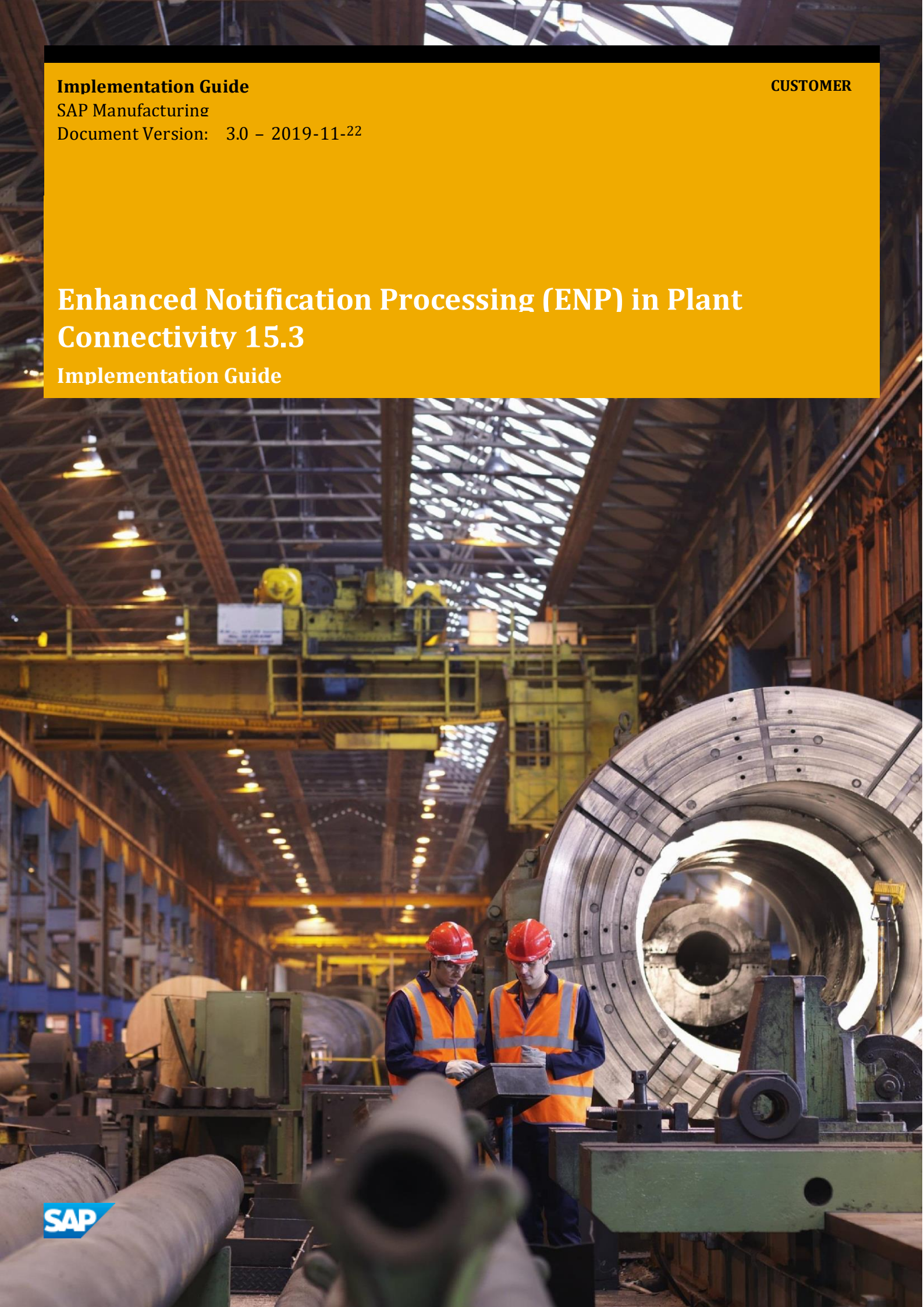
Implementation Guide **CUSTOMER**
SAP Manufacturing
Document Version: 3.0 – 2019-11-22

Enhanced Notification Processing (ENP) in Plant Connectivity 15.3

Implementation Guide



Typographic Conventions

Type Style	Description
Example	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Textual cross-references to other documents.
Example	Emphasized words or expressions.
`EXAMPLE`	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE`	Keys on the keyboard, for example, `F2` or `ENTER`.

Document History

Version	Date	Change
1.0	2015-10-06	Updates for PCo 15.1
2.0	2019-06-14	Updates for PCo 15.3 (SP00)
3.0	2019-11-22	Updates for PCo 15.3 (SP01)

Table of Contents

1 Disclaimer

Document classification for SAP Library: Customer

1.1 Coding Samples

Any software coding or code lines/strings ("code") included in this documentation are only examples and are not intended to be used in a productive system environment. The code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the code given herein, and SAP shall not be liable for errors or damages caused by the usage of the code, except if such damages were caused by SAP intentionally or due to gross negligence.

1.2 Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint where to find supplementary documentation. SAP does not warrant the availability and correctness of such supplementary documentation or the ability to serve for a particular purpose. SAP shall not be liable for any damages caused by the use of such documentation unless such damages have been caused by SAP's gross negligence or willful misconduct.

1.3 Accessibility

The information contained in the SAP Library documentation represents SAP's current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP specifically disclaims any liability with respect to this document and no contractual obligations or commitments are formed either directly or indirectly by this document.

2 Overview

With SAP Plant Connectivity (PCo), SAP provides a software component that enables the exchange of data between an SAP system and the industry-specific standard data sources of different manufacturers, for example, process control systems, plant Historian systems, and programmable logic controller (PLC) systems. With PCo, you can receive tags and events from the connected source systems in production either automatically or upon request and forward them to the connected SAP systems.

The Plant Connectivity component supports the following basic processes:

- **Notification process**: The notification process enables you to monitor production facilities and record any sudden, undesired events (such as rule violations or changes in measurement readings) and report them to a destination system.
- **Query process**: This process enables you to query specific source system tags from a destination system (such as SAP MII). This data can then be displayed on a dashboard, for example.

Enhanced notification processing (ENP) enables you to flexibly control and document the data flow in production in connection with various destination systems, for example, with Web services. In this way, you can connect a third-party system (such as SAP ME) to PCo and transfer data from machine level to the desired SAP ME activity using Web service calls. This makes it possible, for example, starting from PCo, to call a Web service provided by SAP ME, evaluate the result of the call, and then, depending on the results of the Web service call, call an additional Web service or write data back to a source system.

SAP delivers the standard enhancement *Destination System Calls with Response Processing* for enhanced notification processing with which you can execute one or multiple destination system calls and with which you can write back the results of the calls to the data source of the agent instance or other agent instances. This covers the most common requirements in communication between data sources of production and business systems.

If you want to implement requirements that go beyond the function scope of the SAP standard enhancement, you can implement a customer-owned enhancement. PCo provides an interface for this purpose that you can implement. The notification enhancement is mapped in the form of a destination system so that the enhancement is called as part of a notification process.

Enhanced notification processing enables you to do the following:

- You can call one or more destination systems one after the other in any order. This might be, for example, a regular Web service, a RESTful Web service, an SAP ESP destination, an OData destination, or an ODBC destination.
- You can call mass-enabled destination systems (for example, Web services) for multiple object instances, for example, for multiple SFC numbers.
- You can assign the output expressions of a notification to the parameters of a Web service statically or dynamically.
- You can perform a program-controlled evaluation of the results of a destination system call and react to the results accordingly.
- After a notification message is received, the destination system can send information to a specific agent and update, for example, a tag value in the source system.

Note that the ENP was called *Customer-Specific Logic* in PCo releases before 15.1.

3 Prerequisites

3.1 Technical Prerequisites

The following technical prerequisites are necessary in order to build a customer-owned enhancement for PCo 15.3:

- .NET Framework 4.7.2 or higher
- .NET development environment, for example, Microsoft Visual Studio 2017 Professional or higher.

3.2 Required Knowledge and Skills

- .NET development knowledge, preferably C#
- Knowledge in building and creating PCo agents

- Knowledge in consuming Web services

Prerequisites

4 Architectural Overview

4.1 Main Building Blocks

The following diagram shows the main building blocks of Plant Connectivity for a notification process and illustrates how the ENP is embedded into the architecture.

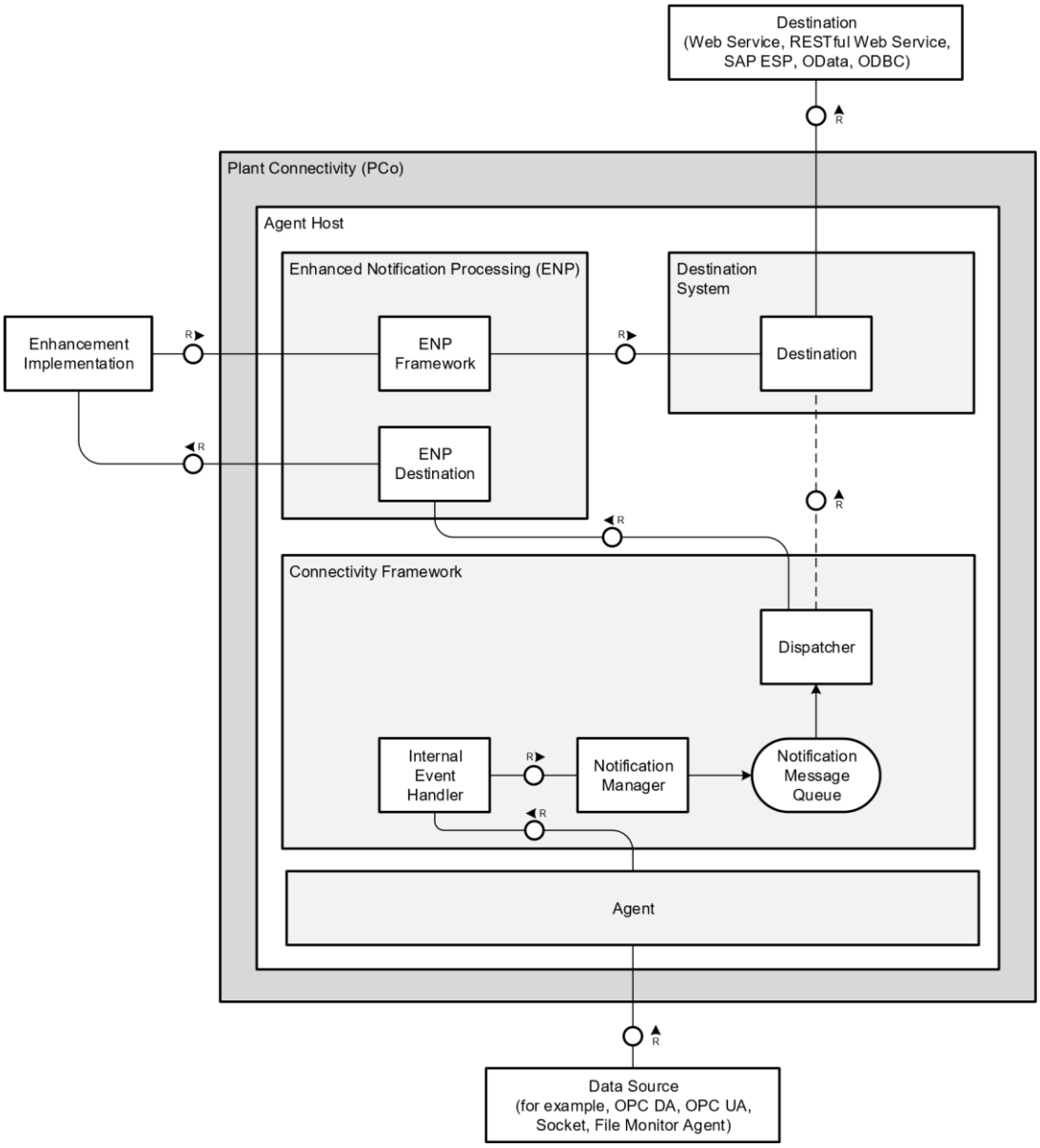


Figure 1: Integration of the ENP Enhancement Implementation into Plant Connectivity 15.1

The existing process flow for a notification coming in from a source system is as follows:

When a notification is received from a data source, an event is raised, and the notification manager takes care that a notification message is built. The message is submitted to the destination system by the dispatcher through the notification message queue so that reprocessing is possible in case of dispatching failures.

The ENP is hooked in as a special destination that is called by the dispatcher. The ENP destination forwards the notification message to the enhancement implementation that implements the controller interface. The enhancement implementation may at first execute a parsing method for the notification message in order to extract variables and their actual values. Then a programmed sequence of steps can be executed; typically, these steps are synchronous calls of Web services. The Web service calls are handled by the ENP framework that offers the service methods. The response of these Web service calls is returned to the controller, interpreted, and can then be used to influence the further program flow and to parameterize further Web service calls. The controller can also call back to the agent instance to write data back to the source system or to read additional data from it. This feature makes it possible, for example, for machines to be controlled based on a decision taken in SAP ME.

If the execution of the enhancement implementation fails, the message is kept in the notification message queue for reprocessing. Since the processing sequence of Web service calls may be important, the notification message queue provides exactly-once-in-order (EOIO) capability, meaning that notification messages are processed in exactly the same sequence they were put into the queue.

PCo 15.1 supports enhancement implementations with the following features:

- A controller interface containing methods that allows you to retrieve module and enhancement variable names from the enhancement implementation during design time
- An ENP framework containing helper methods for destination system calls, agent callbacks, tracing, and reading the ENP configuration
- The integration of the controller interface call into the existing Plant Connectivity architecture by means of an ENP destination
- Configuration possibilities of the enhancement through the Management Console
- Configurable exception handling, so that expected exceptions that occur inside the enhancement implementation do not terminate the notification process
- Documentation and example implementations for the controller interface

4.2 Interfaces

In the architectural overview, the ENP was shown in the overall context of a notification scenario. Below is a drill down into the ENP. The diagram below shows a class diagram of the `CustomLogic` project and its relationship to the enhancement implementation. Note that the technical objects in PCo still carry the term *CustomLogic* in their names, although the general concept has been renamed to *Enhanced Notification Processing (ENP)* as of PCo 15.1.

