7 The Controller

Introduction

At this point, we have several wrapper classes and the web service running. We are now ready to write the main `Controller` class that will take the transaction request from the web service, split it into individual requests, decide where each of the requests goes, and compile and return the results.

We will look at a few major items in this chapter. First, we will discuss the `Controller` component. This is the "traffic cop" that takes in the request sent from the web service and directs it to the appropriate places. We will also examine the `RequestsProcessor` component. In this class, the requests are broken apart and an appropriate `RequestHandler` component is called for the particular request type. Finally, we will look at the `RequestHandlerFactory`. This class has some nifty code to create an instance of the appropriate handler class simply by having a name passed in.

Because we are going to allow clients to send in synchronous or asynchronous requests, we will discuss the .NET System.Messaging namespace. We will illustrate how to send asynchronous requests to the Microsoft Message Queuing system, and we will look at the System.Reflection namespace to find the bit of magic that enables you to create an instance of a request handler class on the fly by just supplying the class name.

Getting to the Controller

I like to follow this basic design tenet when designing web services:

Move as much processing logic out of the web service as possible.

In other words, when an HTTP request comes into the web service, do as little processing in the web service itself as possible, and farm out all remaining logic to classes in "the engine." In this case, the engine begins with a class called `Controller`. Aside from basic XML structural validation and user/session authorization, the vast bulk of the processing occurs in or below this `Controller` class.

The reason for this approach was born out of experience. In the early days of the .NET web service classes, I (and others) had trouble putting complex logic in the web service classes. To be fair, I also ran into problems with the Java Servlet components. Various combinations of "stuff" caused the process to hang. In other cases, the results were not what I had expected, and the debugging task was difficult. As a matter of habit, then, I have come to move the more interesting logic out of the web service itself and to call helper classes instead.

The `Controller` component is an example of a "point man" class—a class acting as the lead in an entire chain of other classes—used to kick off the bulk of the actual processing. This tends to make the code easier to read anyway and greatly reduces the debugging complexity, because you then can write standalone tester programs that call this class.

The issues I noted previously with the web service classes likely have been fixed in the .NET Framework. However, I will continue to use this approach of moving logic out, because doing so brings clarity and easier debugging.

Figure 7-1 is a rather high-level sequence diagram of a request flow. It shows the calling sequence from the initiation of a request through the calling of the `Controller` component. This should put into perspective the context of the remainder of this section, in which we discuss the `Controller` in more detail.

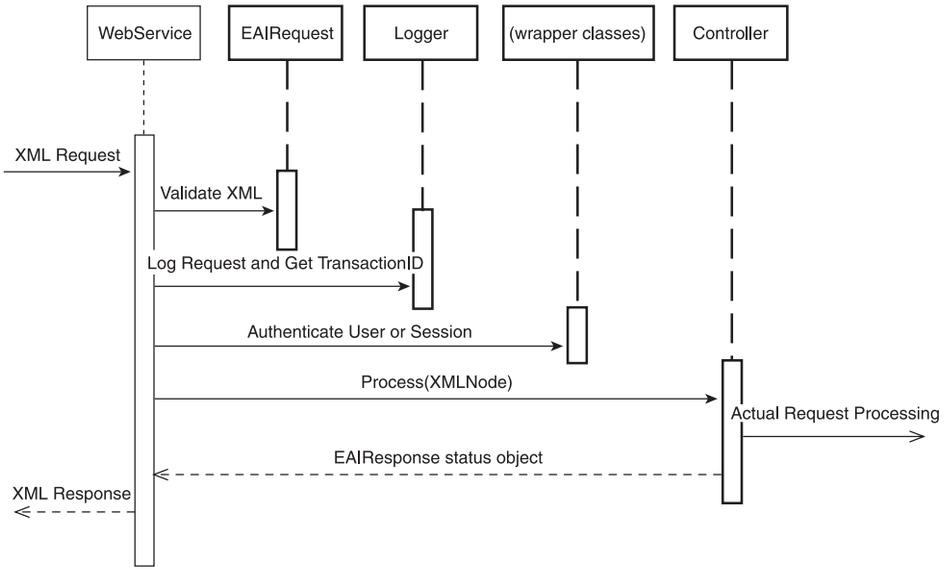

Figure 7-1 Getting to the `Controller`.

The `Controller` Structure

Surprisingly, the `Controller` class is named `Controller`. It takes in as its sole parameter an `XmlNode` object and returns an EAIResponse object. By the time the `Controller` gets the go-ahead to start processing the input via its `process` method, you should know a few things:

- The input request was a well-formed XML message.
- If the request was a new login message, and the username and password were valid.
- If the request was submitted for an existing session, that session was valid.

The `Controller` doesn't actually see login requests. They are handled by the web service itself.

Because we know each of these pieces of information here, it's full steam ahead as we enter the `process` method. The `XmlNode` object is simply a representation of the request string the originator submitted. We expect this to be in the following format:

```
<EAIRequest>
  <SessionID>xyz-123-abc-987</SessionID>
  <Requests>

    <Request Name="DoThis">
      <Needed>info</Needed>
    </Request>
  </Requests>
</EAIRequest>
```

The `Controller` class has a default, blank constructor. It then has a single public method, named `process`.We look at a couple of private helper methods in a moment.

From a high level, the `Controller` component is really responsible for only a few, albeit important tasks:

- Converting an `XmlNode` object into an `EAIRequest` object
- Determining whether the request is synchronous or asynchronous
- Sending an `EAIRequest` object to `MessageQueue` for asynchronous requests
- Creating a `RequestsProcessor` object for synchronous requests
- Firing off the actual `Requests` to the `RequestsProcessor`
- Compiling status info and creating and returning an `EAIResponse` object

Figure 7-2 is a sequence diagram of a simple synchronous request coming into the `Controller`.

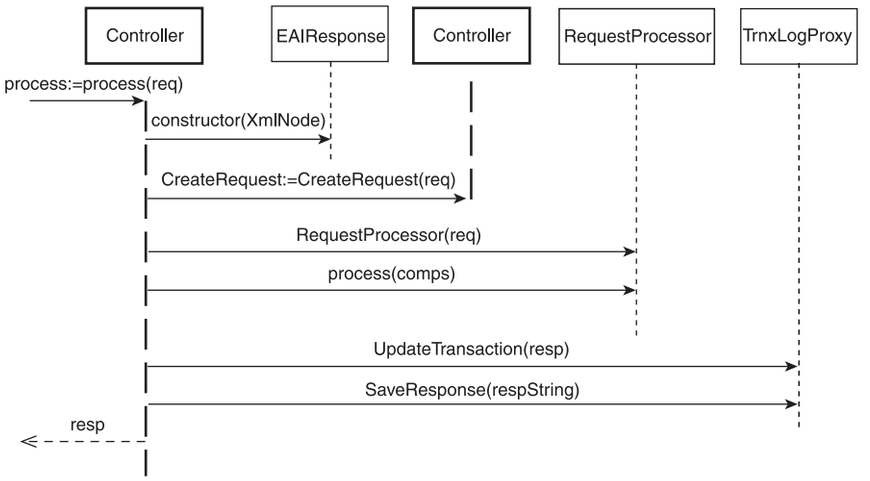

Figure 7-2 `Controller` synchronous request processing.

The `Controller` first creates a new `EAIResponse` object that will be returned from the `process` call. It then converts the incoming `XmlNode` object, representing the original request, into an `EAIRequest` object. This is a helper object that contains various information and objects that make it much easier to process the request. Among the `EAIRequest` members is a `Requests` object that contains `Request` objects. These `Request` objects each represent a single `<Request>` block from the incoming request.

After the `EAIRequest` object has been created, `Controller` checks to see if the request is synchronous or asynchronous. For our purposes, a synchronous request is specified in the incoming message by the originator, and it means that the caller expects the request to be handled while he waits. In other words, the caller expects a response to the request that contains status information about all `Request` block processing for requests sent in. For example, a request specified for asynchronous processing would have the following structure:

```
<EAIRequest>
  <SessionID>123-xyz-987-abc</SessionID>
  <Requests Asynch="true">
     <Request Name="First"> ... </Request>
        ...
     <Request Name="Last"> ... </Request>
  </Requests>
</EAIRequest>
```

To make matters a bit more confusing (but flexible), the caller can also specify that he wants processing to either continue on in the face of `Requests` returning an error or to `FailOnFirstError`. Failing on first error means that processing stops the first time a `Request` block returns an error. This is accomplished by the optional XML attribute `FailOnFirstError` on the `Requests` element. If the attribute is not present, `FailOnFirstError` is set to `false`. For example, if you were submitting a bunch of `Request` blocks in a `Requests` block and you wanted the processing to halt the first time any `Request` returned an error, you would send in something like the following:

```
<EAIRequest>
  <SessionID>123-xyz-987-abc</SessionID>
  <Requests FailOnFirstError="true">
     <Request Name="First"> ... </Request>
        ...
     <Request Name="Last"> ... </Request>
  </Requests>
</EAIRequest>
```

Therefore, you have four ways in which to combine the preceding two attributes. They are shown in Table 7-1.

Table 7-1 `FailOnFirstError` and `Asynch` request attributes

	`FailOnFirstError = TRUE`	`FailOnFirstError = FALSE`
Synchronous	Processing starts from `Controller` and returns when the first error is encountered processing a `Request` block. Therefore, not all requests might even have processing attempted.	Processing starts from `Controller` and continues on with each `Request` block being processed, even if one or more `Request` blocks return an error.
Asynchronous	Processing starts from the Microsoft Message Queuing (MSMQ) and returns when the first error is encountered processing a `Request` block. Therefore, not all requests might even have processing attempted.	Processing starts from the MSMQ and continues with each `Request` block being processed, even if one or more `Request` blocks return an error.

Listing 7-1: Controller.cs DiscussionController.cs

```
using System;
using System.Xml;
using System.Messaging;
using System.Collections;

using EAIFramework.Messages;
using EAIFramework.Util;
using EAIFramework.Handler;


namespace EAIFramework.Controller
{
  /// <summary>
  /// EAIFramework.Controller.Controller
  ///   This class is responsible for taking in an XmlNode object
  ///   and converting it to an EAIRequest object.  Then each
  ///   Request block in the incoming EAIRequest message is
  ///   processed in either a synchronous or asynchronous
  ///   fashion, as specified in the incoming message.
  /// </summary>
  public class Controller
  {
     public Controller(){}//end constructor
```

Listing 7-2 shows the `process` method that will do the bulk of the work.

