obtain data from a UDDI directory. |
| Vocabulary | ESB BAM Policy 1.0.xml | BRE vocabulary used for defining trackpoint configuration when writing rules to define BAM steps. |
| Vocabulary | ESB Resolution Actions 1.0.xml | BRE vocabulary used for defining actions when writing rules to build directives. |
| Vocabulary | ESB Resolution Criteria 1.0.xml | BRE vocabulary used for defining conditions when writing rules to build directives. |
| Vocabulary | ESB UDDI 1.0.xml | BRE vocabulary used for defining actions when writing rules to build directives. |
| Vocabulary | ESB Validation Actions 1.0.xml | BRE vocabulary used for defining actions when writing rules to validate documents or messages. |
| Schema | Esb.Validation.Properties.xsd | BizTalk Server property schema used to define validation result properties. |
| Web Service | Resolver.svc | The Resolution web service definition. |
| Config | Web.config | The config file for the Resolution web service. |

## Resolution Web Service

The Solidsoft.Esb.ResolutionService assembly is a WCF web service. For legal reasons, it must run on a licenced BizTalk Server machine. To install the web service manually under IIS, create a web site (e.g., “Solidsoft Reply ESB Resolution Web Service”) and a corresponding application pool (e.g., “EsbResolution”). It is recommended that the identity of the app pool is set to an account created specifically for the web service. Copy the Solidsoft.Esb.ResolutionService.dll assembly file to the bin folder in the web service folder. Copy Resolver.svc and Web.config to the web service folder.

An example of the web.config file is shown below:

<?xml version="1.0"?>

<configuration>

<appSettings>

<add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />

<!--add key="ESB.BRE.PolicyTester" value="true"/-->

<!--add key="ESB.BRE.Trace" value="false"/-->

<!--add key="ESB.BRE.TraceFileLocation" value="C:\Temp"/-->

</appSettings>

<system.web>

<compilation debug="true" targetFramework="4.5" />

<httpRuntime targetFramework="4.5"/>

</system.web>

<system.serviceModel>

<behaviors>

<serviceBehaviors>

<behavior>

<serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>

<serviceDebug includeExceptionDetailInFaults="true"/>

</behavior>

</serviceBehaviors>

</behaviors>

<protocolMapping>

<add binding="basicHttpsBinding" scheme="https" />

</protocolMapping>

<serviceHostingEnvironment aspNetCompatibilityEnabled="true" multipleSiteBindingsEnabled="true" />

</system.serviceModel>

<system.webServer>

<modules runAllManagedModulesForAllRequests="true"/>

<directoryBrowse enabled="false"/>

</system.webServer>

<system.diagnostics>

<switches>

<add name="XmlSerialization.Compilation" value="1" />

</switches>

</system.diagnostics>

</configuration>

## Other Assemblies

The Solidsoft.Esb.BizTalk assembly is a BizTalk Server application containing pre-defined pipelines. It is provided as a courtesy, and its implementation is option. It can be deployed to BizTalk Server using SolidsoftEsbPipelines.msi file. Double click this file to run the installation and choose default values.

All other assemblies are strong-named and should be placed in the GAC. When installing manually, use GACUTIL.exe to install these assemblies.

## Rule Engine Vocabularies

Use the Business Rules Engine Deployment Wizard to install each of the vocabulary files to the rule store. These vocabularies provide terms that can be used when defining rule sets of directives, BAM trackpoint configuration and validations.

## Property Schema

Add the Esb.Validation.Properties.xsd to a BizTalk Server project in order to access validation properties set when validation is done by the ESB BizTalk pipeline components. This is a BizTalk Server property schema.

Configuring BTSNTSvc.exe.config

The following example illustrates how BTSNTSvc can be configured to work with the Resolution Service. Simialar configuration will be required for any WCF client of the service.

<?xml version="1.0" ?>  
<configuration>  
    <appSettings>  
        <add key="ESB.ServiceEndPoint" value="<http://localhost:8080/Resolver.svc>" />  
        <!--add key="ESB.BRE.PolicyTester" value="true" /-->  
        <!--add key="ESB.CacheExpiration" value="1" /-->  
    </appSettings>   
    <startup useLegacyV2RuntimeActivationPolicy="true">  
        <supportedRuntime version="v4.0" />  
    </startup>  
    <runtime>  
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
            <probing privatePath="BizTalk Assemblies;Developer Tools;Tracking;Tracking\interop" />  
        </assemblyBinding>  
    </runtime>  
    <system.serviceModel>  
        <bindings>  
            <basicHttpBinding>  
                <binding name="BasicHttpBinding\_IResolver" />  
            </basicHttpBinding>  
        </bindings>  
        <client>  
            <endpoint address="<http://localhost:8080/Resolver.svc>" binding="basicHttpBinding"  
                bindingConfiguration="BasicHttpBinding\_IResolver" contract="ResolutionService.IResolver"  
                name="BasicHttpBinding\_IResolver" />  
        </client>  
    </system.serviceModel>  
    <system.runtime.remoting>  
      
        <channelSinkProviders>  
            <serverProviders>  
                <provider id="sspi" type="Microsoft.BizTalk.XLANGs.BTXEngine.SecurityServerChannelSinkProvider,Microsoft.XLANGs.BizTalk.Engine" securityPackage="ntlm" authenticationLevel="packetPrivacy" />  
            </serverProviders>  
        </channelSinkProviders>  
      
        <application>  
            <channels>  
                <channel ref="tcp" port="0" name="">  
                 <serverProviders>  
                     <provider ref="sspi" />  
                        <formatter ref="binary" typeFilterLevel="Full"/>  
                    </serverProviders>  
                </channel>  
            </channels>  
        </application>  
    </system.runtime.remoting>  
      
</configuration>

## Installation MSI

To aid installation, the ESB Libraries provide the Solidsoft.Esb.Setup MSI. This installation file will install the library assemblies into the GAC and set up an IIS web site for the Resolver Service. The current version does not, however, import the vocabularies into the rule store or import the predefined pipelines as a BizTalk application. In the current version, the MSI can simply be used as a convenient starting place for installation.

# Configurable App Settings

The following table defines the app settings that can be configured in the relevant .NET config file. For example, these settings may be added to BTSNTSVC.exe.config in order to configure the ESB Libraries in the context of BizTalk Server processes.

Table 2: App Settings

|  |  |
| --- | --- |
| **App Setting Key** | **Description** |
| ESB.BAM.BufferedConnectionString | The connection string for use by the BAM buffered event stream. Overrides any connection string assigned in the policy. |
| ESB.BAM.DefaultTrackpointPolicyName | The default BAM trackpoint policy. NB. Policy names are case sensitive. |
| ESB.BAM.DefaultTrackpointPolicyVersion | The version number of the default BAM trackpoint policy. If not set, the latest deployed version of the policy is used. |
| ESB.BAM.DirectConnectionString | The connection string for use by the BAM direct event stream. Overrides any connection string assigned in the policy. |
| ESB.BAM.FlushThreshold | The value that determines under what conditions the data will be persisted.  **Less than or equal to 0:** Not allowed. No mechanism for manual flushing exists.  **1:** Each event will be immediately persisted.  **Greater than 1:** Events are accumulated in memory until threshold value is reached.  Overrides any threshold assigned in the policy. |
| ESB.BAM.IsBuffered | Selects between direct and buffered event streams. Overrides any setting assigned in the policy. |
| ESB.BRE.PolicyTester | Indicates whether to use the PolicyTester class to execute rule sets. When set to ‘true’, the rule sets will be executed via the tester. BRE policies do not need to be published or deployed. If no version is provided, version 1.0 will be explicitly executed. Use this setting during development as a convenient way to allow rule sets to evolve without having to mark them as immutable (published) and without the need to deploy them via the Rule Engine Update Service. Use this setting in the Resolution Service config file for directive rule sets. Use in other config files to control the execution of BAM and validation rule sets. |
| ESB.BRE.Trace | Indicates whether to trace the execution of rule sets. When set to true, each execution of a rule set will result in a trace file being created at the location specified by the ESB.BRE.TraceFileLocation setting. If no value is provided for ESB.BRE.TraceFileLocation, the trace file is created local to current executable file. |
| ESB.BRE.TraceFileLocation | The location at which to save trace files generated for each rule engine execution. |
| ESB.CacheExpiration | Expiration of cache in hours. If less than or equal to zero, caching is switched off. |
| ESB.DefaultPolicy | The default service mediation policy. NB. Policy names are case sensitive. |
| ESB.ErrorOnInvalid | Indicates if an error should be thrown automatically if a validation policy detects any invalid data. If true, an exception is raised. |
| ESB.ServiceEndPoint | The endpoint URI for the resolution service. |
| ESB.UDDI.DiscoverSites | Indicates whether UDDI site discovery should be used. If true, the UDDI component will find all UDDI sites registered in Active Directory and log warnings for any invalid sites. NB. The UDDI component is used in the context of the Resolver Service. |
| ESB.UDDI.ExpireDiscoveredSitesAfterHours | Specifies the number of hours after which the cache of discovered UDDI sites will expire. If not set, the default is 24 hours. Part hours can be set as decimal values – e.g., 17.5 hrs. |

# Facts

The ESB resolver component supports policy enforcement at interaction points between services. Service consumers pass a list of values to the resolver. These values are called ‘facts’. The resolver evaluates these facts and returns policy instructions grouped into lists. Each list of instructions is a ‘directive’ identified by a name. The consumer can then apply these directives, and the instructions they contain, as appropriate, to the interchange. Service consumers can enforce policy using the Resolver API or via helper classes that wrap the resolver and apply the policy directives within a given context

The following table lists the facts that can be passed to the resolver from a service consumer. The values are defined in alignment with UDDI v3 and WSDL v2.0.

Table 3: Resolution Values

|  |  |
| --- | --- |
| Value | Description |
| Provider name | A name that identifies a service provider. |
| Service name | A name that identifies a service. |
| Access Point | A physical or virtual address (URL) that can be used to access a service. |
| URL type | The URL scheme for an access point (e.g., HTTP, HTTPS, FTP) |
| Message type | A message type specifier. This should generally follow the pattern used by BizTalk Server for the BTS.MessageType message property. For XML interchanges, this is [namespace#][document element local name] |
| Operation name | The name of an operation supported by a service. |
| Message role | Identifies the role of a message in the context of a message exchange pattern. Equivalent to ‘messageLabel’ in WSDL 2.0. Examples might be ‘GetExistingCover’, ‘GetExistingCoverResponse’ ‘fault’, etc. |
| Message direction | Specifies the direction of the message in the context of a message exchange pattern. Loosely based on WSDL 2.0. However, there is no support for infaults or outfaults (use ‘MsgIn’ or ‘MsgOut’ with appropriate message role specifier), and the value can be set to ‘Both’ or ‘NotSpecified’. |
| General parameters | A list of name-value pairs that can be used to represent application-specific value types. |

NB. Values are optional. Passing a null value or empty string within a programmatic interface will be interpreted as passing no value.

# Directives and Instructions

At run time, facts are evaluated by the rules defined in a ‘directive’ policy. Each rule defines actions. When the rule engine finds a match between the rule conditions and the facts, the rule engine ‘fires’ the rule. The rule actions define directives and instructions.

A directive is a named entity created by rule actions. Each rule action specifies the name of a directive and defines an instruction. Multiple actions can contribute instructions to a single directive, even if those actions are contained in different rules. Rules must match the facts and fire in order to contribute instructions. If a rule does not match, it will not fire. Hence, each rule can be thought of in the following generic terms:

**when** <*some facts*> **have** <*some values*>…  
 **contribute** <*some instructions*> **to** <*some named directives*>

For example, a rule might specify the following:

**when** the message type **has** a value of <http://someorg#somemessage>

**contribute** an instruction to route the message to <http://someorg/someservice.svc>   
 **to** a directive named ‘route’

**contribute** an instruction to use a transport type of ‘WCF-BasicHttp’  
 **to** a directive named ‘route’

A single rule can contribute instructions to more than one directive.

The ESB Libraries provides instruction definitions. Each definition is categorised. The resolver currently supports six distinct directive instruction categories for service mediation. These are listed in Table 4: Service Mediation Directive Instruction Categories.

Table 4: Service Mediation Directive Instruction Categories

|  |  |
| --- | --- |
| Instruction Category | Description |
| Endpoint Resolution | Used to resolve the URL address for a specific endpoint together with any other values required for transporting messages to an endpoint. |
| Transformation | Used to specify a transformation. The current version of the resolver assumes that transformations will be encoded as BizTalk Server maps. This allows transformations to be applied in a uniform fashion and managed through BizTalk Server. Note that the Resolver pipeline component extracts the XSLT from the specified map via reflection and applies it directly. Hence, BizTalk Server assemblies containing maps can be distributed to web servers, etc., without requiring BizTalk Server to be installed on those servers. |
| BAM | Used to specify BAM steps. Any one directive can specify two separate steps at which BAM interception and event observation may occur. If the directive also specifies a transform, the first step will intercept the message before transformation, whilst the second will intercept the transformed message. If no transform is defined, only the first step will be used. |
| Retries | Used to specify retry policy for the interchange. Multiple levels of retry can be specified by using multiple directives for the same interchange. |
| Service Windows | Used to specify a time window during which the target service is available. Time windows are specified against any 24 hour day. Multiple time windows can be defined using multiple directives for the same interchange. |
| Validation | Used to specify a validation policy that can be used to validate documents and messages. The Resolver API does not implement any functionality for invoking the validation policy. However, the BizTalk Server-related components (pipelines and orchestration helper) implement functionality to invoke BRE validation rule sets that use the ESB Validation Actions vocabulary definitions. |

Each instruction category is described in detail in the following sub-sections.

### Endpoint Resolution

Specifies service endpoints.

The following directive instructions belong to this category:

Table 5: Endpoint Resolution Instructions

|  |  |
| --- | --- |
| Instruction | Description |
| Endpoint | A URL specifying the endpoint address. NB, in most cases, the transport type can be determined from the scheme included in the URL. |
| Transport type | This is used as an optional disambiguator where different transport options are available for the same endpoint address. It is supported by some BizTalk Server adapters and is primarily included for use with BizTalk Server R2’s WCF adapters. In a BizTalk Server context, the transport type generally specifies which adapter or WCF binding to use. |
| Configuration Token | This is a string that provides configuration for the endpoint. It may contain raw configuration values or act as a handle (e.g., a URL) to configuration data stored elsewhere. |
| SOAP action | A URI that indicates the intent of the SOAP HTTP request. This value should be specified for any endpoint that supports SOAP interchanges. |

To configure service mediation policy for Endpoint Resolution, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 6: Vocabulary Definitions for Endpoint Instructions

| Definition | Description |
| --- | --- |
| Send to an endpoint  One instance of this instruction, the ‘Send to an endpoint with a specified transport’ instruction or the ‘Send to an endpoint with a specified transport and configuration’ instruction allowed per directive. | Set endpoint attributes on a directive.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Endpoint**  Cannot be null or empty. | URI for resolved endpoint. | |
| Send to an endpoint with a specified transport  One instance of this instruction, the ‘Send to an endpoint’ instruction or the ‘Send to an endpoint with a specified transport and configuration’ instruction allowed per directive. | Set endpoint attributes on a directive. Specify a transport.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Endpoint**  Cannot be null or empty | URI for resolved endpoint. | | **Transport Type**  Cannot be null or empty. | BTS transport type for resolved endpoint. | |
| Send to an endpoint with a specified transport and configuration  One instance of this instruction, the ‘Send to an endpoint’ instruction or the ‘Send to an endpoint with a specified transport’ instruction allowed per directive. | Set endpoint attributes on a directive. Specify a transport and configuration.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty | Key name that identifies a directive.  Case-sensitive string. | | **Endpoint**  Cannot be null or empty. | URI for resolved endpoint. | | **Transport Type**  Cannot be null or empty. | BTS transport type for resolved endpoint. | | **Configuration Token** | Configuration token. This may be literal configuration or a reference. | |
| Set the SOAP Action header  One instance of this instruction allowed per directive. | Set the SOAP Action header on a SOAP interchange.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty | Key name that identifies a directive.  Case-sensitive string. | | **SOAP Action**  Cannot be null or empty. Must be a well-formed URI string. | A URI that indicates the intent of the SOAP HTTP request. | |

### Transformation

Specifies transformation requirements in terms of BizTalk Server maps.

