Embedding Microsoft .NET Code in TIBCO StreamBase Event Processing Applications
# Table of Contents

3 | Introduction  
3 | Benefits of Hosting Code in TIBCO StreamBase  
3 | The TIBCO StreamBase .NET Operator  
4 | How it Works  
5 | Embedding .NET Code Using the Operator  
9 | Performance Considerations  
10 | Benchmark Application  
11 | Conclusion
Introduction
TIBCO StreamBase® event processing is a deployment platform and authoring environment for complex event processing applications.

These are applications that react in real time to incoming events — connecting to the source of the event, collecting event data, analyzing that data, making decisions, and then taking action — often by publishing more events. Examples of event driven applications include algorithmic trading, risk management, network monitoring, and fraud detection. The TIBCO StreamBase event processing platform hosts these applications, which often combine logic written in multiple languages including TIBCO StreamBase StreamSQL, TIBCO StreamBase EventFlow™, Java, C++, R, MATLAB, or Python.

The Microsoft .NET platform has gained a wide user base over the last decade. It includes many libraries, and your organization may already have software or expertise in C#, Visual Basic, and other .NET languages. This paper describes how to embed .NET code into TIBCO StreamBase, invoking your existing or new .NET-hosted assets from a TIBCO StreamBase application, optionally feeding the results back into the application for further processing or automated actions. In this way, non-real-time .NET code can easily be executed against real-time data and trigger real-time reaction.

Benefits of Hosting Code in TIBCO StreamBase
Developing business applications that respond to real-time information can be difficult. Developers may have to deal with messaging protocols, event models, threading, scalability, and fault tolerance. By working on top of the TIBCO StreamBase platform, these applications are much easier to build. Data connectivity, threading, scale out, and other concerns are handled automatically. Developer tools for debugging, profiling, monitoring, and managing systems, appropriate for real-time deployments, are all included with the platform. Developers can thus focus on their unique analytics or business logic. The embedding of .NET code into TIBCO StreamBase brings these same benefits to .NET programmers.

The TIBCO StreamBase .NET Operator
TIBCO StreamBase’s Microsoft .NET Operator is the focal point for integration. It seamlessly loads the .NET runtime into a StreamBase application’s process along with an arbitrary .NET assembly (the “target assembly”) and invokes its code via a simple, predefined interface. Using this interface, your .NET code can process StreamBase tuples without any inter-process or inter-server communication overhead; from the StreamBase application’s perspective the .NET code is just another operator in its execution path.
How it Works

The bulk of a TIBCO StreamBase application executes in the context of a Java Virtual Machine (JVM). The .NET Operator transparently embeds the .NET runtime inside the JVM’s process and allows data to be passed between the two environments seamlessly with minimal overhead.

![Diagram of how data travels through the .NET Operator](image)

*Figure 1 - A tuple’s path through the .NET Operator*

The .NET Operator is created and configured like any other TIBCO StreamBase operator available in TIBCO StreamBase™ Studio. An instance of the operator is dragged from the Operators and Adapters palette to the canvas, and its properties are set via Studio’s Operator Properties tab.
Among the properties available is the .NET strong name (or the full path) of the assembly to load and the fully qualified name of the class contained therein which extends the StreamBase.SB.Operator class. When the application is run, the .NET Operator loads the Microsoft .NET runtime environment into the application’s process, loads the given assembly, and instantiates the given class. As tuples are received by the operator, they are passed on to the .NET object, which can then process them and optionally send tuples of its own to the application for further processing.

**Embedding .NET Code Using the Operator**

In order for the .NET Operator to interact with arbitrary .NET code, the latter must expose an object extending StreamBase.SB.Operator, which is defined in the TIBCO StreamBase .NET Client API library (StreamBase.SB.Client.dll) included in the TIBCO StreamBase base product installation. This class contains several methods that are either called by the operator when interesting events occur (for example, when a tuple is received on one of the input ports), or called by the client code to interact with the TIBCO StreamBase application (for example, to send new tuples on one of the output ports).

**Example**

Let’s say we have some logic to calculate π (pi) to the Nth digit in a C# method defined as

```csharp
string MyCalculatorClass.GetPi(int n)
```

... that is compiled into class library called CalculatorAssembly.DLL. We would like to invoke this code from our TIBCO StreamBase application.
First let’s think about the number of desired input and output ports and their associated schemas for our .NET Operator in the application. In this case we can easily handle the task by having one input port containing a single field of type \texttt{int} (call it \texttt{numDigits}) and one output port containing a \texttt{string} field (call it \texttt{answer}). More involved scenarios could of course require multiple input and output ports with complex schemas, but for this example we have all we need.