Listing 7-2: Controller.cs `process` Method

```
public EAIResponse process(XmlNode req)
{
  Logger.log(Logger.INFO,"------------------------");
  Logger.log(Logger.INFO,"Entering Controller ");
  Logger.log(Logger.INFO,"with str: " +req.OuterXml);
  Logger.log(Logger.INFO,"------------------------");
  TrnxLogProxy tlp = new TrnxLogProxy();

  EAIFramework.Messages.EAIResponse resp =
    new EAIFramework.Messages.EAIResponse();
  resp.OverallStatusCode=StatusCodes.OK;
  resp.OverallStatus = StatusCodes.Descriptions(
                    StatusCodes.OK);


  // Create the EAIRequest object
  Logger.log(Logger.INFO,
    "Controller: Going to create new EAIRequest");
  EAIRequest eaiReq =
    CreateRequest(req);
  resp.TransactionID=eaiReq.TransactionID;
  resp.SessionID=eaiReq.SessionID;
  resp.OriginalXML=req.OuterXml;

  if( ! eaiReq.Synchronous)
  {
    Logger.log(Logger.INFO,
        "THIS IS AN ASYNCHRONOUS TRNX!");
    resp = this.processAsynchRequests(eaiReq);
    // Now save and return the response to the caller

    int nRows = tlp.SaveResponse(resp.ToString(),
      resp.TransactionID);
    return resp;
  }//end Asynch request.


  Logger.log(Logger.INFO,
    "THIS IS A SYNCHRONOUS TRNX!");
  //-------------------------------------------------
  // The rest of this method is handling
  //    a SYNCHRONOUS request
  //-------------------------------------------------
  // Set up a place to compile the results of
  // the <Request> processing.  There will be
  //  one Component (in Components) for each
  //  Request (in Requests).  This will come back in the
  //  ArrayList rreqs, below.

  RequestsProcessor processor = new RequestsProcessor(
    eaiReq);
  ArrayList rreqs = processor.process();
```

```
 if( !(eaiReq.SessionID.Equals("")) )
   resp.RequestingUsername = eaiReq.SessionID;
 else
resp.RequestingUsername =
     eaiReq.RequestingUsername;

 // Populate the EAIResponse object with
 //  the results of the processing

 // Obscure the Password, if it's there
 try{req.SelectSingleNode(
   "/EAIRequest/RequestingPassword").
   InnerText = "*****";}
 catch(Exception exc){}

 resp.RequestResponses = (RequestResponse[])
   rreqs.ToArray(
   new RequestResponse().GetType());

 tlp.UpdateTransaction(resp);
 int nRespRows = tlp.SaveResponse(resp.ToString(),
   resp.TransactionID);
 return resp;
}//end process()
```

The `process` method performs, or at least kicks off, the heavy lifting of `Request` processing. It starts off by doing some diagnostic logging and then creates a `TrnxLogProxy` object. This is used later to update the transaction in the database.

Next, a new `EAIResponse` object, named `resp`, is created that you use to return to the caller all the status information gathered from the various steps needed to process each `Request` block. By default, you set the response object as showing success (`OK`).

The first of the private helper methods, `CreateRequest()`, is called. It takes in the `XmlNode` object and returns a populated `EAIRequest` object. This object can then be used to easily gain access to the various parts of the input request. We take a look at `CreateRequest()` in a moment.

After `CreateRequest()` is called, the main branching for synchronous or asynchronous processing takes place. If the request is not to be handled in a synchronous manner (that is, if it's asynchronous), you call the private helper method `processAsynchRequests()`. This also is discussed in a moment. This basically sends the request to a message queue that watches for asynchronous requests. Before you leave, you save off the response to the database.

On the other hand, if this request is to be handled in a synchronous manner, a new `RequestsProcessor` object is created. The `RequestsProcessor` class, discussed later in this chapter, encapsulates all the functionality necessary to actually process a bunch of requests. You could have written the code for this processing here in the `Controller` class. However, because the request is sent off to a message queue for asynchronous processing, you can call the same `RequestsProcessor` class from both the `Controller` class and the message queue. This leaves you with a single chunk of code, making maintenance and enhancements easier and more reliable.

The process method of the RequestsProcessor class is called next, and you get back an ArrayList of RequestResponse objects. There should be one for each processed Request. When FailOnFirstError is false, you should see a RequestResponse object for each Request. If FailOnFirstError is true, you might see fewer RequestResponse objects than Requests, if one of the Requests fails.

When the RequestsProcessor returns its results, Controller updates the EAIResponse object that will be sent back. You'll recall that the T_TransactionLog table has a column to hold either SubmittingUser or SessionID, because both will never come in at the same time. Controller sets the RequestingUsername member to the value of either a SubmittingUsername or a SessionID, whichever is present. It then sets the Password field to a series of asterisks, if password has a value. This just masks the password in the response message. It's true that, for now, you should never see requests with a username and password hit the Controller. However, to be prepared for a time when you need to send login requests to the Controller, the information will be presented back to the user as he would expect it to be. Finally, the response object is populated with the remainder of the interesting information, including the TransactionID, SessionID, OriginalXML string, and RequestResponse status objects returned by the RequestsProcessor.

To finish out the Controller.process method, the TrnxLogProxy object that was created near the start of the method is called on two different methods. First, the UpdateTransaction() method is called. This takes in the newly-created response object and does an UPDATE command on the row in the T_TransactionLog table that has the TransactionID held in the response object. It at least changes the Status and StatusCode columns.

All that's left to do is return the EAIResponse object to the caller. For now, the only component calling the Controller class is the web service. It then returns to the original caller a string representation of the response object that you just sent back.

Private Helper Methods

Now let's take a look at the private helper methods in the Controller class that the process method called. The longest of these methods is the CreateRequest() method, shown in Listing 7-3. It takes in an XmlNode object and returns an EAIRequest object.

Listing 7-3: The CreateRequest() Method

```
/// <summary>
/// This is a helper method for the
///  process() method that hides all
///  the logic to build the EAIRequest
///  object from the input XmlNode
///  object
/// </summary>
/// <param name="req">The input XmlNode object</param>
/// <returns>EAIRequest object</returns>
private EAIRequest CreateRequest(XmlNode req)
{
  EAIRequest rRet = new EAIRequest();
```

```
rRet.OriginalXML = req.OuterXml;
try {
  rRet.RequestingUsername =
    req.SelectSingleNode(
    "/EAIRequest/RequestingUsername")
    .InnerText;
} catch (Exception exc) {
  Logger.log(Logger.INFO,
   "CreateReq: Couldn't read RequestingUsername: "
    + exc.Message);
  rRet.RequestingUsername="";
}

try{
  rRet.TransactionID =
  Int32.Parse(
  req.SelectSingleNode(
   "/EAIRequest/TrnxID").InnerText);
} catch (Exception exc) {
  Logger.log(Logger.INFO,
      "CreateReq: Couldn't read TrnxID: " +
    exc.Message);
  rRet.TransactionID = -1;
}


try{
  rRet.OverallStatus =
    req.SelectSingleNode(
    "/EAIRequest/OverallStatus")
    .InnerText;
} catch (Exception exc) {
  Logger.log(Logger.INFO,
      "CreateReq: Couldn't read OverallStatus: " +
    exc.Message);
}

try {
  Logger.log(Logger.INFO,
     "CreateReq: Going to get StatusCode now...");
  string sSC = req.SelectSingleNode(
    "/EAIRequest/OverallStatusCode").InnerText;
  if( (sSC==null) || sSC.Equals(""))
    sSC="0";
  Logger.log(Logger.INFO,
    "CreateReq: Just got StatusCode: " + sSC);
  int nSC = Int32.Parse( sSC );
  Logger.log(Logger.INFO,
    "CreateReq: Just converted int to: " +
    nSC.ToString());
  rRet.OverallStatusCode = nSC;
  Logger.log(Logger.INFO,
    "CreateReq: Stuck in OverallStatusCode.");
} catch(Exception exc) {
  Logger.log(Logger.INFO,
```

```
    "CreateReq: Couldn't read OverallStatusCode: "
    + exc.Message);
}//end catch

try {
  Logger.log(Logger.INFO,
    "CreateReq: Going to get SessionID now...");
  string ssn = req.SelectSingleNode(
    "/EAIRequest/SessionID")
    .InnerText;
  rRet.SessionID = ssn;
  Logger.log(Logger.INFO,
    "CreateReq: Stuck in SessionID " + ssn);
} catch(Exception exc) {
  Logger.log(Logger.INFO,
    "CreateReq: Couldn't read SessionID: " +
    exc.Message);
  rRet.SessionID = "";
}//end catch
//rRet.Started =
//  new DateTime( req.SelectSingleNode(
//    "/EAIRequest/Started").InnerText);

XmlNode reqs = req.SelectSingleNode(
        "/EAIRequest/Requests");
XmlAttribute async = reqs.Attributes["Asynch"];
if(async != null) {
  if(async.InnerText.Equals("true"))
    rRet.Synchronous = false;
  else
    rRet.Synchronous = true;
}//End if Asynch != null
else
  rRet.Synchronous = true;

XmlAttribute fof = reqs.
    Attributes["FailOnFirstError"];
if(fof != null) {
  if(fof.InnerText.Equals("true"))
    rRet.FailOnFirstError = true;
  else
    rRet.FailOnFirstError = false;
}//End if FailOnFirstError != null
else
  rRet.FailOnFirstError = false;

// Now build the Requests object
XmlNodeList xnl = req.SelectNodes(
    "/EAIRequest/Requests/Request");
Requests r = new Requests();
if(xnl.Count>0)
{
  int nReqNum = 0;
  foreach(XmlNode xn in xnl)
  {
```

```
        Request rr = new Request(xn.OuterXml);
        rr.TrnxID = rRet.TransactionID;
        rr.Iteration = nReqNum++;
        r.add(rr);
      }//end foreach
      rRet.Requests = r;
    }//end if there are any requests
    rRet.Requests = r;

    return rRet;
  }//end CreateRequest()
```

The `CreateRequest()` method begins by instantiating a new `EAIRequest` that will be returned to the caller. It then steps through, member by member; pulls out the information from the `XmlNode` object sent in; and populates the appropriate member of the request object. Working with the XML is the same as done previously, with the possible exception of pulling in an attribute. You will want to check for a couple of different attributes, including `FailOnFirstError` and `Asynch`.

I will now walk through pulling out a particular XML element, to make sure you understand what is going on in the code. You'll notice that very similar code repeats several times, getting different XML elements. I will then walk through pulling out an XML attribute. It also repeats for subsequent attributes later in the method.

First, let's look at the code to pull out the `RequestingUsername` element (see Listing 7-4). The statement is placed in a `try/catch` block, because the `SelectSingleNode()` method can throw an exception. The code populates the `RequestingUsername` member of the `rRet (EAIRequest)` object, which is a string. It is populated with the `InnerText`—in other words, the value between the XML element tags—for the element matching the XPath search string of `/EAIRequest/RequestingUsername`. Let's take a look at the code that actually does the XPath search now.

Listing 7-4: XPath Example

```
try{
  // First get the XmlNode object for the XPath search
  // of "/EAIRequest/RequestingUsername".
  // This XPath search should find the XML Element at the
  //  following spot:
  //
  //  <EAIRequest>
  //    <RequestingUsername>theUser</RequestingUsername>
  //      ...other interesting XML goodies here...
  //   </EAIRequest>

  System.XML.XmlNode xnReqUser = req.SelectSingleNode(
    "/EAIRequest/RequestingUsername");

  // Now that we have the XmlNode for the RequestingUsername,

  //  get the value for that element.  In this case, strUser
  //  would be set to equal "theUser".
  string strUser = xnReqUser.InnerText;
```

```
  // Finally, populate the RequestingUsername member with the

  //  value we got for strUser.
  rRet.RequestingUsername = strUser;
}
catch( Exception exc)
{
   // error processing here...
}
```

As you might notice with most of the code in this project, I like to split most operations into separate statements. Over the years, I have found that it's much quicker to debug problems if the statements are separated than if you have many methods chained together in a single statement. I also tend to break that rule if I have become particularly comfortable with a series of calls. Crunching through an `XmlNode` is just one of those occasions. I will also concede that code looks "cooler" if it's all smushed into one line, but I long ago (and quite happily, I might add) traded worrying about looking cool for simplifying my life. Splitting multiple calls into separate lines is one such simplification.

The `catch` block traps any errors, including the case in which the element doesn't exist. If this is the case, the response `RequestingUsername` is set to a blank string. The same holds true for several other bits of information, including the `TransactionID`, `OverallStatus`, `OverallStatusCode`, and so on. Each of these works the same way.

