Transactions in .NET Enterprise Services

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

Transaction integrity is critical when dealing with enterprise applications. How does one develop code to guarantee it? It takes non-trivial amount of effort and at the end of this, are we sure we have covered all holes? Passing around a transaction object through out the system is not the best of the solutions available. It should be noted that transaction is an aspect and a crosscutting concern in a large scale system. The best way to handle this aspect may be is to intercept the code. This is exactly the capability of transaction support in Enterprise Services. We discuss the capabilities, some pitfalls and workarounds of the transaction support in the .NET Enterprise Services.

Transaction Integrity and Need

Transaction binds a set of related tasks the either succeed or fail as a unit, i.e., it is atomic. In a large scale system, several classes and components may be participating in the same transaction. Tying each of these classes into one transaction may be a challenge. Guaranteeing the integrity of the transaction requires close examination of the code, and one needs to make sure all related objects and tasks fall into that one transaction. Each participant object needs to have a say in the success or failure of the transaction. How do we achieve this in a fool safe manner? One option is to create a transaction object and pass that object through out the system, through method calls. This process is largely tedious and error prone.

We will illustrate how one could realize these goes with .NET Enterprise Services using a simple example. Let's first write the example without regard to any transactions.

Let's say we have collectors. A collector collects items. We have two kinds of collectors: Big time collector who buys 1000 or more items; a small time collector who buys any number of items. We have created a SQL server database with the information shown in Figure 1.

id	items_count	big_collector
1	10000	1
2	100	0
3	12000	1
4	122	0

Figure 1. Big time and small time collectors shown in the CollectorDB database.

The details of the collectors table is shown in Figure 2.

Column Name	Data Type	Length	Allow Nulls
id	int	4	
items_count	int	4	
big_collector	bit	1	

Figure 2. The columns of the collectors table.

We want to develop a simple ASP.NET application that will let us trade items between collectors. Let's start by creating a blank solution named CollectorExchange. In the blank solution, we create a new C# class library project named CollectorCompLib. Within that project, we create two classes CollectorFactory and Collector as shown below:

//CollectorFactor.cs

```
using System;

namespace CollectorCompLib
{
      public class CollectorFactory
      {
            public Collector getCollector(int theID)
            {
                  Collector ctr = null;

                  ctr = new Collector();
                  ctr.load(theID);
                  return ctr;
            }
      }
}
```

//Collector.cs

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace CollectorCompLib
{
      public class Collector
      {
            private readonly string dbconnection =
                  System.Configuration.ConfigurationSettings.
                        AppSettings["DBConnectionString"].ToString();

            private int id;
            private int count;
            private bool bigCollector;

            public int collectorID
            {
                  get { return id; }
            }

            public int collectionCount
            {
                  get { return count; }
            }

            protected internal Collector() {}
            // You must use the Factory to get Collector

            private void init(int theID, int itemsCount, bool big)
            {
                  id = theID;
                  count = itemsCount;
                  bigCollector = big;
            }

            protected internal virtual void load(int theID)
            {
```

```
                SqlConnection connection =
                        new SqlConnection(dbconnection);
                SqlCommand command = connection.CreateCommand();
                command.CommandText =
                  "SELECT * from collectors where [id] = " + theID;
                connection.Open();
                SqlDataReader reader = command.ExecuteReader();

                if (reader.Read())
                {
                        init(Convert.ToInt32(reader["id"]),
                                Convert.ToInt32(reader["items_count"]),
                                Convert.ToBoolean(reader["big_collector"])
                        );
                }
                else
                        throw new ApplicationException("Invalid id");

                connection.Close();
        }

        public void buy(int numberOfItems)
        {
                if (bigCollector && numberOfItems < 1000)
                        throw new ApplicationException(
                                "That's too small for me to buy");
                else
                {
                        count += numberOfItems;
                        save();
                }
        }

        public void sell(int numberOfItems)
        {
                if (count < numberOfItems)
                        throw new ApplicationException(
                                "Not enough items to sell");
                else
                {
                        count -= numberOfItems;
                        save();
                }
        }

        protected virtual void save()
        {
                SqlConnection connection =
                        new SqlConnection(dbconnection);
                SqlCommand command = connection.CreateCommand();
                connection.Open();
                command.CommandText =
                        "UPDATE collectors SET items_count = " +
                        count + " where [id] = " + collectorID;

