BizTalk BAM Typed API Generator & BAM XML Exporter Usage Guide

Version 2.3

Updated July 8, 2020

Hosted on GitHub at https://github.com/tfabraham/BizTalkTypedBAMAPIGenerator

1. Introduction

Since you're reading this document, you must already know that Microsoft's BizTalk Server includes a framework for collecting and analyzing data about business processes called Business Activity Monitoring (BAM). BAM models (activity and view definitions) are created and managed using Microsoft Excel and saved as a standard Excel workbook (xls/xlsx).

Getting a BAM model deployed and feeding it data at runtime can be somewhat tricky and error-prone, and those are precisely the problems that the BizTalk BAM Typed API Generator and BAM XML Exporter tools can solve.

2. Software Requirements

Both of the tools are command-line based and have minimal software requirements. They require the Microsoft .NET Framework 4.5 or greater.

Working with Excel 2003 files requires the Microsoft JET 4.0 OLE DB driver, which is typically present on machines that have the .NET Framework 2.0 or newer installed.

Working with Excel 2007 or newer files requires the Microsoft Office 2007 Data Connectivity Components or Access Database Engine 2010 Redistributable, both of which can be obtained from Microsoft.com's Download Center. If you're working on a machine that has Excel 2007 or 2010 installed, the components are already present (usually the case for a development machine).

It is not currently possible to install both the 32-bit and 64-bit versions of certain data access components, including the Access Database Engine 2010 Redistributable. As a result, if you have the 64-bit data access components installed (or 64-bit Office), then our BAM tools must also run as 64-bit, and likewise for 32-bit. This distribution contains binaries for both modes.

3. Developing with BAM Models

Once you've defined a BAM model, the next step is obviously to populate it with meaningful data at runtime. Out of the box, BizTalk provides several "EventStream" classes that serve as BAM's data gathering API: DirectEventStream, BufferedEventStream and OrchestrationEventStream. Getting data into BAM typically requires writing code in your orchestrations, custom pipeline components and/or other applications that are part of the business process being monitored.

The problem with directly consuming these classes is that they are *un-typed* and *generic*.

Looking first at data typing, the BAM model itself *is* typed. When you define an activity item, you must specify whether it is a string, date/time, etc. If you write code to assign a string value to a BAM activity item that is defined as an integer, the compiler cannot help you find the error prior to runtime. It would be helpful to work with an API specific to your particular BAM model that is strongly typed, thus allowing the compiler to help you find any errors, particularly as your BAM model changes over time.

The generic nature of the stream classes is the second issue. Your BAM model defines a specific set of activities and items within those activities, and with the EventStream classes you must maintain knowledge of that structure in your code. Further, as the BAM model evolves, you must manually keep your code in sync. Instead, it would be very helpful to have an API that is automatically synchronized with the BAM model, which would not only produce more developer-friendly and readable code, but also allow the compiler to help find problems as the BAM model changes over time.

3.1. The BAM Typed API Generator

The issues discussed above are both solved using the BAM Typed API Generator utility. The utility performs two primary actions:

1. Opens the BAM workbook (XLS/XLSX) and extracts the BAM XML manifest
2. Generates a C# code file containing a set of classes that define a strongly-typed API that mirrors the actual BAM model (as defined by the XML manifest)

The usage is straightforward:

```
 GenerateTypedBAMAPI.exe <ExcelFile> <CodeFile>
<Direct|Buffered|Orchestration> <.NET Namespace> [XSLTFile]
```

`<ExcelFile>` is the path to your BAM workbook, either XLS or XLSX format. The workbook must contain at least one BAM activity definition. If the path contains spaces, it must be wrapped in double-quotes.

`<CodeFile>` is the file path and name where the generated C# code will be saved. If the file already exists, it will be overwritten if it is not marked as read-only. If the path contains spaces, it must be wrapped in double-quotes.

`<Direct|Buffered|Orchestration>` indicates which EventStream class should be used to persist BAM data (DirectEventStream, BufferedEventStream or OrchestrationEventStream). This parameter must contain *only one* of the listed options. Please see the BizTalk documentation for detailed information on when to use the various EventStream classes.

