Chapter 1

Solution: Receiving 850 Data

This chapter will walk through a complete end-to-end solution on how to build out BizTalk to receive 850 (Purchase Order) documents from an external trading partner and send an acknowledgement back. The data will be received via an SFTP adapter and then it will be archived and processed by an orchestration. In the orchestration, it will be determined whether the purchase order needs to be reviewed manually by an internal user prior to approval, or whether it can be delivered automatically as a flat file to the internal order processing application. This will introduce many of the key concepts required in working with inbound data.

The data will be received in unencrypted format on an SFTP site, will be archived by a BizTalk orchestration using a .NET library, and will be mapped to a flat file format. The rules determining whether the data can be automatically approved, or whether it required manual approval, will be handled in the orchestration, but additional ideas on how to make this more robust will be presented at the end of this chapter. The overview of this specific solution is shown in Figure 1-1.

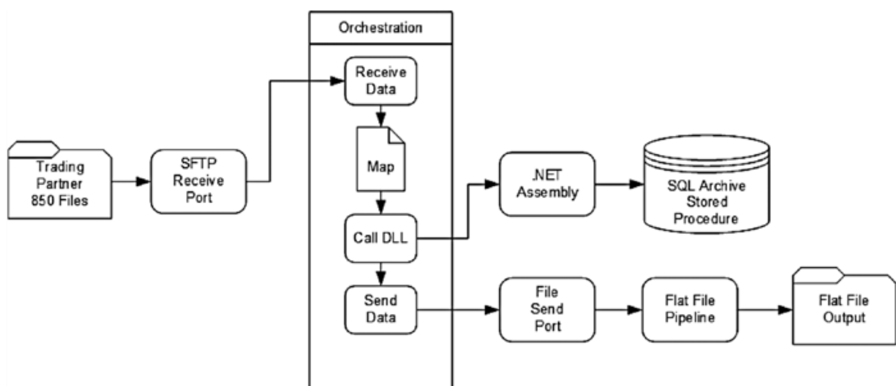

Figure 1-1. Inbound 850 solution overview

Visual Studio Solution

It is critical that your project structure and namespaces are correct from the start. If these are not exactly what you need for the proper architecture and organization of your solution, you'll be spending a great deal of time later in the process rewriting and retesting. For this solution, the namespace is in a structure that you should be able to use directly within your own solutions, substituting the wording, but not the structure. You will also be creating a Visual Studio project structure that will be generic enough to fit within any model you may encounter. In the case of the solution being built out, the following Visual Studio projects and namespaces will be used:

- **Solution Name:** Demo.BizTalk. This is a generic solution that can hold inbound and outbound projects. You may have several projects that are common to many projects, so having everything in one solution can be very helpful. Examples of such projects would be common schemas and .NET helper classes.
- **Schemas:** There are three schemas that will be used in this solution. All three of these will be contained in a single project called Demo.BizTalk.Schemas.X850.
  - The 850 Schema ships with BizTalk. All of the out-of-the-box schemas are contained in the `MicrosoftEdiXSDTemplates.exe` file found in `C:\Program Files (x86)\Microsoft BizTalk Server 2013\XSD_Schema\EDI`.
  - The Target flat file schema, which represents the target data to which the 850 is being mapped. This must be created using the flat file wizard.

> **Note** When setting a namespace, never use a numeric value alone without at least one leading text character (such as the 850 in `Demo.BizTalk.Schemas.850`), as it could result in a variety of potential naming conflicts, unexpected errors, and challenges in testing. If you wish to refer to an EDI document type directly in your namespace, use a pattern such as a leading "X", like X850.

- **Maps:** The map project will contain all maps required by the solution, and will have a namespace of Demo.BizTalk.Maps.X850.
- **Helper Library:** There will be one external .NET assembly project with the namespace of Demo.BizTalk.Helper.X850.
- **Orchestration:** There will be one orchestration used, which will be in its own project called Demo.BizTalk.Orchestrations.X850.
- **Pipeline:** There will be one custom Send pipeline project, which will be called Demo.BizTalk.Inbound.Pipelines.X850.