                command.ExecuteNonQuery();
        }
    }
}
```

Let’s make a few observations from the above code. Ideally, I would move the select statements to the stored procedures. One is not allowed to create an object of Collector.

You can ask the CollectorFactory to get you a collector. The collector's load and save are inaccessible outside the project.

Now, we create a C# ASP.NET Web application named CollectorWebSite. In it we create a CollectorTrading.aspx which has the controls shown in Figure 3.

Figure 3. Layout of the CollectorTrading.aspx page.

The code behind page, CollectorTrading.aspx.cs, is shown below:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using CollectorCompLib;

namespace CollectorWebSite
{
      public class CollectorTrading : System.Web.UI.Page
      {
            protected System.Web.UI.WebControls.Label Label1;
            protected System.Web.UI.WebControls.Label Label2;
            protected System.Web.UI.WebControls.TextBox
                        sellerIDTextBox;
            protected System.Web.UI.WebControls.TextBox buyerIDTextBox;
            protected System.Web.UI.WebControls.Label
                        sellerItemsCountLabel;
            protected System.Web.UI.WebControls.Label
                        buyerItemsCountLabel;
            protected System.Web.UI.WebControls.TextBox
                        sellCountTextBox;
            protected System.Web.UI.WebControls.Label Label3;
            protected System.Web.UI.WebControls.Label messageLabel;
            protected System.Web.UI.WebControls.Button refreshButton;
            protected System.Web.UI.WebControls.Button tradeButton;

            private void Page_Load(object sender, System.EventArgs e)
            {
            }
```

```
#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
      //
      InitializeComponent();
      base.OnInit(e);
}

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
      this.refreshButton.Click +=
new System.EventHandler(this.refreshButton_Click);
      this.sellCountTextBox.TextChanged +=
new System.EventHandler(this.sellCountTextBox_TextChanged);
      this.buyerIDTextBox.TextChanged +=
new System.EventHandler(this.buyerIDTextBox_TextChanged);
      this.sellerIDTextBox.TextChanged +=
new System.EventHandler(this.sellerIDTextBox_TextChanged);
      this.tradeButton.Click +=
new System.EventHandler(this.tradeButton_Click);
      this.Load += new System.EventHandler(this.Page_Load);

}
#endregion

private void sellerIDTextBox_TextChanged(object sender,
                  System.EventArgs e)
{
      Collector seller =
            new CollectorFactory().getCollector(
                  Convert.ToInt32(sellerIDTextBox.Text));
      if (seller != null)
      {
            sellerItemsCountLabel.Text =
                  seller.collectionCount.ToString();
      }
      else
      {
            sellerItemsCountLabel.Text = "Invalid: " +
                  sellerIDTextBox.Text;
            sellerIDTextBox.Text = "";
      }

      enableDisableButtons();
}

private void sellCountTextBox_TextChanged(object sender,
             System.EventArgs e)
{
      enableDisableButtons();
}

private void buyerIDTextBox_TextChanged(object sender,
            System.EventArgs e)
{
      Collector buyer =
            new CollectorFactory().getCollector(
                  Convert.ToInt32(buyerIDTextBox.Text));
      if (buyer != null)
      {
```

```
                buyerItemsCountLabel.Text =
                        buyer.collectionCount.ToString();
        }
        else
        {
                buyerItemsCountLabel.Text = "Invalid: " +
                         buyerIDTextBox.Text;
                buyerIDTextBox.Text = "";
        }

        enableDisableButtons();
}

private void enableDisableButtons()
{
        tradeButton.Enabled = false;
        refreshButton.Enabled = false;
        if (sellerIDTextBox.Text.Trim() != ""  &&
                        buyerIDTextBox.Text.Trim() != "")
        {
                refreshButton.Enabled = true;

                if (sellCountTextBox.Text.Trim() != "")
                        tradeButton.Enabled = true;
        }
}

private void tradeButton_Click(object sender,
                System.EventArgs e)
{
        try
        {
                Collector seller =
                new CollectorFactory().getCollector(
                        Convert.ToInt32(sellerIDTextBox.Text));

                Collector buyer =
                new CollectorFactory().getCollector(
                        Convert.ToInt32(buyerIDTextBox.Text));

                int count =
                        Convert.ToInt32(sellCountTextBox.Text);

                seller.sell(count);
                buyer.buy(count);
        }
        catch(Exception ex)
        {
                messageLabel.Text = ex.Message;
        }
}

private void refreshButton_Click(object sender,
                System.EventArgs e)
{
        try
        {
                Collector seller =
                new CollectorFactory().getCollector(
                        Convert.ToInt32(sellerIDTextBox.Text));

                Collector buyer =
                new CollectorFactory().getCollector(
                        Convert.ToInt32(buyerIDTextBox.Text));
```

```
                        sellerItemsCountLabel.Text =
                              seller.collectionCount.ToString();
                        buyerItemsCountLabel.Text =
                              buyer.collectionCount.ToString();
                  }
                  catch(Exception ex)
                  {
                        messageLabel.Text = ex.Message;
                  }
            }
      }
}
```

Studying the code shown above indicates that clicking the Refresh button displays the number of items owned by the seller and buyer. Clicking the Trade button results in the call to sell on the seller and buy on the buyer.

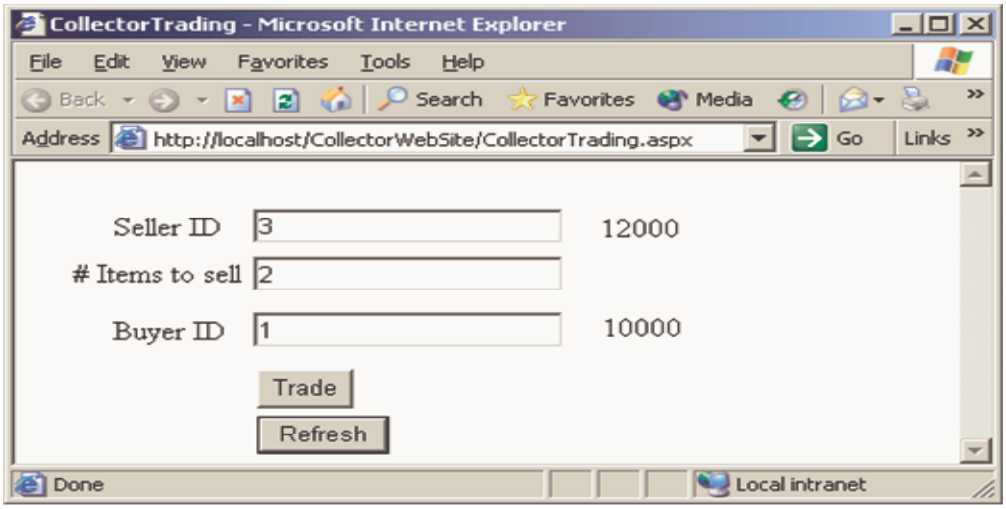
The web.config file of the ASP.NET project was modified to add the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