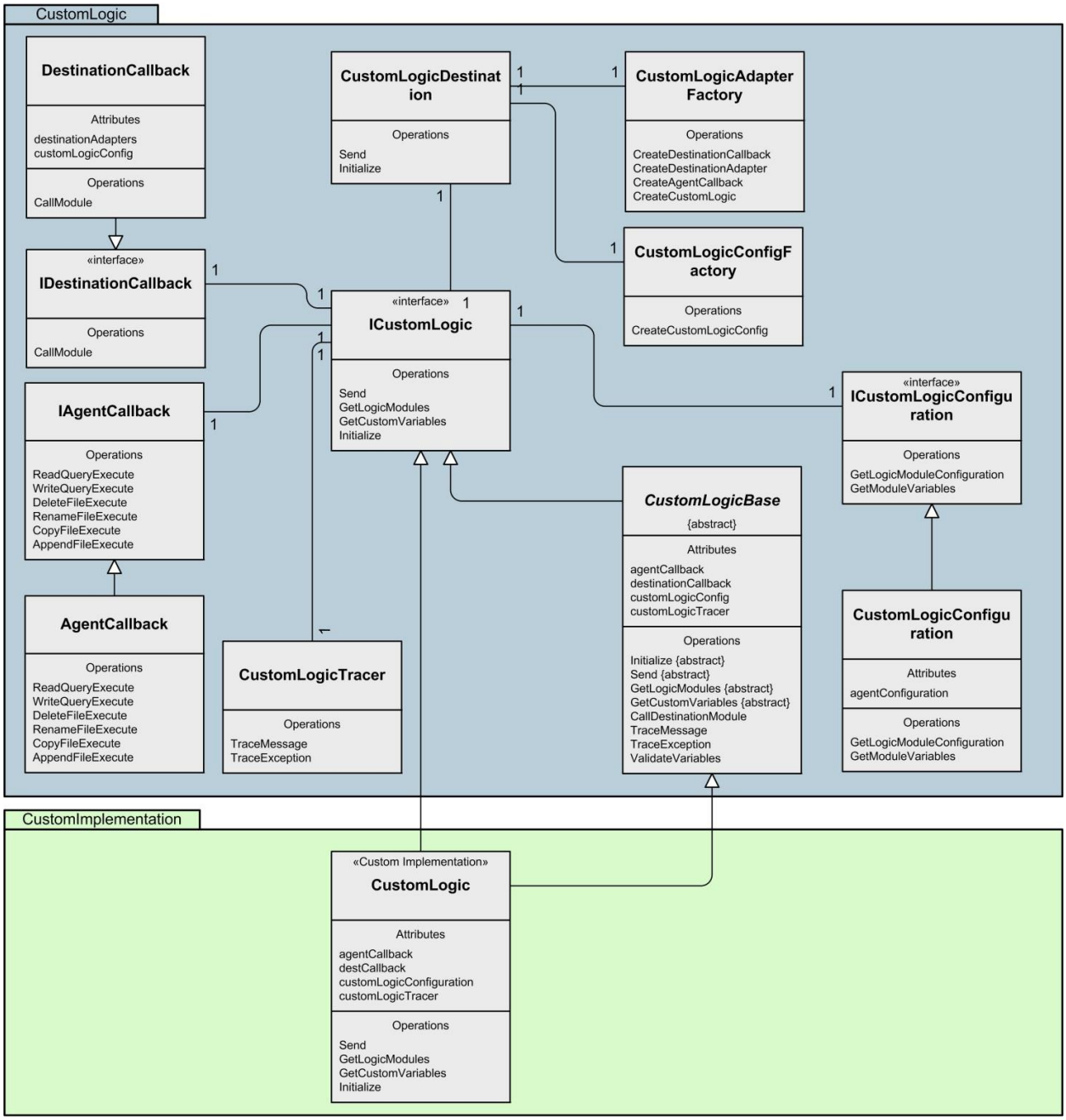


Figure 2: Class Diagram of the CustomLogic Project

The main part of the ENP logic is handled in the `CustomLogic` project. The `ICustomLogic` interface is the central interface that allows the customer to implement his or her own logic to process machine events meaning, for example, parsing of a file and calling SAP ME Web services. In addition to the `ICustomLogic` interface, there is an abstract class `CustomLogicBase` that inherits from the `ICustomLogic` interface and provides some basic implementations. So it is possible to inherit directly from the `ICustomLogic` interface or indirectly from the abstract class `CustomLogicBase` to set up the enhancement implementation.

To facilitate the enhancement implementation, the `ICustomLogic` interface provides four references to interfaces that offer service methods. These interfaces are:

- `IDestinationCallback`
  The interface encapsulates the call of a destination system.
- `IAgentCallback`
  The interface offers read and write methods for tag query agents and additional methods for the File Monitor Agent.
- `ICustomLogicConfiguration`
  The interface provides methods for reading the configuration of the enhanced notification processing destination.
- `CustomLogicTracer`
  A service class that provides methods to trace messages and exceptions so that they appear in the log in the Management Console.

Another central building block of the ENP is the `CustomLogicDestination` class. The `CustomLogicDestination` class forms a special destination system type. It inherits, like all destination systems, from the abstract class `DestinationBase`. This architecture enables notification messages to be sent to the `CustomLogicDestination` class and to be processed within the class. So this class is the link between the standard message processing and the ENP message processing in the notification process.

The `CustomLogicDestination` class sets up the environment for the enhancement implementation, meaning that the `CustomLogicDestination` class calls the different factories (`CustomLogicAdapterFactory` and `CustomLogicConfigFactory` classes) in order to create the instances of all relevant classes.

4.2.1 ICustomLogic Interface (Controller Interface)

The `ICustomLogic` interface contains the definition of how an enhancement implementation has to be implemented. The interface provides an initialization method, two design time methods for the configuration of an enhancement implementation, and one runtime method for processing notification messages.

4.2.1.1 Initialization

The `Initialize` method initializes the enhancement implementation with references to agent callbacks (`AgentCallback`), destination callbacks (`DestinationCallback`), the enhancement implementation configuration (`CustomLogicConfiguration`), and a tracer object for logging purposes (`CustomLogicTracer`). These references should be kept as private members of the enhancement implementation. They have to be used in the notification message processing in order to perform the enhanced notification processing actions.

4.2.1.2 Configuration Methods

In order to understand the use of the configuration methods in the `ICustomLogic` interface, the following diagram illustrates how the implementation of the enhanced notification processing is linked to the configuration of the enhanced notification processing:

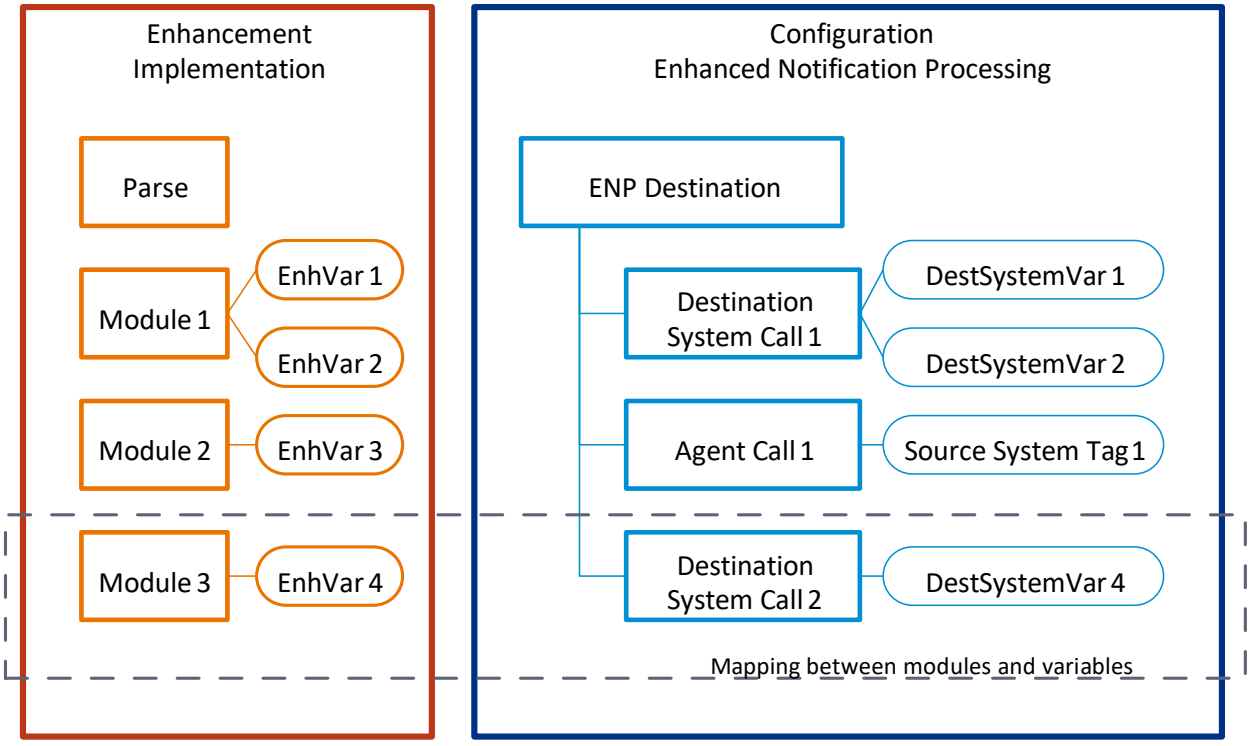


Figure 3: Mapping of Modules and Variables of the ENP

The basic assumption is that the enhancement implementation is done on an abstract level, so that it can be used in multiple PCo installations and destination systems within an enterprise. It is assumed that the enhancement implementation is described by modules and enhancement variables.

The modules represent steps that are executed in the ENP. The modules are described by a set of input parameters and a set of return parameters. Potential modules are Web service calls or agent calls respectively. The modules represent wrappers for the concrete call of a Web service or an agent. They are an abstraction layer that allows communication with the same type of destination system in a generic way. In order to call the concrete destination system, the module has to be mapped to a concrete destination system call or a concrete agent call, and the module parameters have to be mapped to the real module variables.

Parsing could be another process step of the enhanced notification processing. It usually converts machine parameters into parameters of the enhancement. Parsing is an independent, stand-alone, local process step of the enhancement implementation that does not require a wrapper or mapping.

The abstraction of the enhancement implementation is necessary for its reuse. The same implementation is potentially applicable to many concrete processes. The individual processes may have different parameters, depending on the location of the machine or device.

The ENP framework provides mapping of abstract modules to concrete destination system calls or agent calls respectively. In order to call, for example, a concrete Web service that is preconfigured in the Web Service Destination or to call a tag query agent, it is necessary to map the abstract module parameters to the concrete parameters of a Web service call or an agent call.

The mapping has to be performed within the Management Console of SAP Plant Connectivity during design time. In order to build a mapping dialog that fulfills the requirements described above, the `ICustomLogic` interface contains two configuration methods. The `GetLogicModules` method has to be implemented to return the modules of the enhancement implementation, and the `GetCustomVariables` method should return the enhancement variables of the enhancement implementation.

4.2.1.3 Processing of Notification Messages

The processing of the notification messages (for example, machine events) is realized with the enhancement implementation of the `Send` method. This implementation represents a customer-owned process. Typically, the implementation starts with a parsing process step in order to extract parameters from the notification message. In the subsequent program flow, destination system calls or agent methods could be used to build the customerowned process. The service methods of the `ICustomLogic` interface can be seen as a toolset to support the implementation of the ENP.

4.2.2 Custom-Logic Framework

The `CustomLogic` project, which is available as a dynamic link library (DLL) in the PCo installation folder, contains a framework that provides classes and interfaces with service functions. Some of these service functions are available in the enhancement implementation, and help to facilitate the implementation of the customer-owned processes. In addition, there are factory classes available for internal use. The factory classes encapsulate creating instances of helper classes.

4.2.2.1 CustomLogicBase Class

The abstract class `CustomLogicBase` inherits from the `ICustomLogic` interface. The abstract class implements the `Initialize` method of the `ICustomLogic` interface. In addition, it provides helper methods for tracing and Web service calls including enhancement variable validation.

The class offers another option for inheriting from the `ICustomLogic` interface and provides some basic method implementations for the customer-owned enhancement implementation.

4.2.2.2 DestinationCallback Class

The `DestinationCallback` class is a service class that allows communication with a destination system, for example, a given Web Service Destination. On the destination system configuration screen in the PCo Management Console, the destination system is preconfigured with constant values or variables, called destination system variables. These variables are accessible from the enhancement implementation during message processing, after you have configured the ENP destination as a destination system on the notification destination screen of the Management Console. The destination system call should be realized by calling the `CallModule` method in the enhancement implementation.

For each destination system that is used in the enhancement implementation, an instance of the `IDestinationAdapter` interface is created. The `IDestinationAdapter` interface is implemented in many of the PCo destination system classes, for example, the `WSDestination` class that handles the Web service communication. This implementation provides the destination system variables and the information as to whether a destination system variable is an input or an output variable. This information is used to map the abstract enhancement variables of the enhancement implementation to the destination system variables provided and to hand over the values for the destination system variables to the destination system call.

4.2.2.3 AgentCallback Class

The `AgentCallback` class is a service class that allows communication with different source systems represented by their corresponding agent types (`TagQueryAgent` and `FileMonitorAgent`). The `AgentCallback` class offers a restricted but simple interface for this purpose.

The following methods are available for the communication with different source systems:

- `ReadQueryExecute` (`TagQueryAgent`/`FileMonitorAgent`)
  This method reads tag values for a list of specified tags.
- `WriteQueryExecute` (`TagQueryAgent`/`FileMonitorAgent`)
  This method writes tag values for a list of specified tags at given time stamps.
- `DeleteFileExecute` (`FileMonitorAgent`)
  This method deletes a file.
- `RenameFileExecute` (`FileMonitorAgent`)
  This method renames a file.
- `CopyFileExecute` (`FileMonitorAgent`)
  This method copies a file.
- `AppendFileExecute` (`FileMonitorAgent`)
  This method appends a given content to a file.

4.2.2.4 CustomLogicConfiguration Class

The `CustomLogicConfiguration` class is a service class that provides the configuration of the enhancement implementation. The class consists of the following methods:

- `GetLogicModuleConfiguration`
  This method returns the mapping of the logic modules to destination systems or agent instances respectively.
- `GetModuleVariables`
  This method returns the mapping of the destination system variables to the enhancement variables of the enhancement implementation.

