Solidsoft Enterprise Service Bus

Policy-Driven Service Mediation

Component Design

Charles Young

June 2015

# Table of Contents

[Table of Contents 2](#_Toc421724258)

[1. Introduction 3](#_Toc421724259)

[2. The ESB Libraries 5](#_Toc421724260)

[2.1 Policy-Driven Service Mediation 5](#_Toc421724261)

[2.2 Dynamic Models in BizTalk Server 5](#_Toc421724262)

[2.3 The Wider View 5](#_Toc421724263)

[2.4 The Anatomy and Role of a Policy 6](#_Toc421724264)

[2.5 Caching 8](#_Toc421724265)

[3. Installation & Configuration 9](#_Toc421724266)

[3.1 Resolution Web Service 10](#_Toc421724267)

[3.2 Other Assemblies 11](#_Toc421724268)

[3.3 Rule Engine Vocabularies 11](#_Toc421724269)

[3.4 Property Schema 11](#_Toc421724270)

[3.5 Installation MSI 12](#_Toc421724271)

[4. Configurable App Settings 13](#_Toc421724272)

[5. Facts 15](#_Toc421724273)

[6. Directives and Instructions 16](#_Toc421724274)

[6.1.1 Endpoint Resolution 18](#_Toc421724275)

[6.1.2 Transformation 19](#_Toc421724276)

[6.1.3 BAM Steps 20](#_Toc421724277)

[6.1.4 Retries 22](#_Toc421724278)

[6.1.5 Service Windows 23](#_Toc421724279)

[6.1.6 Validation 23](#_Toc421724280)

[6.2 Policy Properties 24](#_Toc421724281)

[7. Implementation Design 25](#_Toc421724282)

[8. Property-Handing for BizTalk Server 31](#_Toc421724283)

[8.1 Creating new properties 33](#_Toc421724284)

[8.2 Routing policy 33](#_Toc421724285)

[8.3 Retry policy 33](#_Toc421724286)

[8.4 Validation policy 34](#_Toc421724287)

[9. Defining BAM Policy 34](#_Toc421724288)

[9.1 BAM Steps and Trackpoints 35](#_Toc421724289)

[9.2 Extraction Strings 36](#_Toc421724290)

[9.3 Extended BAM API 38](#_Toc421724291)

[9.3.1 BAM Step Data 41](#_Toc421724292)

[9.3.2 Orchestration Support for BAM 41](#_Toc421724293)

# Table of Figures

[Figure 1: Assert-Infer-Enforce Cycle for Service Mediation 8](#_Toc421724747)

[Figure 2: Component Architecture 26](#_Toc421724748)

[Figure 3: Resolution Data Properties 33](#_Toc421724749)

# 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

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

Table 1: Artefacts

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| MSI | SolidsoftEsbPipelines.msi | Installs optional pre-defined ESB pipelines. |
| 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 1: 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 2: 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 3: Service Mediation Directive Instruction Categories.

Table 3: 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 4: 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. |

### Transformation

Specifies transformation requirements in terms of BizTalk Server maps.

The following directive instructions belong to this category:

Table 5: 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. |

### 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 6: 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.

### 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 7: 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.

### 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 8: 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 |

### 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 9: 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. |

## 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.

# Implementation Design

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

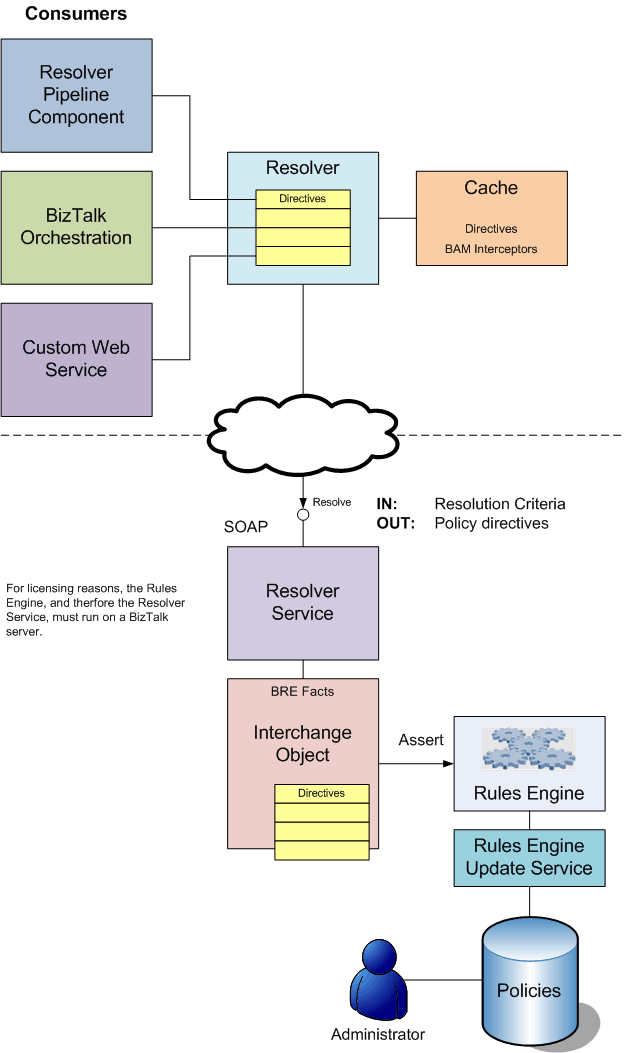


Figure 2: Component Architecture

Table 10: 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 three pipeline components. The ESB Governance component supports endpoint resolution and single-level retry policies via promoted message properties. It also supports dynamic transformation, BAM interception and service windows. XML and Flat File disassemblers are also provided which combine existing Microsoft-provided disassembly functionality with the same features provided by the ESB Governance 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 11: Resolution Data Property Values

|  |  |
| --- | --- |
| Value | Description |
| ValuesOnly | The pipeline component only passes resolution values. See Table 2: 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.