The Schema Project

There will be two schemas required for this project. The first is the 850 schema that ships with BizTalk. BizTalk has thousands of EDI schemas that come with it, crossing all of the document types and versions available. The second is the Target flat file that you'll be mapping the 850 to, which represents the format that the internal order processing application requires. An example of the source 850 schema is show in Figure 1-2.

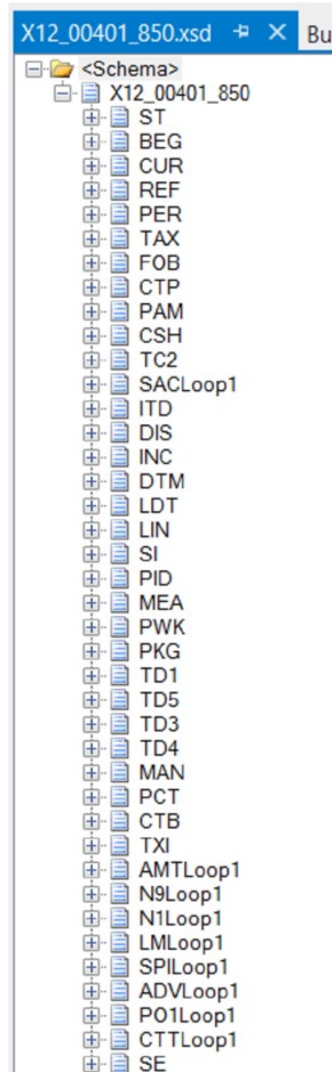

Figure 1-2. The 850 Schema

> **Note** To access the EDI schemas with BizTalk, browse to the Microsoft BizTalk Server 2010 root folder and go to `XSD_Schema\EDI`. In this directory you will find a file called `MicrosoftEdiXSDTemplates.exe`. Running this file will extract all available schemas.

The schema project is the foundation of many of the other projects you will be creating, as these other projects reference them. If the schemas change during the course of development, all of the other projects will be impacted. Do everything you can to get the schemas namespaced and laid out correctly at the start of your development in order to minimize the impact to projects (especially maps) that reference these schemas.

The Map Project

The map project will allow for the mapping of the inbound 850 data into the target flat file format. The 850 data can be mapped into the target format using whatever mapping techniques are required—standard functoids, lookups, even XSLT. Chapter 4 is completely dedicated to mapping, and describes how best to approach this task. For this exercise, the map project structure should be as follows:

- Create a new project in Visual Studio called Demo.BizTalk.Maps.X850.
- Add a reference to the schema project you have created.

The .NET Helper Library Project

The .NET helper library is used by the orchestration to archive the inbound XML version of the EDI file to a database in its native XML format. This is an invaluable way of being able to access and report on data through SQL Business Intelligence (BI) platforms, such as SSRS, without having to push the 850 data to a tradition database model. The .NET class will have a single method in it that looks similar to the code shown in Listing 1-1. You can pass as many or as few parameters as you would like, depending on the needs of your reporting.

> **Note** Always mark your .NET classes as Serializable, so that they can be called from anywhere within BizTalk. To do this, type [Serializable] directly above the class declaration in your helper library.