The following directive instructions belong to this category:

Table 7: Transformation Instructions

|  |  |
| --- | --- |
| Instruction | Description |
| Map full name | Fully qualified name of .NET class and assembly that implements a BizTalk Server map. E.g., TestOrch.Map1, TestOrch, Version=1.0.0.0, Culture=neutral, PublicKeyToken=bc5dedda99dac908 |
| Map type | .NET Type object corresponding to the full name. NB, the full name must be specified in the ‘map full name’ assertion in order to obtain the .NET Type object at runtime. |

To configure service mediation policy for Transformation, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 8: Vocabulary Definitions for Transformation Instructions

| Definition | Description |
| --- | --- |
| Apply a transformation  One instance of this instruction allowed per directive. | Set transformation attributes on a directive.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Map to Apply**  Cannot be null or empty. | Fully qualified .NET assembly and type name of a BTS map. | |
| Apply formatting to XML content  One instance of this instruction allowed per directive. | Set formatting attributes for XML content on a directive.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string | | **XML Format** | True or false. Indicates whether formatting is required for XML content. | |

### BAM Steps

Specifies BAM steps for the service interactions. BAM steps follow the semantics defined by the BAM Interceptor, but may also be used directly with event streams. Steps represent a point in a business activity where BAM tracking occurs. They are named locations in the code. A single step always represents either the full tracking work for a given activity or a discrete fragment of that work which commences with the start of a new activity or the continuation of an existing activity.

These instructions establish BAM steps, but do not specify the details of BAM actions, if any, that will be undertaken by the BAM interceptor or BAM event stream. Actions are specified in a separate BAM policy called a ‘trackpoint’ policy. A trackpoint is a single action that occurs at a given step.

The following directive instructions belong to this category:

Table 9: BAM Interception Instructions

| Instruction | Description |
| --- | --- |
| BAM activity | Name of BAM activity. This must be a pre-defined activity name for which primary import tables exist. |
| BAM step name | Name of BAM activity step. The step name will generally identify a specific type of service interchange. |
| BAM after-map step name | Name of BAM activity step that will be inserted after a transformation. If no transformation is specified in this directive, the step will not be inserted. |
| BAM connection string | Defines a database connection string. If the ‘BAM is buffered’ assertion is true, this value should be a connection string for the BizTalk Server message box. Otherwise, it should be a connection string for the BAM Primary Import database. The default value is a connection string for a BizTalk message box database with a default name running under a local default instance of SQL Server. |
| BAM is-buffered | Indicates if the BAM interceptor should use a buffered or direct event stream. The default is ‘buffered’. Buffered event streams support higher throughput and lower latency for service interactions, but introduce delays in terms of real-time event monitoring. |
| BAM flush threshold | Indicates how many BAM events should be stored in local memory before being flushed to the event store. The Resolver does not support a value of ‘0’. This value would indicate that the interceptor never automatically flushes events, and is not appropriate in the context of the resolver. The default value of ‘1’ indicates that each event is immediately flushed. |
| BAM Trackpoint Policy | Specifies a rule set that will provide BAM trackpoint configuration for a BAM step. NB. Policy names are case sensitive. |

The BAM connection string, BAM is-buffered flag and BAM flush threshold can be set as part of a directive and can vary for individual policy directives. Alternatively, these setting can be configured at the Resolver Service level in the web.config file using the following ‘appSetting’ keys:

* BamBufferedConnectionString
* BamDirectConnectionString
* BamIsBuffered (set to true | false, 1 | 0 or yes | no)
* BamFlushThreshold (set to integer > 0)

In this case, the web.config settings will be returned within all policy directives unless overridden on specific directives by a policy.

The same ‘appSetting’ keys can be configured in consumer’s config files (e.g., BTSNTSvc.exe, or a web.config file for a BizTalk ‘isolated’ host or custom web service). If present, these settings will be used in preference to the settings contained in policy directives.

To configure service mediation policy for BAM, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 10: Vocabulary Definitions for BAM Instructions

| Definition | Description |
| --- | --- |
| Define a BAM activity step  If BAM interception is required, use this and/or a post-transformation BAM activity step  One instance allowed per directive  If a post-transformation activity step is defined, both must be for the same BAM activity name | Set a step for a given BAM activity. The step will be located before any transformation.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **BAM Activity Name**  Cannot be null or empty | Case-sensitive string | |
| Define a post-transformation BAM activity step  If BAM interception is required, use this and/or a BAM activity step  One instance allowed per directive  Ignored if transformation is not used  If an activity step is defined, both must be for the same BAM activity name | Set a step for a given BAM activity. The step will be located after a transformation. The step will only be implemented if the directive defines a transformation.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **BAM Activity Name**  Cannot be null or empty | Case-sensitive string | |
| Extend a BAM activity step  Cannot be included unless a BAM activity step is defined  Multiple instances allowed per directive  If transformation is used, the step extensions are placed before the transformation | Set a step extension for an activity containing a BAM step.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Step Extension Name**  Cannot be null or empty | Case-sensitive string | |
| Extend a post-transformation BAM activity step  Cannot be included unless a post-transformation BAM activity step is defined  Multiple instances allowed per directive  If transformation is used, the step extensions are placed after the transformation | Set a step extension for an activity containing a post-transformation BAM step.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Step Extension Name**  Cannot be null or empty | Case-sensitive string | |
| Configure a BAM interceptor  If omitted, the default values are assumed  One instance allowed per Directive | Set configuration properties for the BAM interceptor.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Connection String**  Cannot be null or empty | Specifies the SQL connection for the BizTalk message box. Defaults to “Data Source=.;Initial Catalog=BizTalkMsgBoxDb; Integrated Security=True” | | **Is Buffered** | If True (default), the ESB Libraries will use a buffered event stream. Otherwise, a direct event stream will be used. | | **Flush Threshold**  Must be > 0 | The size of the in-memory BAM action cache at which a buffered event stream will flush the actions to the Message Box. Ignored if the ‘Is Buffered’ parameter is False. Defaults to 1. This should generally be increased in production environments. | |
| Specify a BAM trackpoint policy  Can be overridden using the ESB.BAM,DefaultTrackpointPolicyName app setting in the local .NET configuration file.  One instance allowed per directive  Cannot have an instance of this instruction and the next instruction in the same directive. | Specifies a trackpoint policy to be used for a BAM step.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Policy Name**  Cannot be null or empty | Case sensitive string. The name of the BizTalk Business Rule Policy that will be invoked for trackpoint configuration. | |
| Specify a BAM trackpoint policy and version  Can be overridden using the ESB.BAM,DefaultTrackpointPolicyName and ESB.BAM.DefaultTrackpointPolicyVersion app settings in the local .NET configuration file.  One instance allowed per directive  Cannot have an instance of this instruction and the previous instruction in the same directive. | Specifies a trackpoint policy and version to be used for a BAM step.  **Parameters:**   |  |  | | --- | --- | | **Directive Name**  Cannot be null or empty | Case-sensitive string | | **Policy Name**  Cannot be null or empty | Case sensitive string. The name of the BizTalk Business Rule Policy that will be invoked for trackpoint configuration. | | **Policy Version**  Cannot be null or empty  Must be in .NET version format (x.x.x.x). NB. Only the major and minor versions numbers will be used. | The version number of the BizTalk Business Rule Policy. This is a two-part version number – e.g., “1.0” | |

### Retries

Specifies the retry policy that should be applied to service interactions. For any one directive, this category of instructions can specify a single interval value and a single count value.

The following directive instructions belong to this category:

Table 11: Retry Instructions

|  |  |
| --- | --- |
| Instruction | Description |
| Retry count | The number of retries to be attempted before abandoning retries at the current level. |
| Retry interval | The interval, in minutes, between each retry. Use 0 to specify immediate retry. |
| Retry level | The retry level specified by this directive. |

Retry levels are designed for scenarios where shorter-interval retries are executed in the context of longer-interval retries. For example, consider the following directive instructions:

Directive A

Retry count = 3

Retry interval = 1 minute

Retry level = 0

Directive B

Retry count = 5

Retry interval = 60 minutes

Retry level = 1

When a failure occurs, the application should perform three initial retries at one minute intervals, then wait for an hour before attempting a further three retries at one minute intervals, repeating the level 1 loop five times.

The ability to perform multi-level retries depends on the context in which the revolver is used. For example, when used in a BizTalk Server pipeline, only level 0 retries will be attempted. Additional levels will be ignored.

As good practice, when applying policy within the context of a BizTalk Server application, use level 0 only for retries which will be applied by a BizTalk Server Send Port. Orchestrations should be coded to use levels 1 and higher.

To configure service mediation policy for Retries, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 12: Vocabulary Definitions for Retry Instructions

| Definition | Description |
| --- | --- |
| Perform retries  One instance of this instruction or the ‘Perform retries at a given level’ instruction allowed per step. | Set a retry policy on a directive (level 0).  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Retry Count**  Must be an integer value of 0 or greater. | The number of retries to perform. | | **Retry Interval**  Must be an integer value of 0 or greater. | The interval, in seconds, between each retry. | |
| Perform retries at a given level  One instance of this instruction or the ‘Perform retries’ instruction allowed per step. | Set a retry policy on a directive for a given level.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Retry Count**  Must be an integer value of 0 or greater. | The number of retries to perform. | | **Retry Level**  Must be an integer value of 0 or greater. | The temporal level of retries. | | **Retry Interval**  Must be an integer value of 0 or greater. | The interval, in seconds, between each retry. | |

### Service Windows

Specifies a time windows during which the service is available. A single directive can provide a single time window. Use multiple directives for multiple windows. Date values are ignored. Time windows are calculated within any one 24 hour period.

The following directive instructions belong to this category:

Table 13: Service Windows Instructions

|  |  |
| --- | --- |
| Instruction | Description |
| Service Window start time | Specifies the time, within a 24 hour period, at which a service becomes available |
| Service Window stop time | Specifies the time, within a 24 hour period at which a service ceases to be available |

To configure service mediation policy for Service Windows, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 14: Vocabulary Definitions for Service Window Instructions

|  |  |
| --- | --- |
| Definition | Description |
| Define a service window  One instance of this instruction allowed per directive.  NB. The start and stop times are expressed as DateTime values, but any date is ignored. You can use the ‘Time: hh:mm:ss’ vocabulary definition to specify a time value, or use ‘Time: Start of Day’ (midnight) or ‘Time: End of Day’ (just before midnight) as shortcuts. | Sets a service window for a directive.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Start Time**  Cannot be null or empty. | Time at which the service window opens. | | **Stop Time** | Time at which the service window closes. | |

Day of week (multiple per directive)

Day of month (multiple per directive)

Month (multiple per directive)

Year (multiple per directive)

Except day of week (multiple per directive)

Except day of month (multiple per directive)

Except month (multiple per directive)

Except date (multiple per directive)

Except year (multiple per directive)

### Validation

Specifies a validation policy that can be used to validate documents and messages. The Resolver API does not implement any functionality for invoking the validation policy. However, the BizTalk Server-related components (pipelines and orchestration helper) implement functionality to invoke BRE validation rule sets that use the ESB Validation Actions vocabulary definitions.

The following directive instructions belong to this category:

Table 15: Validation Instructions

|  |  |
| --- | --- |
| Instruction | Description |
| Validation Policy | Specifies a policy for validation. This will typically be a BRE rule set that exploits the ESB Validation Actions vocabulary definitions. NB. BRE policy names are case sensitive. |
| Error on Validation Failure | Specifies if an error should be thrown when validation errors are determined by application of the policy. |

To configure service mediation policy for Validation, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 16: Vocabulary Definitions for Validation Instructions

| Definition | Description |
| --- | --- |
| Specify a validation policy  One instance of this instruction, the ‘Specify a validation policy and version’, the ‘Specify a validation policy with exceptions’ instruction or the ‘Specify a validation policy and version with exceptions’ instruction allowed per directive. | Specifies a validation policy to be used to validate data or messages.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Policy Name**  Cannot be null or empty. | Name of the validation policy. | |
| Specify a validation policy and version  One instance of this instruction, the ‘Specify a validation policy’ instruction, the ‘Specify a validation policy with exceptions’ instruction or the ‘Specify a validation policy and version with exceptions’ instruction allowed per directive. | Specifies a validation policy and version to be used to validate data or messages.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Policy Version**  Cannot be null or empty. Must be a valid version specifier (e.g., “1.0”). | Validation policy version. | | **Policy Name**  Cannot be null or empty. | Name of the validation policy. | |
| Specify a validation policy with exceptions  One instance of this instruction, the ‘Specify a validation policy’, the ‘Specify a validation policy and version’ instruction or the ‘Specify a validation policy and version with exceptions’ instruction allowed per directive. | Specifies a validation policy to be used to validate data or messages and throw an exception for any logged errors.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Policy Name**  Cannot be null or empty. | Name of the validation policy. | | **Error on Invalid** | True or false. Indicates whether to throw an error if a validation rule policy indicates invalidity. Default is true. | |
| Specify a validation policy and version with exceptions  One instance of this instruction, the ‘Specify a validation policy’, the ‘Specify a validation policy and version’ instruction or the ‘Specify a validation policy with exceptions’ instruction allowed per directive. | Specifies a validation policy and version to be used to validate data or messages and throw an exception for any logged errors.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Policy Version**  Cannot be null or empty. Must be a valid version specifier (e.g., “1.0”). | Validation policy version. | | **Policy Name**  Cannot be null or empty. | Name of the validation policy. | | **Error on Invalid** | True or false. Indicates whether to throw an error if a validation rule policy indicates invalidity. Default is true. | |

## Policy Properties

The Resolver supports two types of property collection that can be returned as part of a directive. One collection is a general-purpose collection of name value pairs. This collection can be used for any purpose, but relies on the consumer to provide the logic to interpret each property and exploit it appropriately. These properties should be used for application- or technology-specific requirements.

A second property collection is supported for BizTalk Server properties. As well as simple names, this collection also supports namespace definitions and a flag to indicate if the property should be marked as ‘promoted’ within message context. Note that, when used within the context of a BizTalk Server orchestration, it will be necessary to introduce correlation sets in order to honour the ‘promotion’ flag. The flag is honoured automatically within the Resolver pipeline component.

Property collections allow the Resolver to be used in virtually any policy enforcement scenario, even if the type of policy is not covered by the existing directive instruction categories.