Midway through this method is the code to pull out the `Request` XML nodes. Here is where you hit the first of the XML attributes, `Asynch`. You tell the system to process this transaction asynchronously by sending in the attribute `Asynch` with a value of `true` on the `Requests` XML element. For example, the tag would look like this:

```
...   <Requests Asynch="true">...</Requests>...
```

The default is to handle requests synchronously so that if the attribute is left off or is set to a value of `false`, you set the `EAIRequest` member `Synchronous` to `true`. Otherwise, you set the `Synchronous` member to `false`, meaning that you want the incoming transaction to be processed in an asynchronous fashion.

To get the `Asynch` attribute, you first get the `XmlNode` for the `Requests` block by using the statement `SelectSingleNode("/EAIRequest/Requests")`, exactly as you did earlier for other pieces of information. The next statement, however, is where you create an `XmlAttribute` object. This is accomplished with the following statement:

```
XmlAttribute async = reqs.Attributes["Asynch"];
```

This is another case in which the .NET Framework enables you to specify an array item by name rather than by an index number. This is a very handy little trick that saves many lines of code by hiding the logic necessary to search through the array of items for a particular entry.

All you need to do now to get the value of the attribute is to look at the `InnerText` member, just like you did for `XmlNode` objects. Check to see if the `InnerText` equals `true` for the `Asynch` attribute. If so, set the `rRet.Synchronous` member to `false`, indicating that this transaction should be processed asynchronously. If the value sent in was not `true` for the `Asynch` attribute, set the `rRet.Synchronous` member to `true`. The other possibility for the `Asynch` attribute is that it wasn't sent in at all. The `else` block traps the situation in which the async (`XmlAttribute`) object is null. This means that the attribute wasn't sent in, so you set the `rRet.Synchronous` member to `true`, because synchronous processing is the default.

At this point, you immediately step into another section of code that checks for the `FailOnFirstError` attribute. It is processed in exactly the same manner as the `Async` attribute, but it sets the `rRet.FailOnFirstError` member. This member is used in the `RequestsProcessor` object to see whether to continue processing if a particular request returns something other than `true`.

The `processAsynchRequests()` Method

We end the discussion of the `Controller` class by taking a look at the `processAsynchRequests()` method. Until now, we have essentially ignored the asynchronous processing of transactions. Well, no more!

Asynchronous transaction processing occurs as follows:

1. An `EAIRequest` object is created and sent to the `processAsynchRequests()` method.
2. A `Message` object is created, along with a `MessageQueue` object.
3. The `Message` is sent to the MessageQueue.
4. If no errors are caught, meaning that the message was sent, a success `EAIResponse` object is returned to the caller.
5. Otherwise, an error `EAIResponse` object is returned.
6. Special code monitoring the MessageQueue takes the incoming request object and processes the individual `Request` blocks.

Figure 7-3 shows a sequence diagram of a successful asynchronous transaction flowing through the system. It is very similar to the synchronous transaction diagram in Figure 7-2, but instead of creating and calling an instance of `RequestsProcessor`, it uses a private helper method to send a `Message` object to a MessageQueue.

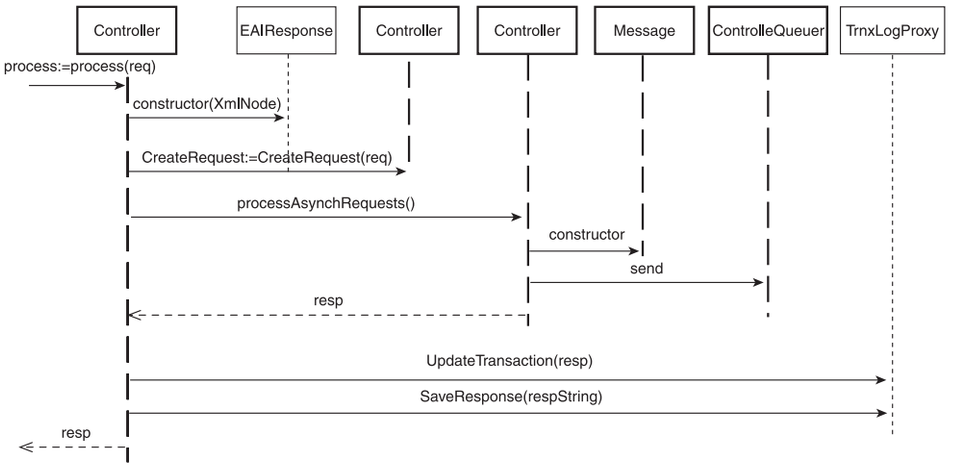

Figure 7-3 Asynchronous request flow sequence diagram.

To keep it all together, we first look at the `processAysnchRequests()` method and then talk about using the message queues. You'll notice that creating a `Message` object and sending it to a message queue is quite easy and straightforward with .NET. You accomplish what previously was fairly complex processing with just a couple lines of code (see Listing 7-5).

Listing 7-5: `processAsynchRequests()` Method

```
/// <summary>
/// This method takes in an EAIRequest object and
///  fires off the request to the RequestsQueueName
///  MSMQ listener.  It will then immediately return
/// </summary>
/// <param name="req"></param>
/// <returns>EAIResponse</returns>
protected EAIResponse processAsynchRequests(
  EAIRequest req)
{
  EAIResponse resp = new EAIResponse();
  resp.SessionID = req.SessionID;
  resp.TransactionID = req.TransactionID;
  ConfigData cdata = new ConfigData();
  string strQueName = cdata.getConfigSetting(
    "RequestsQueueName");

  MessageQueue mq;
  try
  {
    Logger.log(Logger.INFO,
    "Controller: Going to check que: " +
      strQueName);
    if( !MessageQueue.Exists(strQueName)){
```

```
        MessageQueue.Create(strQueName);
      }//end if
    }
    catch(Exception exc)
    {
              Logger.log(Logger.ERROR,
         "Controller: ERROR calling MessageQueue." +
        "Exists(): " + exc.Message);
    }//end catch
    mq = new System.Messaging.MessageQueue(strQueName);
    mq.DefaultPropertiesToSend.Recoverable = true;
    XmlMessageFormatter xmf = (XmlMessageFormatter)mq.Formatter;
    xmf.TargetTypes = new Type[]{typeof(EAIRequest)};
    mq.Formatter = xmf;

    System.Messaging.Message msg = new
      System.Messaging.Message( req );
    msg.Formatter = new XmlMessageFormatter(
        new Type[]{typeof(EAIRequest)});


    try
    {
      mq.Send(msg);
      resp.OverallStatusCode = 1;
      resp.OverallStatus = StatusCodes.Descriptions(
        resp.OverallStatusCode);
      resp.Description = "Requests submitted to " +
        strQueName;
    }//end try
    catch(Exception exc)
    {
      resp.OverallStatusCode = 50;
      resp.OverallStatus = StatusCodes.Descriptions(
        resp.OverallStatusCode);
      resp.Description = exc.Message;
    }//end catch

    return resp;
  }//end processAsynchRequests()


}//end class
}//end namespace
```

Here's how that code breaks down. `processAsynchRequests()` takes in an `EAIRequest` object that was created earlier in the `Controller`. It returns an `EAIResponse` object, so the first thing it does is create an instance of this object and populate a few of the members.

The name of the message queue that will accept transaction requests—that is, the overall incoming request represented by the `EAIRequest` object—is identified in the config file by `RequestsQueueName`. Therefore, you create a `ConfigData` object, get the value for `RequestsQueueName`, and store it in a member called `strQueueName`.

As the first of the real message queue processing steps, you check to see if the particular message queue exists. If it doesn't, it is created. You do the check by calling the static method `Exists()` on the `MessageQueue` class.

The `MessageQueue` and `Message` classes are in the System.Messaging namespace. This namespace is kept in a library that must be added to your Visual Studio .NET project. In the Solution Explorer window in your project, right-click the References item and select Add Reference. On the .NET tab of the dialog box that pops up, scroll down to find the System.Messaging.dll entry. Select this and click Select. You'll see the System.Messaging entry appear in the Selected Components box at the bottom of the dialog box; click OK. You should then be put back in the VS.NET project. System.Messaging should be listed in the References section of the Solution Explorer.

Figure 7-4 shows the Add Reference dialog box, with the .NET library for System.Messaging selected. You can now include the Messaging namespace in your various components in the project with the following line:

```
using System.Messaging;
```

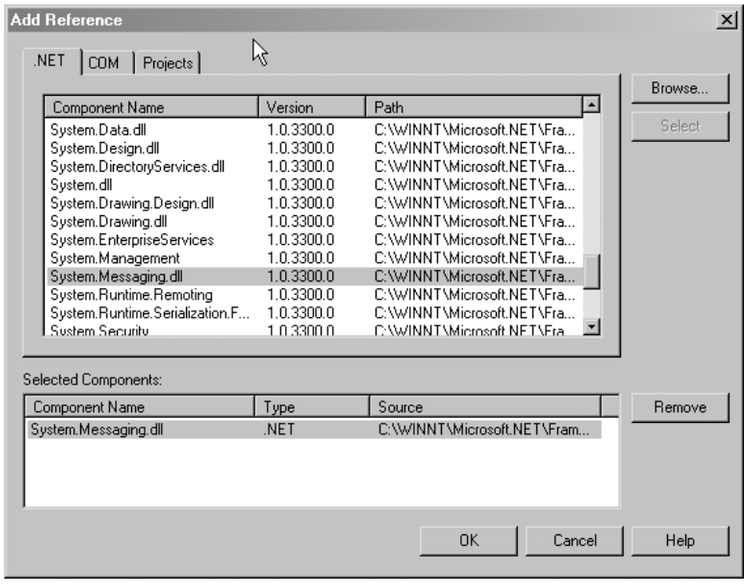

Figure 7-4 Add Reference dialog box.

Now you create a MessageQueue object for the queue and set the `Formatter` to handle our `EAIRequest` object (more on this in a moment). Finally, the message, which contains the `EAIRequest` object, is sent to the message queue for processing. If you don't get an error sending the message, the response object to be sent back to the caller is set with an `OK` status and is returned. If an error was encountered sending the message, you update the response object appropriately in the `catch` block and return the response object. It's

important to keep in mind that the status, either `OK` or an error, is *not* reflective of the status of the request processing, in this case. It merely indicates whether the submission to the message queue was successful. The caller gets back the generated TransactionID in the response, which can be used later to query about the status of the actual processing.

Thus ends the asynch processing—well, okay, the sending of the request. The code to pick up the message and to process the requests is discussed later in this chapter. However, this is another area in which the .NET Framework has made very powerful functionality extremely easy to use. This is great for software developers, because they really don't have much code to write. It's not so great news for book authors hoping to make their code look impressive.

System.Messaging Namespace

This seems like a good spot to take a brief detour to discuss the System.Messaging namespace. All of the classes needed to interact with the Microsoft Message Queuing (MSMQ) system are found here. The MSMQ is message-oriented middleware (MOM) that allows for the robust, asynchronous communication between programs. Classes in this namespace give the functionality to connect to, monitor, administer, send messages to, receive messages from, and peek into messages in a message queue. Clients of message queues need not be on the same physical box as the message queue. When a MessageQueue object is instantiated, an IP can be supplied to indicate where to find the queue. Indeed, the destination machine that is the intended recipient of the message needn't even be running when the message is sent.