You maintain these mappings in the configuration of the ENP in the Management Console.

4.2.2.5 CustomLogicTracer Class

The `CustomLogicTracer` class is a service class that provides methods to trace messages and exceptions. The traced messages appear with respect to the ‘CustomLogicDestination’ source in the PCo application log.

4.2.2.6 CustomLogic Factories

There are factory classes `CustomLogicAdapterFactory` and `CustomLogicConfigFactory`. The `CustomLogicAdapterFactory` class provides the following factory methods:

- `CreateCustomLogicDestination`
  This method creates an instance of the `ICustomLogicDestination` interface.

- `CreateDestinationCallback`
  This method creates an instance of the `IDestinationCallback` interface. Additionally the references to the `IDestinationAdapter` interface are created.
- `CreateDestinationAdapter`
  This method creates an instance of the `IDestinationAdapter` interface.
- `CreateAgentCallback`
  This method creates an instance of the `IAgentCallback` interface.
- `CreateCustomLogic`
  This method creates an instance of the `ICustomLogic` interface.

The `CustomLogicConfigFactory` class provides the following factory method:

- `CreateCustomLogicConfig`
  This method creates an instance of the `ICustomLogicConfiguration` interface.

5 How to Implement a Customer-Owned Enhancement

5.1 Overview of Implementation Steps

The implementation of a customer-owned enhancement consists mainly of two parts:

1. Implementation activities in Microsoft Visual Studio:
   - Create a class that implements the `ICustomLogic` interface.
   - Create implementations of a parameter-less class constructor, and of the interface methods `GetLogicModules`, `GetCustomVariables`, `Initialize`, and `Send`.
   - Build an ENP DLL (dynamic link library) from your class.
2. Configuration activities in the PCo Management Console:
   - Create destination systems for every destination system that you want to call from the customer-owned enhancement implementation.
   - Maintain the destination system settings and operation configuration in the destination system. ○ Define agent instances for the source systems with which you want to communicate within the ENP. ○ Create an agent instance, and link your ENP DLL to the agent instance. ○ Create a notification for this agent and define the ENP as the destination of this notification.
   - Maintain the mapping of modules, variables, and source system tags for the notification destination.

The following sections describe the individual activities in detail.

5.2 Implementation Activities in Microsoft Visual Studio

5.2.1 Prerequisites

To create a customer-owned enhancement, you require an integrated development environment for .NET development. SAP recommends using Microsoft Visual Studio 2012 or higher. The following steps are explained by using screenshots and settings from Microsoft Visual Studio 2012.

If you want to use an ENP DLL provided by a third party, you can skip this section and proceed directly to section 5.3. Note

that the DLL has to be built for .NET framework 4.0 with platform target *Any CPU*. **How to Implement a Customer**

5.2.2 Create a Solution and an Implementing Class

First create a new solution for your implementation. Make sure you choose *.NET Framework 4.0*. Use the *Class Library* template.

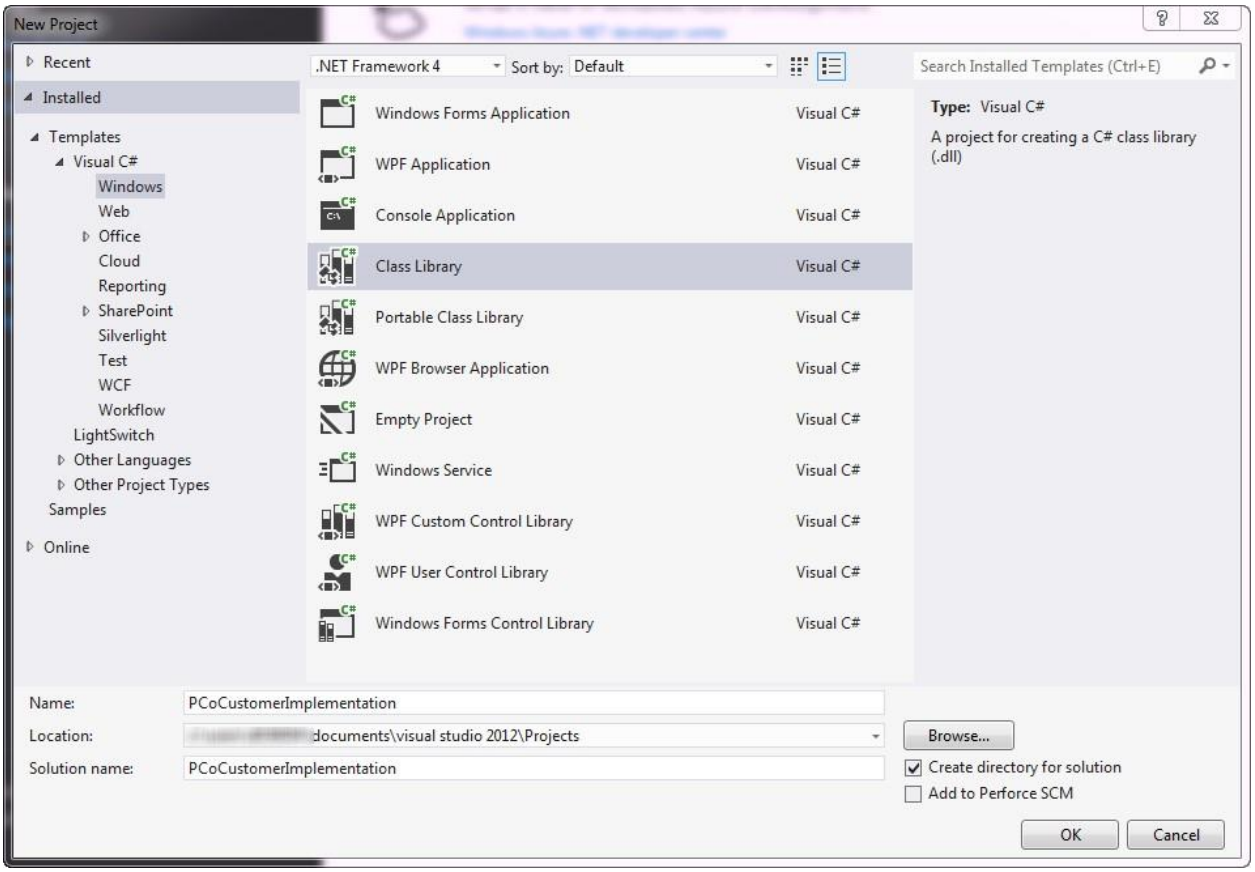


Figure 4: Create a New Solution

Then set the properties of your new project. On the Application Properties tab, check if the target framework is set to *.NET Framework 4.0* and the output type is set to *Class Library*.

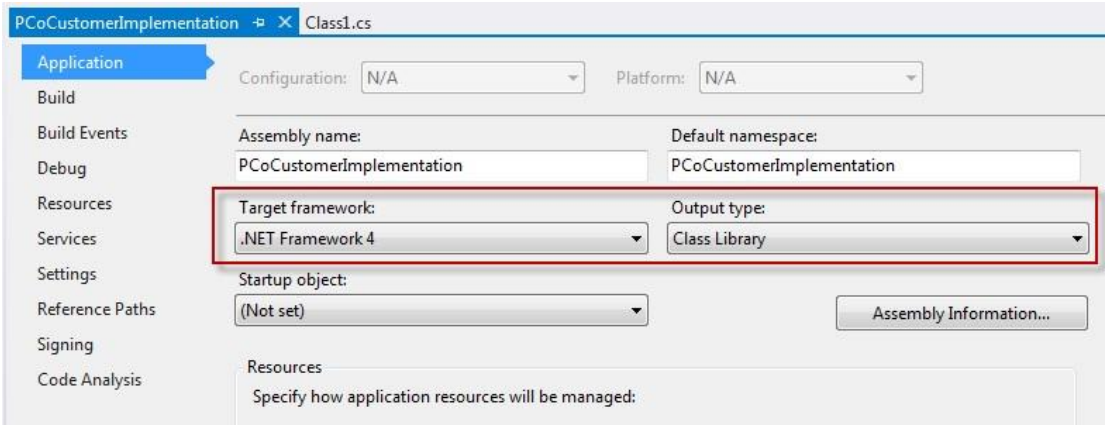


Figure 5: Application Properties

Make sure that the platform target is set to *Any CPU* for all configurations.

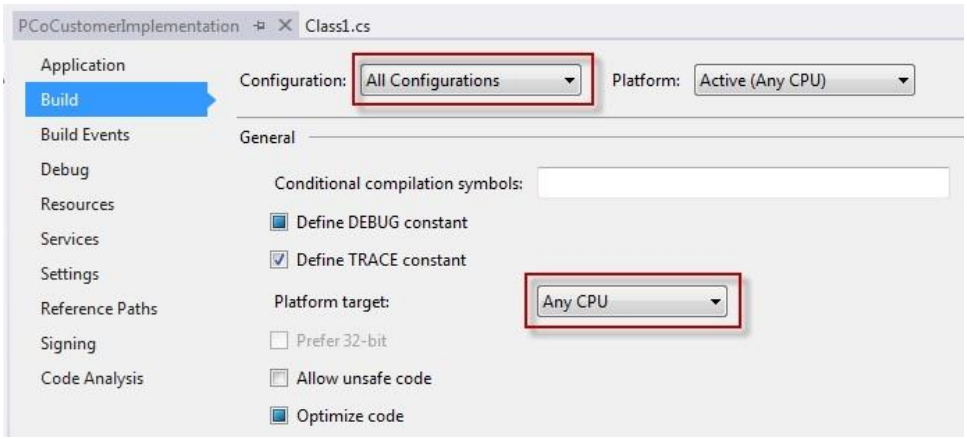


Figure 6: Project Build Configuration

Now create a reference to the PCo `Custom-Logic` DLL for your project.

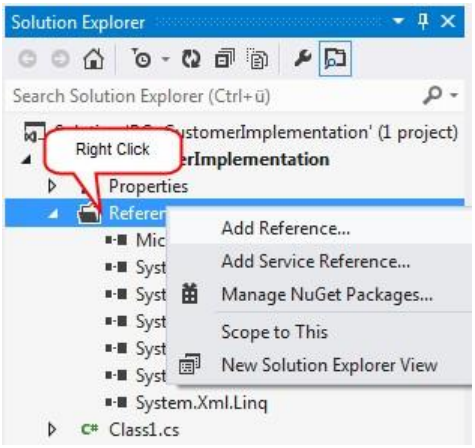


Figure 7: Adding References to Your Project

Choose *Browse* then select the *CustomLogic.dll* from the *System* subfolder of the PCo program folder. The *References* branch of your project should now contain the reference to the CustomLogic.dll.

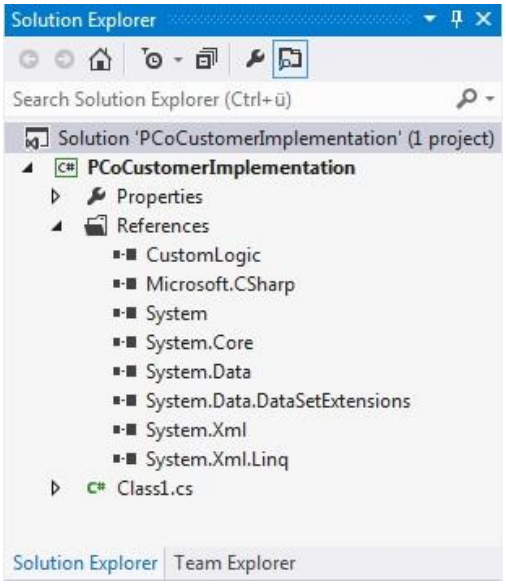


Figure 8: Required Project References

You now have a solution with an almost empty class, for example, `Class1.cs`, which could look as follows:

```
using System; using
System.Collections.Generic; using
System.Linq; using System.Text;
namespace PCoCustomerImplementation
{
    public class Class1
```

How to Implement a Customer-

```
    {
```

Customer

```
    }
}
```

5.2.3 Implement Required Methods

In order to use your implementation as an enhancement implementation in PCo, you have to implement the `ICustomLogic` interface within your class. Start by adding the using directive for the namespace `SAP.Manufacturing.CustomLogic` to your code. Now your class coding could look as follows:

```
using System; using
System.Collections.Generic; using
System.Linq; using System.Text;
using
SAP.Manufacturing.CustomLogic;
namespace
PCoCustomerImplementation
{     public class Class1 :
ICustomLogic
    {
    }
}
```

Then implement the required interface methods. These methods are:

- A parameter-less constructor, for example, `public Class1()`
- `Dictionary<string, Type> ICustomLogic.GetCustomVariables(Guid notificationID)`
- `Dictionary<string, ModuleType> ICustomLogic.GetLogicModules()`
- `void ICustomLogic.Initialize(IAgentCallback agentCallback, IDestinationCallback destinationCallback, ICustomLogicConfiguration customLogicConfig, CustomLogicTracer customLogicTracer)`
- `bool ICustomLogic.Send(CustomLogicNotificationMessage notification)`

Visual Studio can assist you with creating the empty methods, except the constructor: Right-click on the interface name `ICustomLogic` and choose *Implement Interface > Implement Interface Explicitly*.