To configure service mediation policy for Properties and BTS Properties, use the following vocabulary definitions from the ‘ESB Resolution Actions’ vocabulary in your rule actions:

Table 17: Vocabulary Definitions for Property and BTS Property Instructions

| Instruction | Description |
| --- | --- |
| Assign a value to a property | Set a value on a directive for a general purpose property.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Value**  Cannot be null or empty. | Value to assign to the property. | | **Name**  Cannot be null or empty. | BTS property name. | |
| Assign a value to a BTS GlobalSchema property | Set a value on a directive for a BizTalk Server GlobalSchema property. The property may optionally be promoted.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Value**  Cannot be null or empty. | Value to assign to the property. | | **Name**  Cannot be null or empty. | BTS property name. | | **Promoted** | True or false. Indicates whether the property should be promoted. | |
| Assign a value to a promoted BTS GlobalSchema property | Set a value on a directive for a BizTalk Server GlobalSchema property, and mark as promoted.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Value**  Cannot be null or empty. | Value to assign to the property. | | **Name**  Cannot be null or empty. | BTS property name. | |
| Assign a value to a BTS property | Set a value on a directive for a BizTalk Server property. The property may optionally be promoted.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Value**  Cannot be null or empty. | Value to assign to the property. | | **Name**  Cannot be null or empty. | BTS property name. | | **Namespace**  Cannot be null or empty. | XML namespace of the property. | | **Promoted** | True or false. Indicates whether the property should be promoted. | |
| Assign a value to a promoted BTS Property | Set a value on a directive for a BizTalk Server property, and mark as promoted.  **Parameters:**   |  |  | | --- | --- | | **Directive Key**  Cannot be null or empty. | Key name that identifies a directive.  Case-sensitive string. | | **Value**  Cannot be null or empty. | Value to assign to the property. | | **Name**  Cannot be null or empty. | BTS property name. | | **Namespace**  Cannot be null or empty. | XML namespace of the property. | |

# Implementation Design

The following diagram illustrates the various components that are involved in resolution.

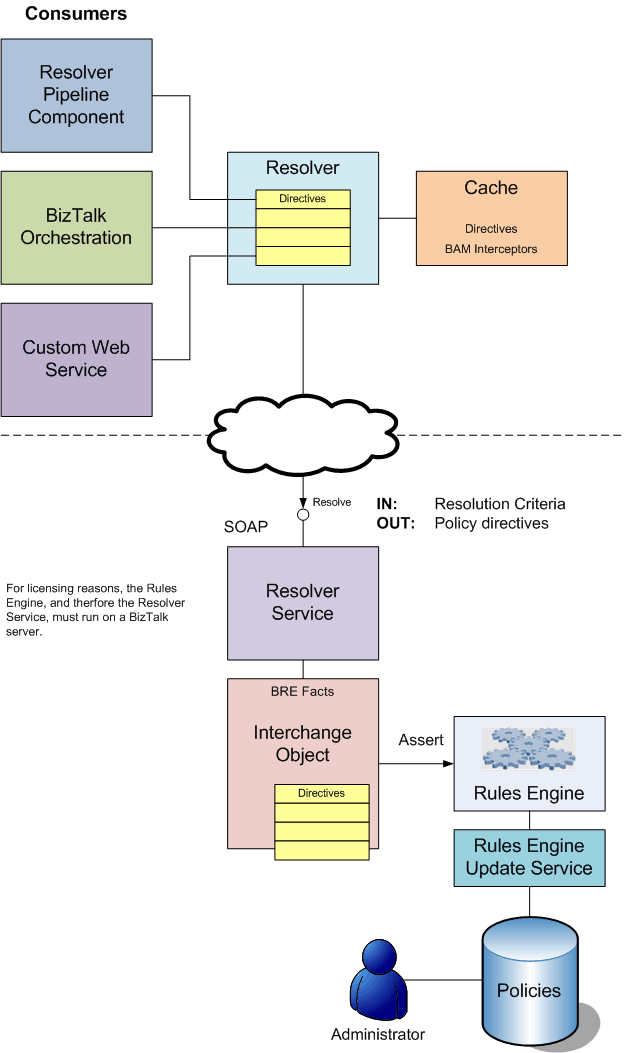


Figure 12: Component Architecture

Table 18: Resolver Components

| Component/Service | Description | Source |
| --- | --- | --- |
| Resolver Service | A SOAP web service (asmx). The service is responsible for creating an Interchange fact and asserting it to the rules engine. It then returns the interchange object.  The web service also supports requests for BAM policy. A BAM policy is returned on a per-activity basis (i.e., per BAM activity defined using BAM tooling). | Resolver |
| Interchange Object | The interchange object created by the Resolver service is asserted to the rules engine as a fact. It is populated with values provided within the call to the service. Once the rule engine policy has been executed, the interchange object contains a collection of directives. | Resolver |
| Directives | Each directive object provides one or more instructions. See section **Error! Reference source not found.** for a complete listing of all instructions categories. As a general rule, any one directive can contain just one instance of any one instruction, and run-time errors will be raised if the policy assigns instructions incorrectly to a directive (NB., there is no mechanism for enforcing design-time validation of directives).  Directives also contain a BTS Property collection and a General Property collection. These are defined in section 6.2. Each collection can contain an unlimited number of entries. | Resolver |
| Rules Engine | The Microsoft Business Rules Engine (MS BRE) ships with BizTalk Server, and for licensing reasons must run on a BizTalk Server box. NB. This implies that the Resolver Service must also run on the same box.  MS BRE is an advanced rules engine that applies pattern matching techniques to data sets (facts). In this case, the Resolver Service asserts a single Interchange fact to the engine, and invokes a policy. NB., in Rule Engine terminology, a ‘fact’ represents a single data tuple (row) that may contain multiple attribute values. The Interchange object represents attributes as properties.  The engine evaluates rule conditions the properties of the Interchange object, and, if all the conditions in a rule match, that rule is ‘fired’. The rule defines actions which are used to add directives and policy assertions to the Interchange object. The policy may contain many rules, and the engine fires each rule for which there is a complete match. Rules may fire in any order.  Once rule execution is complete, the Resolver Service serialises the Interchange object and passes it back to the caller. | Microsoft BizTalk Server |
| Rules Engine Update Service | The Rules Engine Update Service is provided as part of the Microsoft Business Rules Framework. An instance of this service is generally run on each BizTalk Server box. The service polls a central policy store (a SQL Server database) for changes to published policies. When changes are detected, the service is responsible for downloading the new version of the policy (on demand) and caching it locally. This mechanism is used to deploy versioned policies from a central database to multiple BizTalk Services. Policies can be deployed directly into a live environment without requiring a break in service (i.e., without stopping and starting BizTalk or other applications). NB., when a new policy version is deployed, it may take a few minutes for the policy to be fully deployed across multiple machines. | Microsoft BizTalk Server |
| Policy Store | The Policy Store is supplied as part of the Microsoft Business Rules Framework. It is a SQL Server database that stores versioned policies (rule sets) and ‘vocabularies’. Vocabularies are used to map human-friendly terms onto fact attributes, predicate functions and actions used within rule definitions. They provide a mechanism for creating domain-specific rule languages. Vocabularies have been created for the Resolver.  Versioned rule and vocabulary definitions are created, maintained and published using the Microsoft Rules Composer. This tool provides a UI for creating and editing rules and publishing policies. | Microsoft BizTalk Server |
| Resolver | The Resolver component is used by policy consumers to connect to the Resolver Service and request policies. The Resolver de-serialises the returned collection of directives and makes them available programmatically to consumer applications. It also provides helper functionality for applying transforms, implementing BAM steps, etc. | Resolver |
| Resolver Cache | In many scenarios it is vitally important that resolution is a low-latency activity. For this reason, the Resolver uses a local cache to store directives and BAM interceptors. The cache is used to enable re-use of existing objects without having to call repeatedly into the Resolver Service.  This version of the Resolver does not provide an equivalent of the Rules Engine Update Service to allow new policies to be deployed into a live environment with interruption to service. In a BizTalk environment, the Resolver could use the RUES directly without the need for the Resolver Service. However, the Resolver is designed to extend the service bus concept beyond the confines of BizTalk Server; e.g., to enable resolution in interchanges between custom web services. This is why the Resolver Service was introduced. A future version may introduce a REUS-like Resolution Policy Update Service that can be run on any server, and which is not tied by licensing to BizTalk Server.  In this version, BizTalk host processes, IIS worker processes and any other relevant processes must be stopped and re-started in order to invalidate the cache. In any case, the cache uses expiry times on each cache item. However, these should be set to a relatively long interval (e.g., once a day) via an ‘ESB.CacheExpiration’ appSetting key in the local config file. | Resolver |
| Resolver Pipeline Components | The Resolver Pipeline components are designed for use within the BizTalk Server environment. They apply directive instructions to each message that passes through the pipeline. The libraries provide six pipeline components. The Service Mediation component supports endpoint resolution and single-level retry policies via promoted message properties. It also supports dynamic transformation, BAM interception and service windows. The XML, Flat File, BTF, EDI ad AS2 disassemblers are also provided. These combine existing Microsoft-provided disassembly functionality with the same features provided by the Service Mediation pipeline component.  The disassembler components will disassemble XML and flat file data in the normal way. In addition to this, they will further disassemble each disassembled message according to the number of directives returned by the Resolution Service. If a flat file contains 200 rows and is passed through the Flat File Disassembler, the number of messages that will be created by the disassembler is 200 x No. of Directives.  When using the dynamic transformation features of the pipeline components, the components will perform property promotion on the transformed message in a similar fashion to BizTalk Server’s in-built transformation features. | Resolver |
| BizTalk Orchestration | When using the ESB Libraries in the context of BizTalk Server orchestrations, they are accessed via the API. However, a small library of helper classes has been provided. In the current version these provide support for BAM event observation.  For the most part, it is the responsibility of the developer to exploit directive instructions as required in their orchestration code. However, the APIs provide built-in support for dynamic transformation and BAM interception. They also provide BAM event streams that are configured by a directive and trackpoint policy. The orchestration helper library extends this by providing support for the Orchestration Event Stream (OES). | Custom |
| WCF Web Service | The current version does not yet provide support for WCF behaviours. This is planned in a later release. | Custom |

# Property-Handing for BizTalk Server

The ESB Libraries are broadly technology-agnostic, but pr`ovide specific support for use in BizTalk Server environments. This includes support for message properties and property promotion. This section summarises the property-handling behaviour of the libraries when interacting with BizTalk Server messages in the context of an ESB pipeline component.

The ESB pipeline components allow developers to control passing message properties as facts to the Resolver API. By default, no property values are passed. This is an optimisation and safety feature. Values passed as facts are used to resolve directives against a local in-memory cache. The fewer fact values submitted to the Resolver, the less work it needs to do to search for items in the local cache. In addition, if a property value is used as a fact value, but is different for each message, this may result in a large number of cache misses. This can be very expensive and can potentially cause effective memory leaks as the cache fills up with redundant entries. The MessageID property is a good example of a property that has a different value for each individual message. The pipeline components automatically promote this property.

Property passing is controlled by the ‘Resolution data’ and ‘Resolution data properties’ properties of the pipeline component. The ‘Resolution data’ property is an enumerated value, as follows:

Table 19: Resolution Data Property Values

|  |  |
| --- | --- |
| Value | Description |
| ValuesOnly | The pipeline component only passes resolution values. See Table 3: Resolution Values. These properties can be set on the corresponding pipeline properties. The Message Type property is set automatically from the MessageType message property, if it exists, but can be overridden using the corresponding pipeline property. The ‘Resolution data properties’ property is ignored. |
| ValuesWithListedPromotedProperties | The pipeline component passes resolution values and any promoted message properties listed in the ‘Resolution data properties’ property. |
| ValuesWithAllListedProperties | The pipeline component passes resolution values and any message properties listed in the ‘Resolution data properties’ property. |
| ValuesWithAllPromotedProperties | The pipeline component passes resolution values and all promoted message properties. The ‘Resolution data properties’ property is ignored. NB. Avoid this setting unless its implications are fully understood. |
| ValuesWithAllProperties | The pipeline component passes resolution values and all message properties. The ‘Resolution data properties’ property is ignored. NB. Avoid this setting unless its implications are fully understood. |

When set to ValuesWithListedPromotedProperties or ValuesWithAllListedProperties, properties will only be passed to the Resolver as facts if they are explicitly listed in the ‘Resolution data properties’ property. When setting this property, the pipeline component will display a dialog box, allowing developers to build a list of properties. Each property is specified by its namespace and property name.

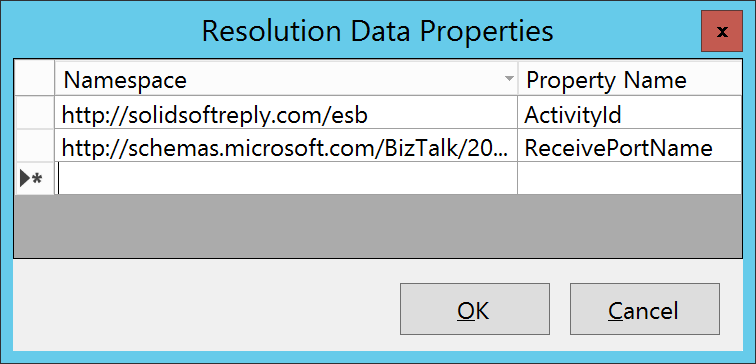


Figure 13: Resolution Data Properties

Each message property is passed to the Resolver as a name-value pair in a ‘parameters’ collection. The qualified property name is represented in the following format:

<namespace>#<property name>

The parameters collection is passed with other facts to the rules engine. You can use the ‘Parameter’ definition in the ESB Resolution Criteria vocabulary to test for a parameter value in the IF part of a rule.

NB. Internally, the ESB XML, Flat File, EDI, A2S and BTF disassemblers call the corresponding Microsoft-supplied disassemblers first before passing each disassembled message to an internal instance of the ESB Service Mediation disassembler. If you use BizTalk Server’s property promotion with one of these disassemblers, properties will be promoted before the Resolver is invoked.

All promoted properties on a message passed to a pipeline component will be promoted on the message returned to the pipeline by that component, even if it performs a transformation on the received message. When a message is transformed, the pipeline component performs property promotion using the functionality of the Microsoft XML Disassembler. This is similar behaviour to the built-in port-level transformation features of BizTalk Server.

There is no need to list the BizTalk BTS.MessageType property as a Resolution Data property. The pipeline components automatically map this property to the Message Type resolution value. Similarly, there is no need to use the BTS Properties (see below) to promote the BTS.MessageType property. The pipeline components always promote this property on each outbound message, together with the BTS.SchemaStrongName property.

## Creating new properties

Properties can be written or promoted by the ESB pipeline component using the following directive actions in a policy:

* Assign a value to a BTS GlobalSchema property
* Assign a value to a BTS property
* Assign a value to a promoted BTS GlobalSchema property
* Assign a value to a promoted BTS Property

These actions populate a collection of properties designed for BizTalk Server which are returned via the Resolver API and accessed via the BtsProperties property of a Directive object. As well as a name and a value, each property also conveys a namespace and a flag indicating if the property should be marked as ‘promoted’. The ‘GlobalSchema’ actions are a courtesy feature that assign values to properties defined in the Microsoft.BizTalk.GlobalPropertySchemas assembly via schemas registered in the BizTalk.System application. When using these actions, there is no need to specify a property namespace, as this is determined dynamically using the property name.