A message queue is a construct running on a machine that listens for and accepts messages. Each individual message queue has a specific name and is addressed by that name. That's why in the code shown in Listing 7.5, we created the `MessageQueue` object by passing in the `strQueueName` value to the constructor.

Basically, the message queue is a hopper into which messages are thrown. These messages accumulate, regardless of whether any code is in place to handle the messages. If and when code is ready to handle messages for a particular message queue, the code takes one message at a time and processes the message. In this case, code is in place to process the `EAIRequest` object sent to the queue through a `RequestsProcessor` instance. Messages are taken from the queue in a first in, first out (FIFO) manner, by default. This is accomplished by using one of the `Receive()` methods. `Receive()` is an overloaded method; we examine its use when we look at the code later in this chapter, in the Reading Messages section, to monitor the queue and process the requests.

It is also possible to set a priority on messages as they are sent into the message queue. If a message is sent in with a higher priority, it takes precedence over other messages and is retrieved before those messages that have been in the queue longer but that have a lower priority. Priorities are set on the `Priority` member of a `Message` object. The acceptable values for priorities are enumerated in the static member `System.Messaging.MessagePriority` and have the following names:

- Highest
- Very High
- High
- Above Normal
- Normal
- Low
- Very Low
- Lowest

For example, if you wanted a particular message to have the highest priority possible, you would issue the following command on the `Message` instance before it was sent into the message queue:

```
msg.Priority = System.Messaging.MessagePriority.Highest
```

To take this a bit further, let's say that you support a set of requests that allow an administrator to suspend a certain request type (we'll cover this later). Maybe a particular subsystem is down and you want to hold all transactions until you know that it's back up. In this case, you would want these types of requests to be processed as soon as possible to reduce the number of requests already queued up that will fire (and fail). This is a perfect example of a request type that should get bumped up to a much higher priority.

Message Queue Types

Two main types of message queues exist: system and user-defined. System message queues are used, as you might guess, by the operating system. Various events can be sent to one of the system message queues. User-defined message queues can be of two types: public and private.

Public message queues are available to all systems connected to the network. These queues are published and can be replicated across the network.

Private message queues, on the other hand, are not published across the network, so access from another machine must be made explicitly with the full pathname of the queue. They mainly are intended for use locally on the machine where the queue is running.

For our purposes here, we use private message queues. Because the web service will be running on the same machine as the message queue, by default, it is a simple matter for the `EAIFramework` engine to send a message to a private queue. The code could be easily changed, however, to use a public queue.

Message queues also can be created to be either transactional or nontransactional. By default, they are created as nontransactional. The code shown in Listing 7-5 creates a nontransactional message queue. A transactional message queue expects every interaction with it to be encapsulated in a transaction. The start of a transaction is signaled by a `Begin` message to the message queue. It then expects some messages and, finally, either a commit or rollback message.

If a message queue is created as transactional, all messages sent into it must be a part of a transaction. If a message arrives that is not in a transaction, an exception is thrown. Likewise, if a message queue is created as nontransactional and a message arrives as a part of a transaction, an exception is thrown.

Transactions ensure that all the messages sent in as a part of a transaction are processed successfully. If they are not, a rollback occurs that basically undoes all of the work that was previously done as a part of the transaction in question.

For example, if you wanted to send in two messages and ensure that both were successfully processed, you could build the following structure to send the appropriate messages:

```
MessageQueueTransaction trnx = new MessageQueueTransaction();
MessageQueue myQ = new MessageQueue(@".\private$\secrets");
trnx.Begin();
try
{
  myQ.Send("Message Number 1", trnx);
  myQ.Send("Message Number 2", trnx);
  trnx.Commit();
}
catch
{
  trnx.Abort();
}
finally
{
  myQ.Close();
}
```

By using the `try`/`catch`/`finally` structure, you can, with minimal code, send messages and then either commit the operations or roll them back via the `Abort()` method. When either the work is done or it has been aborted because of some error, the `Close()` method closes the MessageQueue instance.

For the main `Requests` MessageQueue, we are not using a transactional message queue, because each request being sent in is contained in a single message. It would be perfectly reasonable, however, to create and use a transactional message queue for subsequent `Request` processing, in some cases. If you know that a particular set of `Request` types will be coming in and can be processed asynchronously, and if they all need to be cranked through successfully en mass, you could have the main `Requests` message queue processing send individual `Request` blocks to another transactional message queue.

Until now, the transactions described have been internal (to the message queue) transactions, meaning that they are limited only to MSMQ activities. External transactions also exist that can interact with other systems that can also take part in the transaction. These other systems include databases, such as SQL Server and Oracle. External message queue transactions are part of a specific technology called Microsoft Distributed Transaction Coordinator (MS DTC). This technology enables you to treat database calls as integral parts of a transaction. If a database call fails, the DTC transaction can roll back all previous databases, as well as MSMQ steps. Although a discussion of external transactions is beyond the scope of this book, they are a very powerful feature and are worthy of further study as your needs arise.

Recoverable Messages

One final word on types of messages is in order here before we get to sending and receiving. By default, the entire MSMQ message flow is an in-memory operation. This means that a message is created on a client, sent to the destination message queue, and stored until code is ready to accept and process the message, all in memory. Although it is very fast, this method of message delivery holds certain risks. If the sender machine is taken down for some reason before the message is sent to the destination machine, or if the destination machine accepts the message into its message queue and then fails before it can process the message, the message is lost entirely.

This is not normally a good thing. We would like a way to ensure that a message that we send will be delivered even if one of the machines goes down. This could easily be remedied by saving the message to disk while it is in transit and sitting in the queue waiting for processing. And that's just what the smart people at Microsoft did.

For any particular message, you can set the `Recoverable` member of the message to `true`, indicating that you would like this message to be recovered if a catastrophic event befalls one of the machines during the message's merry trip down the pipe. The following line of code sets the `Recoverable` member on a previously instantiated `Message` object named `msg`:

```
msg.Recoverable = true;
```

This approach requires that each individual `Message` object set the `Recoverable` member. You can also set a member on a MessageQueue indicating that all messages should be recoverable. This is the approach we take here. This is accomplished by issuing the following command on a message queue object named `msgQue`:

```
msgQue.DefaultPropertiesToSend.Recoverable = true;
```

Sending a Message

To send a message to a message queue, you create a `System.Messaging.Message` object. This message object contains the contents that you want sent to the queue. When a message is sent to a queue, a subclass of the `System.Messaging.IMessageFormatter` class is specified. The formatter streams the incoming object (`EAIRequest`, in this case) into the `Message` object. Then on the other side, the code that retrieves the message from the message queue also uses a formatter object to stream the contents of the `Message` object, housed in the `Body` member, into an instance of whatever object was originally streamed into the message.

As you can see in Figure 7-5, a client creates and sends a `Message` object to the message queue. These messages are accepted and stored in the `MessageQueue`. When a process is available, the messages are sent to the registered monitor code for processing.

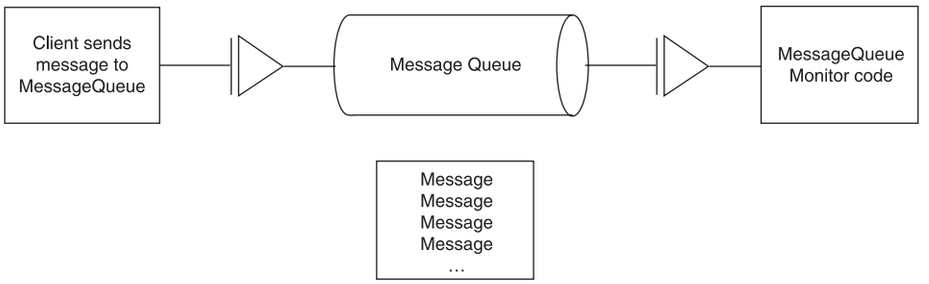

Figure 7-5 Overall message queue flow.

We make use of this service here to handle asynchronous request processing. Certain requests sent into the `EAIFramework` either will not need to be handled on the spot or might take far too long to process for a web service to wait for (keeping the HTTP connection open to the caller so that it can send an HTTP response). These are ideal candidates for asynchronous processing. Because a unique identifier is generated when the initial request is sent in, you can send that ID back to the caller and then not worry about when the actual processing is done. The monitor code attached to the message queue takes care of updating the transaction log when a transaction is processed.

Examining Message Queues

You can see what message queues are running and see what messages are waiting in the queue, as well as manage queues, by using the Computer Management tool. To access it, right-click My Computer on the Windows 2000 desktop, and click Manage. You'll see the Computer Management dialog box pop up. From this screen, you can manage most of the software component configurations on the machine.

On the left side of the screen, expand the Services and Applications item, if it isn't already expanded. You should see several entries under it, including Microsoft SQL Servers and Services. Expand the item named Message Queuing. This is your interface to access the MSMQ service. Figure 7-6 shows the Computer Management dialog box with an arrow pointing to the Message Queuing item.

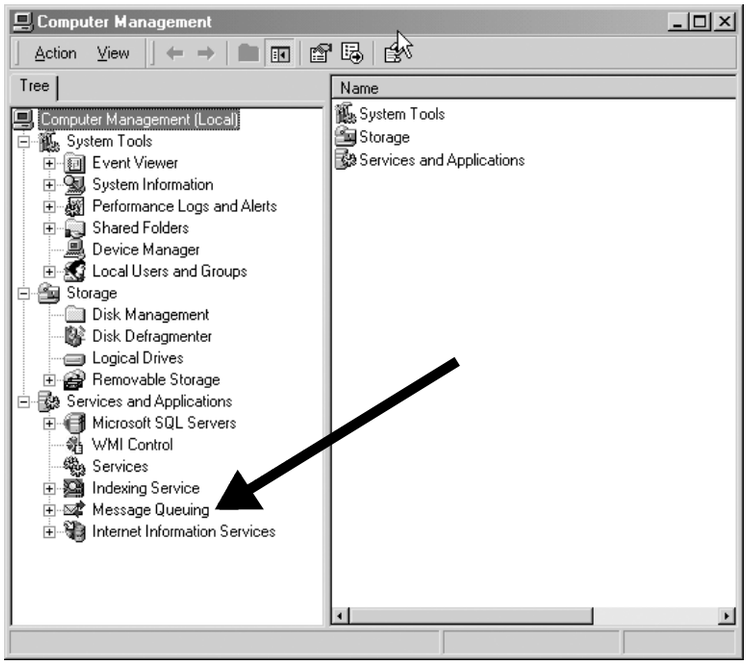

Figure 7-6 Computer Management screen.

Now expand the Private Queues item. An example of this screen is shown in Figure 7-7. This shows you all private queues running on the machine.

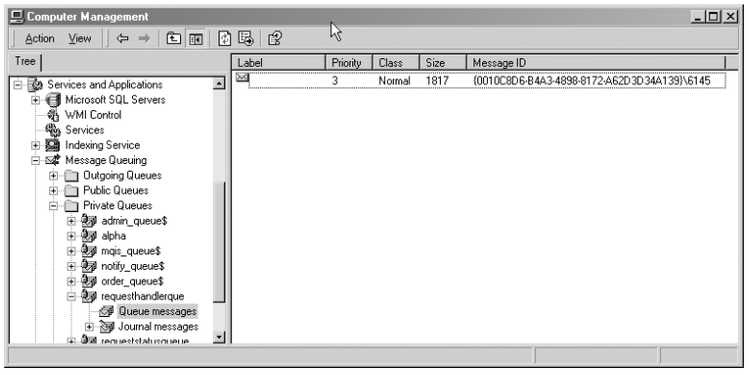

Figure 7-7 MSMQ private queue.