<appSettings>
      <add key="DBConnectionString" value="Data Source=localhost;Initial
Catalog=CollectorDB;User ID=sa;Password=sapwd" />
</appSettings>
  <system.web>
…
```

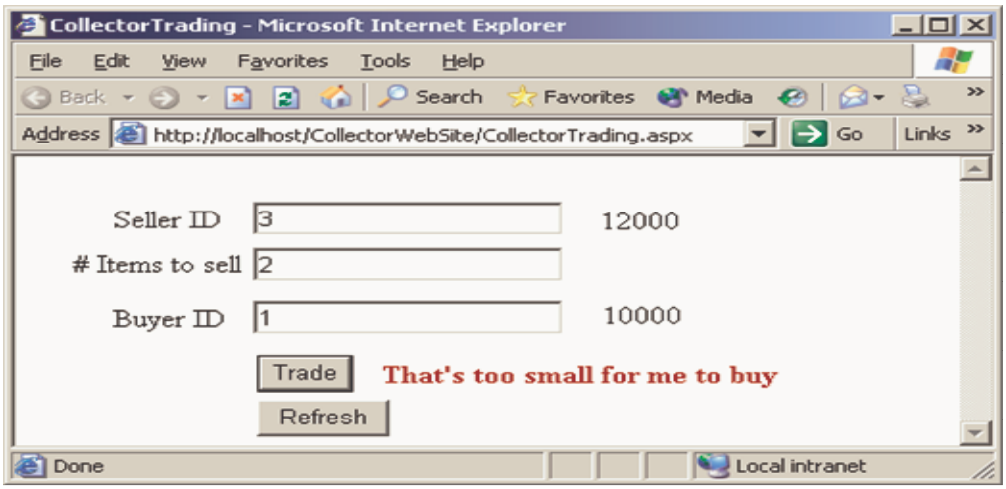
The database connection string is being read from the config file by the Collector class.

Driving the application

Let's run the application. Figure 4 illustrate the response from the application.



(a). Refresh button clicked. Shows quantity owned by collectors.



(b). Trade button clicked. Error message displayed

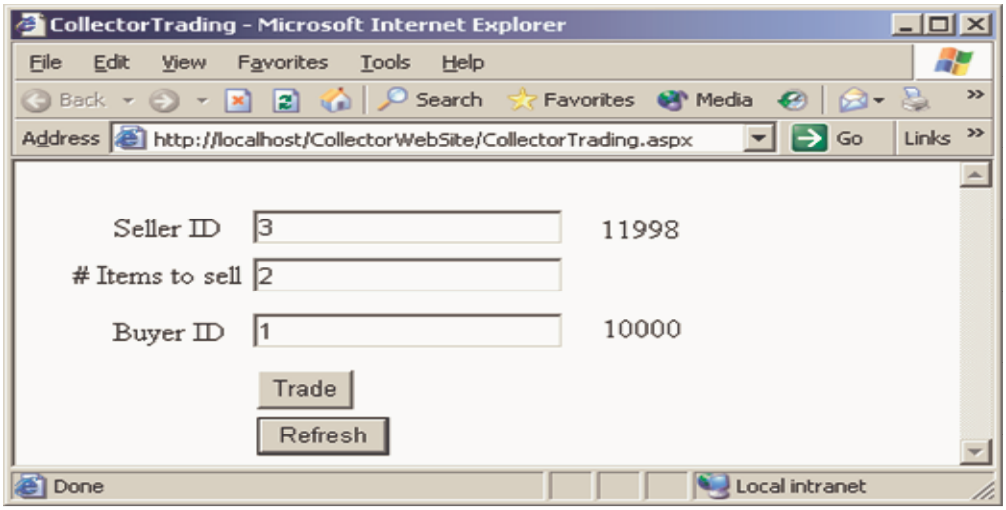


Figure 4 (c). Refresh button clicked. Shows 2 items were "lost" in the failed trading.

As seen from Figure 4, the application as written does not preserve the transaction. Items collector 3 ends up loosing two items in the failed trading. Ideally, if items collector 1 did not buy the 2 items, it should not have been deducted from the stash of collector 3. How do we fix this? One possibility, as mentioned before, is to create a transaction object and pass it through all the functions that are involved in the transaction. .NET Enterprise Services and Serviced Component provides a better alternative to this. We will refine this example to use Enterprise Services after a quick and short introduction to Serviced Component.

Serviced Component

".NET Enterprise services" is the integration of COM+ services into the .NET framework. It brings to .NET the capabilities of COM+: object pooling, just-in-time activation, queued components, transactions, and more.

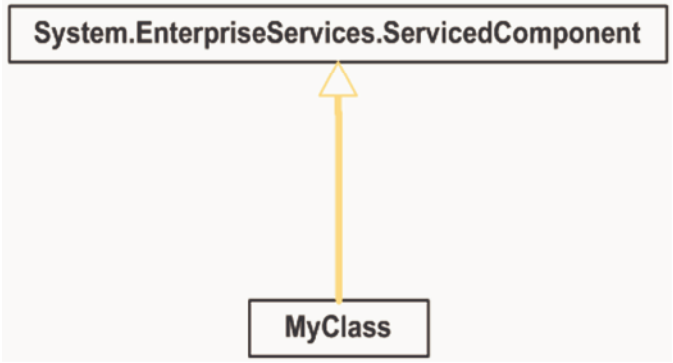


Figure 5. A serviced component "MyClass."

A serviced component is a class that derives from the ServicedComponent class, which is in the System.EnterpriseServices namespace. The class must be written in a CLS-compliant language and must have a public no-argument constructor. Its transaction (and other) requirements are specified declaratively using attributes.

A serviced component is deployed as a COM+ component (may be manually deployed – which is the preferred way or may be automatically deployed – which is what we are doing in this article). You may view and manipulate it using COM+ catalog just like you could manage the traditional COM+ components. When a serviced component object is created, the COM+ environment starts managing it. It monitors calls to these objects. It creates a transaction context under which the object is executed and monitored. We will take a look at this context using the COM+ catalog later.

Implementing the Serviced Component

We will now make the Collector a Serviced Component. Take a look at the set of code changes required and how simple it is to make this transaction aware and compliant.

1. First in the CollectorCompLib project and the CollectorWebSite project add reference to System.EnterpriseServices as shown in Figure 6.

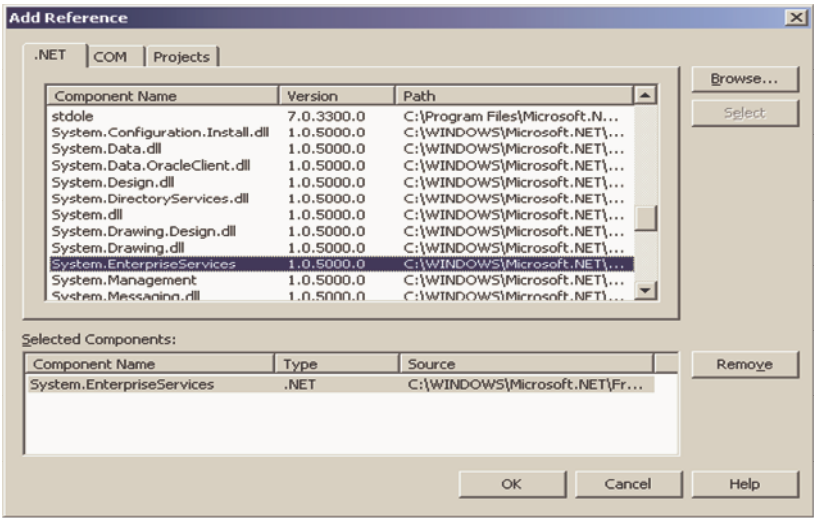


Figure 6. Adding reference to System.EnterpriseServices assembly.

2. Add a using and inherit the class Collector from ServicedComponent as shown below:

```
using System.EnterpriseServices;

namespace CollectorCompLib
{
      [Transaction(TransactionOption.Required)]
      public class Collector : ServicedComponent
      {
```

Note the use of the Transaction attribute above the Collector class.

3. In front of the buy method and sell method, add the AutoComplete attribute as shown here:

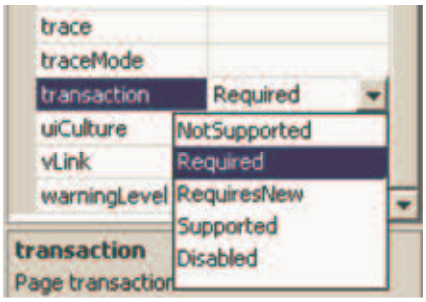
```
[AutoComplete]
public void buy(int numberOfItems)
{
…

[AutoComplete]
public void sell(int numberOfItems)
{
…
```

4. Remember one of the rules for implementing a serviced component is to have a public no-argument constructor. So, we have to modify the constructor of Collector as follows:

```
public Collector() {}
```

The original intent of making it protected internal is to make sure no one creates an object of Collector in an uncontrolled fashion. The only way to create it was using the CollectorFactory. We will compromise this one for a few minutes and then see how we can enforce it again.

5. Bring up the design view of the CollectorTrading.aspx page and go to the properties. Modify the transaction property to "**Required**" as shown below:

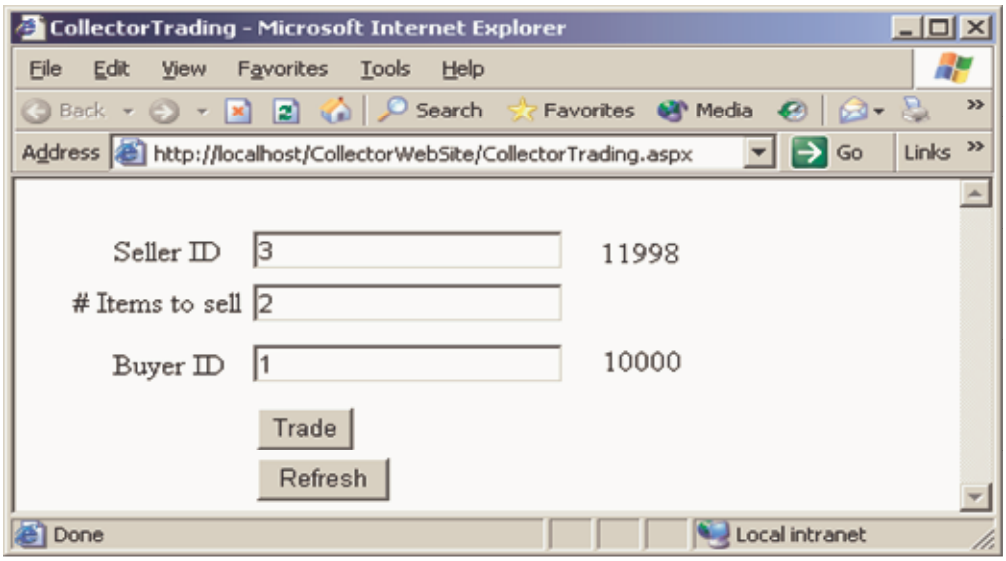


6. Since we are going to rely on automatic registration of our enterprise service, our ASP.NET page needs to have permission to do that. Modify the web.config to add an identify element as shown below:

```
<system.web>
<identity impersonate="true"/>
```

…

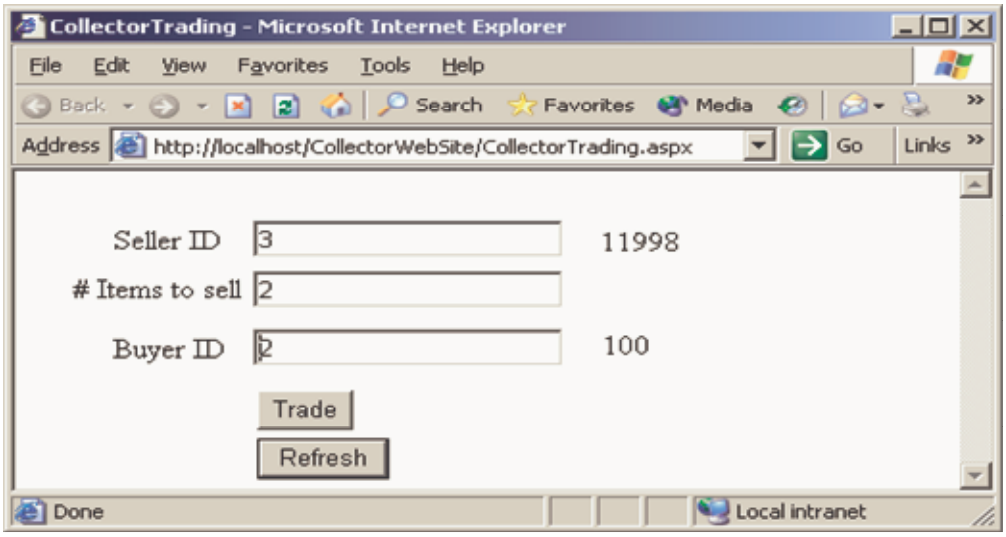
That's pretty much the change that is needed. Let's run the application and see how it differs from the earlier run.



(a). Refresh button was clicked to display initial items in stash.

(b). Trade button was clicked to receive the error message as expected.

(c). Refresh button was clicked. Note that the quantity has not been affected this time



(d). Shows quantity for different collectors before a successful trade.

Figure 7 (e). Both the Trade and Refresh button were clicked to show successful trading.

So, with minimal change, we were able to preserve the integrity of transaction. Hopefully, this example illustrates the power of Enterprise Services and the ease with which transactions can be realized.

Problem with AutoComplete

You noticed that we marked the sell and buy methods with the AutoComplete attribute. The effect of AutoComplete is a call to either ContextUtil.SetComplete() or ContextUtil.SetAbort(). If the method is successful (as defined by the fact that it did not throw any exceptions), then ContextUtil.SetComplete() is called. If the method were to throw an exception, then ContextUtil.SetAbort() is called. These two methods let your transaction bound object to set its vote towards the success or failure of the transaction. If any component or object that is part of the transaction context sets the negative vote, the transaction will be rolled back. The transaction is committed only if all the involved components cast their positive vote.

I hear you saying, "This sounds reasonable, what's your problem, Venkat?" Let's make a slight change to the code. After the trade button is clicked, it will be nice if the updated quantity is displayed without us having to click on the Refresh button, isn't it? So, here is the change to the CollectorTrading.aspx.cs to accommodate that.

```
private void tradeButton_Click(object sender,
                                    System.EventArgs e)
{
      try
      {
            Collector seller =
            new CollectorFactory().getCollector(
                  Convert.ToInt32(sellerIDTextBox.Text));

            Collector buyer =
            new CollectorFactory().getCollector(
                  Convert.ToInt32(buyerIDTextBox.Text));

            int count =
                  Convert.ToInt32(sellCountTextBox.Text);

            seller.sell(count);
            buyer.buy(count);