Now implement the individual methods:

`GetLogicModules()`: This method returns the list of agent and destination system call modules mapped to their module types *Agent Call* or *Destination System Call*. The modules appear later in the PCo Management Console in the assignment of agent instances and destination systems to the ENP modules (see section 5.3.5.1). The method returns a dictionary where the key is the module name, and the value is the module type. For example, in order to define the modules `Start`, `Complete`, `DataCollection`, and `AgentCall`, you could write the following:

```
public Dictionary<string, ModuleType> ICustomLogic.GetLogicModules()
{
        Dictionary<string, ModuleType> modules =
new Dictionary<string, ModuleType>();
modules.Add("Start", ModuleType.Destination);
modules.Add("Complete", ModuleType.Destination);
modules.Add("DataCollection", ModuleType.Destination);
```

```
modules.Add("AgentCall", ModuleType.Agent);
return modules;
}
```

`GetCustomVariables()`: This method returns the list of enhancement variables mapped to their technical types. The variables appear later in the PCo Management Console in the assignment of enhancement variables (see section 5.3.5.2). The method returns a dictionary for which the key is the enhancement variable name, and the value is the technical type. If you use more than one notification in your scenario you can evaluate the parameter `notificationID` and return only those variables that belong to that specific notification. For example, in order to define the variables `site`, `operation`, `returnScrap`, `shopFloorControl`, `collectionParameters`, `resource`, and `revision`, you could write the following:

```
public Dictionary<string, Type> ICustomLogic.GetCustomVariables(Guid
notificationID)
{
       Dictionary<string, Type> variables = new Dictionary<string, Type>();
variables.Add("site", typeof(string));        variables.Add("operation",
typeof(string));        variables.Add("returnScrap", typeof(bool));
variables.Add("shopFloorControl", typeof(string));
variables.Add("collectionParameters", typeof(double[]));
variables.Add("resource", typeof(string));
variables.Add("revision", typeof(string));        return variables;
}
```

`Initialize()`: This method is called by the ENP destination when the agent instance is started. It receives references to the agent callbacks, destination callbacks, the ENP configuration, and the tracer. The tracer object is used to write messages to the agent log. You should keep the references in class member fields so that you can use them later in the `Send()` method. For example:

```
public void ICustomLogic.Initialize(
            IAgentCallback agentCallback,
            IDestinationCallback destinationCallback,
            ICustomLogicConfiguration customLogicConfig,
            CustomLogicTracer customLogicTracer)
{             this.agentCallback = agentCallback;
this.destinationCallback = destinationCallback;
this.customLogicConfig = customLogicConfig;
this.customLogicTracer = customLogicTracer;
}
```

`Send()`: This method contains the ENP processing logic. Within this method, you can implement your parsing of input data, destination system calls, and agent callbacks, for instance.

How to Implement a Customer-

5.2.4 Build the ENP DLL

After you have finished the implementation, you have to create the ENP DLL that is used later in the PCo

Management Console. In Visual Studio, choose a configuration (for example, *Debug* or *Release*), then choose *Build Solution*. If the build was successful, you should find the ENP DLL in the output directory defined in the project properties.

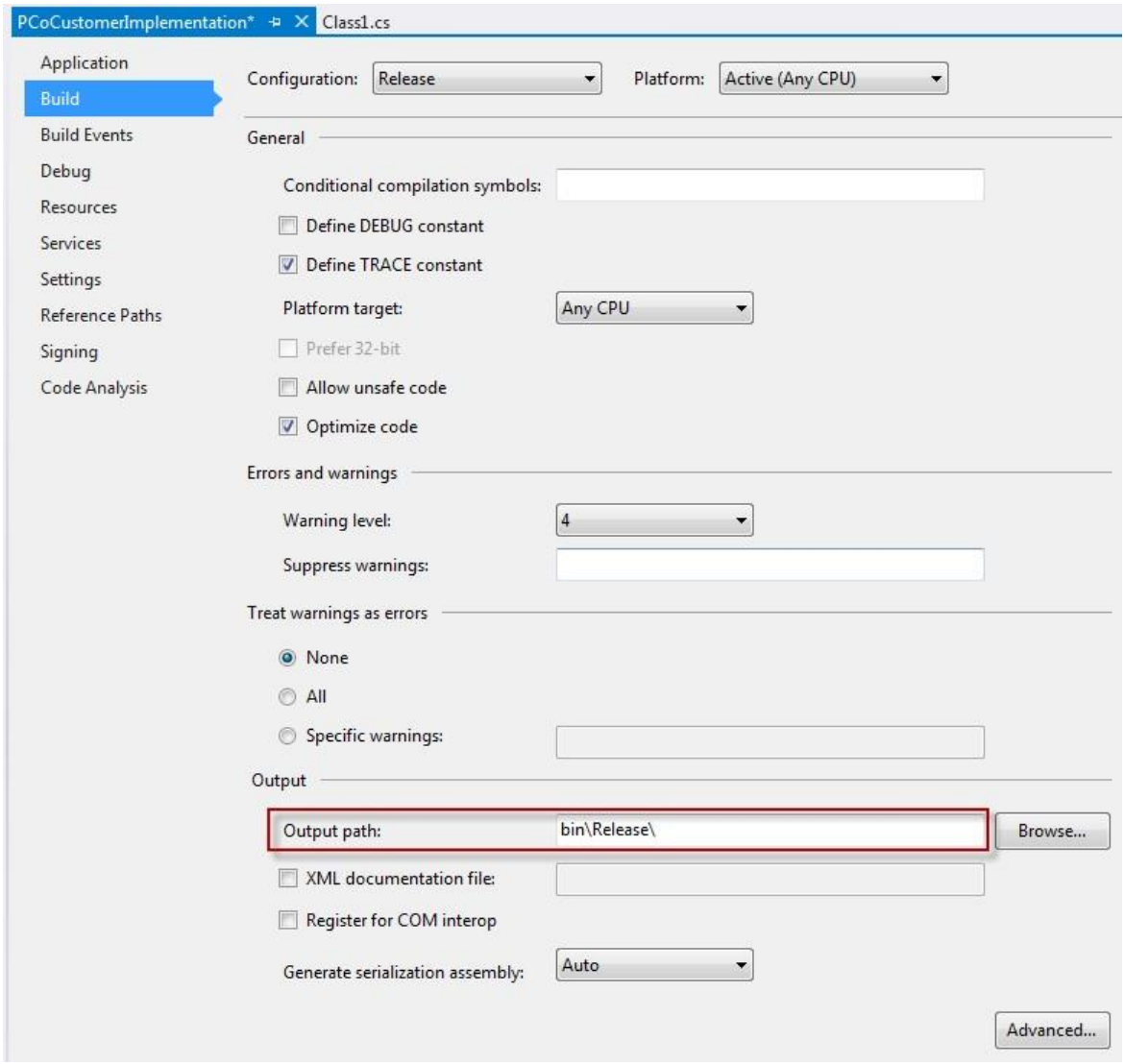


Figure 9: Check the Output Path for the ENP DLL

The resulting DLL is named `<Project Name>.DLL`, for example, `PCoCustomerImplementation.dll`

It is recommended that you copy the DLL into the System folder of the PCo installation. On Windows 32-Bit installation, this folder is named *<Installation Drive>:\Program Files\SAP\Plant Connectivity\System*, in 64-Bit systems *<Installation Drive>:\Program Files (x86)\SAP\Plant Connectivity\System*. If the ENP DLL resides in a different folder, the dependent PCo DLLs are duplicated into this folder once the ENP DLLs are loaded during the runtime of the Management Console.

5.3 Configuration Activities in the PCo Management Console

5.3.1 Prerequisites

Before you can configure the ENP, you create the ENP DLL as described in the previous section 5.2; in particular all of the methods mentioned in section 5.2.3 have to be implemented. Alternatively, you could use a third-party DLL that implements a compatible version of the `ICustomLogic` interface and is compiled for .NET framework
4.0 and platform target *Any CPU*.

Furthermore, you have to define an agent instance based on a source system from which you receive the data that is to be processed in the ENP. The source system can be configured as usual; there is nothing specific to be taken into account regarding the ENP.

5.3.2 Defining Destination Systems

Define a destination system for each destination system that you plan to call in your enhancement implementation. For example, for a Web Service destination, enter the WSDL URL and the required authentication settings, and then retrieve the service information by pressing *Retrieve Services*.

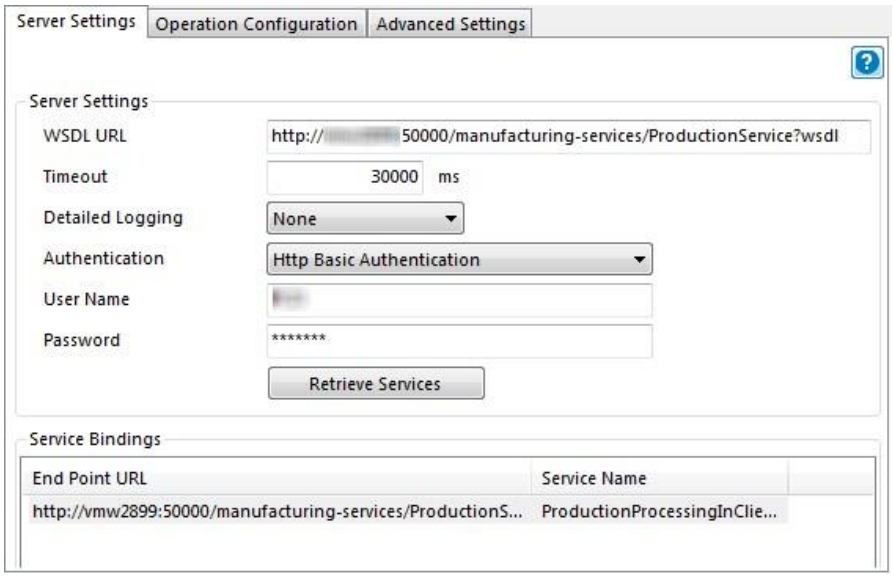


Figure 10: Web Service Destination Configuration

5.3.2.1 Flat Variables

In your destination system, you can define destination system variables that will be used to pass request variables to the destination system and fill response variables from the destination system at runtime.

For example, for a Web Service Destination, on the *Operation Configuration* tab, you enter variables for those Web service fields that are to be dynamically filled from the ENP. These variables are the destination system variables that will show up later in the ENP configuration screens. To define destination system variables, select a service operation, select a Web service operation field, enter a destination variable name in the *Value* column, and finally select the *Variable* checkbox. After you have done the variable mapping for the request message, press *Test Request Message*. A popup appears where you enter values for the request destination variables. Then the Web service operation is called with these parameters.

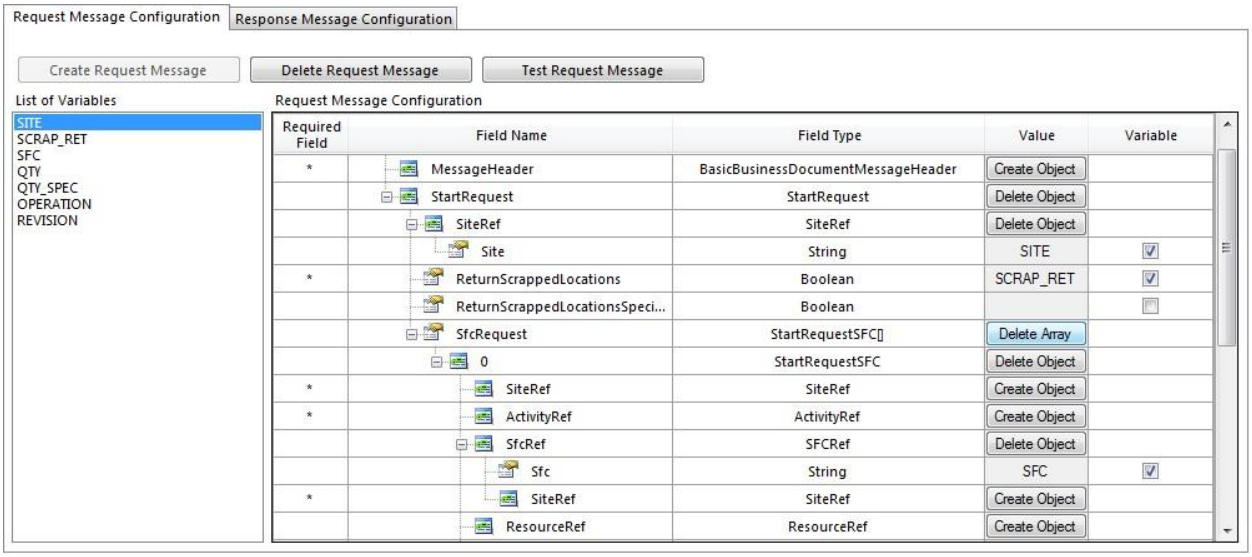


Figure 11: Mapping of Flat Variables

If the call was successful, you can continue with defining response variables and calculated variables on the *Response Message Configuration* tab.

5.3.2.2 One-Dimensional Arrays

In some cases, you want to call a Web service operation for multiple business documents. For example, you want to call the SAP ME Web service Start for *multiple* SFCs. You have two possibilities to model this.

One possibility is to call the Web service for a fixed number of documents. In this case, you press the *Create Array* button next to an array field (for example, `SfcRequest`), and create an **array with a fixed size**:

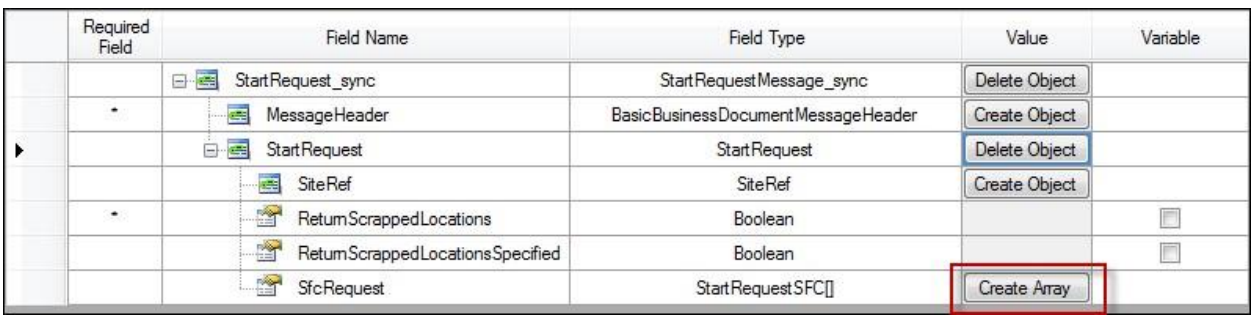


Figure 12: Create Array

When you are prompted to enter the array size, enter a numeric value, for example, 3.

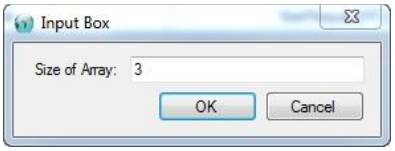


Figure 13: Declare an Array of Fixed Size

Then a fixed number of document nodes are created.

