Automatic SQL Server Transactions with .NET

Introduction

Coming from a web application programming background (using COM+/ASP), one of the (many) attractive features of .NET that I recognized early on was the ability to write attributed COM+ components. Features such as the `[AutoComplete]` attribute which would call `SetComplete` or `SetAbort` for you automatically meant that you had to worry less and less about writing plumbing code in the middle tier and almost every line you did write could contain business logic.

One problem I found with using Enterprise Services in .NET to write managed COM+ components was that of performance. Although we've all been told a thousand times by now that Enterprise Services don't use COM interop to do their thing, the fact remains that they are still comparatively slow because they provide support for much more than I normally need. For example, it is rare that I need to code transactions against more than one resource manager – it's usually just SQL Server – so using the DTC is really overkill.

Wouldn't it be nice to write attributed code that handled automatic enlistment into transactions and automatic commits/rollbacks but that just used SQL Server's own transaction support? That's what I set out to do here.

Hasn't this been done before?

That's nice, you say, sounds familiar too. Hasn't someone already written this? I'm sure I've seen it done in a few places.

Well, yes, this is yet another SQL transaction library, but I hope with some differences. For example, much of the code that I've seen in the past didn't really answer the n-tier question for me: how do you use this to write business layer and data layer components that work well together in a consistent manner.

My approach makes a clear distinction between business and data components. A number of assumptions are made about the architecture of an n-tier system:

- Data component methods can only be instantiated and called from business components. The presentation layer is not allowed to call into data components and data components can not call into each other.
- Business component methods may call other methods in the instance, may call into other business components, and may call into data components.
- Data components will always enlist into the current transaction if one is active.
- When a business method is called from the presentation layer, its attributes determine whether a transaction will be used. If it is marked with a `[RequireTransaction]` attribute, then all of the code that follows will run under a single transaction whether it appears in other business components or data components. If the method returns successfully then a commit occurs and if an exception is thrown then a rollback occurs.

Working with COM+ components in the unmanaged world, we typically had to have matching read-only and transactional business components, the former marked as supporting transactions and the latter requiring transactions so that simple read operations

wouldn't cause a transaction to be started. With my transaction library, individual methods can be marked to indicate whether they need to run under a transaction and this makes the code simpler.

Interception using ContextAttribute

Consider how your unmanaged code runs against COM+: objects that are enlisted into the same transaction appear in the same "context". We use `GetContextObject()` to access the properties of the context and this allows us to call `SetComplete` or `SetAbort` for example.

In .NET we will also make use of the concept of a context: each business component that we create from our presentation layer will be assigned into a new context. When a method call is made to this object, we will determine if a transaction will be required and any business or data components created within the method call will automatically be added to the same context. As the method returns and leaves the context, we will automatically commit or rollback any transaction as appropriate.

To do this, we make use of the `ContextAttribute` class. Although full support for this class was dropped during the .NET beta process and you'll find MSDN advises that "This type supports the .NET Framework infrastructure and is not intended to be used directly from your code." messages, the code still works and still appears even in the .NET 1.1 runtime.

The context support is implemented as follows. First our business components must derive from `ContextBoundObject`:

```
[TransactionalObject]
public class BusinessComponent : ContextBoundObject {
      ...
}
```

The `[TransactionalObject]` attribute instructs the .NET runtime to call into our interception code. It is derived from `ContextAttribute`:

```
[AttributeUsage(AttributeTargets.Class)]
public class TransactionalObjectAttribute :
            ContextAttribute, IContributeObjectSink {

      public const string PropertyName = "SqlTxAspect";

      public TransactionalObjectAttribute() : base(PropertyName) {
      }

      public override bool IsContextOK(Context ctx,
                  IConstructionalCallMessage ccm) {
            return ctx.GetProperty(PropertyName)!=null;
      }

      public override void GetPropertiesForNewContext(
                  IConstructionalCallMessage ccm) {
            ccm.ContextProperties.Add(this);
      }

      public IMessageSink GetObjectSink(MarshalByRefObject o,
                  IMessageSink next) {
            return new SqlTxAspect(o,next);
      }
}
```

By attaching this attribute to a class derived from `ContextBoundObject`, we get to decide which context a new instance is created in. When the instance is created, `IsContextOK` is called to ask if the current context is acceptable – if we return false, a new context will be created. Here, we check to see if the current context contains a reference to our "context property"; if so, the context already contains one of our business objects so we are being created from inside a business method and we keep the context. Otherwise, we signal that we want a new context.

When a new context is created, `GetPropertiesForNewContext` is called and this allows us to add our context property, which in this case is the attribute class itself.