If you select one of these queues, you'll see two entries appear on the right side of the screen. Queue Messages is the item in which any currently queued messages are stored. You can double-click any queued message and examine the body, as well as its other properties. Figure 7-8 shows the message dialog box, with the Body tab active.

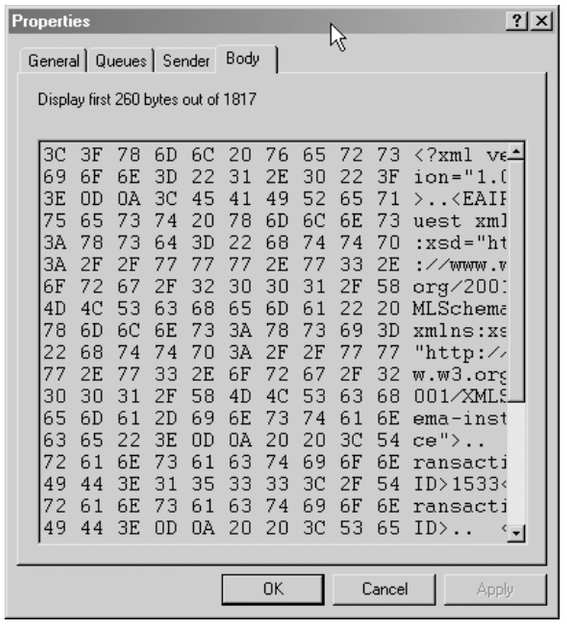

Figure 7-8 Examining the body of a message.

One good way to test that your code is actually submitting messages to the queue is to not have any queue-monitoring code running so that nothing is performing a `Retrieve()` on the queue. When your client-side code submits messages to the queue, they stay in the queue. You can then use this tool to ensure that the messages are indeed being queued up in the correct message queue. You can also open the messages and examine their contents.

Formatters

For messages to flow smoothly in and out of message queues, meaning that both the sender and receiver understand what the other is trying to say, you use a subclass of the `IMessageFormatter` class. This class is used to serialize the object in the `Body` member of the `Message` object.

Three formatters are supplied with the .NET Framework:

- `ActiveXMessageFormatter`
- `BinaryMessageFormatter`
- `XmlMessageFormatter`

The default formatter is the `XmlMessageFormatter`, which suits our needs. For information on the `ActiveXMessageFormatter` or the `BinaryMessageFormatter`, you can explore MSDN.

The `XmlMessageFormatter` serializes and deserializes messages. It streams in and out of the `Body` member of a message. Therefore, you can insert any object into the `Body` of a message with the `XmlMessageFormatter`. You cannot, however, also use a default `XmlMessageFormatter` to `Retrieve()` a message. To accomplish this, you must either set the Formatter on the message that you want to get back and explicitly tell the message what object to expect back, or set the Formatter on the MessageQueue. This is a cleaner way of handling retrieving so that you have to specify only what kind of object(s) will be returned once on the queue. After that, the appropriate conversion takes place and the returned message contains the correct type of object.

If you want to set the formatter on a `Message` object to specify that you expect back an `EAIRequest` object at retrieval time, you would execute the following statement:

```
msg.Formatter = new XmlMessageFormatter(
    new Type[]{typeof(EAIRequest)});
```

If instead you want to set the default Formatter for the MessageQueue to be an `XmlMessageFormatter` that uses the `EAIRequest` object, you would execute the following statements:

```
mq = new System.Messaging.MessageQueue(strQueName);
XmlMessageFormatter xmf = (XmlMessageFormatter)mq.Formatter;
xmf.TargetTypes = new Type[]{typeof(EAIRequest)};
mq.Formatter = xmf;
```

When the message queue has been set up in this way, it renders the previous formatter setting unnecessary. Now, when a message is retrieved from the queue, it uses an `XmlMessageFormatter` object that speaks `EAIRequest`.

Reading Messages

Now that you have created a `MessageQueue` appropriately, sent a message to it, and verified that the message arrived and is being held in the correct queue, it's time to read it in. This is accomplished by connecting to the message queue in exactly the same manner as explained earlier, and then calling the `Receive()` method of the `MessageQueue` instance. This overloaded method pulls in the first message, based upon arrival time in the queue and its priority, and returns it to the caller. It also removes the message from the queue. It is then up to the calling code to process the message and take all necessary actions.

A second way to read a message is nondestructive. As with the `Receive()` method, the `Peek()` method returns a `Message` object for a message but does not remove the message from the message queue. Because we want to process the incoming requests once, we call the `Receive()` method so that it removes the request from the message queue.

The simplest version of the `Receive()` method takes no arguments. When the call is issued, it blocks until a message is available to be returned from the message queue. This means that if a message never appears, the call to `Receive()` never returns. In most cases, this is not the behavior you want. Fortunately, another version enables you to supply a `TimeSpan` object that indicates how long the `Receive()` method should wait for a message. This call can throw one of two exceptions. It can throw `ArgumentException` if the `TimeSpan` argument sent in is invalid. In this case, the `TimeSpan` must be greater than zero.

It also can throw a `MessageQueueException`. This occurs when either there was a problem accessing the message queue, or the specified amount of time passed without seeing a message to return. Therefore, you can create a processing loop that can do some kind of work or check to see if it should exit in a `while` loop. For example, a processing `Receive()` loop could look something like the following:

```
while( bKeepTrying )
{
  try
  {
    msg=mq.Receive(new System.TimeSpan(0,0,5));
    EAIFramework.Messages.EAIRequest req =
      (EAIFramework.Messages.EAIRequest)msg.Body;
    // do some processing here...
  }
  catch(MessageQueueException mqe)
  {
    // This is the message that the Receive() timeout
    //  throws.  In this case, no message is ready
    //  to be read within 5 seconds of issuing the
    //  Receive() call.
    Console.WriteLine("Just caught a MsgQueExc: " +
      mqe.Message);
  }//end catch
}//end while()
```

In fact, this is exactly the way we will watch the `Requests` message queue in the RequestQueMonitor project, which we will discuss in the next section. This project contains a Windows Form application with a few controls to make it easy to watch for messages and to process them. We shall discuss the most interesting aspects of the code here, and I encourage you to take a look at the application.

RequestQueMonitor Project

The RequestQueMonitor application is a standalone Windows Form application that monitors the `Requests` queue to which the `Controller` class sends an asynchronous message. We want the asynchronous processing for the `EAIFramework` to happen in a Windows service so that it takes advantage of the benefits of service features such as restarts after a crash, and so on. For now, though, the RequestQueMonitor will serve that purpose. The nice thing about running it as a Windows application is that you can watch the processing flow happen. With the RequestQueMonitor running, you can submit an

asynchronous `EAIRequest` XML message to the web service and see a message in the text box of the application as the message is picked up from the queue and processed. Later, you will convert this monitor to a Windows service.

A main text box in the application shows status information. Figure 7-9 shows the application as it exists just after launch.

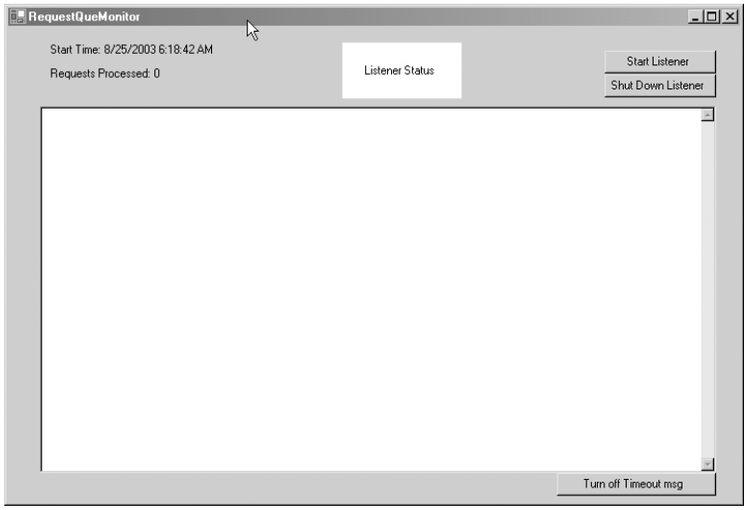

Figure 7-9 RequestQueMonitor screen.

When the application starts, no monitoring is occurring. To start watching the `Request` message queue, click the Start Listener button. You will see a few things happen on the screen. First, a message is printed on the top-left side of the screen telling what time the monitor started watching the queue. Below that, a line is printed showing how many messages have been read off the queue. At the top center of the screen, you should see the white box turn green, with the message "Listener Status" printed in the center. Green signifies that the monitor has successfully connected to the message queue and has entered into the `while` loop, waiting patiently for messages to appear. Finally, you will see a couple of messages in the status text box. The first tells you that the monitor has connected to the message queue. It prints the name of the message queue, which was pulled from a `ConfigData` instance, just like in the `Controller` instance earlier.

The code enters a `while` loop and makes a call to the `Receive()` method. It gives a new `TimeSpan` object of 5 seconds, which means that the code will block for 5 seconds, waiting for a message to become available. If a message arrives within the time specified, it is read in, and you can process it in whatever way you need to. On the other hand, if the time period expires, the message queue object throws a `MessageQueueException`. In the `catch` block, a message is printed telling you that the timeout for the requested operation has expired.

Figure 7-10 shows the RequestQueMonitor screen after having read a message from the queue, processed three `Request` blocks within that message, and then listened for a couple of timeout cycles without having another message to read and process.

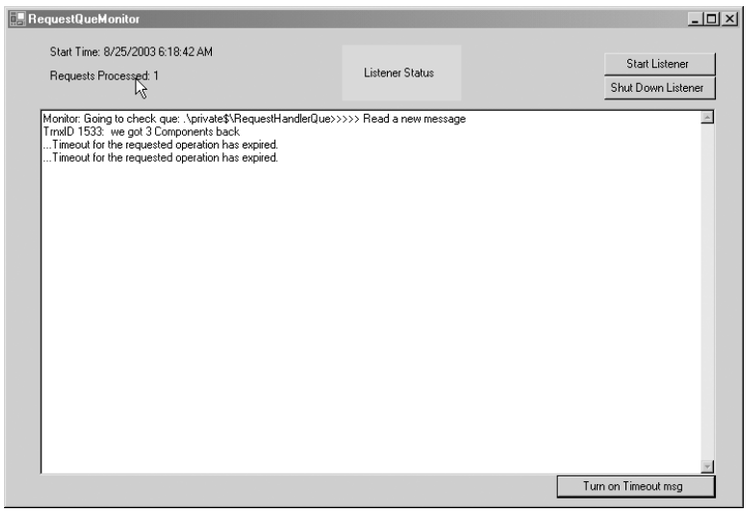

Figure 7-10 RequestQueMonitor processing a transaction.

That's really good information, for a while. A button on the bottom-right side of the screen enables you to toggle this timeout message on and off. When the application starts, a Boolean flag is set to print the timeout message. As soon as you click the toggle button, the Boolean flag is set to `false` and the messages are no longer printed.

The only component on the screen that I haven't talked about yet is the Shut Down Listener button. As you might have surmised, clicking this stops the message queue monitoring. It works by setting a class-level Boolean member that is used in the main receive `while` loop. As you will see in a moment, if the flag is set appropriately, the `while` loop exits and the connection to the message queue is closed. When you click the Shut Down Listener button, you might notice a several-second lag. This is because the shutdown doesn't take effect until the current time `TimeSpan` expires.