            sellerItemsCountLabel.Text =
                  seller.collectionCount.ToString();
            buyerItemsCountLabel.Text =
                  buyer.collectionCount.ToString();
      }
      catch(Exception ex)
      {
            messageLabel.Text = ex.Message;
      }
}
```

The last two statements (highlighted) will accomplish the goal of updating the quantities right after the successful trade, at least in theory!

Now, let's give this a run. The quantities after the trade button is clicked are shown below in Figure 8.

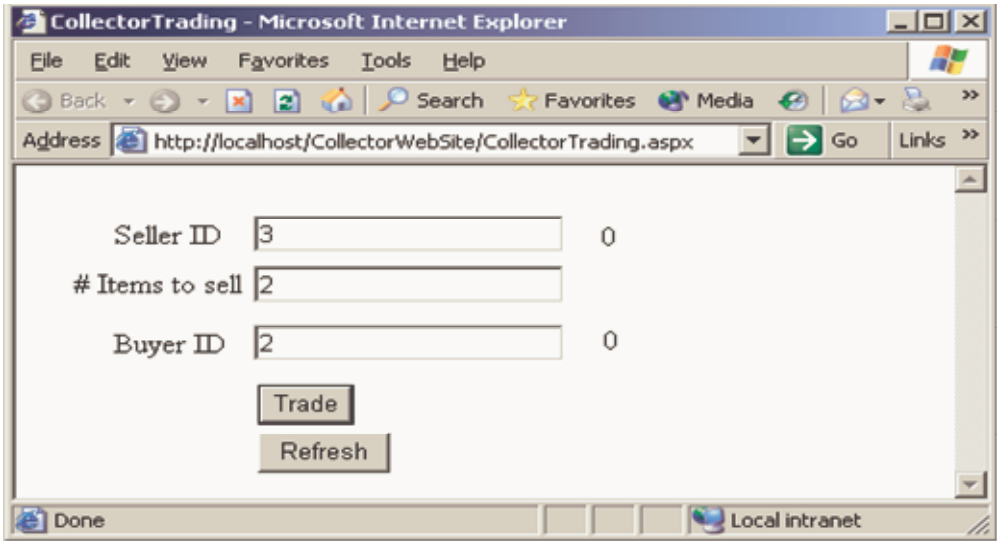


Figure 8. Problem with AutoComplete is illustrated here.

So what went wrong? The problem is that the calls to SetAbort and SetComplete have a side effect. In addition to casting their vote, they also implicitly set the ContextUtil.DeactivateUponReturn to true. This flag tells the COM+ run time to dispose the object upon return from the method call (kind of like working for the Mafia?!). The reference held in the aspx page is not a real reference to the object, but a reference to a proxy. Much like how the object behaves when Just-in-time activation is utilized, the next call to a method using the reference results in a brand new object being instantiated to serve the request. This new object does not have the data loaded and hence results in the erroneous response. While we want to set the vote for transaction commit or abort, we may not want the object to be disposed.

Fix – Do not use AutoComplete

Instead of using AutoComplete, it is better to directly set your vote as shown below:

```
//[AutoComplete] // Not used
public void buy(int numberOfItems)
{
      ContextUtil.MyTransactionVote =
            TransactionVote.Abort;
      // First set it to abort. If successful,
      //set it to success.
      if (bigCollector && numberOfItems < 1000)
            throw new ApplicationException(
                        "That's too small for me to buy");
      else
      {
            count += numberOfItems;
            save();
      }

      // Looks good, so let's vote positive now
```

```
            ContextUtil.MyTransactionVote =
                TransactionVote.Commit;
        }
```

Note that similar change is effected in the sell method as well. Running the program now will show that the transaction integrity is preserved and the updated quantity is displayed right after a successful trading as well.

Controlling the Object Creation

The reason for initially making the constructor of Collector protected internal is to eliminate the possibility of a developer creating objects of our collector in an uncontrolled fashion. We wanted to make sure the only way to create the object is using our CollectorFactory. Unfortunately, a ServicedComponent is required to have a public no-argument constructor. How do we satisfy this requirement without compromising our goal?

It is actually pretty easy to do that! First, try the following line of code in the `refreshButton_Click` method.

```
        private void refreshButton_Click(object sender,
              System.EventArgs e)
        {
              try
              {
                    Collector shouldNotWork = new Collector();

                    Collector seller =
                    new CollectorFactory().getCollector(
                          Convert.ToInt32(sellerIDTextBox.Text))
…
```

Ideally, a compilation error should occur at the statement where shouldNotWork reference is created. However, if you compile the code as is, you will notice no error is generated.

Now let's modify the collector class as follows:

```
[Obsolete(
  "Please use the CollectorFactory to get a Collector object",
  true)]
public Collector() {}
```

We have set the "Obsolete" attribute with a true flag on the constructor. The true flag tells the compiler to generate an error (instead of a warning). This prohibits any code to utilize the constructor, but at the same time works fine with the Enterprise Services framework.

One small caveat is, the code within our CollectorFactor will not compile. This can be fixed using a reflection trick as shown below:

```
public Collector getCollector(int theID)
{
      Collector ctr = null;