## Routing policy

The ESB pipeline components automatically set the BTS.OutboundTransportLocation and BTS.OutboundTransportType properties on messages in order to support dynamic routing via a Send Port. These provide external endpoint and optional transport (adapter) selection. Attention must still be given to routing messages to the dynamic Send port, e.g. by promoting properties via the BTS Properties mechanism. In addition, depending on the transport type, it will generally be necessary to write additional adapter-specific configuration properties on the outbound message. Again, this can be handled using the BTS Properties mechanism.

The ESB pipeline components provide specific support for SOAP web service calls. Developers can optionally specify a SOAP Action header in the policy. In this case the ESB pipeline components write the following three adapter-specific properties on a message:

Table 20: Routing Policy Properties

|  |  |  |
| --- | --- | --- |
| Namespace | Property Name | Adapter |
| http://schemas.microsoft.com/BizTalk/2003/system-properties | SOAPAction | Legacy SOAP |
| http://schemas.microsoft.com/BizTalk/2003/wse-properties | SoapAction | WSE |
| http://schemas.microsoft.com/BizTalk/2006/01/Adapters/WCF-properties | Action | WCF |

## Retry policy

The ESB pipeline components implement support for multi-level retry policy. Each level is modelled using a different directive and specifies a retry count and interval. To implement multi-level retry approaches, developers must use some run-time mechanism to store and forward messages at the designated intervals. This is generally done using loops in Orchestration code. The Orchestration engine will persist state between retries.

No equivalent mechanism has been implemented in the ESB pipelines. The pipeline components will perform a level 0 retries only. NB, if you define retry policy at level 0 in multiple directives, the ‘winning’ directive is arbitrary, depending on the order in which directives appear in the collection returned by the Resolver Service. The ESB pipeline components assign the following properties to the message. These properties are honoured on Send ports and override any configured retry settings. They are supported on Dynamic Send ports.

Table 21: Retry Policy Properties

|  |  |
| --- | --- |
| Namespace | Property Name |
| http://schemas.microsoft.com/BizTalk/2003/system-properties | RetryInterval |
| http://schemas.microsoft.com/BizTalk/2003/system-properties | RetryCount |

## Validation policy

Developers can define directives to handle validation using the definitions in the ESB Validation Actions vocabulary. When validation is applied in the context of an ESB pipeline component, the results are automatically communicated using message properties. The ESB pipeline components report the counts of error, warning and information records generated by the validation policy using message properties. These properties are promoted and can be used to route messages appropriately. In addition, the pipeline components output error, warning and information messages via ‘written’ (non-promoted) properties. Each property value contains a list of messages separated on different lines. The six properties are listed in the following table:

Table 22: Validation Policy Properties

|  |  |  |
| --- | --- | --- |
| Namespace | Property Names | |
| Namespace | Counts | Messages |
| http://solidsoftreply.com/esb/2015/validation-properties | ErrorsCount | Errors |
| http://solidsoftreply.com/esb/2015/validation-properties | WarningsCount | Warnings |
| http://solidsoftreply.com/esb/2015/validation-properties | InformationCount | Information |

# Defining BAM Policy

The ESB Libraries provide a rich mechanism for implementing dynamic observation of events through the BAM framework. Developers can use the libraries to instrument their code to collect data. Service mediation policy defines interception points which can be applied as required in BizTalk Server or other services.

## Motivation

Business Activity Monitoring (BAM) is a framework for observing, recording and processing events of interest to the business. It addresses the concept of a business activity which is modelled as a discrete set of events and the data associated with those events. BAM provides tooling for Business Analysts to define business activities, milestones and data values, together with views. This information is used by BizTalk developers to provision sets of ‘primary import tables’ in a SQL Server database provided by BizTalk Server. BizTalk developers then have a choice. They can use ‘interception’ tooling such as TPE (the BizTalk Tracking Profile Editor) or the WCF BAM Interceptor to configure event observation, or they can write BAM code to directly observe and record events using the BAM Event Observation API.

The direct coding approach, using direct or buffered ‘event stream’ objects, is widely used and very flexible. However, it has obvious drawbacks. The main problem is that leads to tightly-coupled solutions in which event observation is ‘hard-wired’ in application logic, normally in the context of a BizTalk orchestration. Any evolution of the use of BAM over time will generally require costly re-engineering of the business logic, leading to inflexibility and reducing business agility.

The alternative approach is to use an interception framework. Interception defines ‘steps’ within the business activity at which event observation is carried out. These ‘interception points’ use configurable general-purpose components or functionality. Instead of writing code, developers configure the event observation that should occur at each interception point. Changes can then be introduced easily and quickly in the production environment.

The ESB Libraries use the BAM Interception framework provided as part of the Event Observation API. They provide a highly flexible approach. Like any interception framework, the work done at each step can be configured, in this case using a BAM Interception (or ‘trackpoint’) policy. In addition to this, the Service Mediation capabilities allow the definition of the interception points, themselves, to be driven via policy. Developers can implement rules that allow interception points to be switched on or off, made conditional on message type or other criteria or even overlaid at the same location as other interception points. The work done at each interception point can then be individually controlled and changed.

BAM is flexible with regards to the use of event streams. Event recording depends on the use of unique ‘activity IDs’ to identify discrete fragments of work within a given instance of business activity. The ESB Libraries exploit this is ensure that developers can ‘mix’ the interception model with explicit coded use of event streams. This includes intermediate models where extended versions of event streams can be used with BAM configuration data supplied by the ESB Libraries.

## Terminology

Terminology and semantics can be barriers to understanding the BAM model. Different overlapping terms are used, sometimes without adequate precision. This section provides the definition of terms used by the ESB Libraries and this document.

### Business Activity

A business activity is an identifiable, ongoing operation that an organisation undertakes in order to create value. Business activities are composed of repeatable tasks. The ESB Libraries and the BizTalk BAM framework are applicable to business activities which depend on services provided on the service bus. Because the tasks in a given business activity are repeatable, we can think of a specific set of tasks as an ‘instance’ of a business activity – e.g., processing a single sales order is an instance of the sales order processing business activity.

### Milestone

A business activity is structured around the notion of milestones. At a minimum, each activity has identifiable start and end milestones. Milestones serve as, or represent, ‘states’ within a business activity. They may represent locations in the flow of the activity where responsibility is handed off to a different part of the organisation, where a significant communication occurs with external entities (e.g., customers) or where a system of record is updated. Milestones often represent the hand-off boundaries between automated tasks and human workflow. The BizTalk BAM framework represents milestones as simple date/time values.

### Step

A step is a sub-unit of a business activity. It represents a discrete set of actions that occur together at some location within the activity. These actions are often associated with identified milestones within the business activity flow. They begin after a previous milestone and must be carried out in order to ‘arrive’ at the next milestone. A step, therefore, often represents the work that is done to transition an instance of a business activity form one milestone (state) to another. This association between steps and milestones is loose, and it is quite normal to have steps that do not map directly to milestones or which map to multiple milestones.

### Interception Point

An interception point is a concept defined by a monitoring technology such as the BAM interception framework. Each interception point is a location within an automated business activity task where the task flow is ‘intercepted’ by configurable functionality that observes events and records relevant information. Interception points are strongly associated with the notion of steps. The BizTalk BAM interception model assumes that all monitoring work associated with a given step will happen at that one location in the task flow. The interception point is therefore linked to the definition of a step.

Interception points provide just one model for event observation. Another approach is to tightly couple event observation and data recording with the flow of the automated task. In the BizTalk BAM framework, this is achieved programmatically using the BAM Event Observation API to instrument automated tasks, such as BizTalk orchestrations, using explicit calls to event stream objects.

### Trace Fragment

The term ‘fragment’ has a specific definition within the BizTalk BAM interception framework. It refers to a set of configuration items, called trackpoints, which define the BAM actions for a given interception point. Fragments, therefore, map strongly to the concept of a ‘step’. At run-time, each instance of each fragment is represented by a unique identifier. See Activity ID and Continuation ID below.

A trace fragment represents a complete unit of work done by the BAM framework at an interception point. Each fragment has a well-defined beginning and end and may specify a number of additional actions for recording milestones and other data. Fragments can be chained together using an approach called ‘continuation’. The entire chain often represents a complete business activity. A trace fragment can also be related to other fragments. Each of these additional fragments may be the start of another chain of fragments. This is termed ‘relationship’ and may be used to model sub-activities or other forms of association.

### Root Fragment

A single business activity may contain many steps and interception points, and may therefore define many trace fragments that are chained together through continuation. A fragment that represents the beginning of a business activity is termed a ‘root’ fragment. Logically, it is the first fragment is a sequence of trace fragments that represent an entire business activity. Each non-root fragment represents a subsequent step in the ‘continuation’.

### Activity ID

In the BizTalk BAM interception framework, an activity ID is a unique string identifier up to 128 characters in length that identifies an instance of a root fragment at run time. It is used for that one fragment instance, only, and not for continuation fragments. However, it can be thought of as an identifier for the entire chain of continuations.

Activity IDs are central to the use of the BAM framework, and are not specific to the ESB Liraries interception framework. They are used with event streams in the Event Observation API where they identify groups of actions that represent a step. These actions are invoked through a series of API calls, but they must, when taken together, represent a complete unit of work, just as is the case with fragments. They must begin a new activity instance and end that instance.

### Continuation ID

A continuation ID is the equivalent to an Activity ID for a non-root fragment. It is a unique string identifier up to 128 characters in length that identifies an instance of a fragment at run time. The fragment continues a previous fragment. The previous fragment must register its continuations at run time. It does so using the Continuation ID under which it will be continued. Continuation IDs are sometimes referred to as ‘continuation tokens’.

The generation and communication of continuation IDs can be a problem in BAM. In some cases, it may be necessary to create or use additional mechanisms to pass the each Continuation ID from one fragment to another. For example, they may be communicated as a BizTalk message property or recorded in a custom data store. For business activities that define a large number of fragments in a continuation, this can be difficult to manage. Continuation IDs must be unique, and cannot be the same as an Activity ID.

One way to simplify the model is to use a convention for manufacturing Continuation IDs. For example, the single Activity ID can be communicated through the business process flow in some appropriate fashion. Each Continuation ID can be manufactured by prefixing or suffixing the Activity ID with a token string representing a fragment. This convention (specifically, prefixing) is explicitly supported by the BizTalk BAM interception framework and the ESB Libraries.

### Trackpoint

A trackpoint is a configuration item used by the BizTalk BAM interception framework. It represents the configuration for a single BAM action within a fragment. The fragment is configured as a set of trackpoints that, together, describe a single discrete unit of work. A set of trackpoints, therefore, represents the entire configuration for a single step.

The BAM interception framework does not mandate any mechanism for serialising and storing trackpoint data. It simply provides a .NET class called ActivityInterceptorConfiguration which must be populated programmatically at run time. This is handled by the ESB Libraries using a BAM Interception policy, executed using the BizTalk Business Rules Engine, to provide the trackpoint configuration data.

### Continuation

Continuation plays an important role within the BAM framework. It is the mechanism by which trace fragments are chained together to represent an entire Business Activity. Continuation allows each fragment to run safely on a different thread, in a different process or even on a different machine. The run-time initialisation of each step must happen in chronological order. At the first step, BAM registers a continuation in the Primary Import Database under a given Continuation ID. This record must exist in the database before the activity can be continued at a subsequent step. However, writing events to the database can happen out of order. BAM will ensure data consistency using its knowledge of the sequence of continuation.

This capability allows the use of efficient event stream buffering within the BAM framework. In buffering mode, event streams serialize batches of BAM actions to an intermediate database (the BizTalk Message Box). This greatly reduces the amount of network and database traffic. Once persisted, each batch of actions is then decoded on a separate background thread on a SQL Server machine. Total ordering is imposed, and the actions are replayed to the Primary Import Tables.

When using the event streams directly via the Event Observation API, developers need only use continuation when the event stream is buffered and when the business activity spans multiple threads, processes or machines. BAM also supports a ‘direct’ mode in which events are written, one by one, to the Primary Import Database, rather than as batches to the event buffer tables in the BizTalk Message Box. Direct mode can be inefficient, but is a very simple model that may be appropriate when BAM is constrained to small amounts of work on single threads or where there is a need for real-time event observation.

The BAM interception framework models trace fragments with configurable trackpoints. The only safe way to chain fragments together is to use continuation. Because developers are removed from direct interaction with event streams, the full implication of using fragments as discrete units of work would not be obvious, and any mechanism that avoided continuation when chaining fragments would be inherently unsafe. Hence, the BAM interception framework requires continuation to be used to chain fragments together, even if all the code runs on a single thread.

### Extension

Extension is a concept introduced by the ESB Libraries. It is not defined by the BizTalk BAM framework. It can be used with the trackpoint event stream classes provided by the ESB Libraries. There are two such streams. Both are BAM event streams and can be used like any other event stream. The TrackpointDirectiveEventStream defines the core functionality to exploit trackpoint configuration accessed via a directive defined by the ESB Library service mediation features. The OesTrackpointEventStream is derived from this class, but uses an instance of the OesEventStream class internally to co-ordinate event capture with the Orchestration Event Stream functionality provided by BizTalk Server. The OES is not a true BAM event stream, but rather a set of static methods that developers use to coordinate BAM actions with the BizTalk orchestration engine. The OesEventStream wraps calls to these static methods in a BAM event stream object. The OesTrackpointEventStream class drives the internal use of this event stream from trackpoint configuration returned by the ESB Library functionality.

Extension is implemented using continuation. However, it hides the process of generating Continuation IDs and manages some of the BAM actions automatically. It allows developers to chain fragments by switching the trackpoint event stream between different directives at run time. Each directive defines a separate BAM step. An alternative approach is to define a set of named extensions to a BAM step within a single directive. Developers can chain these ‘step extensions’ by switching between them at run-time. Because extension uses BAM continuation behind the scenes, it offers a safe model which may be more intuitive than using the continuation model explicitly.

### Relationship

The BizTalk BAM framework allows developers to define relationships between different business activities. In this case, one activity is not a continuation of another, and there is no requirement for total ordering of BAM actions between the two activities (there is, of course, a total ordering requirement for the actions within any one activity). Relationships can be used to model a number of requirements, including the concept of sub-activities within a parent activity and the modelling of association between different activity types.

The following diagram illustrates the use of terminology described above.