RequestQueMonitor `startListening()` Method

Now let's take a look at the `startListening()` method (see Listing 7-6). The loop for reading messages from the queue lives here.

Listing 7-6: The `startListening()` Method

```
/// <summary>
/// Helper method, called when the Start Listening
///  button is clicked.  It gets the name of the
///  Requests message queue, opens a connection, and

///  then starts a while loop to wait for messages.
///  Currently, the timeout for any specific waiting
///  period is 5 seconds, set in a TimeSpan instance.
/// To break out of the while loop and stop Listening,
///  a class member, named bShutdown, is set in a button
```

```
///  click event handler elsewhere.  When the user
///  presses the Shut Down Listener button, the flag
///  is set to true.  The next time the while loop
///  fires, it sees that the Boolean is true and exits
///  the read (while) loop.
/// </summary>
private void startListening()
{
  ConfigData cdata = new ConfigData();

  string strQueName = cdata.getConfigSetting(
    "RequestsQueueName");

  MessageQueue mq;
  RequestsProcessor rProc;

  // See if we need to create the queue
  try
  {
    textBox1.Text =
      "Monitor: Going to check que: " +
      strQueName;
    if( !MessageQueue.Exists(strQueName)){
      MessageQueue.Create(strQueName);
    }//end if
  }
  catch(Exception exc)
  {
            textBox1.Text =
       "Monitor: ERROR calling MessageQueue." +
      "Exists(): " + exc.Message;
  }//end catch

  // Create a MessageQueue instance, set it as
  //  Recoverable - to make the messages stored to
  //  disk during their trip, and set the Formatter
  //  to XmlMessageFormatter that understands the
  //  EAIRequest object.
  mq = new System.Messaging.MessageQueue(strQueName);
  mq.DefaultPropertiesToSend.Recoverable = true;
  XmlMessageFormatter xmf =
    (XmlMessageFormatter)mq.Formatter;
  xmf.TargetTypes = new Type[]{typeof(EAIRequest)};
  mq.Formatter = xmf;

  // Now that we're listening, change the background
  //  color of the Listener Status box to Green, to
  //  graphically indicate such.
  lblStatus.BackColor = Color.LightGreen;

  System.Messaging.Message msg;

  while( !RequestQueMonitor.bShutDown )
  {
    try{
```

```
        msg=mq.Receive(new System.TimeSpan(0,0,5));

        textBox1.AppendText(
          ">>>>> Read a new message\r\n");
        this.updateCount(++nCount);

                  EAIFramework.Messages.EAIRequest req =
          (EAIFramework.Messages.EAIRequest)
          msg.Body;

        rProc = new RequestsProcessor(req);
        ArrayList al = rProc.process();

        textBox1.AppendText( "TrnxID " +
          req.TransactionID + ": " +
          " we got " + al.Count +
          " Components back\r\n");
      } catch (MessageQueueException mqe) {
        // Only print out this message to the text
        //  box if the user wants to see it
        // - This Boolean is controlled by the

        //  event handler method for the
        //  Turn on/off Timeout msg button.
        if( bShowTimeout)textBox1.AppendText
           ("..." + mqe.Message + "\r\n");
      } catch (Exception exc) {
        textBox1.AppendText(
          "..." + exc.Message + "\r\n");
      }//end catch()
    }//end while()

    mq.Close();
    // Now that we're not listening anymore, change
    //  the background of the Listener Status box
    //  to pink.
    lblStatus.BackColor = Color.Pink;
  }//end startListening()
```

The method starts by getting the name of the `RequestsQueueName` from the `EAIConfig` file, just like in the engine, and stores the value in the member `strQueName`. This keeps the submitter code and the listener code in synch. If you want to change the name of the queue, you simply make the change in one place and (in this case) restart the RequestQueMonitor application.

The method next checks to see if the message queue exists. If so, it just drops through and continues processing. If it doesn't exist, it creates a new message queue, named after the value of `strQueName`. This is accomplished by a single statement:

```
MessageQueue.Create(strQueName);
```

Next, the `MessageQueue` instance is created. As discussed earlier, you set the `Recoverable` property of the message queue to `true` to force MSMQ to write each request to disk as it flows through the system. In this way, it is much less likely that messages will be lost if a power failure or other catastrophic problem occurs.

Finally, the Formatter for the message queue is set. You'll notice that we set the `TargetTypes` of the Formatter to type `typeof(EAIRequest)`. This tells the message queue that any messages read back in from the queue should contain `EAIRequest` objects within the `Body` member of the messages. Again, it is a way of setting default processing behavior on the message queue, and it eliminates the need for explicitly setting the Formatter type for every `Message` object created for this message queue.

With the connection to the message queue started, change the background property of the status box, in the top middle of the screen, to `LightGreen`. I personally like being able to glance at the screen and see the status of the listener. `green` means go; `red` (well, okay, `Pink`) means stop.

You now enter the main `while()` loop for reading messages from the queue. The class-level Boolean member is named `bShutDown`. When the program starts, this member is given the value of `false`. When the processing hits this `while` loop each time, it checks the value of `bShutDown`. In the event handler method for the Shut Down Listener button, the value of `bShutDown` is set to `true`. When this happens, the `while` loop exits.

The first statement in the `while()` loop is a call to the `Receive()` method on the `MessageQueue` instance. This returns a `Message` object if a message is available. Notice that you send in a new `TimeSpan` object with a value of 5 seconds. If a message is read, you print a message to the text box, update a counter, and pull in the `EAIRequest` object that should be contained in the `Body` member of the `Message` instance. This `EAIRequest` object gets sent to a `RequestsProcessor` instance, just the same as when the `Controller` calls `RequestsProcessor` for synchronous request processing. In this case, you're not really doing anything with the response yet. That will come later. For now, you just want to make sure that you're reading in the messages, converting the contents to an `EAIRequest` object, and sending it on for processing.

If there is no message to be returned within the specified 5 seconds, a `MessageQueueException` is thrown. In a way similar to `bShutDown`, the `bShowTimeout` member indicates whether to display the timeout messages. This Boolean member is maintained in the event handler for the Turn on(off) Timeout msg button.

When the `bShutDown` member is set to `true`, it tells you that the user wants to quit listening. All you need to do here is close the `MessageQueue` to free up the resources held by it, and change the background of the status box to `Pink`. The application is then ready for the user to either exit or press the Start Listening button again.

`RequestsProcessor` Discussion

We finish this chapter with a discussion of the `RequestsProcessor` class. As we look at `RequestsProcessor`, we jump briefly to a helper class that it calls `RequestHandlerFactory`, because it has some interesting and powerful code. As mentioned earlier, the `RequestsProcessor` class actually kicks off the processing efforts for each `Request` block in an incoming transaction. If all goes well, each `Request` block, identified by a separate `<Request> . . . </Request>` element in the incoming XML message, is processed by an appropriate `RequestHandler`, and an `EAIResponse` message is generated for return.

If one or more request blocks fail, however, `RequestsProcessor` needs to call the `rollback()` method on the `RequestHandler` objects. Any specific `RequestHandler` does not need to do any real rollback processing, but all `RequestHandler` components do need to implement (override) the abstract `RequestHandlerBase.rollback()` method.

The `FailOnFirstError` attribute, which can decorate the `<Requests>` XML element in the `EAIRequest` message, tells the `RequestsProcessor` whether to continue processing when the first `Request` processing returns an error. If `FailOnFirstError` is `true`, the overall processing is halted at that point and the `rollback()` method is called on the offending request by calling its `RequestHandler` object. Figure 7-11 shows an activity diagram detailing the important actions that take place, mostly in the `RequestsProcessor` class, for a synchronously processed transaction.

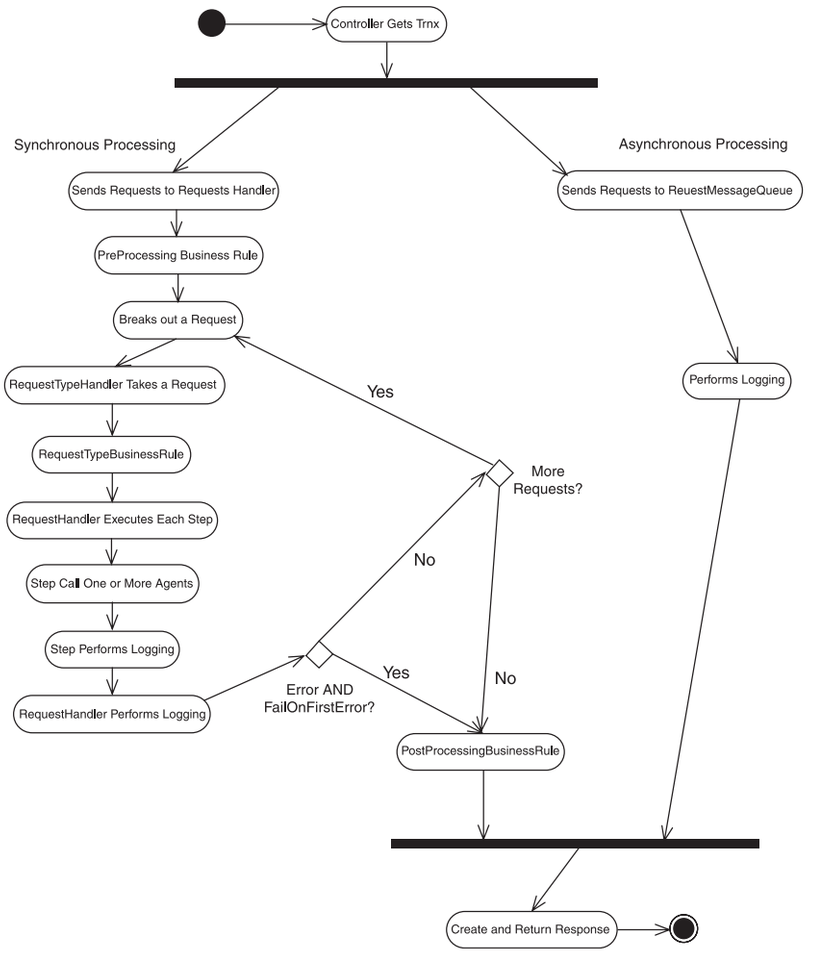

Figure 7-11 Synchronous processing activity diagram.

One bit of sorcery called from the `RequestsProcessor` class is really fun. The name of the request handler class, which is subclassed from the `RequestHandlerBase` class, either is explicitly defined in the database table T_RequestDefinitions or is derived from the name of the `Request` sent in. This name is identified in the `Name` attribute of each `<Request>` XML element. For example, a request to catalogue the MP3 files on a machine could be called like so:

```
<EAIRequest>
  <SessionID>987-cba-321-zyx</SessionID>
  <Requests>
    <Request Name="Catalogue">
       <OutputFilename>MyMP3Files.txt</OutputFilename>
    </Request>
  </Requests>
</EAIRequest>
```

Alright, so this might not solve a burning industry-wide need, but I wanted to compile a list of MP3 files on my server. Sure, I *could* have used the Search utility that comes with Windows 2003, but what fun would that be? Being the geek that I am, I instead wrote a `RequestHandler` to perform this function. When the server is connected to the Internet, I can get a list of files from anywhere in the world. Now, when I'm in a dark alley in Rome, and a thug says "Gimme a list of all yer MP3 files on yer server, or else," I can fire off an `EAIRequest` message and display the results. (Remind me to always have my wireless device in all dark alleys from now on.)