      //ctr = new Collector();
      ctr = Activator.CreateInstance(
                  typeof(Collector)) as Collector;
      ctr.load(theID);
      return ctr;
}
```

Of course, you may ask, what prohibits from a user of our class from doing the same thing, say from the aspx page. A disciplined user would see the error message and use the factory to create the object.

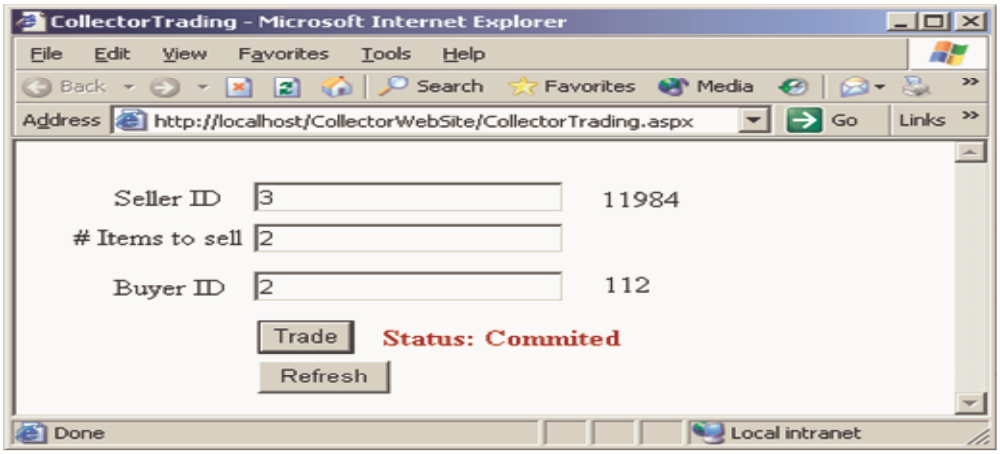
Windows 2003 Version Enhancement

While the above solution worked great, we are required to inherit from the ServicedComponent class. A true interception should not require this. In .NET Framework 1.1 and on Windows 2003 server, a new feature is available. ServiceDomain creates a stack of the transaction context and manages without the need to inherit from ServicedComponent. Here are the changes to the code to use the ServiceDomain.

1. Do not inherit Collector from ServicedComponent. Here is how it looks after this change:
```
[Transaction(TransactionOption.Required)]
public class Collector
{
```
2. Set the transaction property for the aspx page to empty.
3. Modify the trade button handler as follows:
```
private void tradeButton_Click(object sender,
             System.EventArgs e)
{
      ServiceConfig cnfg =
            new ServiceConfig();

      cnfg.TrackingEnabled = true;
      cnfg.TrackingAppName = "CollectorWebSite";
      cnfg.TrackingComponentName =
            "CollectorWebSiteContext";
      cnfg.Transaction =
                  TransactionOption.Required;

      ServiceDomain.Enter(cnfg);

      try
      {
            Collector seller =
            new CollectorFactory().getCollector(
                  Convert.ToInt32(sellerIDTextBox.Text));

            Collector buyer =
            new CollectorFactory().getCollector(
                  Convert.ToInt32(buyerIDTextBox.Text));
```

```
                int count =
                      Convert.ToInt32(sellCountTextBox.Text);

                seller.sell(count);
                buyer.buy(count);

                sellerItemsCountLabel.Text =
                      seller.collectionCount.ToString();
                buyerItemsCountLabel.Text =
                      buyer.collectionCount.ToString();
          }
          catch(Exception ex)
          {
                messageLabel.Text = ex.Message;
          }

          TransactionStatus status =
                      ServiceDomain.Leave();

          messageLabel.Text += " Status: " +
                status.ToString();
    }
```

Compiling and running this version displays as shown in Figure 9.



(a) After a successful trade.

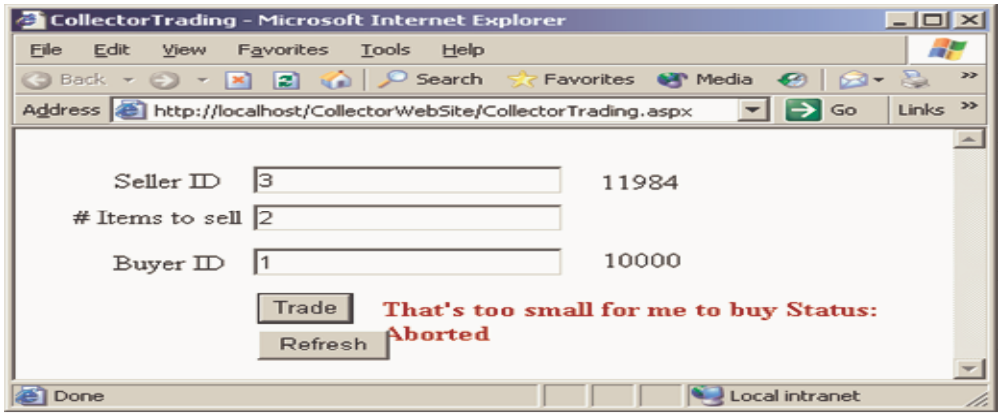


Figure 9. (b) After a failed trade.

Eliminating the call to Save

We are invoking save from buy and sell. What if there are several methods that need to access the data. It will be nice if the save happens towards the end. One option is to move the save to Dispose (of course there is the disadvantage of the user forgetting to call Dispose). Here are the changes made:

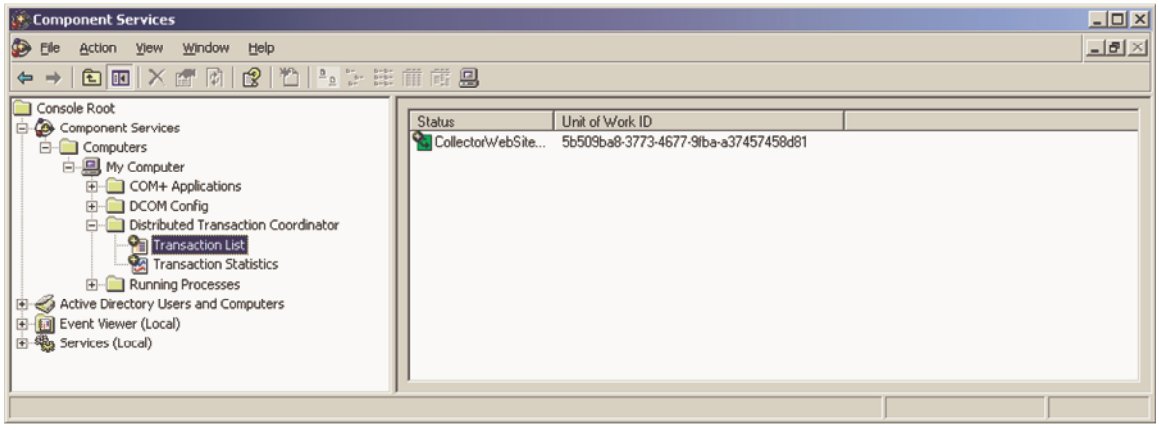
1. Modified the Collector class as follows:
   ```
   public class Collector : IDisposable
   ```
2. Added a field to the Collector class:
   ```
   private bool dirty = false;
   ```
3. Modified the buy method (*and the sell method as well*) to set a dirty flag:
   ```
   public void buy(int numberOfItems)
   {
         ContextUtil.MyTransactionVote =
                     TransactionVote.Abort;
         // First set it to abort. If successful,
         //set it to success.
         if (bigCollector && numberOfItems < 1000)
               throw new ApplicationException(
                     "That's too small for me to buy");
         else
         {
               count += numberOfItems;
               dirty = true;
         }