Now we need to create some TIBCO StreamBase client code in C# to extend StreamBase.\SB.\Operator. Your Microsoft Visual Studio project should:

- Be set to produce a class library
- Include a reference to \texttt{StreamBase.SB.Client.DLL} (located in <streambase_home>\bin), with its ‘Copy Local’ property set to false
- Include a reference to \texttt{CalculatorAssembly.DLL} (to use \texttt{MyCalculatorClass})
- Target the full .NET 4.0 Framework Profile as opposed to the Client Profile (this is true for any StreamBase .NET Client API code)

We’ll call the resulting assembly \texttt{MyDotNETOperator.DLL}. The implementation of the \texttt{StreamBase.SB.Operator} subclass is as follows (error checking is omitted for clarity):

```csharp
using StreamBase.SB;
namespace MyNamespace {
    public class MyDotNETOperator : Operator {
        MyCalculatorClass calc = null;
        public override void Init() {
            // Create a calculator instance
            calc = new MyCalculatorClass();
        }

        public override void ProcessTuple(int port, Tuple tuple) {
            // Get the input value from the tuple
            int numDigits = tuple.GetInt("numDigits");

            // Calculate the answer
            string answer = calc.GetPi(numDigits);

            // Place the answer in a new tuple
            Tuple returnTuple = GetOutputSchema(0).CreateTuple();
            returnTuple.SetString("answer", answer);

            // Send the tuple on the operator's output port #0
            SendOutput(0, returnTuple);
        }

        // The class needs a public default constructor (compiler-generated or explicitly defined),
        // which will be used by the .NET Operator to create the instance
        public MyDotNETOperator() {
        }
    }
}
```
When the MyDotNETOperator.DLL assembly is compiled, we place it alongside CalculatorAssembly.DLL in a directory (call it C:\MyAssemblies\). The accompanying .PDB files may also be included for debugging purposes.

Now we have all the pieces needed by the .NET Operator, and we can create the TIBCO StreamBase application. The EventFlow will be simple: one input stream connected to a .NET Operator, in turn connected to an output stream. The input stream’s schema contains a numDigits field of type int.

On the operator’s Operator Properties tab the full path to the assembly is entered (C:\MyAssemblies\CalculatorAssembly.DLL), as well as the fully-qualified class name (MyNamespace.MyDotNETOperator). On the Ports tab the number of input ports and output ports are both set to 1. On the Schemas tab the schema of the output port is made to contain one field: answer (string).

Everything is now set and ready to run. When TIBCO StreamBase Studio’s Run button is clicked, the application is started, the .NET runtime environment is loaded along with all the assemblies, and an instance of MyDotNETOperator is created. In the Debug Perspective’s Manual Input pane, you can enter a value for numDigits and hit the Send Data button, which will route the tuple to the .NET code and calculate the value of p to the specified digit. The .NET code will generate a new tuple with the answer, and this will be emitted on the operator’s output stream, as evidenced by the Application Output pane.
Assembly Loading Considerations

The .NET runtime environment has a special set of rules in determining locations to examine when looking for assemblies. For starters (and rather counter-intuitively) it doesn't look in the path defined in the PATH environment variable, and the rules only get more complex after that.

The .NET Operator's ability to locate and load your assembly depends on several factors, outlined in this table:

<table>
<thead>
<tr>
<th>How the assembly name is specified to the operator's Assembly Name property</th>
<th>How the operator locates the assembly</th>
<th>How the operator locates other referenced assemblies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the assembly's exact location on disk (C:\MyDir\MyAssembly.DLL)</td>
<td>Looks in the specified directory. If a relative path is specified, the current working directory of the StreamBase application is used as a reference. If the assembly is not located in the specified directory, the load will fail.</td>
<td>If normal .NET loading rules fail to locate the assembly, the directory containing the operator's assembly is probed. If the assembly is found there, it will be loaded. Otherwise the load will fail.</td>
</tr>
<tr>
<td>Using the assembly's simple name (&quot;MyAssembly&quot;) or fully qualified name (&quot;MyAssembly, Version=1.0, ...&quot;)</td>
<td>Normal .NET loading rules apply. If the assembly is not found using these rules, the load will fail.</td>
<td>If normal .NET loading rules fail to locate the assembly, the directory containing the operator's assembly is probed. If the assembly is found there, it will be loaded. Otherwise the load will fail.</td>
</tr>
</tbody>
</table>

If your assembly refers to other assemblies, the normal .NET loading rules will be used in locating them; however should these rules fail, a last attempt will be made by looking in the directory where your initial assembly was found.

Clearly, the most expedient solution to ensure seamless operation is to place all the assemblies in a specific directory whenever possible and use the full path to that directory as a value to the operator’s Assembly Name property.