`<.NET Namespace>` indicates the name of the .NET namespace that should wrap the generated C# code.

`[XSLTFile]` is an **optional** parameter that allows a complete override of the built-in code generation. Internally, the utility contains an XSLT file that takes in the BAM XML manifest and converts it to C# code. To override the built-in code generation, provide a path to a custom XSLT file. As a starting point, the built-in XSLT file may be obtained via the source code download package and modified for your particular needs.

Here's a proven method to integrate the utility into your BizTalk solution (in this example, it is assumed that your solution is maintained in a source control system):

1. Create or locate a C# project in your BizTalk solution where you keep C# helper classes (strong-named and deployed to the GAC)
2. Add to the project a new C# code file that will contain the BAM API code
3. Replace the entire contents of the code file with a single comment:
   ```
   // Placeholder: generated by BizTalk BAM Typed API Generator
   ```
4. Add and check in the placeholder code file to your source control system (with just the comment included)
5. Add GenerateTypedBamApi.exe to your source control system, preferably within the solution directory structure
6. Add your BAM workbook (XLS/XLSX) to your source control system, preferably within the solution directory structure
7. Open the project properties for the C# project and go to the Build Events tab
8. In the Pre-build event box, enter:
   ```
   attrib -r ..\..\BamApi.cs
   ..\..\..\Tools\GenerateTypedBamApi.exe ..\..\..\BAM\BAM.xlsx
   ..\..\BamApi.cs Orchestration MyNamespace.Components
   ```
   Of course, you'll need to change the relative paths to reflect the actual locations where you checked in the EXE and BAM workbook, your desired .NET namespace, code file name and which EventStream to use.
9. Add a project reference to <BizTalkInstallDir>\Tracking\Microsoft.BizTalk.Bam.EventObservation.dll if you are using Direct or Buffered, otherwise add a project reference to <BizTalkInstallDir>\Tracking\Microsoft.BizTalk.Bam.XLANGs.dll.
10. Rebuild your solution
11. If everything looks good, check in the rest of the changes (and ensure that only the placeholder comment is checked in for the BAM API code file)

The net effect of this method is that you can keep your BAM API code file checked in to source control as an empty placeholder, and whenever your solution is built, the latest BAM model will be used to overwrite the placeholder with the actual BAM API code. This ensures that you always build against the latest BAM model, and prevents confusion by keeping generated code out of the source control system.

3.2. The BAM Typed API

Let's review the structure of the generated API code by looking at a simple example. The BAM model in this case has one activity named POTransform containing three activity items: ReceiveMsg, SendMsg and TotalPrice. Assume that we have chosen BufferedEventStream.

The C# code file contains a container class that encapsulates a set of activity classes -- one class for each activity defined in the BAM model.

In this example, we have a container class named BufferedEsApi which contains a nested activity class named POTransform. If we had multiple activities defined in the BAM model, there would be additional activity classes below POTransform. The code sample below gives a picture of what the API code would look like, with much of the implementation detail removed for clarity.

```
// Most code, comments and parameters removed for brevity
namespace Sample.Components
{
    public static class BufferedESApi
    {
        [Serializable]
        public partial class POTransform
        {
            public virtual DateTime ReceiveMsg {}
            public virtual DateTime SendMsg {}
            public virtual string TotalPrice {}
            public POTransform() {}
            public POTransform(string activityId) {}
            public virtual void BeginPOTransformActivity() {}
            public virtual void CommitPOTransformActivity(){}
            public virtual void EndPOTransformActivity(){}
            public virtual void AddReferenceToAnotherActivity(){}
            public virtual void AddCustomReference(){}
            public virtual string EnableContinuation(){}
            public virtual void Flush(){}
        }
    }
}
```

The underlying EventStream methods such as BeginActivity() are already documented by Microsoft, so we will focus on what is unique to our generated code.

The container class BufferedESApi is static and contains a couple of helper methods and a simple struct (not shown). It mainly serves as a wrapper for one or more activity classes.

To start adding data to your activity, create an instance of the nested activity class (`new BufferedESApi.POTransform()`). In an orchestration, create a variable that is typed as one of the nested activity classes.