Listing 1-1. Method Called from Orchestration to Archive Data to SQL

```
public void ArchiveInboundData(string
strSourceFileName, string
strTradingPartner, XmlDocument xmlSource, string
strApprovalStatus, string strConnectionString)
{
    SqlConnection sqlConnection = new
SqlConnection(strConnectionString);
    SqlCommand sqlCommand = sqlConnection.CreateCommand();
    sqlCommand.CommandText = "spInsertInboundData";
    sqlCommand.CommandType = CommandType.StoredProcedure;
    sqlConnection.Open();

    SqlParameter sqlParameter = new SqlParameter();

    sqlParameter.ParameterName = "@vchSourceFileName";
    sqlParameter.SqlDbType = SqlDbType.VarChar;
    sqlParameter.Direction = ParameterDirection.Input;
    sqlParameter.Value = strSourceFileName;
    sqlCommand.Parameters.Add(sqlParameter);

    sqlParameter = new SqlParameter();
    sqlParameter.ParameterName = "@vchTradingPartner";
    sqlParameter.SqlDbType = SqlDbType.VarChar;
    sqlParameter.Direction = ParameterDirection.Input;
    sqlParameter.Value = strTradingPartner;
    sqlCommand.Parameters.Add(sqlParameter);

    sqlParameter = new SqlParameter();
    sqlParameter.ParameterName = "@xmlSourceData";
    sqlParameter.SqlDbType = SqlDbType.Xml;
    sqlParameter.Direction = ParameterDirection.Input;
    sqlParameter.Value = new XmlNodeReader(xmlSource);
    sqlCommand.Parameters.Add(sqlParameter);

    sqlParameter = new SqlParameter();
    sqlParameter.ParameterName = "@vchApprovalStatus";
    sqlParameter.SqlDbType = SqlDbType.VarChar;
    sqlParameter.Direction = ParameterDirection.Input;
    sqlParameter.Value = strApprovalStatus;
    sqlCommand.Parameters.Add(sqlParameter);

    sqlCommand.ExecuteNonQuery();
    sqlConnection.Close();
}
```

The stored procedure called from this method is shown in Listing 1-2. It simply takes the data passed to it and inserts it into a table. Once the data in in the table, it can be queried using standard T-SQL and XQuery.

Listing 1-2. Stored Procedure to Archive XML Data

```
CREATE PROCEDURE [spInsertInboundData]
     @vchSourceFileName As varchar(500)
     ,@vchTradingPartner As varchar(50)
     ,@xmlSourceData As xml
     ,@vchApprovalStatus As varchar(50)
AS
BEGIN

     SET NOCOUNT ON;
     INSERT tblInboundData
     (
          vchSourceFileName
          ,vchTradingPartner
          ,xmlSourceData
          ,vchApprovalStatus
          ,dtmCreateDate
     )
     VALUES
     (
          @vchSourceFileName
          ,@vchTradingPartner
          ,@xmlSourceData
          ,@vchApprovalStatus
          ,getdate()
     )
END
```

Once you have the .NET helper library built, you can reference the DLL in the orchestration project and call it to archive the inbound XML data. The next section shows how to call this referenced DLL from within an orchestration.

The Orchestration Project

In this solution, the orchestration will receive the 850 directly, map it, validate whether it can be automatically sent through or whether it must be manually reviewed, archive it to the database, and send it out in the final target format to a file directory. If archiving and the check on whether it can be automatically processed or not were not requirements, all of this could be accomplished without the use of an orchestration; the mapping could occur directly on either the receive port

or send port. For this solution, however, the orchestration is used, and an example of it is shown in Figure 1-3. In order for this orchestration to work, you must add a reference to the .NET Helper DLL, the schema project, and the map project, all created earlier in this chapter.