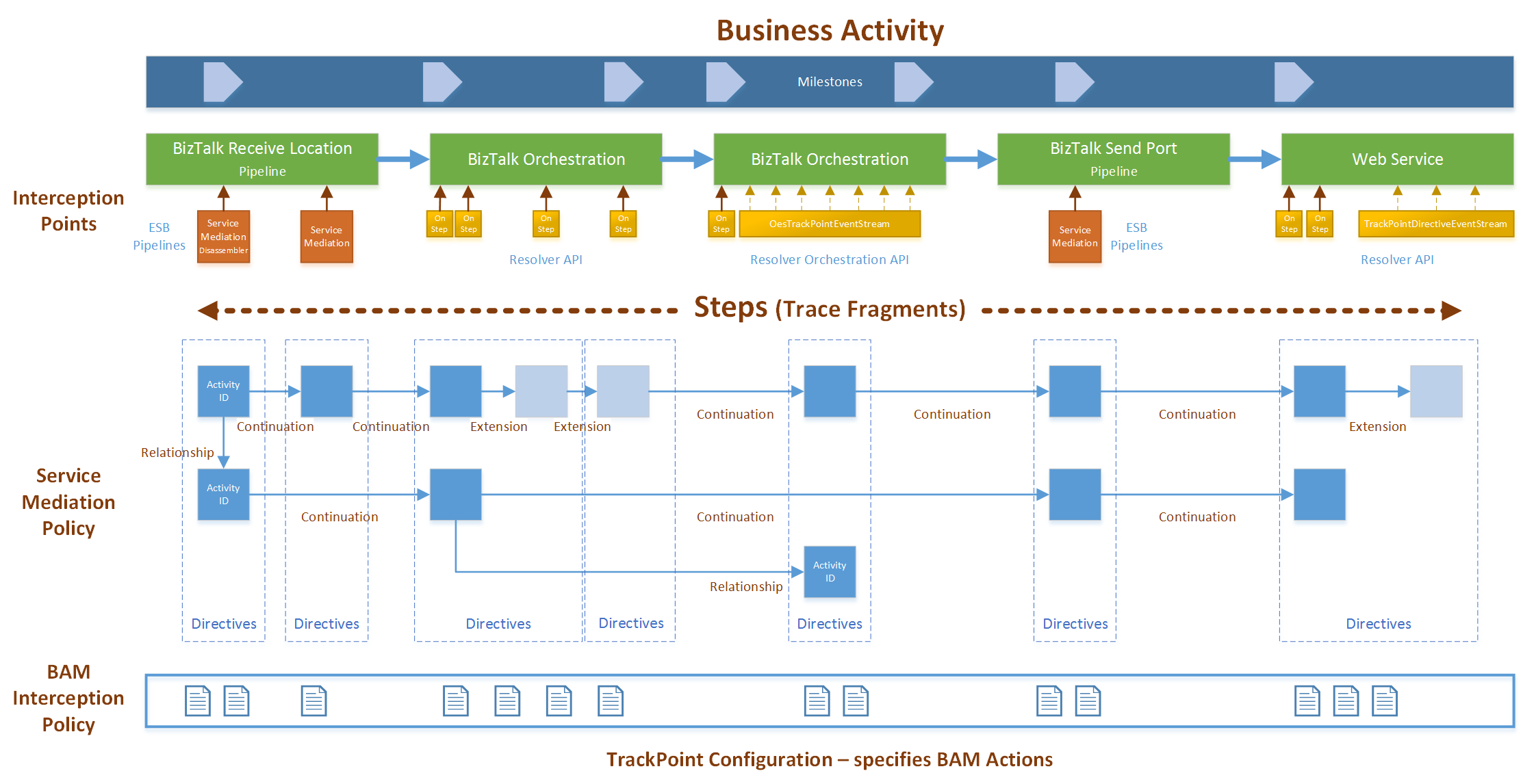


Figure 14: BAM Terminology in ESB Libraries

## Advantages of using the ESB Libraries BAM Interception

The ESB Libraries support a number of strategies for collecting BAM event data at run time. The primary motivation for using the libraries is to build dynamic solutions that adapt to changing requirements while minimising the need for costly re-engineering. This includes strategies that allow the use of BAM to evolve and change over time.

BizTalk developers often tightly couple their BAM logic with orchestration code using the Event Observation API. This is a familiar and versatile model. However, it assumes that BAM requirements are well defined from the start, and that these requirements will not change over time. Any change is likely to require costly re-engineering of orchestration logic.

BizTalk Server offers alternative an alternative approach using the BizTalk Tracking Profile Editor (TPE). Developers use this tool to define interception configuration and apply it to BizTalk Orchestrations and messaging ports. TPE configures a BAM interception framework built into BizTalk Server and supported by data held in SQL Server tables. Configuration is defined by mapping message payload items and property data from locations in orchestrations or messaging ports to BAM-defined milestones or data values, or to a BAM actions that begin activities, continue existing activities or define relationships.

The TPE provides a simple user interface for defining mappings and manages the deployment of interception configuration into the BizTalk Server environment. However, it has some limitations. It can only track data within message payloads or modelled as message properties. It cannot access data provided in orchestration state without constructing a message to contain that data. Also, the TPE can only instrument BizTalk Server components and cannot be used more widely across services that reside outside of BizTalk server.

The ESB Libraries provide an alternative BAM interception framework with the following advantages:

* Developers have more control over interception locations. For example, they may implement multiple interception points in a single pipeline. In combination with the dynamic transformation features of the ESB Libraries, this allows them to intercept messages before and after transformation. They can also perform BAM actions on each disassembled message in a pipeline.
* Developers have more control when combining direct use of event streams with interception. The ESB Libraries define a graduated approach in which event streams can execute entire steps in a single method call or spread the step over multiple method calls.
* Developers can instrument their own code, including code that resides outside BizTalk Server, using a common interception mechanism driven from centralised policies. Note that Microsoft’s BAM Event Observation API assembly, which defines the BizTalk BAM interception framework on which the ESB Libraries depend, is freely redistributable to support these scenarios.
* Developers can resolve steps dynamically at interception points. The steps can be resolved according to message type, location or any other parameter provided to the ESB resolver.
* Developers can resolve BAM configuration dynamically. Configuration can be resolved according to activity name or step name.
* Developers can apply multiple steps at a single interception point. This makes it easy to implement multiple overlapped BAM tracking profiles. For example, a single interception point may be used to record business-specific order processing information together with custom health tracking information captured as a different BAM activity.
* Developers can use an ‘extension’ model that hides much of the detail of continuation from view, making it easier to handle continuation across multiple steps.

## BAM Strategies

The ESB Libraries support many different strategies for instrumenting solutions. This section discusses a representative set of ideas that developers can consider and discusses how the ESB Libraries might be exploited for BAM.

### Pre-define steps for future extension

Organisations taking on BizTalk Server for the first time often find it difficult to articulate detailed requirements for BAM from the start of their journey. Even where organisations understand, in a general sense, their need to monitor and track business activities, they may not be able to provide sufficient detail to developers to allow them to implement BAM activities that adequately meet that need. One strategy is to allow the development team to implement basic instrumentation that capture obvious and significant milestones and data items. Once these events are being observed in the production environment, the ability to report on this data leads naturally to a broader understanding across the business of the capabilities of the middleware. Additional detailed requirements then flow from this better understanding as the business starts to naturally demand better and more detailed information.

This strategy works best if emerging business requirements can be met rapidly without the need for costly re-engineering of the business logic. The use of an interception framework is crucial in enabling this, and the ESB Libraries provide a very flexible approach. However, to avoid the need for re-engineering, the code must contain instrumentation to define interception points. In this strategy, developers introduce service mediation pipeline components and ‘On Step’ method calls to define a range of interception points. These can be introduced liberally across the code at any point that might be of interest in the future for BAM event observation. To begin with, many or all of these interception points may be left un-configured. At run-time, they act as ‘null returns’ and introduce no significant overhead. At a later time, they can be configured through policies in the BizTalk Rules Framework to start to collect event data.

When BAM interception locations are defined using the ESB Libraries, they can be created to use a single named directive or left ‘open’ to use multiple directive, each of which can define a separate step. Leaving the interception points open provides the greatest flexibility, but there may be scenarios where developers prefer to associate their interception points with a specific named step. This step does not need to be immediately defined or configured in the service mediation and BAM interception policies. If the step name is not defined, the run-time code will continue without error.

### Address overlapped tracking needs

BAM is often used to meet multiple overlapped tracking requirements. For example, a single flow of messages through BizTalk Server may be instrumented to collect business-specific information that records total volumes and values of orders. At the same time, the code may be instrumented to maintain an audit trail of order processing and also to provide an optional diagnostic trace that can be switched on in production when required. The ESB Libraries make it easy to configure these as multiple overlapped steps, each of which can be managed independently. Where appropriate, a single interception point in the code can be used to handle these overlapped steps.

### Extend through activity versioning

One of the issues with BAM is that, as activity definitions evolve and change, this sometimes requires re-provisioning of BAM primary import tables. BAM supports in-place updates of BAM activity definitions, but some changes, such as deletion of milestones and data items, requires dropping the existing tables and re-provisioning of new tables. This is a disruptive change which generally requires careful thought with respect to data backup and migration of data from earlier to a later versions of activity tables.

One strategy for handling these issues is to ‘version’ activities by retaining the older activity tables and introducing new tables (i.e., a new activity) for the new version. The desirability of this approach depends on a number of factors, one of which is the impact on existing BizTalk applications which may need to be re-engineered. The ESB Libraries provide a model which supports this approach, allowing applications to switch seamlessly to the new version of the activity with no re-engineering.

### Mix dynamic and static approaches

Interception is a powerful model, but it can be problematic in some respects. It can be difficult to adequately integrated step-based interception with procedural logic – e.g., decision points and loops in a BizTalk orchestration. Interception frameworks are also constrained with regard to running custom logic and accessing custom data sources to obtain relevant event data.

When using the TPE, BizTalk developers will generally deal with these issues by using the BAM Resolution API alongside TPE interception points. This requires careful management of activity and continuation IDs to ensure that API code works seamlessly with configured interception. The solution uses multiple BAM event streams to track a single activity and its continuations.

The ESB library improves on this model by providing rich event stream classes that utilise the trackpoint configuration returned by a BAM interception policy. The libraries provide a general-purpose TrackpointDirectiveEventStream class and a derived OesTrackpointEventStream, class for use in BizTalk orchestration code. These event streams and can be used like any other BAM event stream, and can be initialised for either buffered or direct mode. In addition, they can be used to perform ‘all-at-once’ processing of an entire fragment in a single OnStep method call.

The classes two provide intermediate modes of execution. The first provides a mechanism for that allows multiple actions of a single type (e.g., updates, continuations) within a given fragment to be executed in a single method call. Execution of the entire fragment is therefore spread over multiple method calls, but is still driven entirely by configurable trackpoint data.

The second intermediate approach overloads a number of the per-action method calls defined by the base event stream class. These methods provide fine-grained control in which each action in a fragment is executed individually as a separate method call. However, the overloaded method is parameterised internally using the trackpoint data.

The trackpoint event streams provide extended control of BAM processing that can exploit the configured trackpoint data provided by a BAM interception policy. They support the directive-driven step ‘extension’ method which hides much of the detail of continuation. They can be switched freely between different directives when using the extension model. They allow developers to freely mix dynamic and static approaches at different levels on a single event stream in order to meet different solution demands.

### Support the extended service bus

BizTalk BAM is supported as part of the BizTalk Server product and depends on the use of BizTalk Server database functionality. However, it can be used beyond the boundary of BizTalk Server machines. This is well-aligned to a core design principal of the ESB Libraries, namely that it provides good support for the extended service bus. The TrackpointDirectiveEventStream class described above is implemented as part of the Resolver library which is designed to be used in any .NET container or code location. For example, it could be used to instrument a custom web service to observe and record event information. Event observation is driven through the same step-wise trackpoint configuration used within the BizTalk environment.

The ability to use the ESB Libraries across different BizTalk Server and non-BizTalk Server services is an important sect of building the extended service bus that is not tied to any one specific technology or platform. Note, however, that the ESB Libraries are currently limited to running within .NET or Mono CLRs.

## Configuring Service Mediation Policy for BAM

The definition of BAM policy has two distinct aspects, and involves two separate policy types. BAM steps are defined in service mediation policy as part of directive definition. Trackpoint configuration is handled in BAM interception policies.

Service mediation policies define BAM steps as simple names. At runtime, these names are treated as interception points applied as specific locations in the code. A directive will generally define a single step, but they can define a second step for post-transformation work. The following step configuration styles are valid in any one directive:

Table 23: Directive Step Configuration Styles

| Step Configuration Style | Description |
| --- | --- |
| No steps | The directive defines no BAM steps. |
| Single step; no transformation | The directive defines a single BAM step which is applied to each message. |
| Single step with transformation | The directive defines a single BAM step which is applied to each message before it is transformed by the ESB Libraries. |
| Single post-transformation step with transformation | The directive defines a single BAM step which is applied to each message created as a result of transformation by the ESB Libraries. |
| Two steps with transformation | The directive defines two BAM steps. The first is applied to each message before it is transformed by the ESB Libraries. The second is applied to each message created as a result of transformation by the ESB Libraries. |
| Any of the above with step extension | Each of the previous configuration styles can be extended with one or more step extensions. Step extensions can be applied to pre- and post-transformation steps. |

**NB.** It is possible to configure a directive with a post-transformation step even if there is no transformation. In this case, the post-transformation step and any extensions are ignored. Other configurations will result in a run-time exception.

All steps in any one directive must be assigned to the same named BAM interception policy. If an interchange represents more than one activity, introduce multiple directives to handle BAM requirements and configure each one with a different BAM interception policy.

## Configuring BAM Steps and Trackpoints

The directives provided by a service mediation policy can specify the inclusion of named BAM steps. The Resolver component supports the implementation of configurable BAM Interception points for these steps. BAM Interception is used in the pipeline components provided as part of the ESB Libraries. It is supported using the TransformWithInterception() and OnStep() methods of the Resolver Directive class. Steps can also be used with specialised BAM event streams provided by the Resolver and BizTalk.Orchestration components.

A step defined in a service mediation policy specifies the type of type of event stream (buffered or direct) that will be used, together with SQL Server connection information, threshold values, etc. However, the service mediation policy does not directly specify the tracking that should be done at each step. Additional BAM interception policies can be defined for this purpose. These policies can be specified as part of a directive. A default trackpoint policy can be configured in a configuration file. On approach is to configure the ESB Libraries to use the same policies for BAM interception that are used for service mediation.

BAM interception policies define ‘trackpoints’ for business activities and steps. Each activity and step is identified by name. Any one activity may include multiple message exchanges or may be related to one part of a message interchange. A BAM interception policy defines a set of trackpoints for a given activity. Each trackpoint is related to a named step. A single step can define multiple trackpoints. Each trackpoint definition registers a given BAM action such as starting an activity, continuing an activity, defining a relationship between two activities or extracting a data item from the message.

When steps are used to drive BAM Interception via the pipeline components or the TransformWithInterception() or OnStep() methods of the Resolver Directive class, they have the same semantics as ‘steps’ in BAM Interception. Each step must contain a ‘begin’ action. This will either mark the start of an activity or the continuation of an existing activity. Internally, each step must contain an ‘end’ action. However, this is inferred and is not stated explicitly in the BAM interception policy. For BAM Interception, a step is an entire unit[[4]](#footnote-4) of tracking work undertaken by the BAM framework.

When steps are used in conjunction with event streams, they have looser semantics. They may define only part of a unit of tracking work. The developer exploits the specified trackpoints as required. NB. When using the BAM Interceptor, trackpoints may be defined in any order in trackpoint policy. The BAM Interceptor automatically groups and sorts trackpoints internally. This is also true when using the extended observation API discussed below.

At run time, the Resolver will determine a collection of steps based on the definitions in the Service Bus policy. If steps have been defined for a given interchange, the Resolver will retrieve the trackpoint policy for each step. It will create BAM interceptors and cache the BAM trackpoint information. The lifetime of each interceptor is tied to the lifetime of the Resolver instance. Be careful not to cache and re-use Resolver instances when using BAM Interception, as this may lead to incorrect behaviour and incomplete activities recorded in the BAM database. Remember that the Resolver already makes use of extensive caching of policy directives. Resolver instances should be instantiated per message interchange, and released immediately after the interchange has been completed.