         // Looks good, so let's vote positive
         ContextUtil.MyTransactionVote =
               TransactionVote.Commit;
   }
   ```
4. Implemented the Dispose method as shown here:
   ```
   public void Dispose()
   {
         if (dirty) save();
   }
   ```
5. Finally, modified the aspx.cs page as shown below:
   ```
   private void tradeButton_Click(object sender,
         System.EventArgs e)
   {
         ServiceConfig cnfg =
               new ServiceConfig();

         cnfg.TrackingEnabled = true;
         cnfg.TrackingAppName = "CollectorWebSite";
         cnfg.TrackingComponentName =
                     "CollectorWebSiteContext";
         cnfg.Transaction =
                     TransactionOption.Required;

         ServiceDomain.Enter(cnfg);

         try
         {
               using(Collector seller =
               new CollectorFactory().getCollector(
                     Convert.ToInt32(sellerIDTextBox.Text))
               {
   ```

```
                        using(Collector buyer =
                        new CollectorFactory().getCollector(
                          Convert.ToInt32(buyerIDTextBox.Text))
                        {

                              int count =
                                    Convert.ToInt32(
                                          sellCountTextBox.Text);

                              seller.sell(count);
                              buyer.buy(count);

                              sellerItemsCountLabel.Text =
                               seller.collectionCount.ToString();
                              buyerItemsCountLabel.Text =
                               buyer.collectionCount.ToString();
                        }
                  }
            }
            catch(Exception ex)
            {
                  messageLabel.Text = ex.Message;
            }

            .TransactionStatus status = ServiceDomain.Leave();

            messageLabel.Text += " Status: " + status.ToString();
      }
```

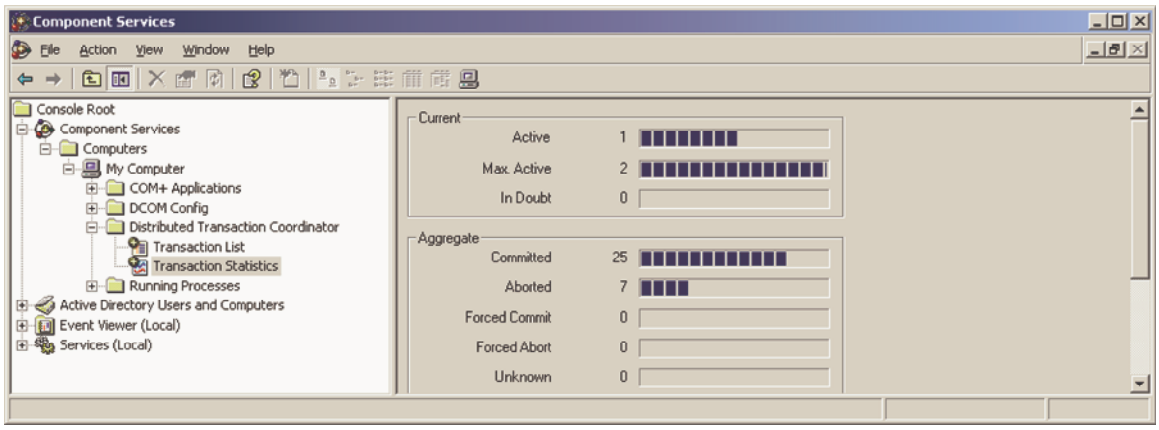
The using clause forces the call to Dispose on the objects at which time the save is invoked.

Viewing the transaction in COM+ catalog

Put a break point in the tradeButton_Click method within the try block and run the application in debug mode. Click on the Trade button. When the break point is reached, bring up the Component Services (from Control Panel, Administrative Tools, and Component Services). Navigate to the Distributed Transaction Coordinator link and you will see the following:



Notice the name of the transaction is what we hard coded for TrackingComponentName. The transaction statistics shows the following:



Conclusion

Properly handling transactions is a challenge. .NET Enterprise Services provides an infrastructure to take care of this with great ease, robustness and reliability. We have introduced transaction handling using an example, presented the issues and workarounds as well. Even though this article was a bit length, we hope you enjoyed reading it and benefit from it. Please write to us to let us know your opinion.

References

1. Microsoft Visual Studio .NET 2003 documentation – refer to "using System.EnterpriseServices namespace."