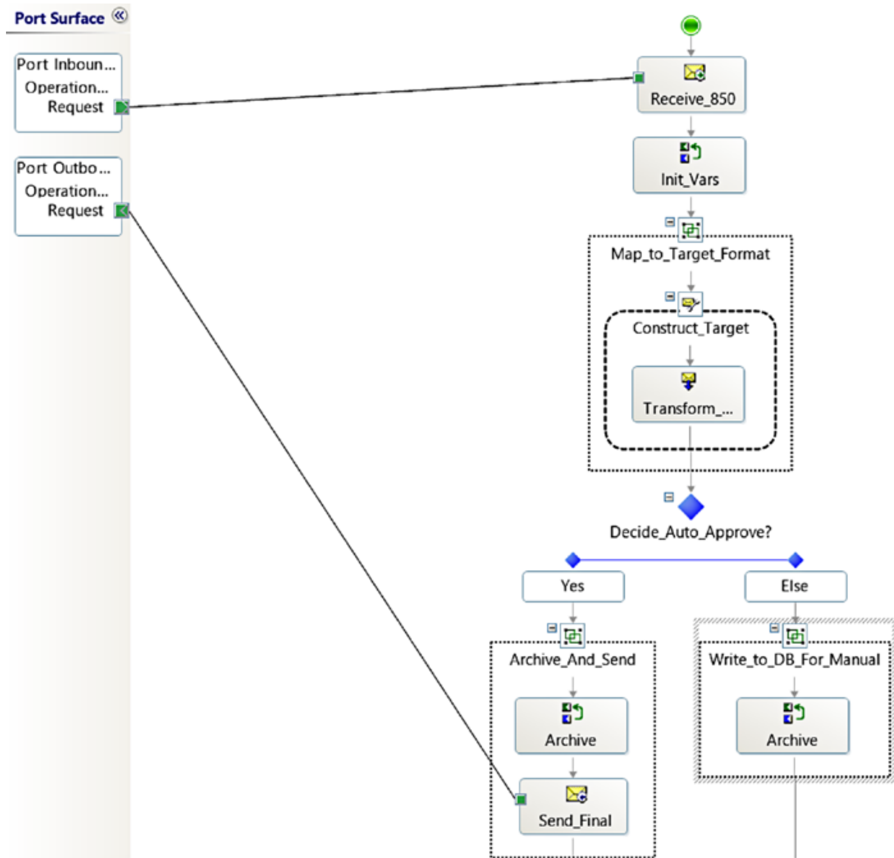

Figure 1-3. The orchestration

Details behind each of the shapes in this orchestration must be given, as several of them are Expression shapes and have important code behind them. There is no restriction as to how you populate your Expression shapes, and no requirements as to naming standards. In the case of this orchestration, all of the Expression shapes could be merged into one, but keeping them separate will allow you to see how you could add additional message types to this orchestration and be able to reuse common code.

The Receive_850 Shape

This is a simple receive shape that must be set to an orchestration message type of the 850 schema (you'll need to add a reference to this schema project from your orchestration project). Call this message msg850. The Activate property on this shape must be set to True.

The Init_Vars Shape

This shape sets variables that are specific to the document just received, which are immediately available to the orchestration (as opposed to the Init_Vars shape, which gets its data from a configuration file). The code behind this shape sets several of the fields that are passed as parameters to archive the data. The code, with notes, is shown in Listing 1-3.

Listing 1-3. Init_Vars Expression Shape Code

```
// these fields are set here so that they can be easily
written to the database
strType = "850 Inbound";
strTradingPartnerID = msg850(EDI.ISA06);
strReceivedFileName = msg850(FILE.ReceivedFileName);
```

This shape is an excellent place to read from the BizTalk configuration file. There are often variables that are best kept configurable; in this case, the configurable field is the database connection string used to connect to the SQL Server database where the data will be archived. Adding a key/value pair to your configuration file allows for rapid access and alteration of this key. Using the BizTalk configuration file can be done as follows:

- Browse to the root BizTalk Server folder.
- Open `BTSNTSvc.exe.config` in a plain text editor.
- You can add new configurable fields to the <appSettings> node of this document. An example of storing a connection string would be

```
<add key="Demo.BizTalk.Archiving.ConnectionString"
value="Data Source=BTSSQLSERVER;Initial
Catalog=Archiving;Integrated Security=SSPI;" />
```

- Save the modified config file and restart the BizTalk host instance. The field can now be referenced from an Expression shape using the code shown in Listing 1-4.

Listing 1-4. Init_Vars Expression Shape Code

```
strConnectionString = System.Configuration.
ConfigurationSettings.AppSettings["Demo.BizTalk.
Archiving.ConnectionString"];
```

The Map_to_Target_Format Shape

This is the map that converts the inbound 850 into the target file format. It consists of a Construct Message shape and a Transform shape. The Transform shape references the map that was created earlier in this project.

The Decide_Auto_Approval Decision Shape

This step does the check as to whether the data can be sent straight through to the order processing system, or whether it needs to be written to a database for manual review and approval. The left-hand branch of this shape contains actions that will map it from the 850 format to the target format, archive the data, and then send the final flat file on to the final location. The right-hand branch of the shape has the steps for mapping it to the target schema and writing this XML to the database where it can be manually reviewed.