To configure BAM interception, use the following trackpoint instructions in your rule actions. These are defined in the ‘ESB BAM Policy’ vocabulary:

Table 24: Vocabulary Definitions for BAM Interception Trackpoint Instructions

| Definitions | Description |
| --- | --- |
| Start Activity  One instance of this instruction, the ‘Continue Activity’ instruction, the ‘Continue Activity with Unique ID Prefix’ instruction or the Extend Step instruction allowed per step | Register the start of a BAM activity. This instruction indicates that the step is a root fragment.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | |
| Continue Activity  One instance of this instruction, the Start instruction, the ‘Continue Activity with Unique ID Prefix’ instruction or the Extend Step instruction allowed per step | Register a BAM continuation for an activity step.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | |
| Continue Activity with Unique ID Prefix  One instance of this instruction, the Start instruction, the ‘Continue Activity’ instruction or the Extend Step instruction allowed per step | Register a BAM continuation for an activity step using a unique ID prefix.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | | **Prefix**  Cannot be null or empty | Case-sensitive string | |
| Extend Step  One instance of this instruction, the Start instruction, the ‘Continue Activity’ instruction or the ‘Continue Activity with Unique ID Prefix’ instruction allowed per step  The step name cannot be the same as the extended step name | Register the extension of a previous activity step. This is really a form of BAM continuation, automated via an ESB directive at run time.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Extended Step Name**  Cannot be null or empty | Case sensitive string. | |
| Record Milestone | Register data extraction for a BAM milestone for an activity step.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Milestone Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | |
| Extract Data | Register BAM data extraction of a named data item for an activity step.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Data Item Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | |
| Enable Continuation | Register the enablement of a BAM continuation for an activity step.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | |
| Enable Continuation with Unique ID Prefix | Register the enablement of a BAM continuation for an activity step using a unique ID prefix.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | | **Prefix**  Cannot be null or empty |  | |
| Define Relationship | Register the extraction of relationship information between BAM activities.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Related Activity Name**  Cannot be null or empty | Case sensitive string. | | **Extended Step Name**  Cannot be null or empty | Case sensitive string. | |
| Define Reference | Register the definition of a reference to an external document or location.  **Parameters:**   |  |  | | --- | --- | | **Step Name**  Cannot be null or empty | Case-sensitive string | | **Reference Name**  Cannot be null or empty | Case sensitive string. | | **Reference Type**  Cannot be null or empty | Case-sensitive string | | **Extraction Info**  Cannot be null or empty | XPath or macro | |

At runtime, each instance of a named BAM activity is identified by an Activity ID. Each BAM Trackpoint policy allows an extraction string to be defined as part of registering the start of an activity. These extraction strings can either be XPaths (for XML data), macros (see below) or .NET format strings. Format strings are extended to support macros as well as positional arguments.

Instead of defining an Activity ID, a policy may define continuations of a given activity at specific steps. The TransformWithInterception() method is designed to use extraction strings. This method will perform transformation if specified. In addition, an OnStep() method is provided for direct invocation of BAM interception. This also supports extraction strings.

## Extraction Strings

Extraction strings are supported when defining BAM Tracking policy. They support three forms:

* **XPaths**XPaths can be used to extract data from XML messages or other XML content.
* **Format Strings**These are .NET format strings with additional support for macros. See <http://msdn.microsoft.com/en-gb/library/system.string.format.aspx> for information about format strings. The resolver will expand macros automatically. To define a format string, enclose it in braces. Specify positional paramters using integers enclosed in braces. Enclose any additional macros (see below) is braces. E.g.,

{PO {1} was received on {now:D} for activity id {0}}

Format string positional parameters may not be available for use in all contexts. For example, the pipeline components provided by the ESB libraries do not support their use. In the context of a BizTalk Server orchestration, or other locations that use the API directly, use the overloaded OnStep method of the Resolver’s Directive class to assign values for each parameter.

* **Macros**Macros are expanded automatically to yield values. They can be used individually or in the context of a format string (see above). When used individually, simply enclose in braces:  
    
   {guid:D}

The following macros are defined by the resolver. Each of these is formattable. A formattable macro has the following form:  
  
 {<macro>[:<format>]}

The macro name is not case sensitive. However, the format specifier is.

Table 25: BAM Trackpoint Extraction String Macros

|  |  |
| --- | --- |
| Macro | Description |
| date | The date component of the current date and time. |
| day | The current day of the month. |
| dayofweek | The current day of the week. |
| dayofyear | The current day of the week. |
| hour | The hour component of the current date and time. |
| millisecond | The millisecond component of the current date and time. |
| minute | The minute component of the current date and time. |
| month | The month component of the current date and time. |
| now | The current date and time on this computer, expressed as the local time. |
| second | The seconds component of the current date and time. |
| ticks | The number of ticks that represent the current date and time. |
| timeofday | The current time of day. |
| today | The current date. |
| utcnow | The current date and time on this computer, expressed as the Coordinated Universal Time (UTC). |
| year | The year component of the current date and time. |
| guid | A new GUID instance. <http://msdn.microsoft.com/en-us/library/97af8hh4(v=vs.110).aspx> |

When formatting date and time values, see the existing documentation at the following locations:

<http://msdn.microsoft.com/en-us/library/az4se3k1(v=vs.110).aspx>

<http://msdn.microsoft.com/en-us/library/8kb3ddd4(v=vs.110).aspx>

<http://msdn.microsoft.com/en-us/library/ee372286(v=vs.110).aspx>

<http://msdn.microsoft.com/en-us/library/ee372287(v=vs.110).aspx>

NB. Unformatted macros return data in the appropriate datatype for their .NET equivalent. E.g., {now} returns a DateTime value. However, when using format specifiers, macros always return strings. When setting a milestone value, use an appropriate unformatted macro to return a DateTime.

The following macro names are reserved:

* call
* eval
* if
* regex

Additional macros may be supported by additional components such as the pipeline components provided with the libraries.

* **Properties**The ESB Libraries support a ‘property’ macro. This is used to access a populated dictionary of name/value pairs. The macro evaluation engine does not restrict the semantics of these properties. However, in a given context, properties may have specific semantics. For example, the pipeline components provided by the ESB Libraries will automatically populate the property dictionary with BizTalk Server message property values. Similarly, when using the BizTalk.Orchestration library, developers can assign a BizTalk Server message to a BamStepData object to populate the properties dictionary with BizTalk message properties. In the context of an orchestration, developers are free to add additional properties to the list. To access property values in rules, use the following macro form:

{property:<name>}

## Modelling Step Extension

Microsoft’s BAM Interceptor defines steps as discrete units of work bounded by a begin action (‘start activity’ or ‘continue activity’) and an end action. The ESB Libraries implement the notion of an ‘extend step’ action (really a form of continuation). Step extensions are defined in section 9.2.11. Section 9.5 describes how BAM steps and step extensions are configured in Service Mediation policies.

The concept of step extension is specific to the ESB libraries. It is a form of continuation in which much of the housekeeping work, including management of continuation IDs and automated enablement of continuations, is managed through an instance of one of the ESB Libraries’ trackpoint event streams. These are the TrackpointDirectiveEventStream and the OesTrackpointStream. When instrumenting BizTalk Server orchestrations, developer should generally favour the use of the OesTrackpointStream.

Step extension is designed specifically for scenarios where continuations are performed in a single process. When using Microsoft’s Event Observation API, continuation is normally used to continue activities across different processes. However, the BAM interception model often requires continuation within a single process. This comes as a surprise to experienced BizTalk developers who have not previously coded with the BAM interception model. Because each step is a discrete and well-defined unit of work in which all BAM operations are done at one location (a single method call) in the code, continuation is necessary for activities that involve multiple steps.

BAM continuation provides the only inherently safe mechanism for performing multiple interception steps. Even in a single process, steps may be executed on different parallel threads of execution. BizTalk orchestrations add the further complication of dehydration and re-hydration. A long-lived orchestration may be executed across multiple processes on different machines. Of course, the BizTalk orchestration engine provides Orchestration Event Stream (OES) functionality which synchronises the buffered writing of BAM events with persistence points. However, the use of OES is optional, and edge cases may still arise where continuation is necessary to ensure that integrity is maintained across steps.

BAM continuation involves careful management of tokens (continuation IDs). Tokens may be manufactured according to convention, or may be communicated between different continuations of a given activity. The lifetime of each continuation must also be managed carefully, ensuring that it is started and ended correctly. This housekeeping adds to the development burden. When using multiple steps in a single process or BizTalk Server orchestration, the additional effort feels cumbersome and unnecessary to developers.

The step extension model helps to reduce this burden. Several housekeeping tasks are delegated to the specialised BAM trackpoint event streams. This includes the manufacture of unique continuation IDs according to a pre-defined convention and automated lifetime management for each step. Step extension must be modelled using directives defined in Service Mediation policies. Directives are then used to configure step extension at run-time.

A step extension is itself a step that continues a previous step. There are two ways to model step extension in Service Mediation policy. The first is to define each extension as a normal step. This maximises flexibility when composing steps. Any one step can be used to continue any other step. However, step extensions ignore any use of the ‘Configure a BAM interceptor’ directive instruction. They extend the root fragment (see 9.2.6) using the same connection string, buffering and flush threshold. This better supports the semantics of step extension.

To use this first model, create a number of directives in one or more Service Mediation policies and define each one using BAM instructions. Each directive specifies a step name and, optionally, a BAM interception policy. Any steps that may be used as root fragments may optionally define interceptor configuration. Publish and deploy the Service Mediation policy(s) and all related BAM interception policies. In code, create an instance of the TrackpointDirectiveEventStream or OesTrackpointStream class over a directive that specifies a root fragment. To extend the root fragment or any additional fragment, use the ExtendActivity() method, passing it the directive for the next extension.

The second model is less flexible, but may be easier to understand in many scenarios. In this model, all step extensions are defined in the same directive that defines the root fragment as a BAM step. This ties the extensions to a single root fragment. Use the ‘Extend step’ directive instruction to define one or more extensions to a named BAM step. Each extension is defined by name and will use the same configuration and BAM interception policy as the root fragment.

In code, create an instance of the TrackpointDirectiveEventStream or OesTrackpointStream class over a directive that specifies a root fragment, as for the first model. To extend the root fragment or any additional fragment, use the ExtendActivity() method, passing it the name of the extension.

For either model, you can use step extensions in any order you like, and revisit previous extensions if required. You can even use the same extension repeatedly – e.g., in a loop. The SB Libraries will ensure that each time a step is extended, a unique continuation ID is generated internally.

When using step extensions, you cannot use the OnStep() method of Directive objects or Directives collections to perform step actions. This may surprise developers, but is logical. Remember that step extension is controlled by trackpoint event streams and that the semantics of step extension suggests that a step is not ‘complete’ until all required extensions have been performed. The OnStep() Directive method performs a discrete unit of work according to trackpoint configuration provided by a BAM interception policy. However, directives are not aware that they may be being used by a trackpoint event stream and therefore cannot synchronise with the housekeeping provide by these objects. Step extension can only be used with the methods implemented by trackpoint event streams. Any attempt to mix the use of the OnStep() method of directives with step extension will result in a run-time error.

## Using the Extended BAM API

The ESB Libraries provide a graduated approach to programmatic instrumentation of custom code and BizTalk orchestrations. Section 9.8 described the concept of step extension and related it to programmatic use of the OnStep() method of Directive objects and ExtendActivity() method of trackpoint event stream objects. This section provides a wider description of the extended BAM API provided by the ESB Libraries.

The ESB Libraries support three levels of API support for BAM instrumentation:

### Interception Points

At this level, the extended API supports Microsoft’s BAM Interception framework in conjunction with directives and additional trackpoint configuration provided through BAM interception policies. This centres on the use of the OnStep() method of Directive objects. Programmers can also call OnStep() on a Directives collection in order to perform step-wise processing for each directive in the collection.   
  
In this model, each call to OnStep() results in the performance of a discrete unit of work bounded by a begin action (‘begin activity’ or ‘continue activity’) and an end action. The actions for each step are provided as trackpoint configuration defined in a BAM interception policy. Each step is defined by a directive and identified using a simple name. Steps can be continued or related to other steps as required. However, this model is an ‘all-at-once’ approach in which all step actions are executed within a single method call.

### Trackpoint-Aware Event Observation

The ESB Libraries support extensions to Microsoft’s Event Observation API that utilise the same trackpoint configuration used by interception points. In this case, however, the ESB Libraries do not use Microsoft’s BAM Interception framework. When working at this level, developers instrument their code using specialised BAM event stream objects. These implement additional methods and overloaded versions of existing event stream methods. The following event stream methods are overloaded with trackpoint-aware implementations:

* BeginActivity()
* ContinueActivity()
* UpdateActivity()
* EndActivity()

In addition, the trackpoint event streams provided by the ESB Libraries include new methods that will perform all operation of a given type configured in trackpoints.

* AddReferences()
* AddRelatedActivities()
* EnableContinuations()
* UpdateActivityAll()

Step extension, described in section 9.8, applies at this level. Step extensions are a form of continuation managed by trackpoint event streams. Use the following method for step extension to a current activity.

* ExtendActivity()

### Standard Event Observation

The trackpoint event streams provided by the ESB Libraries support Microsoft’s standard event observation APIs and can be used as alternatives to the event stream classes provided by the Event Observation API. The support both buffered and direct stream processing, as specified by a Service Mediation directive.

## Trackpoint Event Streams

The ESB Libraries provide trackpoint-driven BAM event streams. The following diagram illustrates the class hierarchy and its association with the Microsoft BAM Event Observation API.