The available constructors vary slightly depending on the chosen EventStream. DirectEventStream and BufferedEventStream both require a SQL Server connection string, so the activity class provides a constructor override that takes in a connection string (a default is also provided). OrchestrationEventStream does not use an explicit connection string. The default constructor creates an activity ID equal to `Guid.NewGuid.ToString()`.

With the activity object instance created, it's time to add some data. Notice the properties ReceiveMsg, SendMsg and TotalPrice in the code sample above. Those properties directly correspond to the activity items defined in the BAM model.

Before setting any properties, start the activity by calling BeginPOTransformActivity(). Now, go ahead and set (and re-set if needed) the values of the activity item properties. When you are ready to save the values to BAM, call the CommitPOTransformActivity() method. **If you do not explicitly commit the values, they will be lost.** When you have finished the activity, call the EndPOTransformActivity() method.

You may also span orchestration or Windows process boundaries by enabling continuation on the activity. This allows processes other than the one that started the activity to add additional data to it. To enable continuation, call the EnableContinuation() method after calling BeginPOTransformActivity(). Even with continuation enabled, you **must** complete your own processing with a call to the EndPOTransformActivity() method.

The generated C# code is well commented, so please feel free to open up the code file and take a look at what is happening under the covers.

4. Deploying BAM Models

One of the tricks to deploying a BAM model is avoiding the need to install Excel on your BizTalk servers. The BAM model is really two parts: the Excel workbook with the familiar Excel features like PivotTables, etc. and a hidden XML manifest that is stored separately inside the Excel workbook file.

If you attempt to copy the workbook to your BizTalk server and deploy it using bm.exe, you'll quickly find that Excel is required. However, if you extract the XML manifest and deploy it on your server, you'll find that it works just fine without Excel.

As a result, the ideal situation is to maintain your BAM model in the binary XLS/XLSX file, but deploy to your BizTalk servers using the XML manifest. Unfortunately, the only built-in way to get the XML out of the workbook is a menu option in the BAM add-in. That means a manual step before every deployment, and no way to automate BAM deployments.

Enter the BAM XML Exporter utility. This simple command-line application will open your binary Excel workbook, extract the XML manifest and save it to disk. Since the utility does not require Excel, it's perfect for use on an automated build server that does not have Excel installed.

The usage is simple:

```
ExportBamDefinitionXml.exe <ExcelFile> <XMLFile> [UseLegacyExport]
```

`<ExcelFile>` is the path to your BAM workbook, either XLS or XLSX format. The workbook must contain at least one BAM activity definition. If the path contains spaces, it must be wrapped in double-quotes.

`<XMLFile>` is the file path and name where the BAM XML will be saved. If the file already exists, it will be overwritten if it is not marked as read-only. If the path contains spaces, it must be wrapped in double-quotes.

`[UseLegacyExport]` is an optional (and rarely necessary) option that tells the exporter to use Excel Automation, which **does** require Excel to be installed. This option should be used only as a last resort if exports are not working for a problematic workbook. To enable this option, pass the value "True" (without the quotes).

A typical command-line may look like:

```
ExportBamDefinitionXml.exe BAM\MyBAMModel.xls BAM\MyBAMModel.xml
```

The Deployment Framework for BizTalk, also available on GitHub at https://github.com/BTDF/DeploymentFramework, provides automated deployment for BizTalk solutions, including BAM models, and already includes the BAM XML Exporter.

5. About the Authors

The original GenerateTypedBamApi utility was created by Darren Jefford and featured in the 2007 Wrox Press book "Professional BizTalk Server 2006" which he co-wrote. Darren is an experienced Solutions Architect with Microsoft in the United Kingdom. Darren's last release on CodePlex was Version 1.1 in June 2007.

In July 2008, with Darren's permission, Thomas F. Abraham picked up the project and continues to enhance and maintain it today. Thomas's contributions include several releases featuring significant enhancements to the original utility, along with the addition of the BAM XML Exporter and this documentation. Thomas is a long-time software architecture and development consultant in Minneapolis/St. Paul, Minnesota, USA, and the primary developer for the Deployment Framework for BizTalk and Environment Settings Manager projects. Thomas maintains a blog at http://www.tfabraham.com.