The code within the Decision shape that determines whether the data can be automatically approved or not is based on the value of BEG02 (the Purchase Order Type Code). This can be done in a variety of ways. Some fields can be promoted as distinguished fields, and accessed directly through code. Those fields that cannot be promoted (such as repeating elements or elements of certain types) can be accessed using xpath. For this demo, the decide shape's code is based on the BEG02 field being distinguished, and is `msg850.BEG.BEG02 != "AB"`. This means that anything that is not AB in the BEG02 field will allow the document to be automatically approved and sent on. Anything that has the AB value in the BEG02 fields will go down the right-hand branch of the Decision shape and will have to be manually approved. For reference, Figure 1-4 shows some of the enumerations that are available for BEG02.

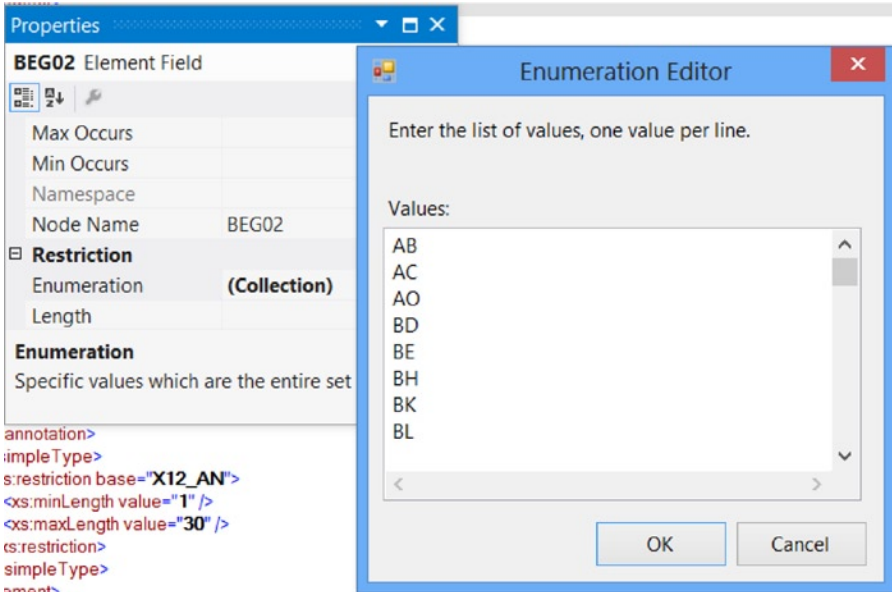

Figure 1-4. The BEG02 enumerations

The Archive Shape

This shape calls the code in the .NET library to do the actual archiving, passing in several fields as parameters. These parameters include the source file name, the trading partner, and the data to be archived. The data being archived in this case is the XML version of the 850 data, readily available to the orchestration in the inbound message. The code for archiving is shown in Listing 1-5. Notice that the inbound message (which is the orchestration message of type Demo.BizTalk.Schemas.X850 received on the Receive shape) can be converted to XML and sent straight in as a parameter.

Listing 1-5. Archive Shape Code

```
objHelper.ArchiveInboundData(strReceivedFileName,
strTradingPartnerID, (System.Xml.XmlDocument)msg850,
"Approved",strConnectionString);
```

Note that the fourth parameter is “Approved” in the left-hand block of the Decision shape, and “Needs Manual Approval” (or similar value) in the right-hand block of the Decision shape.

The Send_Final Shape

Only the left-hand block of the decision code sends an actual flat file out. This Send shape is tied to a send port, which in turn is bound to a file send or other physical port. The output is the final flat file. In the case of the right-hand block of the Decision shape, the writing of the data to the database ends the orchestration.

The Pipeline Project

The final project required is a simple custom flat file pipeline using the Flat File Assembler component that ships with BizTalk. The use of this pipeline on a send port will allow the outbound target document to be output in flat file format. The steps to create this pipeline are as follows, and the pipeline is shown in Figure 1-5.

- Create a new project called Demo.BizTalk.Inbound.Pipelines.X850.
- Add a new send pipeline to this project.
- In the pipeline GUI interface, drop a Flat File Assembler component on the Assemble stage.