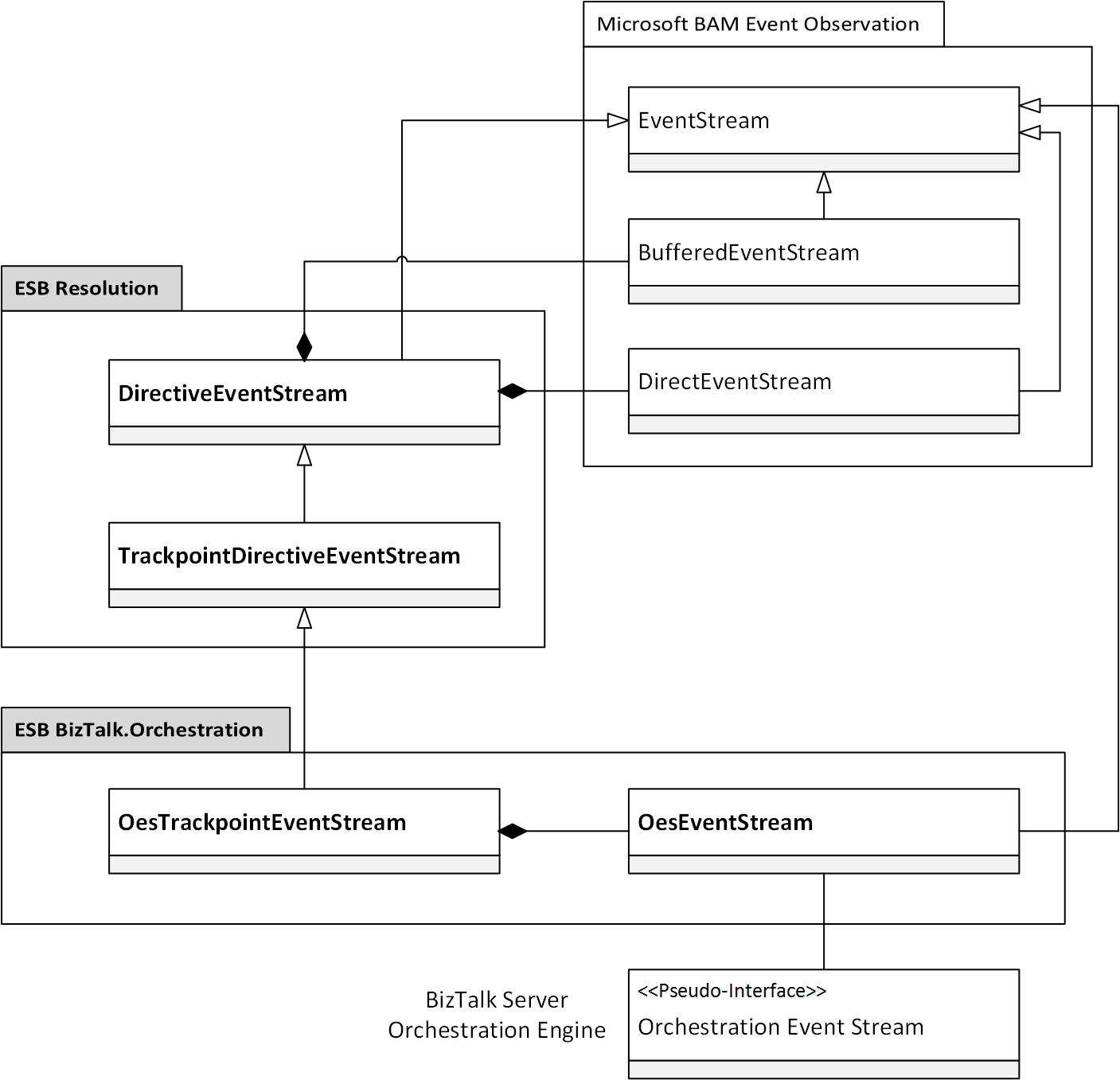


Figure 15: BAM Event Stream Class Hierarchy

**DirectiveEventStream**This class is derived directly from Microsoft’s EventStream class. It is constructed over a directive. The directive instructs the event stream to use either a buffered or direct inner event stream and provides SQL Server connection and BAM threshold settings. The following table lists the API members.

Table 26: DirectEventStream Members

|  |  |
| --- | --- |
| **Member** | **Description** |
| AddReference | Provides the current activity instance with a reference to additional data. This method is overloaded to support references to items containing up to 512 KB of Unicode characters. |
| AddRelatedActivity | Specifies a relationship between the current activity instance and another BAM activity instance. |
| BeginActivity | Starts a BAM activity. A new activity record will be created if data is tracked using the UpdateActivity method. |
| Clear | Clears the buffered data. |
| ContinueActivity | Continues a BAM activity under a new continuation token, as specified in the trackpoints for the current step. |
| Directive | Ends the BAM activity specified by the current directive. |
| EnableContinuation | Enables the continuation of the current BAM activity instance using a continuation token. Data tracked in a different context can contribute to the current activity record. |
| EndActivity | Ends the current BAM activity instance. Indicates that there are no more events expected for the given activity instance or continuation token. |
| Flush | Flushes the event stream. |
| InnerEventStream | Read-only property. Provides direct access to the inner event stream |
| StoreCustomEvent | Stores a custom serialized event. |
| UpdateActivity | Updates or inserts an activity record for a named milestone or data item. |
| UpdateDirective | Updates the current directive used by the event stream. |

**TrackpointDirectiveEventStream**This class is derived from DirectiveEventStream. It supports trackpoint policy in addition to other directive configuration. In order to exploit trackpoints, it provides an extended API that overloads some existing methods and adds additional members. The following table lists the extended API. The inherited members are not listed, except where overloaded.

Table 27: TrackpointDirectiveEventStream Members

|  |  |
| --- | --- |
| **Member** | **Description** |
| AddReferences | Provides the current activity instance with references to additional data, as specified in the trackpoints for the current step. |
| AddRelatedActivites | Specifies relationships between the current activity instance and other BAM activities, as specified in the trackpoints for the current step. |
| BeginActivity (overloads) | Starts a BAM activity instance using the activity ID specified in the trackpoints for the current step. |
| ContinueActivity (overloads) | Continues a BAM activity instance under a new continuation token, as specified in the trackpoints for the current step. |
| EnableContinuations | Enables continuations of the current BAM activity instance using the continuation tokens specified in the trackpoints for the current step. |
| EndActivity (overloads) | Ends the BAM activity instance specified by the current directive. |
| ExtendActivity | Extends the current activity with a step extension defined in a directive. This may be a named step, or an extension defined for an existing step. |
| UpdateActivity (overloads) | Updates or inserts a named milestone and data for the current activity instance, as specified in the trackpoints for the current step. |
| UpdateActivityAll | Updates or inserts all milestones and data items for the current activity instance, as specified in the trackpoints for the current step. |

The extended APIs provide method overloads to handle two concerns:

* Use of ‘after map’ BAM locations specified by directives in conjunction with transformation. BAM steps can be specified before and/or after a transformation. Use the afterMap parameter to select ‘after map’ steps where defined.
* Activity instance identifiers. A running process may handle multiple instances of the same activity simultaneously (e.g., in different branches of a parallel shape or as a result of looping code). In this case, you can easily distinguish between different instances of the activity by providing a local instance token or identifier. This is not the same as the activity ID, although the activity ID may be used, if known.

### BAM Step Data

The BAM interception framework supports extraction of data items from data provided by applications. For example, an extractor component may support the use of XPaths against XML data. The ESB libraries provided an extraction component that supports the following data sources:

* XML documents, using XPaths
* Message properties, using the {property} macro
* Value list (format string arguments)

In addition the extraction component supports a number of macros, mainly concerned with time and dates.

To pass data to the TrackpointDirectiveEventStream, developers can populate an instance of the BamStepData class and pass it to the constructor along with a directive. The BamStepDataClass provides the following members:

Table 28: BamStepData Members

|  |  |
| --- | --- |
| **Member** | **Description** |
| Properties | A set of name-value pairs. These may represent message properties. |
| ValueList | A list of values. These are passed, in the order they appear in the collection, as arguments to format strings. |
| XmlDocument | An XML document from which data will be extracted using XPaths. |

### Orchestration Support for BAM

The ESB Libraries implement support for BAM instrumentation in the context of BizTalk orchestrations. The SolidsoftReply.Esb.Libraries.BizTalk.Orchestration component provides two event stream classes that allow developers to access the capabilities of the Orchestration Event Stream (OES) implemented in BizTalk Server in conjunction with directives and track points. In addition, it provides a derived version of the BamStepData class with orchestration-friendly features.

Please note that in the current version of the ESB Libraries, it is necessary to include a reference to the SolidsoftReply.Esb.Libraries.Resolution library as well as the BizTalk.Orchestration library. This requirement may be removed in a later version.

**OesEventStream**

Microsoft’s Orchestration Event Stream (OES) class provides a set of static methods designed to be used in the context of a BizTalk orchestration. The OES is not a ‘true’ BAM event stream, in that it does not derive from the EventStream class. However, it broadly follow the same pattern as the BAM event stream API. The OES synchronises BAM operations with the BizTalk Server Orchestration Engine, ensuring that events are flushed to the message box automatically at suitable persistence points.

The OesEventStream class, provided by the SolidsoftReply.Esb.Libraries.BizTalk.Orchestration library, is a ‘true’ BAM event stream which acts as a wrapper around the OES. It can be used directly by developers if they wish. However, its main role is to provide an orchestration-friendly alternative to the buffered and direct event streams when used as an ‘inner’ event stream by the OesTrackpointEventStream.

**OesTrackpointEventStream**

The OesTrackpointEventStream is an orchestration-friendly specialisation of the TrackpointDirectiveEventStream, and should generally be used in BizTalk orchestrations. Developers can, of course, elect to use the TrackpointDirectiveEventStream. However, this class is not designed to take advantage of the OES. The OesTrackpointEventStream provides identical functionality to the TrackpointDirectiveEventStream, but co-ordinates its operations with the BizTalk Server Orchestration Engine via an inner instance of the OesEventStream class. In effect, therefore, it is always buffered, regardless of configuration settings in the directive.

**BamStepData**

The SolidsoftReply.Esb.Libraries.BizTalk.Orchestration library implements a derived version of the BamStepData class defined in the SolidsoftReply.Esb.Libraries.Resolution library. See section 9.10.1 for further information about the base class. The derived class provides two additional features:

* **Support for BizTalk Orchestration messages**Developers can assign a BizTalk Server message to an instance of the BamStepData class. In this case, the XML content, if any, of the body part or first message part is used as the XML document. Each message property is added to the Properties collection.
* **Orchestration-friendly helper methods for the Value list**The BamStepData class provides additional Orchestration-friendly methods for writing to and reading from the Value list, as well as a count property for the Value list.

The members of the BamStepData class are listed below:

Table 29: Orchestration BamStepData Members

|  |  |
| --- | --- |
| **Member** | **Description** |
| Properties | A set of name-value pairs. These may represent message properties. |
| ValueList | A list of values. These are passed, in the order they appear in the collection, as arguments to format strings. |
| XmlDocument | An XML document from which data will be extracted using XPaths. |
| BizTalkMessage | A BizTalk Server orchestration message which will be used to assign the XmlDocument and a list of properties. |
| ValueListCount | The number of values in the Value list. |
| ValueListRead | Reads a value, by position, from the Value List |
| ValueLstWrite | Writes a value to the Value list, either appending it to the end of the list or inserting it as a given position. |

### Examples of Orchestration BAM Code

The first task when instrumenting BizTalk Server orchestration code for BAM is to obtain a directive for a BAM step. In addition, the code should generate or obtain a unique identifier for the current instance of the BAM activity:

//Initialize BAM activity id

bamActivityId = System.Guid.NewGuid();

// Specify facts against which to resolve directives

factsForCanonicalPO = new SolidsoftReply.Esb.Libraries.BizTalk.Orchestration.Facts();

factsForCanonicalPO.ServiceName = "PurchaseOrderProcess";

factsForCanonicalPO.OperationName = "Track Canonical PO";

factsForCanonicalPO.MessageType = msgCanonicalPO(BTS.MessageType);

// Get directives for canonical message from latest deployed version

// of the Service Mediation policy

directivesForCanonicalPO = SolidsoftReply.Esb.Libraries.BizTalk.Orchestration.Resolver.Resolve(  
 factsForCanonicalPO,  
 "PurchaseOrderProcess Policy");

// Get the BAM directive. This defines the root fragment step for the BAM activity

// and any additional step extensions

directiveTrackCanonicalPO = directivesForCanonicalPO.GetDirective("Track Canonical PO");

Now we have the BAM directive, we need to create any data required for BAM tracking

// Specify data for BAM to use with the 'Track Canonical PO' directive at the first step.

// Assign the BizTalk message to obtain the XML document and message properties.

bamTrackCanonicalPOStepData  
 = new SolidsoftReply.Esb.Libraries.BizTalk.Orchestration.BamStepData();

bamTrackCanonicalPOStepData.BizTalkMessage = msgCanonicalPO;

To exploit the BAM interception model, execute the step in a single method call. This approach cannot be used with step extension later in the orchestration:

// Track the target system activity at this step

directiveTrackCanonicalPO.OnStep(bamTrackCanonicalPOStepData);

Alternatively, to use the Trackpoint-aware event observation model, first initialise a BAM event stream for OES:

// Initialize a BAM event stream for OES using the directive and step data

bamTrackCanonicalPOEventStream  
 = new SolidsoftReply.Esb.Libraries.BizTalk.Orchestration.OesTrackpointEventStream(  
 directiveTrackCanonicalPO, bamTrackCanonicalPOStepData);

The OES Trackpoint event stream provides an extended API and support for step extension. The following code adds the activity ID generated earlier to the BamStepData as an additional property, begins the root fragment of the activity and then executes all the updates operations specified by the BAM interception policy specified by the directive.

// Add the activity ID value as a property to the BAM step data

bamTrackCanonicalPOStepData.Properties.Add("ActivityId", bamActivityId);

// Begin the Purchase Order Process activity

bamTrackCanonicalPOEventStream.BeginActivity();

// Update the activity at the root fragment step

bamTrackCanonicalPOEventStream.UpdateActivityAll();

To extend the current step, use the ExtendActivity() method. Further data can be added to the BamStepData to support extension. The current step can be extended by a step defined in another directive:

directiveTrackCanonicalPOHeader   
 = directivesForCanonicalPO.GetDirective("Track Canonical PO - Header");

// Add additional data properties for the extended step

bamTrackCanonicalPOStepData.Properties.Add("FileType", msgCanonicalHeader.Body.FileType);

bamTrackCanonicalPOStepData.Properties.Add("MessageId", msgCanonicalHeader.Body.MessageID);

bamTrackCanonicalPOStepData.Properties.Add("UniqueId", msgCanonicalHeader.Body.UniqueId);

bamTrackCanonicalPOStepData.Properties.Add(

"ReceiveLocationName",

msgCanonicalPO(Common.Schemas.ReceiveLocationName));

// Extend the current activity to track the canonical PO at the 'Header' step.

bamTrackCanonicalPOEventStream.ExtendActivity(directiveTrackCanonicalPOHeader);

Alternatively, if the current BAM directive specifies step extensions, the current step can be extended using step extension names:

// Extend the current activity to track the canonical PO at the 'Header' step.

bamTrackCanonicalPOEventStream.ExtendActivity("Track Canonical XML - Header");

Once extended, use the event stream API to perform Trackpoint-aware operations for the step extension :

// Perform all updates at the 'Header' step

bamTrackCanonicalPOEventStream.UpdateActivityAll();

You can extend the activity as many times as you wish. Once all operations are complete for this instance of the activity, explicitly end it:

// End the "Track Canonical PO" activity

bamTrackCanonicalPOEventStream.EndActivity();

# Defining Validation Policy

Section 6.1.6 describes how service mediation policies can be configured to specify validation policies. When these directive instructions are applied within a Service Mediation pipeline component, the pipeline component will execute the specified validation policy in the expectation that it will return a list of validation records in accordance with the logging actions specified in this section. The validation records specify errors, warnings and other information created by the execution of the validation policy.

For each category of record (errors, warnings and information), the Service Mediation pipeline components creates two message properties. The first is a promoted property containing the count of error records in that category. The second is a non-promoted (non-routable) property containing a delimited list of textual messages. Each message represents an individual record. The ‘|’ symbol is used to delimit each message.

The six properties are defined in the following table. Note that, for any one category, these properties are only created in the count of records in that category is greater than 0. All properties are in the <http://solidsoftreply.com/esb/2015/validation-properties> namespace.