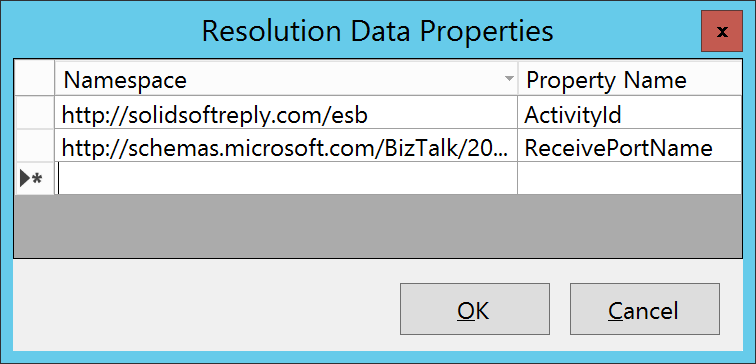


Figure 3: Resolution Data Properties

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. The ESB XML, Flat File, EDI, A2S and BTF disassemblers call the corresponding Microsoft-supplied disassemblers first before passing each disassembled message to the ESB 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 12: 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 13: 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 14: 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 definition of BAM policy has two distinct aspects, and involves two separate policies. The Service Bus policy for BAM steps, described earlier, defines activities and steps. It is used to introduce named locations into specific interchanges. These locations are called ‘steps’. A directive can define up to two steps. The following directive configuration styles are valid:

Table 15: BAM Interception Policy Configuration Styles

|  |  |
| --- | --- |
| Configuration Style | Description |
| No steps | The interchange contains no BAM interception locations. |
| Single step; no transformation | The interchange introduces a single BAM interception location which operates on the message. |
| Single step with transformation | The interchange introduces a single BAM interception location which operates on the message before it is transformed |
| Single post-transformation step with transformation | The interchange introduces a single BAM interception location which operates on the results of the transformation. |
| Two steps with transformation | The interchange introduces two BAM interception locations. The first operates on the message before it is transformed. The second operates on the results of the transformation. |

**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 is ignored. Other configurations will result in a run-time exception.

All steps in any one directive must be assigned to the same named policy. If an interchange represents more than one activity, there may be a need to introduce multiple directives to handle BAM requirements.

## BAM Steps and Trackpoints

Service Bus policy directives specify the inclusion of named BAM locations. These locations are called ‘steps’. The Resolver component supports the use of BAM Interception with these steps. BAM Interception is used in the pipeline components provided as part of the ESB Libraries and via the TransformWithInterception() and OnStep() methods of the Resolver Directive class. Alternatively, steps can be used in conjunction with BAM event streams. The Resolver component provides specialised event streams that are configured by steps.

A step 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, they do not directly specify the tracking that should be done at each step. Additional policies can be defined for this purpose. These can be specified as part of a directive. A default trackpoint policy can be configured in a configuration file.

Trackpoint policies define BAM trackpoints for specific business activities and steps. Activities and steps are identified by name. Any one activity may include multiple message exchanges or may be related to one part of a message interchange. A BAM Trackpoint policy defines a set of track points 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 trackpoint 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.

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. Enclose macros 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 Sever orchestration, or other locations that use the API directly, use the overloaded OnStep method of the Resolver’s ResoverDirective class to assign values for each parameter.

* **Macros**Macros are automatically expanded 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 cases sensitive. However, the format specifier is.

Table 16: 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.

## Extended BAM API

We saw earlier that the BAM Interceptor expects steps to represent entire units of work bounded by a begin action (‘start activity’ or ‘continue activity’) and an end action. This is true when using the OnStep() or TransformWithInterception() methods on a Directive. When writing BAM event observation code as part of an orchestration or in some other process flow, the BAM Interceptor limits the flexibility of BAM. For example, developers often want to instrument an orchestration to collect different information at different points. The orchestration includes multiple events which need to be observed as part of an activity. When using OnStep() or TransformWithInterception(), the developer is forced to break the work down into units and use continuation within the single orchestration to link each step.

In order to maximise the flexibility of the framework, the ESB Libraries support a more fine-grained approach that exploits Microsoft’s event observation API directly without using the BAM Interceptor. It provides two event stream classes for this purpose:

**DirectiveEventStream**This class is derived from 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 provide SQL Server connection and BAM threshold settings. The following table lists the API 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.

|  |  |
| --- | --- |
| **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. |
| UpdateActivity (overloads) | Updates or inserts a named milestone and data for the current activity instance, as specified in the trackpoints for the current step. |
| UpdateActivities | Updates or inserts 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:

|  |  |
| --- | --- |
| **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 using BAM in the context of BizTalk orchestrations. The Solidsoft.Esb.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.

OesEventStream

Microsoft’s OES comprises a set of static methods that follow the same pattern as the BAM event stream API. They allow BAM tracking to be synchronised with orchestration persistence points

OesTrackpointEventStream

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) 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)