Figure 1-5. The flat file pipeline and Assembler component

Setting up the BizTalk Components

With all of the Visual Studio projects completed, you will have a number of BizTalk components to deploy and configure. The first step will be to deploy the DLLs that are created when you compile your Visual Studio projects. This can be done in a variety of ways, but the one that will allow you the most control (and is the quickest) is as follows. This can be used for adding new DLLs and for updating existing DLLs. After you deploy, always restart the BizTalk host instance.

- Open BizTalk Administration Console and browse to the application where your code will be deployed. If one doesn't exist, create one called "Demo.BizTalk."
- Right-click the application and select Add and then BizTalk Assemblies.
- In the window that opens, click the Add button and add all of the assemblies for this solution: two schema DLLs, one map DLL, one orchestration DLL, and the .NET helper DLL.
- Click the Overwrite all checkbox.
- Click each DLL and make sure the first and third checkbox in the Option window is selected for each one. You must do this for every DLL that you are adding. Figure 1-6 shows the checkboxes being set (the first and third also need to be checked for the .NET assembly, though it will have several additional boxes).

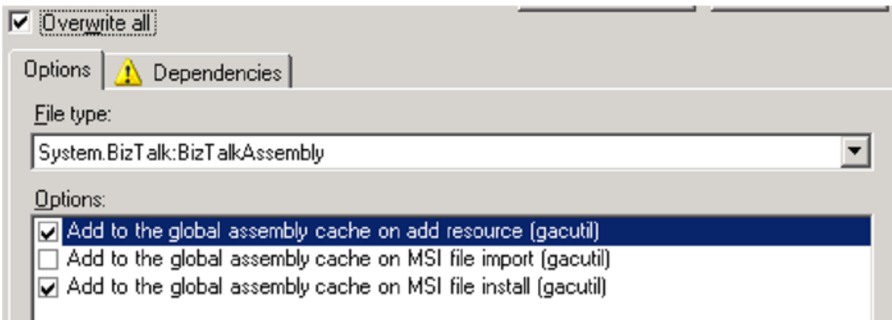

Figure 1-6. The options to select when deploying assemblies

Note It can be helpful to have all of your Visual Studio projects write their compiled DLLs to the same directory, as you will be deploying them to BizTalk frequently for testing. This can be done by going into the project properties for each project and setting the Output Path of the build to a common directory, like a custom folder called Binaries.

Party Settings and Agreements

The steps to configure the BizTalk Party settings that contain all of the information about how the data will be validated and what the 850 envelope settings should be are as follows:

- Create a new BizTalk Party that represents your home organization. For now, this will be called Company. All you need to set is the name.
- Create a new BizTalk Party for your trading partner; this will be named Trading Partner for this solution. You'll need to set one Party up for every trading partner you will be doing business with. Only the name needs to be set.
- Create one Agreement that represents the exchange of information between your trading partner and your company. Right-click the Company Party and create a new Agreement. Set the Protocol type to X12; the first party will automatically be set to Company, and the second party should be set to your trading partner party. The moment this is done, two additional tabs will appear within the Agreement. One tab is for inbound data from the trading partner to your company, while the other is for outbound data from your company to the trading partner.
- On the tab representing inbound data from the trading partner to your company, click the Identifiers, Envelopes, and Character set tabs and enter the appropriate information as required by the 850 envelope. You can easily access this either in a sample instance coming from the trading partner (just open in notepad and match the values) or by referencing the trading partner implementation guide (if you have one).
- On the Validation tab, set the Transaction Type property to 850 – Purchase Order.
- On the Envelopes tab, set the following values:
  - Transaction Type should be 850 – Purchase Order.
  - Version/Release should be 00401 (or 00501 if you are using the 5010 version)
  - Target namespace should be `http://schemas.microsoft.com/BizTalk/EDI/X12/2006`
  - GS1 should be PO-Purchase Order (850)
  - GS2 through GS7 should be set based on what your trading partner requires (again, see the trading partner implementation guide or a sample 850 instance from them)
  - GS8 should be 004010 (for 4010)

Note The GS8 property value can be tricky to get right. If you are batching, or you have a trading partner that requires a certain value here, you may find that you have to create a custom pipeline to override this value right before it is sent to the trading partner.