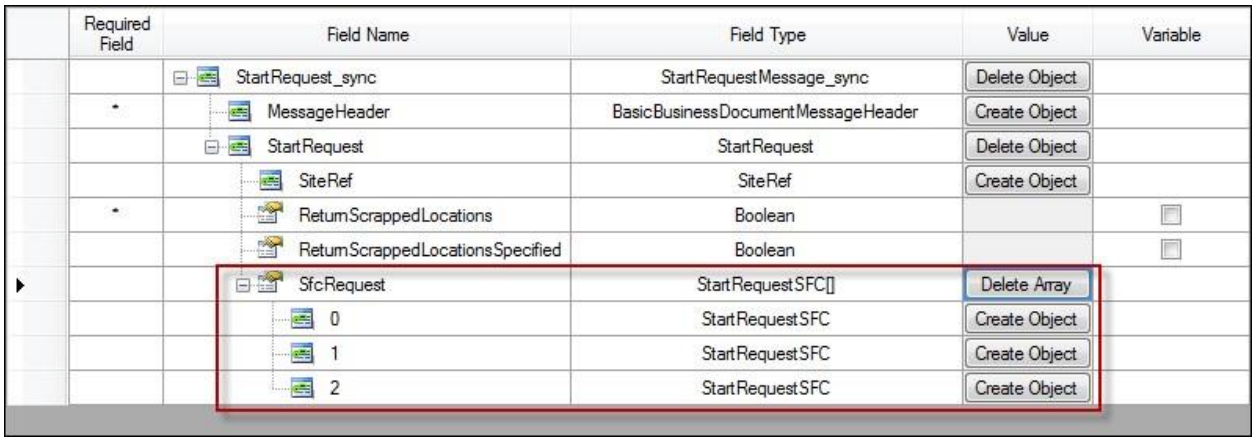


Figure 14: An Array with a Fixed Size

Variable-inside arrays with a fixed size require unique variable names. For example, the Site variable inside SfcRequest 0 could be named SITE0, whereas the Site variable inside SfcRequest 1 would need to be named differently, for example, SITE1.

Another possibility is to use **arrays with a dynamic size**. In order to define a dynamic array, declare an index variable instead of a fixed size. The index variable is a nonnumeric character. Enter the index variable instead of a numeric value when you are prompted for the array size, for example:

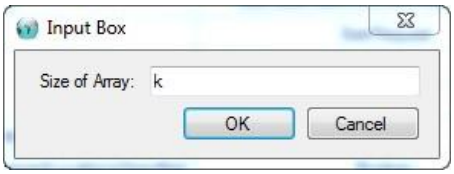


Figure 15: Declare an Index Variable for an Array with a Dynamic Size

A subnode with the name of the index variable is created:

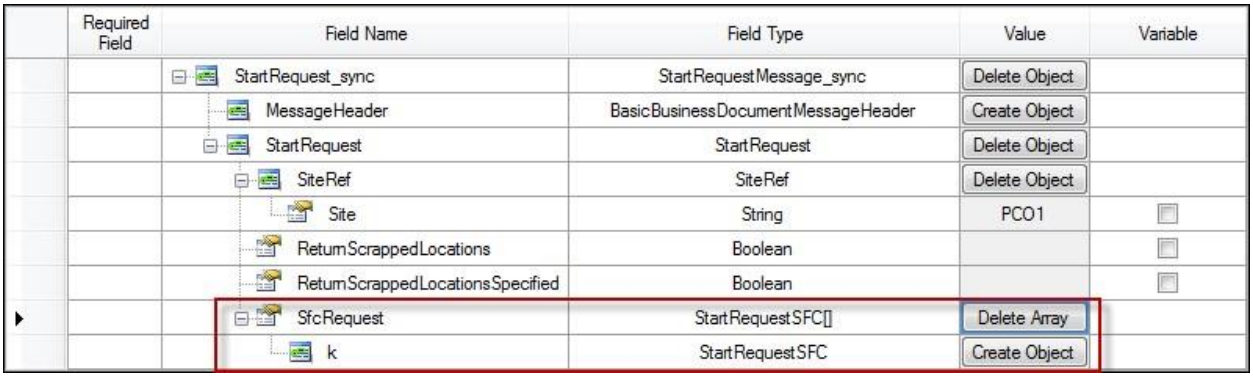


Figure 16: An Array with a Dynamic Size

An indexed variable inside the array would then be named `<Variable Name>[<index variable>]`, for example, `SFC[k]`.

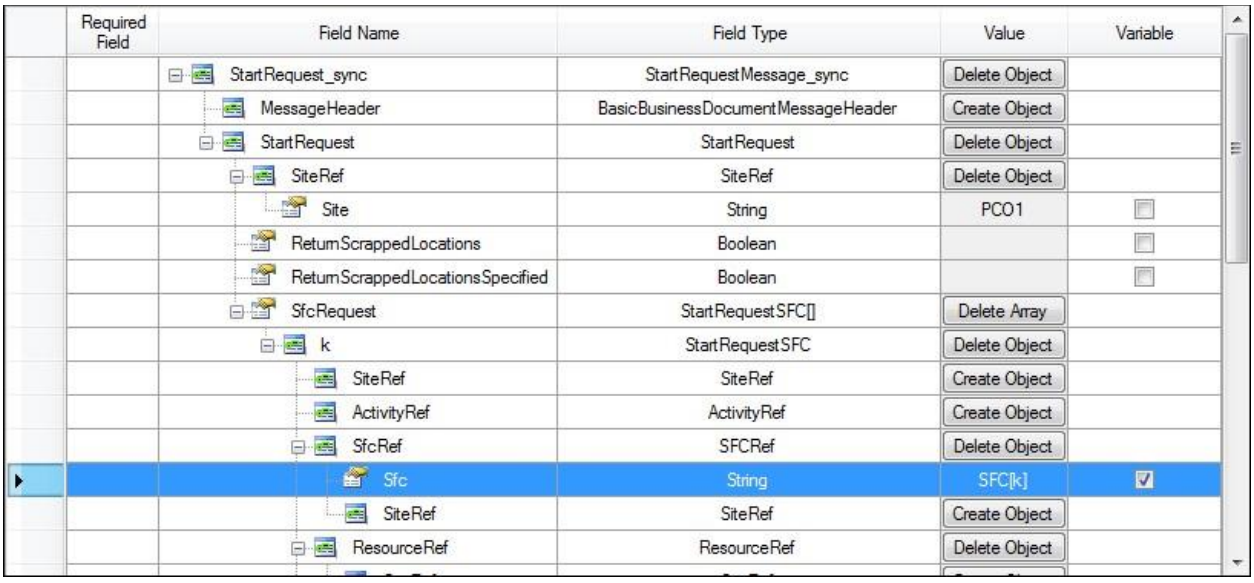
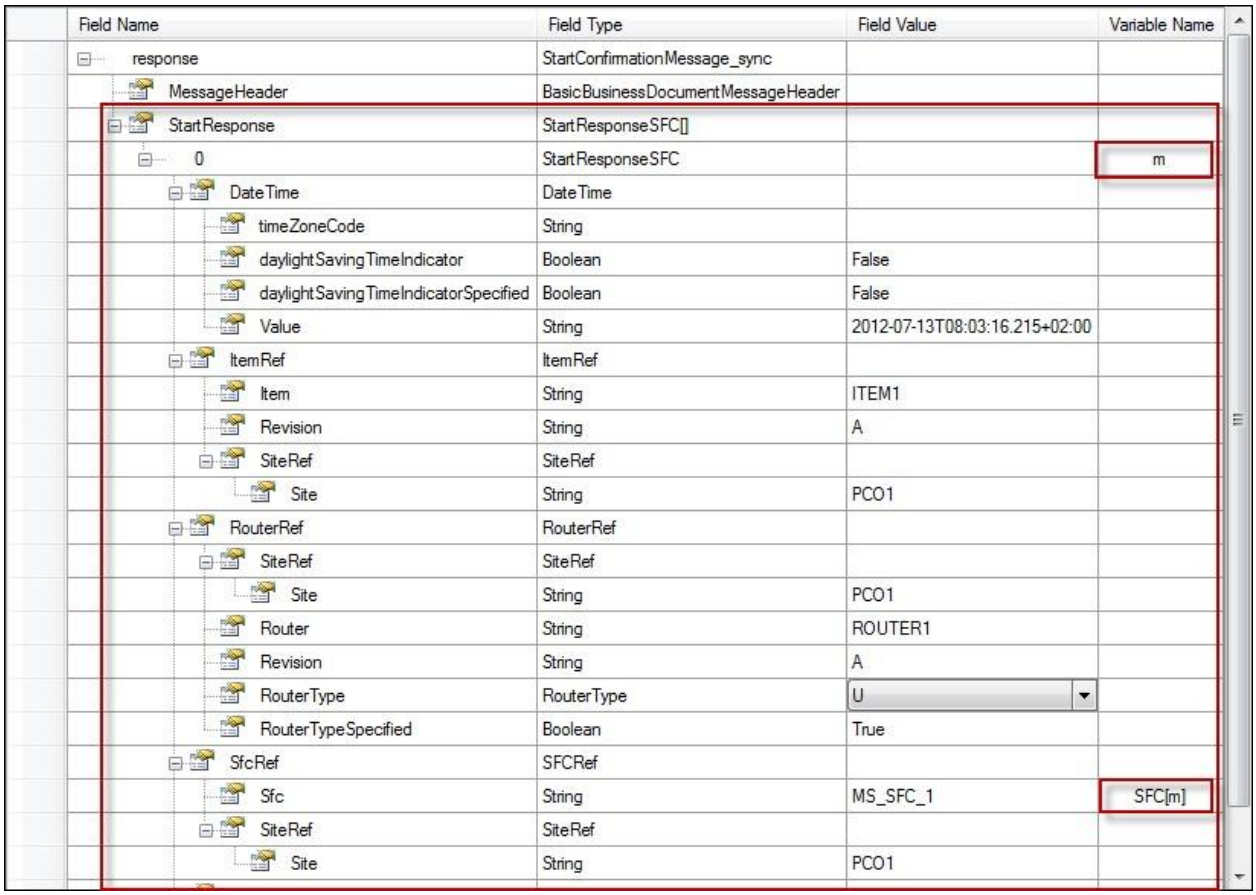


Figure 17: How to Declare an Indexed Variable

If you want to address **array variables in response messages**, you can do the following. Declare an index variable for the direct subnode of the array field, for example: For the subnode `0` of `StartResponseSFC[]`, declare the

index variable `m` in the column *Variable Name*. Then you can address an array variable as `<Variable Name>[<index variable>]`, for example: `SFC[m]`.



Field Name	Field Type	Field Value	Variable Name
response	StartConfirmationMessage_sync		
MessageHeader	BasicBusinessDocumentMessageHeader		
StartResponse	StartResponseSFC[]		
0	StartResponseSFC		m
DateTime	DateTime		
timeZoneCode	String		
daylightSavingTimeIndicator	Boolean	False	
daylightSavingTimeIndicatorSpecified	Boolean	False	
Value	String	2012-07-13T08:03:16.215+02:00	
ItemRef	ItemRef		
Item	String	ITEM1	
Revision	String	A	
SiteRef	SiteRef		
Site	String	PCO1	
RouterRef	RouterRef		
SiteRef	SiteRef		
Site	String	PCO1	
Router	String	ROUTER1	
Revision	String	A	
RouterType	RouterType	U	
RouterTypeSpecified	Boolean	True	
SfcRef	SFCRef		
Sfc	String	MS_SFC_1	SFC[m]
SiteRef	SiteRef		
Site	String	PCO1	

Figure 18: Array Variables in Response Mapping

5.3.2.3 Two-Dimensional Arrays

You can define **arrays of up to two dimensions**. For example, a one dimensional array `NcLog[]` with index variable `m` has another array `NcCustomData[]` with index variable `n` inside. So a variable for `Location` with one dimension `m` could be named `LOC[m]`, whereas a two-dimensional variable inside `NcCustomData[]` variable could be named `TEXT[m][n]`.

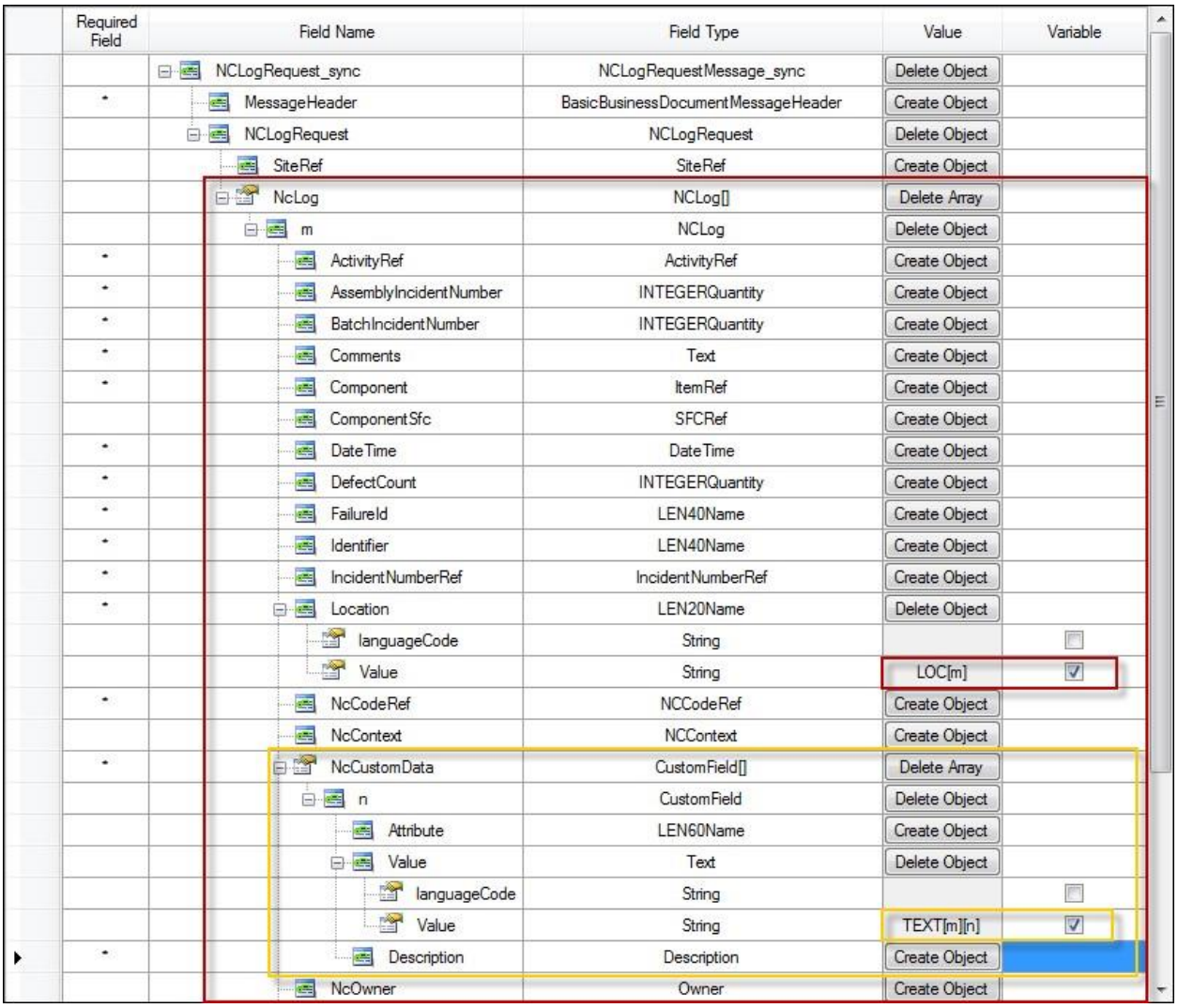


Figure 19: Two-Dimensional Arrays

5.3.3 Creating and Configuring an Agent Instance

Create a source system from where you would like to obtain data tags. Then create an agent instance for the source system. For the ENP it is recommended that you use the agent option *Process Notification Messages Exactly Once in Order* for notification messages on the *Host* tab. Otherwise, it is not guaranteed that notification messages are processed in exactly the same order in which they came from the source system.