If a situation is encountered where either the target assembly or one of its dependents fails to load, determining the exact cause can be tricky. One thing to double-check in such cases is for the target assembly’s reference to StreamBase.SB.Client.DLL to have its ‘Copy Local’ property set to false. Beyond that there are a number of online resources to help with diagnosis, most of which revolve around the use of a free Microsoft tool called Fusion Log Viewer.
Debugging the .NET Operator’s Target Assembly

Provided you have a copy of Microsoft Visual Studio installed and debugging symbols are available (the assembly’s .PDB file), debugging the .NET client code is easy:

• Open the target assembly’s solution in Visual Studio
• Place a breakpoint in the desired location (for example in the code’s implementation of Operator.ProcessTuple())
• In the menu go to Debug | Attach to Process… and select sbd-java.exe in the list of processes, then click Attach

The debugger should load the process and install the breakpoint. From then on debugging proceeds as with any other .NET application.

If you wish to debug your operator’s Init() method, an additional step will be required because this gets called very early in the application’s lifecycle, and it is doubtful there will be enough time to attach the .NET debugger before its invocation. The solution here is to run your TIBCO StreamBase® Server with the --suspend option to start your application in suspended mode, then use TIBCO StreamBase’s sbadmin utility(sbadmin resume) to resume your application after having attached your .NET debugger to the process.

Performance Considerations

Interoperability between TIBCO StreamBase and .NET code involves translating data from StreamBase’s memory structures to .NET’s and vice versa, a process called data marshaling. This extra conversion step consumes some processing cycles and affects the overall performance of the operator (target assembly’s processing time notwithstanding, of course). Specifically:

• The more data there is to convert in a given StreamBase tuple, the more processing cycles it will cost to marshal.
• Complex data types (nested tuples, lists, capture fields, functions) require more cycles to marshal than simple data types (boolean, integer, long, double, string, timestamp, blob).

Given these observations, we can look at some benchmark tests involving simple and complex schemas (both containing the same total number of fields) in an attempt to characterize performance.

Both benchmark tests are identical save for their respective input and output schemas. In each test, one hundred thousand tuples are first sent through to prime both the Java and .NET JIT compilers. Then one million tuples are sent as fast as possible while timing the operation. The target assembly performs minimal work but returns a tuple for each tuple sent so that timing values represent full StreamBase-.NET-StreamBase roundtrips.

The simple schema is defined as follows:

• 7 int fields
• 7 long fields
• 6 string fields
• 6 double fields
The complex schema is defined as:

- int field
- long field
- tuple field with a schema of \{(int, long, double, string)\}
- list-of-tuples field (tuple schema is again \{(int, long, double, string)\}), five elements in the list

Note that both schemas contain the same number of fields overall (26).

**Benchmark Application**

The TIBCO StreamBase project described above includes two separate but nearly identical TIBCO StreamBase applications, one that uses the .NET Operator and one that uses a Java Operator performing the same work. In this way a valid comparison can be made for the performance overhead of running custom .NET code vs. custom Java code in a TIBCO StreamBase application.

**Benchmark Results**

Here are the results from running the .NET Operator benchmark test application:

<table>
<thead>
<tr>
<th></th>
<th>Tuple roundtrip time (SB -&gt; .NET -&gt; SB)</th>
<th>Roundtrip tuples per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Schema</td>
<td>~26.3 microseconds</td>
<td>~38,000</td>
</tr>
<tr>
<td>Complex Schema</td>
<td>~44.0 microseconds</td>
<td>~22,700</td>
</tr>
</tbody>
</table>

By comparison, an identical test run against a Java Operator performing exactly the same work exhibits the following performance:

<table>
<thead>
<tr>
<th></th>
<th>Tuple roundtrip time (SB -&gt; .NET -&gt; SB)</th>
<th>Roundtrip tuples per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Schema</td>
<td>~1.65 microseconds</td>
<td>~606,000</td>
</tr>
<tr>
<td>Complex Schema</td>
<td>~2.7 microseconds</td>
<td>~370,400</td>
</tr>
</tbody>
</table>

These tests were run on a Dell Latitude E6510 quad-core laptop with 6GB of RAM running Windows 7 64 bits SP1, with StreamBase 7.3.1 64 bits. During the test the processor load hovered around 25%, although no one core was at 100% at any time. The application was run from the command line to avoid the additional overhead introduced by TIBCO StreamBase Studio’s runtime environment.

**Performance Conclusions**

Hosting .NET code in TIBCO StreamBase adds about 24-42 microseconds of latency compared to a similar Java operator – a total end to end latency of about 26-44 microseconds.
Conclusion

The TIBCO StreamBase platform is a powerful and productive way to write event driven applications that acquire and process data and perform automated actions. The platform enables the StreamBase developer to focus on business logic. The addition of the .NET Operator further enhances the overall platform productivity through seamless integration of existing or new .NET application logic. Productivity gains are achieved by leveraging existing code and enabling developers to use their existing skills while also benefitting from using TIBCO StreamBase to coordinate event-oriented processing.