Finally, the runtime calls the `GetObjectSink` method which allows us to add a new `IMessageSink` implementation to the sink chain. When a method is called on an object in another context, the runtime passes the call information to the item at the top of a linked list of `IMessageSink` instances, the sink chain. Each one can pre-process the call before passing it on to the next (or in fact manufacturing the return value without even making the call). By adding a new sink, we get to participate in that process which allows us to intercept the calls and deal with the transaction commit/rollback. In the code above, we create a `SqlTxAspect` object which provides our `IMessageSink`. `SqlTxAspect` will cache the next value in the sink chain and a reference to our sink is returned for the runtime to call.

```
public class SqlTxAspect : IMessageSink {
      private IMessageSink m_next;

      public SqlTxAspect(MarshalByRefObject o,IMessageSink next) {
            m_next = next;
      }

      public IMessageSink NextSink {
            get { return m_next; }
      }

      public IMessage SyncProcessMessage(IMessage msg) {
            bool bRequirePostProcess = PreProcess(msg);
            IMessage returnMessage = m_next.SyncProcessMessage(msg);
            if(bRequirePostProcess) PostProcess(msg,returnMessage);
            return returnMessage;
      }

      public IMessageCtrl AsyncProcessMessage(IMessage msg,
                  IMessageSink replySink) {
            // async not supported
            throw new InvalidOperationException();
      }

      private bool PreProcess(IMessage msg) {
            // TODO: pre process
            return false; // no post-processing
      }

      private void PostProcess(IMessage msg,IMessage msgReturn) {
            // TODO: post process
      }

}
```

Out implementation of `IMessageSink` stores the reference to the next sink in the chain in the constructor. When a method is called, the `IMessage` containing the details of the call is passed to `SyncProcessMessage` where we can pre- and post-process the call.

We call `SyncProcessMessage` on the next sink in the chain to make sure the method is ultimately executed.

What we have now is a `BusinessComponent` class that allows us to intercept its method calls. This provides the foundation for out automatic transactions.

Transaction Context

There are two main items that we want to flow automatically between our objects: the SQL Server connection string and the active transaction if there is one. I call this the "transaction context", it will be associated with our `ContextAttribute` context when an object is created in a new context, and we will use a class called `TxCtx` to hold the details.

The `TxCtx` class keeps track of the current connection and keeps it open if a transaction is active. On the other hand, if it is being used just for read-only methods, the connection is closed after each use and relies on traditional connection pooling for scalability. Take a look at the source code to see exactly how it works.

Our business component base class will look like this:

```
[TransactionalObject]
public abstract class BusinessComponent : ContextBoundObject {
	public BusinessComponent(string connString) {
		TransactionSupport.TxCtx = new TxCtx(connString);
	}

	public BusinessComponent() {
		TransactionSupport.AssertContext();
	}
}
```

When a new business object is created from the presentation later, the connection string will be passed into the constructor. `TransactionSupport` is a helper class that provides the link between the current context and its `TxCtx` object. In the code above, a new transaction context is assigned if a connection string is passed. If the default constructor is used, it is assumed that this component is being created from within another business method and so the `AssertContext` method ensures that this is the case.

The data component base class is simple because it always receives the connection string from the context:

```
public abstract class DataComponent : ContextBoundObject {
	public DataComponent() {
		TransactionSupport.AssertContext();
	}

	...
}
```

The data component class contains a host of helper methods for creating and executing SQL commands using the current transaction context. This makes it very easy to implement a real data component without having to worry about how any of the underlying infrastructure works.

Putting it into practice

I haven't gone into a great deal of detail about the remaining implementation of the SQL transaction code. With the information above, it should be fairly easy to follow the source

code – there is nothing particularly difficult going on provided you understand the basics of ADO.NET and how to use the managed SQL Server data provider.

It is probably easiest to see what we have achieved with a small example:

```
public class Country : DataComponent {
      public int Add(string name) {
            SqlCommand cmd = CreateStoredProcCmd("AddCountry");
            CreateParameterWithValue(cmd,"@Name",name);
            CreateIntReturnValue(cmd,"@ID");
            RunWithCheck(cmd);
            return (int)cmd.Parameters["@ID"].Value;
      }

      public void Info(int ID,out string Name) {
            SqlCommand cmd = CreateStoredProcCmd("GetCountry");
            CreateParameterWithValue(cmd,"@ID",ID);
            CreateStringReturnValue(cmd,"@Name",100);
            RunWithCheck(cmd);
            Name = (string)cmd.Parameters["@Name"].Value;
      }
}

public class RegistrationManager : BusinessComponent {
      public RegistrationManager() { }
      public RegistrationManager(string conn) : base(conn) { }

      [RequireTransaction(IsolationLevel.ReadCommitted)]
      public int AddCountry(string name) {
            Data.Country c = new Data.Country();
            return c.Add(name);
      }

      public string GetCountryName(int id) {
            Data.Country c = new Data.Country();
            string name;
            c.Info(id,out name);
            return name;
      }
}

public class TestApp {
      private static readonly string CS =
            "SERVER=(local);DATABASE=RegMan;UID=sa;PWD=;"

      public static Main() {
            RegistrationManager rm = new RegistrationManager(CS);
            int id = rm.AddCountry("United Kingdom");
            Console.WriteLine(rm.GetCountryName(id));
      }
}
```

Things to note:

- The only attributes needed in your code are to mark business methods requiring a transaction.
- You must supply a default constructor and a connection string constructor in your business component classes. The default is used if one business component instantiates another – the connection string flows between them and so isn't needed in the constructor.