On agent instance level, you define which enhancement implementation is to be used. Go to the *Notification Processing* tab on agent instance level, select *Customer-Owned Enhancement*, and browse for the ENP DLL.

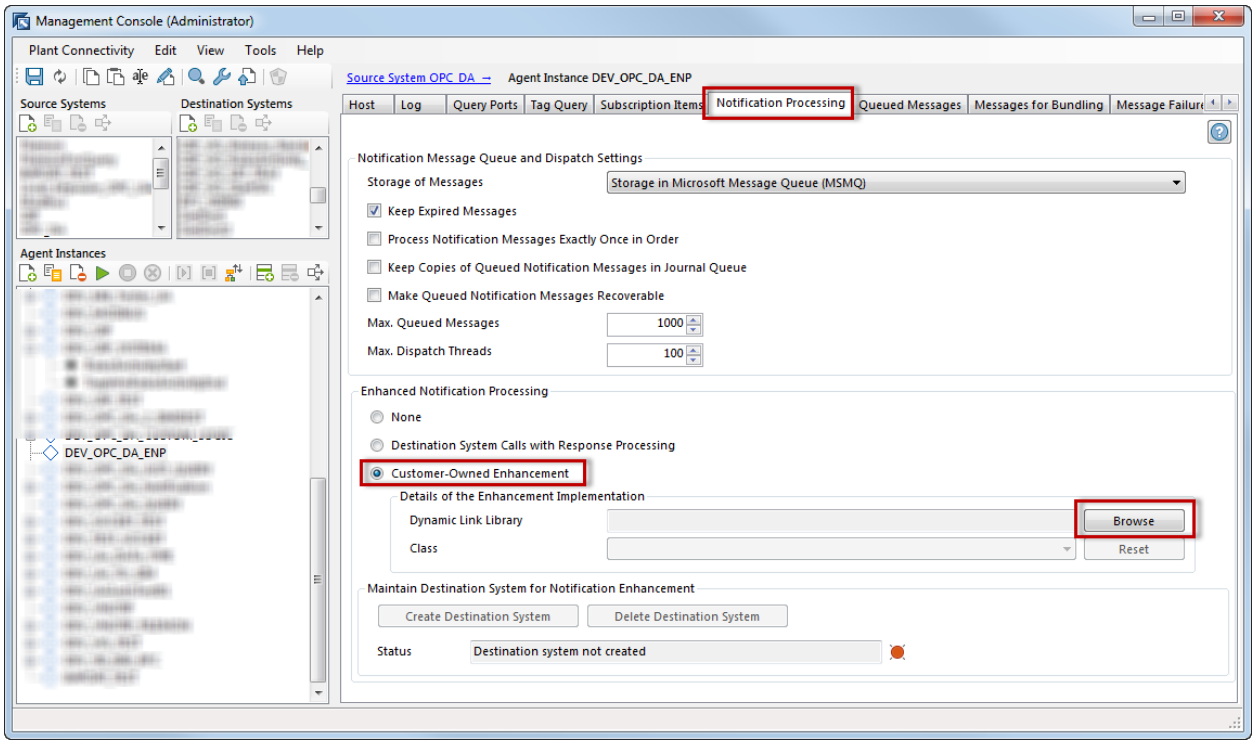


Figure 20: Browse for the enhanced notification processing

The system only accepts DLLs that implement the `ICustomLogic` interface (see section 5.2).

If there is only one class in the DLL that implements `ICustomLogic`, the system automatically proposes this class in the corresponding field. Otherwise, select the desired class from the dropdown box. Use the *Reset* button to undo the configuration of DLL and class.

The ENP is hooked in as a special destination system into the Plant Connectivity framework (see the sections about architecture). Therefore, you have to create a destination system for the ENP. You do this by clicking on the corresponding button on the *Configuration* tab.

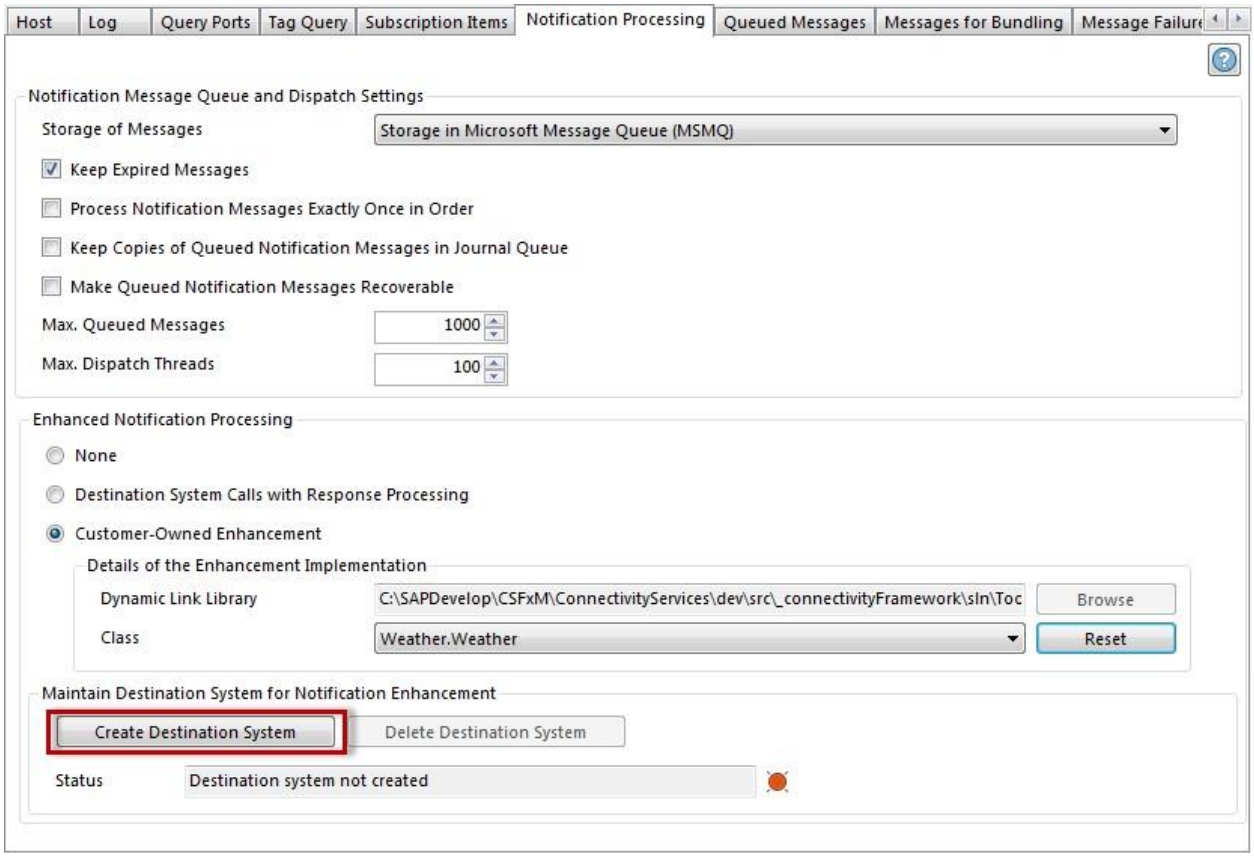


Figure 21: Create Destination System for the ENP

The ENP destination system technically behaves like other destination systems but is not displayed in the list of destination systems in the Management Console. Once the destination system is created, you receive a corresponding message and a flag informs you about its existence. You can only delete the ENP destination system if it is not in use as a destination of a notification.

Figure 22: Created ENP Destination System

5.3.4 Defining the ENP as Destination of a Notification

Create a static or a versioned notification for the agent instance you created previously:

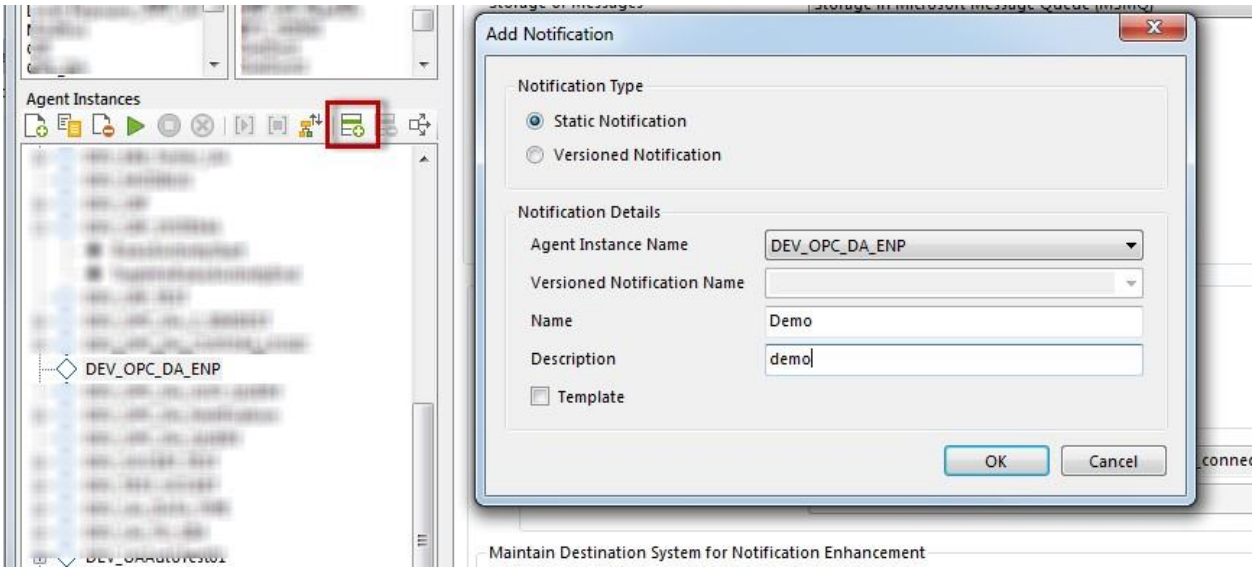


Figure 23: Add Notification to the Agent Instance

Select the notification, change to the *Destinations* tab, and add a destination to this notification. The */Enhanced Notification Processing* destination is offered in the dropdown box at the beginning of the list of all available destinations.

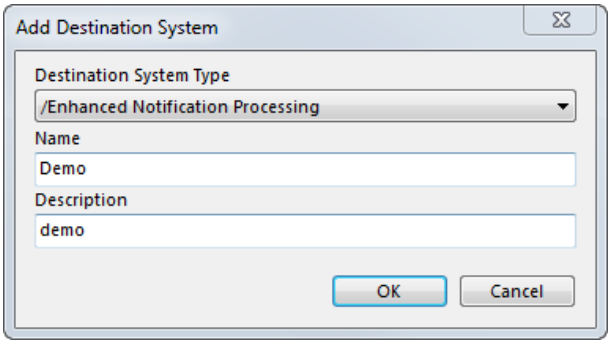


Figure 24: Adding a Destination to a Notification

5.3.5 Mapping of Modules and Variables

5.3.5.1 Mapping of Modules

Click on the *Module and Variable Assignment* tree node to access the configuration screen of the notification destination for modules and variables.