Table 30: Validation Message Properties

|  |  |  |
| --- | --- | --- |
| Category | Property | Description |
| Error | ErrorsCount | Promoted (routable). The number of error records. |
| Errors | The ‘|’ delimited list of error records. |
| Warning | WarningsCount | Promoted (routable). The number of warning records. |
| Warnings | The ‘|’ delimited list of warning records. |
| Information | InformationCount | Promoted (routable). The number of information records. |
| Information | The ‘|’ delimited list of information records. |

To configure Validation rules, use the following instructions in your rule actions. These are defined in the ‘ESB Validation Actions’ vocabulary:

Table 31: Vocabulary Definitions for Validation Policy Instructions

| Definition | Description |
| --- | --- |
| Log a validation error - Description only | Logs a validation error with a description.  **Parameters:**   |  |  | | --- | --- | | **Description** | The entry description. | |
| Log a validation error - Description and severity | Logs a validation error with a description and severity.  **Parameters:**   |  |  | | --- | --- | | **Severity**  May be an integer or null. | The severity of the entry. | | **Description** | The entry description. | |
| Log a validation error - Description and code | Logs a validation error with a description and code.  **Parameters:**   |  |  | | --- | --- | | **Code**  May be a long integer or null. | A code representing the entry. | | **Description** | The entry description. | |
| Log a validation error - Description, code and severity | Logs a validation error with a description, code and severity.  **Parameters:**   |  |  | | --- | --- | | **Severity**  May be an integer or null. | The severity of the entry. | | **Code**  May be a long integer or null. | A code representing the entry. | | **Description** | The entry description. | |
| Log a validation error - Description and type | Logs a validation error with a description and type.  **Parameters:**   |  |  | | --- | --- | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation error - Description, type and severity | Logs a validation error with a description, type and severity.  **Parameters:**   |  |  | | --- | --- | | **Severity**  May be an integer or null. | The severity of the entry. | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation error - Description, code and type | Logs a validation error with a description, code and type.  **Parameters:**   |  |  | | --- | --- | | **Code**  May be a long integer or null. | A code representing the entry. | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation error - Description, code, type and severity | Logs an entry.  **Parameters:**   |  |  | | --- | --- | | **Severity**  May be an integer or null. | The severity of the entry. | | **Code**  May be a long integer or null. | A code representing the entry. | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation warning - Description only | Logs a validation warning with a description.  **Parameters:**   |  |  | | --- | --- | | **Description** | The entry description. | |
| Log a validation warning - Description and code | Logs a validation warning with a description and code.  **Parameters:**   |  |  | | --- | --- | | **Code**  May be a long integer or null. | A code representing the entry. | | **Description** | The entry description. | |
| Log a validation warning - Description and type | Logs a validation warning with a description and type.  **Parameters:**   |  |  | | --- | --- | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation warning - Description, code and type | Logs a validation warning with a description, code and type.  **Parameters:**   |  |  | | --- | --- | | **Code**  May be a long integer or null. | A code representing the entry. | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation information - Description only | Logs a validation information with a description.  **Parameters:**   |  |  | | --- | --- | | **Description** | The entry description. | |
| Log a validation information - Description and code | Logs a validation information with a description and code.  **Parameters:**   |  |  | | --- | --- | | **Code**  May be a long integer or null. | A code representing the entry. | | **Description** | The entry description. | |
| Log a validation information - Description and type | Logs a validation information with a description and type.  **Parameters:**   |  |  | | --- | --- | | **Type** | The type of entry. | | **Description** | The entry description. | |
| Log a validation information - Description, code and type | Logs a validation information with a description, code and type.  **Parameters:**   |  |  | | --- | --- | | **Code**  May be a long integer or null. | A code representing the entry. | | **Type** | The type of entry. | | **Description** | The entry description. | |

# UDDI Support

Microsoft’s ESB Toolkit supports endpoint resolution using either the BizTalk Rules Engine or UDDI. The Solidsoft Reply ESB Libraries always perform resolution via the BizTalk Rules Engine, but allow UDDI to be used in conjunction with rules. This is supported through the SolidsoftReply.Esb.Libraries.Uddi helper library. This library supports any UDDI 3-complaint registry, but is most commonly used in conjunction with Microsoft UDDI 3 directory service included with BizTalk Server. The library an also be used in scenarios where searches must span multiple UDDI registries maintained across the enterprise. This scenario is enabled through the use of registration of each UDDI registry in a Windows Active Directory. For more information on multi-directory registration, see ‘[Publish a Site in Active Directory domain Services](https://msdn.microsoft.com/en-us/library/cc731799.aspx)’.

The UDDI helper library, and the supporting ‘ESB UDI’ vocabulary, provides support for resolving endpoints and other access point types for given UDDI binding keys, service keys and names, and provider keys and names. It also provides support for amending the ‘authority’ component of access point URIs returned by look-ups within UDDI registries. To find an access point, nest a ‘Find \*’ vocabulary definition within a ‘Send to an endpoint \*’ definition in the ‘ESB Resolution Actions’ vocabulary. For example, a rule action might read as follows:

[ Directive PurchaseOrdering ] Send the message to a UDDI-registered access point for service named POProcessing

To specify provider or service names or keys within ‘Find \*’ definitions, use the ‘A named provider’, ‘A provider key’, ‘A named service’ or ‘A service key’ definitions.

**Note:**

The current version of the UDDI helper library supports case-sensitive searches, only.

The following table lists the vocabulary definitions defined in the ‘ESB UDDI’ vocabulary:

Table 32: Vocabulary Definitions for UDDI Helpers

| Definition | Description |
| --- | --- |
| Find an access point type for a binding key | Uses UDDI to find an access point for a given binding key.  **Parameters:**   |  |  | | --- | --- | | **Binding Key**  Cannot be null or empty.  Must be a valid URI no more than 255 characters in length. | A registered binding key.  Case-sensitive string. | |
| Find an access point for a provider and a service | Uses UDDI to find an access point (an endpoint) for a given provider and service.  **Parameters:**   |  |  | | --- | --- | | **Provider**  Use the ‘A named provider’ vocabulary definition. | The provider key or name. | | **Service**  Use the ‘A named service’ vocabulary definition. | The service key or name. | |
| Find an access point type for a provider and a service | Find an access point type for a provider and a service.  **Parameters:**   |  |  | | --- | --- | | **Use Type** | The use type of the access point.  Can be:   * Unspecified * EndPoint * BindingTemplate * HostingRedirector * WsdlDeployment | | **Provider**  Use the ‘A named Provider’ vocabulary definition. | The provider key or name.  Case-sensitive string. | | **Service**  Use the ‘A named service’ vocabulary definition. | The service key or name. | |
| Find an access point for a service | Uses UDDI to find an access point for a given service.  **Parameters:**   |  |  | | --- | --- | | **Service**  Use the ‘A named service’ vocabulary definition. | The service key or name. | |
| Find an access point type for a service | Uses UDDI to find a specific access point type for a given service.  **Parameters:**   |  |  | | --- | --- | | **Use Type** | The use type of the access point.  Can be:   * Unspecified * EndPoint * BindingTemplate * HostingRedirector * WsdlDeployment | | **Service**  Use the ‘A named service’ vocabulary definition. | The service key or name. | |
| Find an access point and amend the URL authority | Sets the URL authority for an access point.  **Parameters:**   |  |  | | --- | --- | | **Access Point**  Cannot be null or empty.  Use a ‘Find \*’ vocabulary definition, as defined above. | The access point. | | **Base URL**  Cannot be null or empty | The base URL. The ‘authority’ contained in this URL will be used to amend the access point URI. | |
| A provider key | A provider identified by a key value.  **Parameters:**   |  |  | | --- | --- | | **Key**  Cannot be null or empty.  Must be a valid URI no more than 255 characters in length. | The key value.  Case-sensitive string | |
| A service key | A service identified by a key value.  **Parameters:**   |  |  | | --- | --- | | **Key**  Cannot be null or empty.  Must be a valid URI no more than 255 characters in length. | The key value.  Case-sensitive string | |
| A named provider | A provider identified by name.  **Parameters:**   |  |  | | --- | --- | | **Name**  Cannot be null or empty.  Must be between 1 and 255 characters in length. | The provider name.  Case-sensitive string | |
| A named service | A service identified by name.  **Parameters:**   |  |  | | --- | --- | | **Name**  Cannot be null or empty.  Must be between 1 and 255 characters in length. | The service name.  Case-sensitive string. | |

# Pipeline Components

The ESB Libraries implement six BizTalk pipeline components. These include the Service Mediation component which implements functionality that applies directives to individual BizTalk messages. It also includes a library of five dissembler components based on corresponding components provided by Microsoft as part of the BizTalk Server product.

* XML Disassembler
* Flat File Disassembler
* BizTalk Framework (BTF) Disassembler
* EDI Disassembler
* AS2 Disassembler

Each of these ESB disassemblers provides identical functionality to their Microsoft counterparts, and uses the same component code. However, they are extended to support the functionality included in the Service Mediation pipeline component.

The Service Mediation component can be used as a normal pipeline component or as a Disassembler component. Each of the other Disassembler components uses the Service Mediation functionality is ‘Disassembler’ mode. It is important to understand the behaviour of the Service Mediation component in these different modes.

**Normal Mode**

In ‘Normal’ mode, the Service Mediation component applies the instructions is each directive to the single message. Consider a scenario where the ESB Libraries resolve a given message in the pipeline to a collection of three directives. In this case, each directive is processed in turn, applying the instructions within the directive to the single message. The order in which the directives are processed is arbitrary by default, and care must therefore be taken to avoid race conditions and inconsistent approaches. For example, if two of the directives contain transformation instructions, these are likely to clash. The order in which the transformations are applied is arbitrary by default, so there may be no guarantee that the second map will accept the output of the first map as its input.

If multiple directives must be used, you can control the order in which the directives are processed by specifying appropriate priorities for rules within the service mediation policy. Rules with a higher priority value will result in directives that are placed earlier in the order in which they are applied. Hence, if Rule A defines a directive called ‘Directive 1’ and Rule B defines a directive called ‘Directive 2’, and if Rule B is given a higher priority that Rule A, then ‘Directive 2’ will be applied before ‘Directive 1’. Multiple rules can contribute to a single directive. In this case, the directive order is determined by the highest priority rule that first contributes to a given directive.

If a rule contributes to multiple directives, and if that rule is the highest priority rule that first contributes to those directives, the order in which the directives will be applied corresponds to the order in which rule actions first specify each directive.

**Disassembly Mode**

In ‘Disassembly’ mode, the behaviour of the Service Mediation component changes. In this case, the pipeline component creates a separate copy of the message for each individual directive in the collection of directives that it applies. Each message is returned to the pipeline in turn as it calls the GetNext() method of the component. In this case, if the rules define one directive with a transformation instruction and a second directive with a different transformation instruction using a different map, each message that the component receives, and which the rule engine matches to the two directives, will result in two outbound messages, each transformed using a different map.

In a similar way, if the pipeline component applies three different directives to a message, and each directive specifies a different endpoint or set of routable properties, the component will create three copies of the same message, each with a different set of promoted properties. By itself, this does not provide any significant advantage over in0built routing capabilities of the product, except for the advantage of implementing a dynamic approach that can be changed easily by amending a rule set. However, when routing is combined with other capabilities such as transformation and BAM, this can provide a powerful mechanism for implementing advanced patterns, reducing the number of message box hops and exploiting dynamic Send ports.

The additional five Disassembler pipeline components listed earlier all exploit the Disassembly mode of the Service Mediation component. In effect, each of these components wraps an instance of a Microsoft disassembler with an instance of the Service Mediation component. Each message provided by Microsoft’s disassembler is handed off to the Service mediation component which works in Disassemble mode.

Consider a scenario which the Flat File Disassembler provided by the ESB Libraries is used to process a flat file containing 100 records. Internally, the flat file is first passed to an instance of Microsoft’s Flat File Disassembler. This ensures complete fidelity with the behaviour of Microsoft’s component. Each ESB Disassembler supports identical configuration to its Microsoft-provided counterpart.

In this scenario, the instance of Microsoft’s Disassembler returns 100 separate messages. Each of these messages is handed off internally to an instance of the Service Mediation component. If that component applies three directives to each message, then the ESB Flat File Disassembler will return a total of three hundred messages to the pipeline.

**Note:**

The ESB Flat File Disassembler uses an internal instance of Microsoft’s Flat File Disassembler component. This internal Disassembler creates each outbound message and passes it on the internal instance of the Service Mediation component. However, Microsoft’s Flat File Disassembler implements a complex arrangement in which the disassembled message is   
linked back to the underlying stream containing the entire flat file message and to the XSD document specification for the disassembled XML. Unfortunately, the XML Reader for this   
XSD can be prematurely disposed. This can happen if the disassembled message is   
processed by a downstream component that replaces the message with a new message.   
The Service Mediation component, which is itself a disassembler and must therefore create   
clones of the original message, replaces messages in this way.

This problem arises as a result of garbage collection at run-time. If a garbage collection   
occurs at specific points in the processing, the XML Reader is disposed as part of internal   
state transition handling. If this happens before Microsoft's Flat File Disassembler has fully disassembled the original flat file, a run-time error occurs. The issue is related to garbage collection, and is therefore effectively non-deterministic. Different behaviour may be   
exhibited on different machines or at different times under different loads.

The problem has been noted several times by BizTalk Server users over the years, but has yet to be addressed by Microsoft. The only documented workaround is to switch on recoverable interchange handling on the Microsoft Flat File disassembler. This works because the   
problem is only triggered when recoverable interchange handling is switched off. Switching it on avoids the line of code that triggers the state transition that results in the disposal of the XML Reader. However, this workaround is unacceptable for the ESB Flat File Disassembler because it would force the use of recoverable interchanges in all cases.

The only way in which the ESB Flat File Disassembler can work around this issue is to first   
fully disassemble the original flat file content in memory and only then process each record   
in turn through the internal instance of the Service Mediation component. This avoids   
garbage collection-related issues but is memory-intensive and may not scale adequately   
under heavy load. Unfortunately, using Microsoft's Flat File Disassembler with a separate downstream instance of the Service Mediation component will not solve the issue. This may still result in run-time exceptions, and has the added problem of not supporting Disassembly mode in the Service Mediation component.

A second issue with Microsoft’s Flat File disassembler concerns problems that arise when processing header records. Although this is a different bug, it is similar to the first, and is   
again related to garbage collection. The work around adopted by the ESB Flat File Disassembler for the first problem also avoids this second problem.

1. A future version of the ESB libraries may include WCF behaviours for enforcement of service mediation policies. [↑](#footnote-ref-1)
2. In development environments, the cache is typically left unconfigured which effectively switches caching off. [↑](#footnote-ref-2)
3. A future version may introduce a more sophisticated fabric to allow near real-time policy changes without the need to recycle services. [↑](#footnote-ref-3)
4. The BAM Interceptor refers to these units as ‘trace fragments’. It is designed to allow a single instance of the interceptor to handle multiple steps. In this case, the BAM Interceptor represents a ‘trace instance’ composed of multiple fragments. Unfortunately, a logical error in the BAM Interceptor (at the time of writing, this includes all versions from 2004 to 2013 R2) causes this model to break down when one or more of the steps represents a continuation. See <http://geekswithblogs.net/cyoung/archive/2014/06/02/using-the-bam-interceptor-with-continuation.aspx>. The ESB Libraries contain a work-around for this fault and will function correctly. [↑](#footnote-ref-4)