The name of the `RequestHandler` for the `Request` named `Catalogue` would be `CatalogueHandler`, in the EAIFramework.Handler namespace. `RequestsProcessor` generates the name, creates an instance of the class on the fly, and finally calls the `process` method. This is all accomplished through classes in the System.Reflection namespace. In the next section, you will see how you can use these classes to instantiate an appropriate handler class on the fly, given just the name of the handler class.

System.Reflection Namespace

The System.Reflection namespace contains a large number of utility classes that are very handy in certain situations. They enable you to interact with and manipulate types, assemblies, and so on. Major classes in this namespace include `Assembly`, `Module`, `ConstructorInfo`, `MemberInfo`, and `MethodInfo`.

With the tools available in System.Reflection, you now can plug in new functionality and allow for new `<Request Name="xxx">` blocks simply by writing a subclass of `EAIFramework.Handler.RequestHandlerBase` and making it accessible to the `EAIFramework` engine. The framework figures out what class should be handling the request, attempts to create an instance of it, and then calls its `process` method.

RequestHandlerFactory.cs

Before we look at the entire `RequestsProcessor` class, let's just pull out the code that creates the instance of the handler class (see Listing 7-7). It is fairly small and performs quite a bit of functionality in these few lines. It has been placed in a factory class named RequestHandlerFactory.cs, and it is in the EAIFramework.Handler namespace.

Listing 7-7: RequestHandlerFactory.cs

```
using System;
using System.Xml;
using System.Runtime.Remoting;
using System.Reflection;

using EAIFramework.Messages;
using EAIFramework.Util;

namespace EAIFramework.Handler
{
  /// <summary>
  /// RequestHandlerFactory class follows a factory pattern
  ///  and has a single static public method: getHandler(Req).
  /// It also has a private constructor.
  /// </summary>
  public class RequestHandlerFactory
  {
    private RequestHandlerFactory(){}

    /// <summary>
    /// The only public method in this class, getHandler,

    ///  takes in a Request object.  This method then
    ///  checks to see what type of request it is, checks to
    ///  see what the handler name is (from the database),
    ///  and instantiates and returns the new object.
    /// If it cannot instantiate an appropriately named
    ///  object, it will return null.
    /// </summary>
    /// <param name="rIn">Request object to be processed
    ///   </param>
    /// <returns>Subclassed object of RequestHandlerBase
    ///   </returns>
    public static
      EAIFramework.Handler.RequestHandlerBase
        getHandler(Request rIn)
    {
      // First, get the Handler name for this request
      EAIFramework.Util.DBUtilProxy tdp = new
        DBUtilProxy();
      string strHandlerName= tdp.getHandlerName(rIn.Name);
      // *** If we don't have a specific Handler Name

      //  from the database, use the Request.Name
      if( (strHandlerName == null) ||
```

```
        (strHandlerName.Equals(""")))
        strHandlerName = rIn.Name;

      // Now get an ObjectHandle for the appropriate type
      //  of RequestHandler object
      ObjectHandle oh = Activator.CreateInstance(
        "EAIUtilities",
        "EAIFramework.Handler." +
        strHandlerName + "Handler");
      Logger.log(Logger.INFO,
        "New Handler obj = " + oh.ToString());

      // Next, get a Type object
      Type ht = oh.Unwrap().GetType();
      Logger.log(Logger.INFO,
         "New Type from Handler = " +
        ht.Name + "-" + ht.ToString());

      // Finally, get a ConstructorInfo object for this
      //  Type.  With this, we will Invoke the CI and

      //  get a 'ready-to-run' object that we can return
      ConstructorInfo ci = ht.GetConstructor(new Type[]{
        new EAIFramework.Messages.Request().GetType()});
      Logger.log(Logger.INFO,
        "ConstructorInfo () returned: " +
        ci.ToString());

      // Now all we have to do is invoke the
      //  ConstructorInfo object to get the handler
      EAIFramework.Handler.RequestHandlerBase rhb =
        (EAIFramework.Handler.RequestHandlerBase)
          ci.Invoke(new object[]{rIn});

//      ------------------------------------------------
//         For testing, uncomment these lines, and the
//        newly created RequestHandlerBase (or
//        descendent class) will process the Request.
//        This method can be called from a standalone
//        test program to test a specific Handler or
//        to ensure that the above code works as expected.

//
//      EAIFramework.Messages.Component cBack =
//        rhb.process();
//      Logger.log(Logger.INFO,
//        "Component returned = " + cBack.Name + ", " +
//        cBack.Status + ", " + cBack.ToString());
//      ------------------------------------------------

      return rhb;
    }//end getHandler()
  }//end class
}//end namespace
```

RequestHandlerFactory has a single public, static method named getHandler(). It takes in a Request object and returns an instance of the Handler class that it determines should be handling this type of request, or it returns null. Because all handlers should be subclassed from the RequestHandlerBase class, the abstract type RequestHandlerBase is returned. All concrete subclasses have process and rollback() methods that are called by the RequestsProcessor class during actual execution.

We include the System.Runtime.Remoting namespace to get access to the ObjectHandle class, used in the getHandler() method. Also included is the System.Reflection namespace for access to the ConstructorInfo class.

As the method is entered, it instantiates the DBUtilProxy class, used to interact with some of the support tables in the database. In this case, we want to see if a specific handler name is associated with the request name. The table T_RequestDefinitions has four columns:

- RequestName
- Description
- Status
- HandlerName

You might be asking why we have this table: We have already said that you get the name of the handler from the name of the request, right? Well, you might want to have several different requests handled by the same handler. For example, in this instance, we want to provide the following kinds of requests for EAIFramework reporting and statuses:

- ListAllRequests (used to get a list of supported request types)
- TransactionStatus (used to get the status/results of a given TransactionID)
- ListToDo (used to get all requests sitting in the ToDo database table)
- ListUsers (used to get a list of all authorized EAIFramework users)

This is just a brief list of the many possible support request types that the EAIFramework could provide to its users and administrators. Because each will have a different request name, each would have to have a separate RequestHandler component. However, each of the requests will be quite small in terms of processing, and it would be unnecessary overhead for each one to have its own handler. Instead, the T_RequestDefinitions table tells you what RequestHandlerBase subclass to use. In this way, you can process more than one request type in a single RequestHandler. For example, the rows in the T_RequestDefinitions table for the preceding scenario might look something like Table 7-2.

Table 7-2 Requests supported by Admin request handler

RequestName	Description	Status	HandlerName
`ListAllRequests`	List supported requests	`A`	`Admin`
`TransactionStatus`	Return saved status info for `TrnxID`	`A`	`Admin`
`ListToDo`	Snapshot of current ToDo requests	`A`	`Admin`
`ListUsers`	Return list of all `EAIFramework` users	`A`	`Admin`

You can see that if any of the four `Request` blocks comes in for processing, they all will be handled with a handler named `Admin`. This handler name, as described earlier, is used to generate the full handler name. In this case, all four requests would generate the full handler name:

```
EAIFramework.Handler.AdminHandler
```

Then, in the `process` method of the `AdminHandler` class, a quick check of the request name is made to determine which functionality to perform.

After you have checked the database for a handler name for the supplied request name, a check is made to see if the database call returned null or a blank string. If either of these were returned, it means that the request name supplied does not exist in the table. If that's the case, just use the Request Name as the handler name. Essentially, the name of a request is the default handler name; you have to insert rows in the T_RequestDefinitions table only if more than one Request Name will be supported by a handler. For example, if the `<Request Name="Catalogue">` request mentioned earlier is the only request handled by the `CatalogueHandler` class, it doesn't need to be in the database, because `Catalogue` will be used to generate the name of the handler on the fly.

Next, you get an `ObjectHandle` object for the handler class. The static method `CreateInstance()` of the `Activator` class is used for this purpose. You pass in the name of an assembly and the name of the class to be instantiated. In this case, you build the class name by using the `strHandlerName` member that should contain either the request name or the handler name returned from the database for this request and a constant. This is done with the following code:

```
strHandlerName + "Handler"
```

You use the `ObjectHandle` instance for the required handler to get a `Type` object. This is accomplished by calling the `Unwrap()` method on the `ObjectHandle` returned earlier and immediately calling the `GetType()` method. You are left with a `Type` object that represents the handler class you need.

Finally, you create a `ConstructorInfo` object from the System.Reflection namespace. This instance is used to create the actual handler instance that gets returned to the caller. The constructor for the `ConstructorInfo` class takes in an array of `Type` objects. This is used when the constructor for the object in question is called. In this case, the constructor for a `RequestHandler` subclass takes in a single parameter, a `Request` object. Therefore, the `ConstructorInfo` object takes in a `Type` array that contains a single `Type`, the `EAIFramework.Messages.Request` type.

With the newly created `ConstructorInfo` instance for the handler you need, you are ready to fire up the constructor and get an instance of the handler class. A call to `Invoke()` on the `ConstructorInfo` class accomplishes this. The last line of real code in the `getHandler()` method creates the instance:

```
EAIFramework.Handler.RequestHandlerBase rhb =
        (EAIFramework.Handler.RequestHandlerBase)
           ci.Invoke(new object[]{rIn});
```

The member `rhb` is returned to the caller. The `Request` object, sent into the `getHandler()` method, is sent into the constructor in the `Invoke()` call. `Invoke()` takes an array of `Object` objects. In this way, you could call any constructor with any signature by setting the appropriate input parameter list in the object array.

`RequestsProcessor` Code

Now we look at the `RequestsProcessor` class (see Listing 7-8). I'm going to blow past some of the helper methods, because they are really self-explanatory. However, I do want to spend some time on the `process` method. This is called from the `Controller` class when a transaction is sent in that needs to be processed synchronously. It is also called from the `RequestQueMonitor` class that watches the request message queue to process asynchronous transactions.

Listing 7-8: The `RequestsProcessor` Class

```
using System;
using System.Collections;

using EAIFramework.Util;
using EAIFramework.Messages;
using EAIFramework.Handler;
using EAIFramework.BusinessRules;

namespace EAIFramework.Controller
{
  /// <summary>
  /// Summary description for RequestsProcessor.
  /// </summary>
  public class RequestsProcessor
  {
    protected EAIFramework.Messages.EAIRequest eaiReq = null;
    protected Requests reqObjects = new Requests ();

    /// <summary>
    /// Default constructor, takes only a single argument,
    ///   An EAIRequest.
    /// </summary>
    /// <param name="rIn"></param>
    public RequestsProcessor(
      EAIFramework.Messages.EAIRequest rIn)
    {
      eaiReq = rIn;
      reqObjects = eaiReq.Requests;
    }//end constructor()
```

There is nothing remarkable about the first portion of the class. It has a single constructor that takes an `EAIRequest` as its only parameter, which it stores in an instance member. Now on to the `process` method (see Listing 7-9).