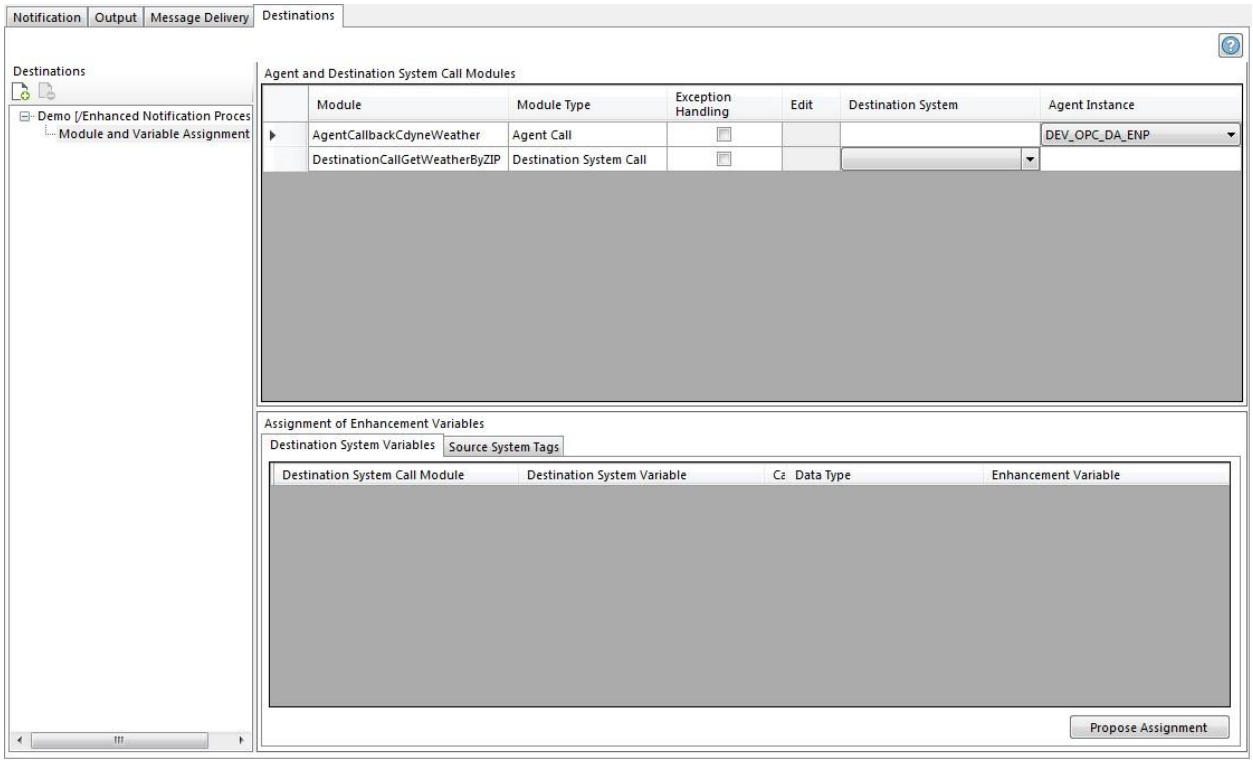


Figure 25: Open the Module and Variable Assignment Screen

In the table at the top of this screen, you can assign the destination systems and agents to the modules you defined in the enhancement implementation (see section 5.2.3). You can choose the destination system or agent by clicking on the corresponding dropdown box. The system only shows destination systems that are prepared to interact with the ENP.

5.3.5.2 Assignment of Enhancement Variables

Once you have selected a destination system in the module mapping part of the screen, the system shows you the destination system variables that you defined for your destination system.

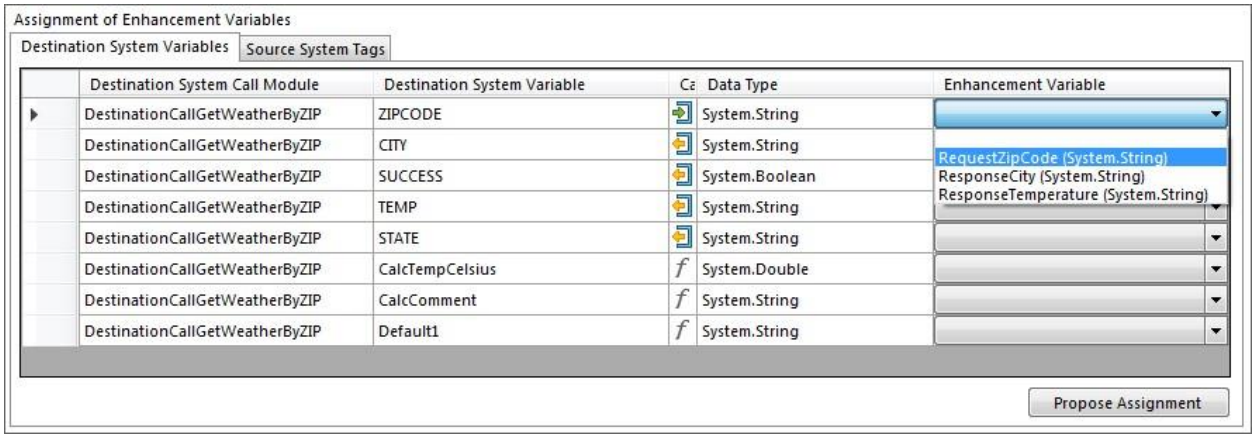


Figure 26: Assigning Enhancement Variables to Destination Variables

The system only displays the enhancement variables that you defined in your interface implementation (see section 5.2.3). You can only assign enhancement variables to destination system variables with matching data types. Enhancement variables with data types that do not match are not displayed by the system.

Note: The Web Service destination allows the definition of calculated variables. With calculated variables, you can convert the data type of Web Service response variables to the data types of enhancement variables.

5.3.5.3 Mapping of Source System Tags

After assigning an agent instance to an agent call module, you can browse for tags and assign them to enhancement variables. You can then, for example, write the contents of enhancement variables back to data tags of the source system. Change to the *Source System Tags* tab and insert or append a new row. Select an agent instance module from the dropdown box and start the browsing dialog.

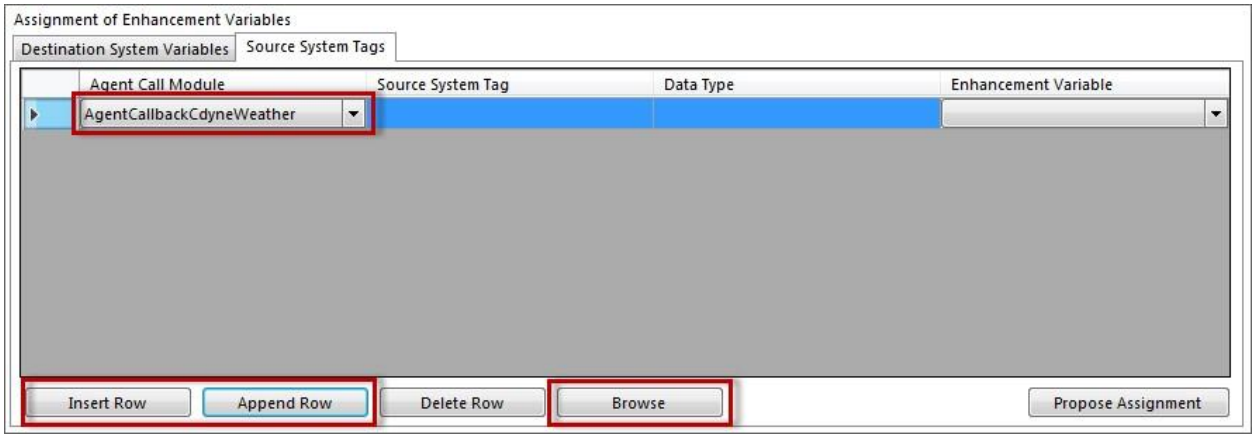


Figure 27: Start the Browsing Dialog for Tags of an Agent

Owned Enhancement

Select one or more tags from the tree and choose *Add Selected Items.*

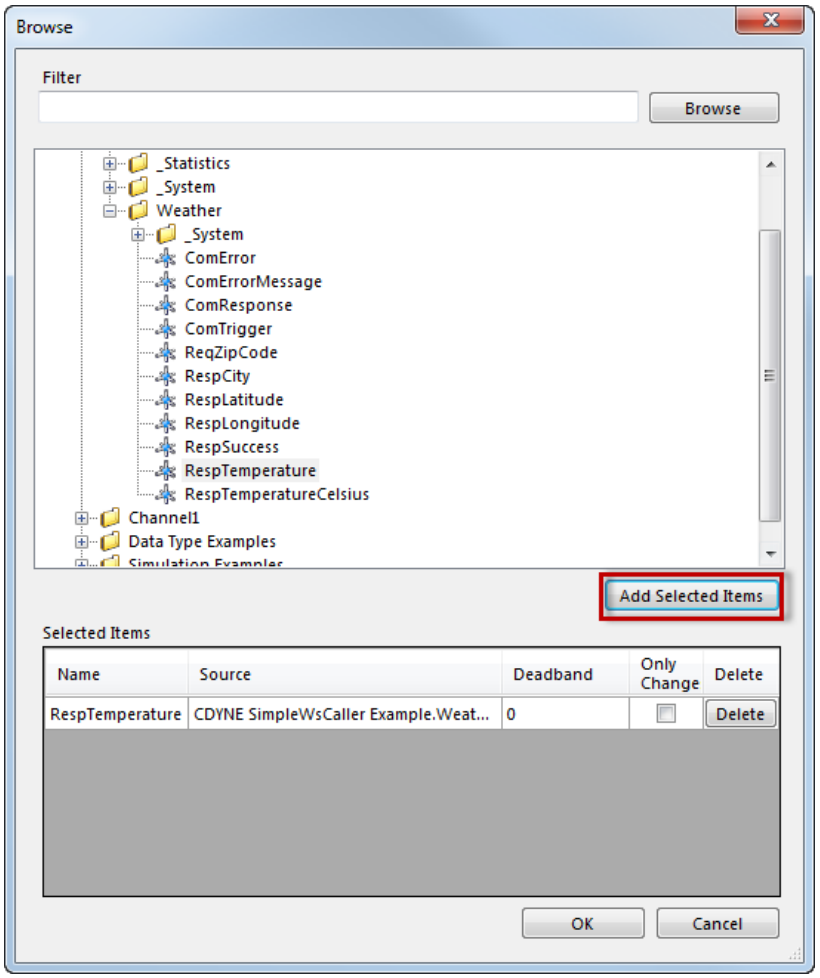


Figure 28: Browsing for Tags

When you close the browsing window with *OK*, the selected tags are copied into the configuration window. If you selected multiple tags, the system creates new rows accordingly. Assign the correct enhancement variables by choosing them from the dropdown box.

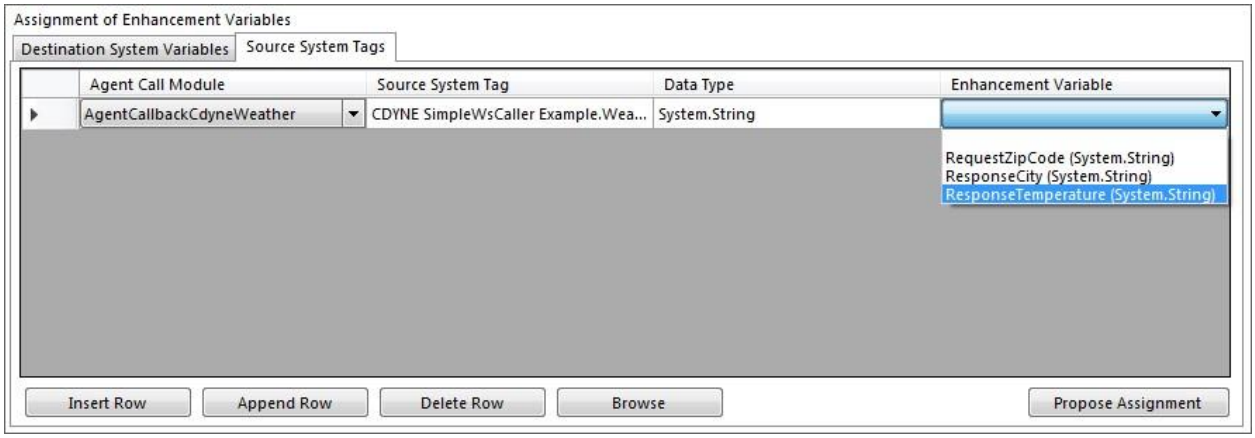


Figure 29: Assign Enhancement Variables to Source SystemTags

Note that the correct data type for the enhancement variable can only be determined immediately after browsing the tags. If you do not assign variables before leaving the screen, the rows without variable assignment cannot be configured later. In this case, delete the rows without data type information or browse again for tags.

5.4 Troubleshooting

5.4.1 Naming Conflicts

If the names or data types of modules and variables are changed after you have configured them in the Management Console, you may receive a warning message if you enter the configuration dialog again. The system automatically removes invalid configurations and allows you to correct the configuration.

5.4.2 Avoiding Thread Safety Problems

Unless you chose the option *Process Notification Messages Exactly Once in Order* in the agent instance configuration, the `Send` method of the ICustomLogic interface is called by different parallel dispatching threads. These parallel threads use the same instance of the class that implement the enhancement. Therefore global class attributes are used by all threads, which can cause unpredictable effects. It is recommended that you use local variables in the `Send` method only, except for references to agent callbacks, destination callbacks, ENP configuration, and the tracer. If access to global class attributes is unavoidable, try to enclose the coding parts with the keyword `lock`.

Owned Enhancement

6 Sample Customer-Owned Enhancement Implementation

6.1 Scenario

The sample customer-owned enhancement implementation that is described in this section demonstrates how to call a Web service from the PCo ENP. The goal is to realize a simple Web service call with as few technical prerequisites as possible. This means that you do not have to have an SAP ME system in place, for example. The implementation calls a free SOAP Web service instead, which is the **CDYNE Weather** Web service that provides you with weather information in the United States. For more information about the CDYNE Weather Web service, see http://wiki.cdyne.com/index.php/CDYNE_Weather.

We use the service operation GetCityWeatherByZIP that returns the up-to-date weather information for a city in the United States identified by its ZIP code. The ZIP code is contained in an XML input file that is passed to the enhancement implementation by a PCo File Monitor Agent. We then read the weather information from an output file created by the enhancement implementation.

To run the sample implementation properly, you need to be connected to the Internet to call the CDYNE Web service.

6.2 Implementation

As described in section 5, the implementation involves coding and configuration activities. We start with the coding activities in Visual Studio (see section 5.2).

1. Start Visual Studio, choose *File > New > Project*, and create a new solution named **`Weather`**. Choose **`.NET Framework 4.0`**. Use the *Class Library* template.