The Receive Port

The inbound data will be received on an SFTP receive port. The details for configuring this port (with and without decrypted data) are given in Chapter 5. What is important to know here is that the port must not only be created, but must be bound to the orchestration. The steps for creating and binding this port are as follows:

- Create a receive port called Company.BizTalk.Receive.X850.TradingPartner. This allows for a pattern that will support multiple trading partners, if needed.
- Add a receive location to this port and call it the same thing as the receive port. Make it of type SFTP and configure the appropriate settings for the SFTP connection. Since unencrypted 850 data will be received here, set the receive pipeline to EdiReceive.

The File Send Port

The send port will be used to send the final ECSIF flat file that was mapped in the orchestration. The send port should be of type File, and should have the Send Pipeline property set to the Demo.BizTalk.Pipelines.X850.Outbound pipeline created earlier. This send port should be called Demo.BizTalk.Send.TargetData.

Orchestration Binding

The orchestration is set to pick up a file of type 850, map it to the proprietary target file, archive it, and then send out the target flat file document. Once it has been deployed, it needs to be bound to the appropriate receive and send ports. Take the following steps to bind the orchestration to the ports that were just created:

- In the BizTalk Administration Console, in the application where you are working, click the Orchestrations folder and open the Demo.BizTalk.Orchestrations.X850.Inbound orchestration you have built and deployed.
- Click the Bindings tab and set the Host property to the BizTalk Server Application host that is available.

- Set the Receive Port property to the receive port you created called Demo.BizTalk.Receive.X850.Receive.
- Set the Send Port property to the send port you created called Demo.BizTalk.Send.TargetData.
- Click OK to save these settings.

Enabling and Running the Solution

At this point, all of the components have been deployed and set up. All you need to do is enable your receive location for the inbound SFTP, start your send port for the outbound flat file, and enable your orchestration. All of this can be done through the BizTalk Administration Console. Restart the BizTalk host instance so that all of the most current settings and assemblies are loaded into memory.

Extending this Solution

There are a number of things you might want to do to make this solution more robust. The most obvious are how to handle the data that has been written to for manual approval, and improvements around determining what would cause the data to be automatically approved or not (instead of just looking at the BEG02 field). The following sections look at both improvements.

Extending this Solution with an Approval Web Site

In this demo solution, the final step of the orchestration for data that must be manually reviewed and approved is to write the data to the database with a "Needs Manual Approval" status. Now that it is in the database, something needs to be done to allow a user to view it, modify the data, and approve it. One solution is to have an ASP.NET application that shows all of the unapproved data in a grid, and loads the EDI data into human readable columns that allow for editing. The code in Listing 1-6 shows the columns of the grid being populated with data from the corresponding fields in the 850 XML, while Figure 1-7 illustrates the data grid with approval check box. This XML has been loaded directly from the database in XML format, and can be parsed for specific data.

Listing 1-6. Populating the ASP.NET Grid

```
public void InitializeDataSet(DataTableDemo dtDemo,
DataSet dsRecords, bool blnAddColumns)
{
 // Create data columns
 dtDemo.AddTextColumn("strID");
 dtDemo.AddTextColumn("vchTargetID");
 dtDemo.AddTextColumn("vchPartnerName");
 dtDemo.AddTextColumn("vchLocation");
 dtDemo.AddTextColumn("vchPONumber");
 dtDemo.AddTextColumn("vchDate");
 dtDemo.AddTextColumn("vchErrorDescription");

 dtDemo.Rows.Clear();

 // Populate rows with data
 if (dsRecords != null)
 {
  if (dsRecords.Tables.Count > 0)
  {
   foreach (DataRow row in dsRecords.Tables[0].Rows)
   {
    String strXmlDoc = row["xmlData"].ToString();

    DataRow newRow = dtDemo.AddNewRow();
    newRow["strID"] = row["uidID"].ToString();
    newRow["vchPartnerName"] = row["vchPartnerName"].
ToString();
    newRow["vchLocation"] = strGetValueFromXml(strXmlDoc,
"HAD", "Name");
    newRow["vchPONumber"] = strGetValueFromXml(strXmlDoc,
"HAD", "PONumber");
    newRow["vchTargetID"] = strGetValueFromXml(strXmlDoc,
"HAD", "TargetID");
    newRow["vchDate"] = strGetValueFromXml(strXmlDoc,
"ENV", "DateSent");
    newRow["vchErrorDescription"] =
row["vchOrigErrorDescription"].ToString();
   }
  dtDemo.AcceptChanges();
  }
 }
}
```