Listing 7-9: The `process` Method

```
/// <summary>
/// This method adds each Request to the RequestsToDo
///  table and then processes each Request.

///  When it completes, each Request should remove
///  its own entry from the RequestsToDo table.
/// </summary>
    public ArrayList process()
{
  // Set up the ArrayList we'll return to the caller
  ArrayList alRet = new ArrayList();

  // Get the list of Request objects
  reqObjects = eaiReq.Requests;

  // Set up a place to store handlers
  ArrayList alHandlers = new ArrayList();

  // Before anything gets added to the ToDo list,
  //  see if this set of Requests passes
  //  PreProcBusinessRules
  PreProcBusinessRules preBR =
       new PreProcBusinessRules();
  EAIFramework.Messages.BusinessRuleResponse brr =
    preBR.Check(eaiReq);
  if( brr.StatusCode != StatusCodes.OK )
  {
    // This is an error.  There was something wrong
    //  with the PreProcessing Business Rule checks!
    // Generate response and return w/o processing.

    Logger.log(Logger.ERROR,
      "PreProcBusinessRules.Check Failed: " +
      brr.Description);

    RequestResponse rBack = new RequestResponse();
    rBack.StatusCode =
      StatusCodes.FAILED_PREPROCBUSINESSRULES;
    rBack.Status=
      StatusCodes.Descriptions(rBack.StatusCode);
    rBack.Description =
      "Failed PreProcBusinessRules checking: " +
      brr.Description;
    rBack.Name = "PreProcBusinessRules";
    alRet.Add(rBack);
    return alRet;
  }//end if PreProcBusRules.Check()
```

```
//Otherwise...
Logger.log(Logger.INFO,
  "PreProcBusinessRules.Check Passed");

// Add to the RequestsToDo list
this.addToToDo(reqObjects);

// Now process each Request object
foreach(Request req in reqObjects.RequestList)
{
  RequestResponse rBack = null;
  Logger.log(Logger.INFO,
    "Processing Request # " +
    req.Iteration);
  try{
    // Here we check to see if the user is
    //  authorized to execute this type of
    //  request.
    Authenticator auth = new Authenticator();
    EAIUser theUser =
          auth.GetUserFromSessionID(
                    eaiReq.SessionID);
    if(theUser.Group.ToLower().Equals("admin"))
    {
      // this is okay, they're an ADMIN
    }
    else
    {
      // See if they have rights to process
      //  this type of Request now.
      // If not, send them on their way w/o
      //  processing this Request
      if( ! auth.UserCanExecuteRequest(
        theUser.Username,
        req.Name))
      {
         // If we're here, then the user is
         //  NOT an ADMIN, *AND* they're not
         //  authorized for this Request type.
         // Therefore, create an error and
         //  continue to the next one.
        rBack = new RequestResponse();
        rBack.StatusCode =
    StatusCodes.USER_NOT_AUTHORIZED_FOR_REQUEST;
        rBack.Status=
        StatusCodes.Descriptions(
          rBack.StatusCode);
        rBack.Description =
  "User: " + theUser.Username +
  " attempting Unauthorized Request Type: " +
  req.Name;
        rBack.Name = req.Name;
      alRet.Add(rBack);
      continue;
      }//end if authed == false
```

```
  }//end else (they're NOT an ADMIN)


   RequestHandlerBase hand =
     RequestHandlerFactory.getHandler(req);
   if(hand == null) {
    rBack = new RequestResponse();
    rBack.StatusCode =
      StatusCodes.UNKNOWN_HANDLER;
    rBack.Status=StatusCodes.Descriptions
      (rBack.StatusCode);
    rBack.Description =
      "Cannot find Handler: " +
      " EAIFramework.Handler." +
      req.Name + "Handler";
     rBack.Name = req.Name;
   }//end if handler == null
   else {
     // The Handler is not null, so add it
     // to the list, and then call
    // the process() method to do the work
      alHandlers.Add(hand);
    rBack = hand.process();
   }//end else
} catch (Exception exc) {
  // Need to build a failed RequestResponse
  //  here to return good info.
  rBack = new RequestResponse();
  rBack.StatusCode =
    StatusCodes.UNKNOWN_HANDLER;
  rBack.Status=
   StatusCodes.Descriptions(rBack.StatusCode);
  rBack.Description =
    "Cannot find Handler: " +
    "EAIFramework.Handler." +
    req.Name + "Handler - " + exc.Message;
  rBack.Name = req.Name;
}//end catch()
alRet.Add(rBack);

// If there is an error AND we're supposed to

//   FailOnFirstError, then break out and don't
//   process any more Request objects.
if( (rBack.StatusCode != StatusCodes.OK ) &&
  eaiReq.FailOnFirstError)
{
  // However, b/f we go, we need to remove all
  //  the Requests that are currently in the
  //  T_RequestToDo table for this TrnxID
  this.removeAllFromToDo(
    req.TrnxID.ToString());

  // Now call roll back on any of the Requests
```

```
      //  that have already been fired
      for(int x = 0; x < req.Iteration; x++)
      {
        Request rRoll = (Request)
          reqObjects.RequestList[x];
        RequestHandlerBase hRoll =
          (RequestHandlerBase) alHandlers[x];
        RequestResponse rrRoll = hRoll.rollback();
        alRet.Add(rrRoll);
      }//end for
      break;
    }

    // Otherwise, just remove this processed Request
    //  from the T_RequestsToDo table
    this.removeFromToDo(req);
  }//end foreach

  // Now, right before we leave,
  //  see if this set of Requests passes
  //  PostProcBusinessRules
  PostProcBusinessRules postBR =
       new PostProcBusinessRules();
  brr = postBR.Check(reqObjects);

  if( brr.StatusCode != StatusCodes.OK )
  {
    // This is an error.  There was something wrong
    //  with the PreProcessing Business Rule checks!
    // Generate response and return w/o processing.

    Logger.log(Logger.ERROR,
      "PostProcBusinessRules.Check Failed: " +
      brr.Description);

    RequestResponse rBack = new RequestResponse();
    rBack.StatusCode =
      StatusCodes.FAILED_POSTPROCBUSINESSRULES;
    rBack.Status=
      StatusCodes.Descriptions(rBack.StatusCode);
    rBack.Description =
      "Failed PostProcBusinessRules checking: " +
      brr.Description;
    rBack.Name = "PostProcBusinessRules";
    alRet.Add(rBack);
    return alRet;
  }//end if PostProcBusRules.Check()

  //Otherwise...
  Logger.log(Logger.INFO,
    "PostProcBusinessRules.Check Passed");

  return alRet;
}//end process()
```

`process` returns an `ArrayList` instance filled with `RequestResponse` objects. These are the `RequestResponse` instances sent back from each called `RequestHandler` instance. The method starts by creating the `ArrayList` that will be returned. It then pulls out the `Requests` object from the `EAIRequest` object and creates an `ArrayList` to hold the `RequestHandler` instances to be used for each `Request`. You create and store a list of handler objects, because it might be necessary to call the `rollback()` method on each one.

Next, all of the requests in the `Requests` object are added to the ToDo table in the database. This is accomplished by calling the helper method `addToToDo()`. It makes the code a little cleaner in this method to stick the call to the `ToDo` proxy class in a helper method.

A `foreach` loop then blasts through each `Request` contained in the `Requests` object. The handler object, cast as a `RequestHandlerBase` object, is created by calling the `RequestHandlerFactory.getHandler()` method, discussed earlier. Providing that the handler is found, instantiated, and returned, you can send off the request for processing. If all is well, the handler is added to the `ArrayList` of the handler, and a new `RequestResponse` object is created by calling the `process` method of the handler. If there is an error along the way for this `Request`, an error `RequestResponse` instance is created. You drop out of the code for this `Request` and add the `RequestResponse`, good or bad, to the `ArrayList` to be returned, named `alRet`.

Next, if the status was unsuccessful for any reason and this transaction is set for `FailOnFirstError`, you delete all requests for this transaction from the ToDo table, because you're bailing on this one. Then each handler that has already been called is cycled through, and the `rollback()` method is called. A `RequestResponse` object is returned from the `rollback()` method, just as the `process` method does. These status objects are added to the `alRet` list along with those returned from `process`.

Otherwise, you remove the current `Request` from the ToDo table, whether the response was successful or unsuccessful. When the `foreach` loop completes, either because of running through each `Request` processed or because of a failure and `FailOnFirstError` being set to `true`, the compiled `ArrayList` of `RequestResponse` objects is returned to the caller (see Listing 7-10).

Listing 7-10: `addToToDo()` Method

```
/// <summary>
/// Helper method to insert all Request objects into the
/// RequestToDo table.
/// </summary>
/// <param name="rsIn"></param>
private void addToToDo( Requests rsIn )
{
  DBUtilProxy tdprox = new DBUtilProxy();
  foreach( Request req in rsIn.RequestList)
  {
    tdprox.addRequest(req);
  }//end foreach
}//end addToToDo()
```

```
    private void removeAllFromToDo(string strReq)
    {
      DBUtilProxy dbup = new DBUtilProxy();
      int nRow = dbup.removeToDoRequests(strReq);
    }//end removeAllFromToDo()

    private void removeFromToDo(Request req)
    {
      DBUtilProxy dbup = new DBUtilProxy();
      int nRows = dbup.removeRequest(req);
    }//end removeFromToDo()

  }//end class
}//end namespace
```

As you can see in Listing 7-10, the method `addToToDo()` takes in a `Requests` object. It instantiates the `DBUtilProxy` class, which is used to interact with some of the utility tables. In this case, it inserts rows into the T_RequestsToDo table. This table is used to hold any `Request` steps that need to be processed. Because all the processing is happening in memory, if the box takes a nosedive for some reason, you would lose all traces of submitted but as yet unprocessed requests. Therefore, the first thing you must do is add each step to a database table. Now, if the cleaning crew decides to unplug your server so that it can use the floor polisher, you can query the ToDo table and continue processing (more or less) as if nothing had happened.

The `removeAllFromToDo()` method is called if the request processing is being aborted. You already know that you are not going to continue processing any subsequent requests, so just remove all of the requests in the T_RequestsToDo table.

`removeFromToDo()` performs essentially the opposite function as the `addToToDo()` method. It is called when an individual request has been processed. When that happens, you can safely remove this request from the T_RequestsToDo table so that it's not in danger of being reprocessed later.

Summary

We've covered quite a bit of ground in this chapter. We started by looking at XML attributes in the `EAIRequest` message to signify whether the client wants the transaction processed synchronously or asynchronously. The other attribute signified whether you should abort processing when you hit the first error while processing a `Request` type.

Next, we examined the code for the `Controller` class. The `Controller` takes a request from the web service and creates an `EAIRequest` object from the incoming XML message. This object is then passed around the system to various subcomponents.

The `Controller` checks what type of processing has been requested. If the client specifically requests `Asynch="TRUE"`, you send the entire request on to a Message Queue in the MSMQ service. All of the classes needed to interact with the MSMQ are housed in the System.Messaging namespace. A response message is sent back to the caller with the TransactionID that was created for this request. Because processing takes place asynchro-

nously, the caller must send a message later to get the status of the processing for this transaction.

The RequestQueMonitor project, a Windows application, uses the classes in the System.Messaging namespace to watch the appropriate message queue for the integration messages. When it finds one, it sends the message to an instance of the `RequestsProcessor` class. Status information is printed to a text box on the screen to let you know what's going on.

If the request is synchronous, the client expects processing to happen now. The HTTP connection is held open to the web service while you work on the back end. When you send back the response, it is sent back to the calling client.

Next we described `RequestsProcessor`, the class that takes the input request and actually pieces out the work to various appropriate handler classes. In the case of synchronous requests, it is called from `Controller`. In the case of asynchronous requests, it is called by Message Queue monitoring code. One of the main reasons for approaching the architecture this way is so that you have a single spot where code can be changed if you want to modify the behavior of the system. When a change is made, both synchronous and asynchronous paths call the `RequestsProcessor` class.

What's Coming Up

In the next chapter, we cover the `RequestHandler` set of classes. You saw an instance of these classes created in the `RequestsProcessor` class. A subclass of the `RequestHandlerBase` will be invoked to process each `Request` block sent into the system. As shown earlier in this chapter, classes in the System.Reflection namespace are used to create an instance of the needed subclass on the fly. This is powerful (and cool) code that lets you quickly add functionality to your system without having to change any of the code on the calling, engine side.