Column0	Column1	Column2
abc	abc	abc
abc	abc	abc
abc	abc	abc
abc	abc	abc
abc	abc	abc

☐ Approved

Save Cancel

Figure 1-7. Basic grid layout

While this focuses on top level fields, you should be able to envision how this could be done with repeating nodes, such as the line items. The web application can have extensive functionality that allows for editing of data and removal of line items, if required. Once the data has been fully reviewed and edited by a user, and has been marked as approved, the web application can drop a copy of the final edited 850 XML into a file drop where BizTalk can pick it up, map it to the target flat file, and deliver it to the target system—following the same path and using the same map that you created earlier for the automatic approval process in the orchestration. In this case, you would just need a receive port, receive location, and a send port (along with the map configured on either the receive or send port) in order to complete this mapping and delivery of the final data.

Extending this Solution with Better Business Rules

Currently, the only business rule that is used to determine whether the inbound 850 can be automatically approved or not happens by checking a single distinguished fields (the BEG02). Real-world scenarios will require much more robust capabilities, including checking to see if specific line items have acceptable quantities and code. For example, you may want to force manual approval when there is a quantity greater than 100 for an individual line item. You may also want to determine if the inbound location codes are valid.

In cases where you have multiple business rules that need to be reviewed, you have several options. The first is using the BizTalk Business Rules Engine (BRE). The BRE has extensive functionality and allows for any variety of rules—but it also comes with a learning curve. Frequently, in EDI environments, users have deep SQL skills and limited BizTalk skills. Building a configurable set of business

rules using SQL is a compelling approach, and one that can be developed and supported fairly easily.

An example of implementing this for the location code would be the following steps:

- Get the inbound location code that you want to validate from the 850. This is found in the N104 node, and can be accessed either through xpath in the orchestration, or via a map.
- Pass the location code into a stored procedure where the lookup and validation can be done. The code in Listing 1-7 shows the C# for this call to the database. By encapsulating the call within a method in an assembly, it can be called from either within a map or within an Expression shape in an orchestration.
- Once the value has been returned, the logic in the decision shape can be based on whether it is valid or not. Any number of fields can be added into the logic of this decision shape.

Listing 1-7. Validating the Location Code

```
public int BusinessRuleLocationCodeIsValid(string
strLocationCode, string strConnectionString)
{
 SqlConnection sqlConnection = new
SqlConnection(strConnectionString);
 string strStoredProcedure =
"spBusinessRule_LocationCodeIsValid";
 SqlCommand sqlCommand = sqlConnection.CreateCommand();
 sqlCommand.CommandText = strStoredProcedure;
 sqlCommand.CommandType = CommandType.StoredProcedure;
 sqlConnection.Open();

 SqlParameter sqlParameter = new SqlParameter();
 sqlParameter = new SqlParameter();
 sqlParameter.ParameterName = "@vchLocationCode";
 sqlParameter.SqlDbType = SqlDbType.VarChar;
 sqlParameter.Direction = ParameterDirection.Input;
 sqlParameter.Value = strLocationCode;
 sqlCommand.Parameters.Add(sqlParameter);

 // value is returned
 int result = (int)sqlCommand.ExecuteScalar();
 sqlConnection.Close();

 return result;
}
```

Conclusion

You have just worked through a full implementation for receiving 850 data and mapping it to an internal proprietary flat file format. Receiving data and getting it into your system is sometimes all there is to an implementation. Other times, sending data back out to trading partners is required. The next chapter focuses on a specific implementation of sending EDI data to a trading partner, and introduces a number of topics not covered in this chapter.