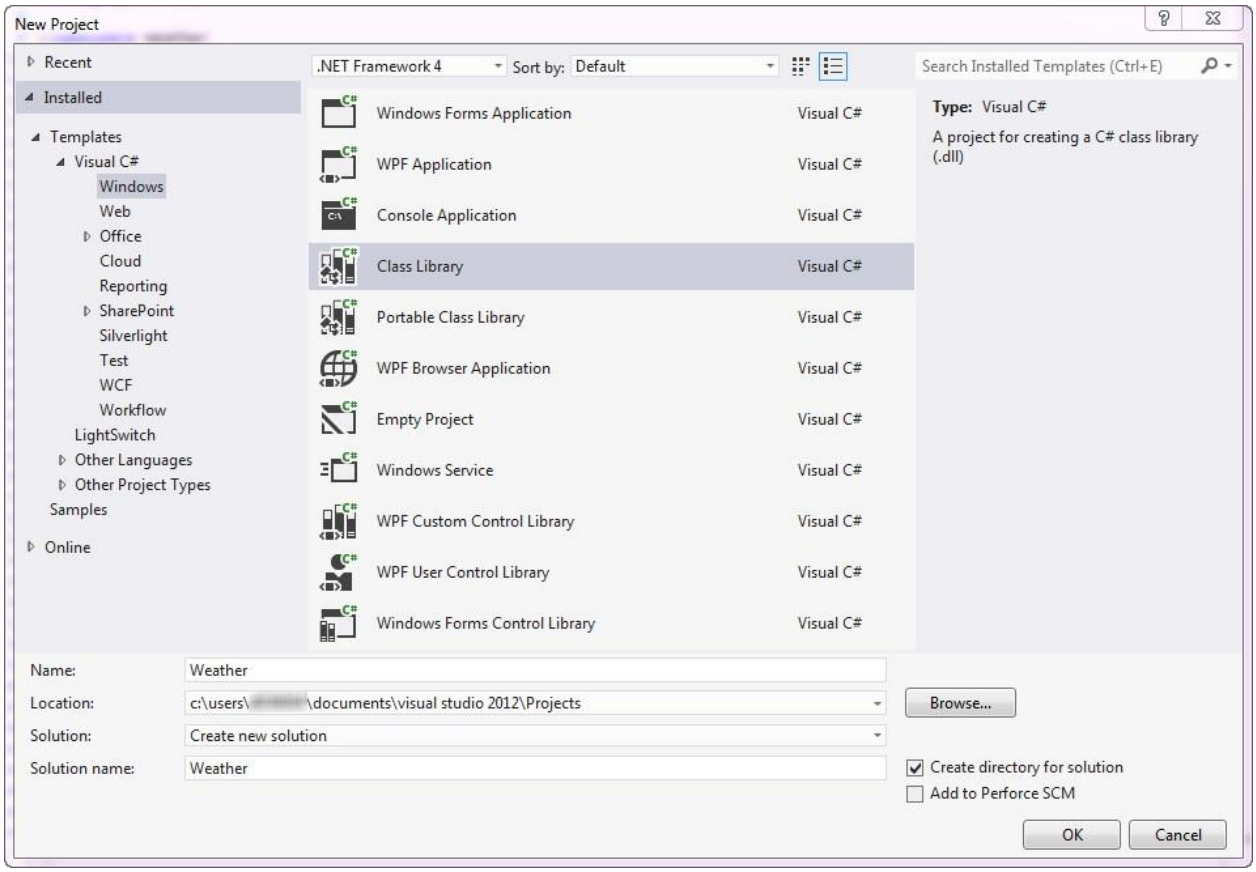


Figure 30: Create a New Solution

2. Set the properties of your project **`Weather`**. In the *Build* section, choose *Configurations*: **`All Configurations`**, and set *Platform target* to **`Any CPU`**.

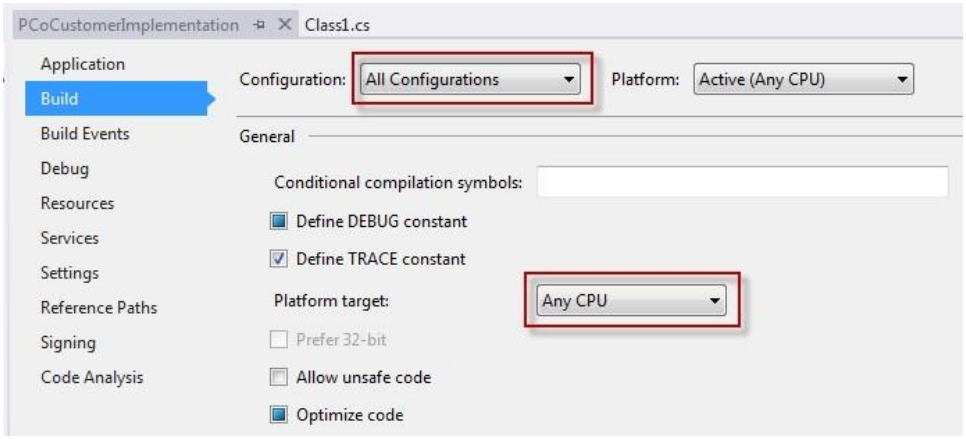


Figure 31: Set the Build Configuration of Your Project

Create a reference to the PCo ENP DLL. In the solution explorer, right-click on the *References* folder below the project `Weather`, choose *Add Reference*, go to the *Browse* tab, and choose the *CustomLogic.dll* from the *System* subfolder of the PCo program folder.

The *References* branch of your project should now contain the reference to `CustomLogic`.

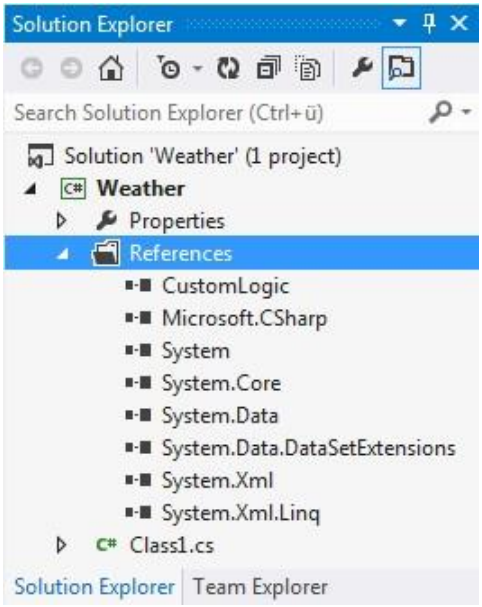


Figure 32: Project References

3. Copy the complete coding of section 0 into the `Class1.cs` file.

Let us analyze the coding briefly before we continue with the implementation.

The first region `Internal constants for modules, variables, and others` contains definitions of constants for the ENP module names, enhancement variable names, destination system variable names, notification message output expression, and the XML tags used in the input file. It is good practice to use constants for the names later on in the coding rather than literals, since some of the names are used multiple times.

The `Weather` class, which implements the `ICustomLogic` interface, resides in the second region `Customer-Owned implementation of the ICustomLogic interface`. The `Weather` class declares some private member attributes that are used to keep references to the agent callback, destination callback, ENP configuration data, and the tracer. All of these references are passed to the ENP in the `Initialize()` method. You can use the tracer object to write messages into the agent log. In addition, two dictionaries are declared that keep the module and enhancement variable names.

Inside the `Weather` class, the `Public interface methods to be defined` region keeps the implementation of the public interface methods. The first public method is the parameter-less class constructor where the dictionaries for enhancement modules and variables are filled. The constructor calls the private methods `initLogicModulesSet()` and `initVariablesSet()`.

In method `initLogicModulesSet()`, we define a destination system call module named `DestinationCallGetWeatherByZIP` and an agent call module `AgentCallbackCdyneWeather`.

In method `initVariablesSet()`, we define one enhancement variable for the request message (`RequestZipCode`) and three enhancement variables for the message response (`ResponseSuccess`, `ResponseTemperature`, and `ResponseCity`).

In the next public method `Initialize()`, we receive references that we copy to the class members `agentCallback`, `destinationCallback`, `enhancementConfig`, and `enhancementTracer` for later use. `Initialize()` is called when the agent is started.

The two public methods `GetLogicModules()` and `GetCustomVariables()` just pass the module and enhancement variable dictionaries to the caller. The caller could be the PCo management console during design time or the ENP framework during runtime.

The public method `Send()` holds the processing sequence of your customer-owned enhancement implementation. It is called when a notification message is to be delivered to the Web Service Destination. Here you could parse the contents of a notification message, map information from the notification message to Web service fields, call Web services or other destination systems, analyze the destination systems responses, or write information back to the agent.

Our implementation performs the following steps:

1. **Extract the ZIP code from the notification message**. Since we are using a FileMonitorAgent (see below), the notification message contains the complete content of the input file. In the input file, the ZIP code is enclosed in XML tags. Therefore an XML reader is used to extract the ZIP code from the input file.
2. **Call the Web service**. First we retrieve the destination system variables from the ENP configuration, then we build the `destinationVariableValues` dictionary where we assign the ZIP code value to the destination system variable `RequestZipCode`. Finally, we call the Web service operation.
3. **Evaluate the Web service response and create the output file**. The Web service response is enclosed in the `result` dictionary. This dictionary contains the destination system response variables (`ResponseSuccess`, `ResponseTemperature`, and `ResponseCity`) as keys together with their values. These response values are written to a file named `weather.txt` in the `Desktop` folder of your local file system.

   If you passed a valid ZIP code (for example, 12345) to the Web service operation and if the call was successful, a message like this is written to the file:

   ```
   11.07.2012 11:55:12: Weather for 12345 Schenectady: Temperature 75F
   ```

   Otherwise, for an invalid ZIP code like 11111, the file contains a message like this:

   ```
   11.07.2012 11:55:12: Invalid ZIP code 11111
   ```

   The output file is created using the `AppendFileExecute()` method of the agent callback reference. Before creating the output, we delete an existing file using the `DeleteFileExecute()` method of the agent callback.

Now we continue with the implementation.

4. After you have implemented the coding correctly, you have to build the ENP DLL. In order to do that, choose *Build > Build Solution* in Visual Studio. Check the output path of your build configuration. This is where you should find your ENP DLL `weather.dll`.

   For DLLs that are to be used in production, SAP recommends copying the DLL into the System folder of the PCo installation. For our test example, we can keep the DLL where it is, and reference to the current path later on in the agent configuration.

We have finished the implementation part in Visual Studio. Now open the Plant Connectivity Management Console and start configuring a Web Service Destination and an agent instance. Remember that we want to read an XML file from a directory that is monitored by a File Monitor Agent.

5. On your local file system, create two directories, preferably on your Windows Desktop. Name the first folder **`FileMonitorFolder`** and the second folder **`FileMonitorFolderProcessed`**.
6. Create a new source system of connection type `File Monitor Agent`. Set the agent properties as follows:

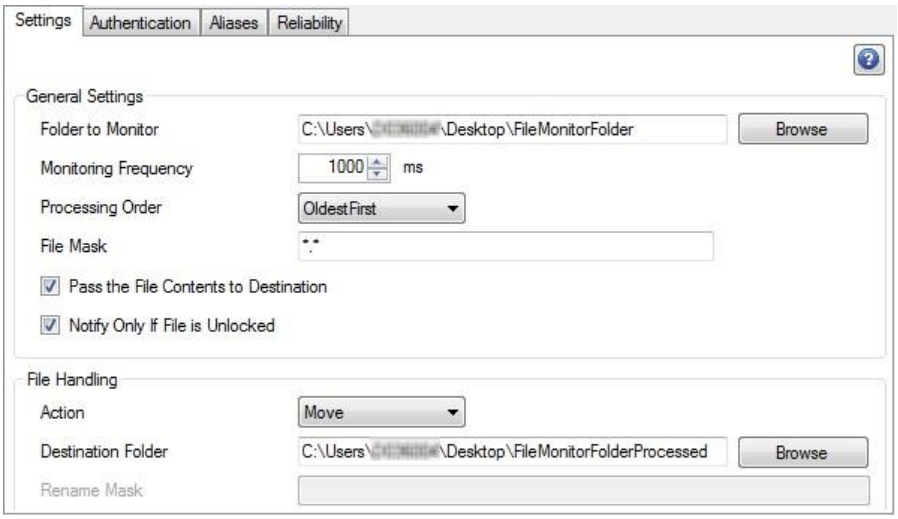


Figure 33: File Monitor Agent Configuration

You can leave the other tabs *Authentication*, *Aliases*, and *Reliability* with their default settings.

7. Create a destination system of type `Web Service Destination`. Enter the following URL to the WSDL of the Web service:

   `http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl`

   No authentication is required for the CDYNE weather service. Then press *Retrieve Services*. If the CDYNE weather service could be contacted, the available endpoint URL and service name should be displayed in the table *Service Bindings*. Save the new destination system before you proceed.

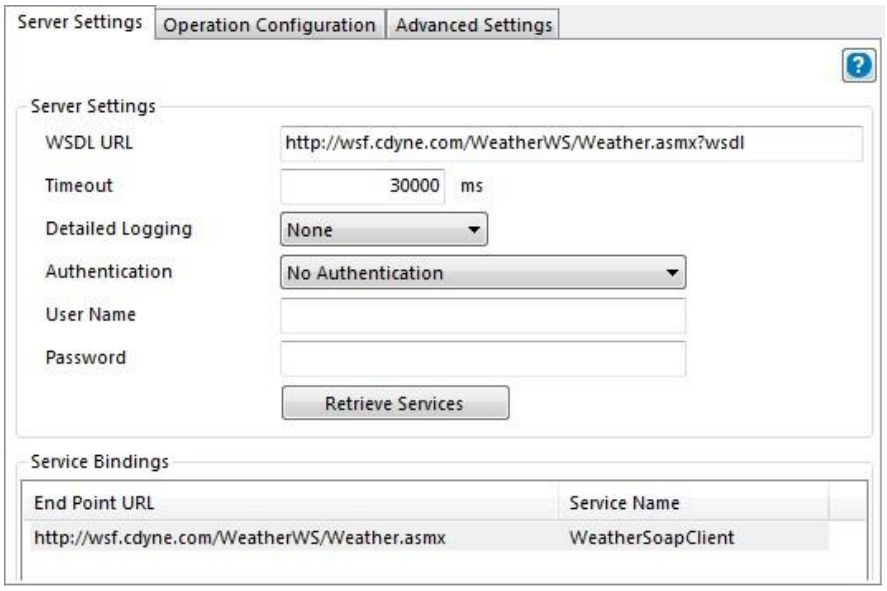


Figure 34: Web Service Destination Server Settings

8. Now do the destination variable mapping. Change to the *Operation Configuration* tab of the new destination system, and select the service operation `GetCityWeatherByZIP`. Then click *Create Request Message*. Remember that we defined the following destination variable names in our implementation coding:

```
internal struct DestinationVariableNames
```



```
    {          internal const string RequestZipCode =
"ZIPCODE";          internal const string ResponseSuccess =
"SUCCESS";          internal const string
ResponseTemperature = "TEMP";          internal const string
ResponseCity = "CITY";
    }
```

So we use the same variables in the variable mapping configuration UI. First do the mapping of the `ZIP` field of the Web service request message to the destination variable name `ZIPCODE`. Do not forget to select the `Variable` checkbox. Check if the destination variable `ZIPCODE` appears in the list of variables.

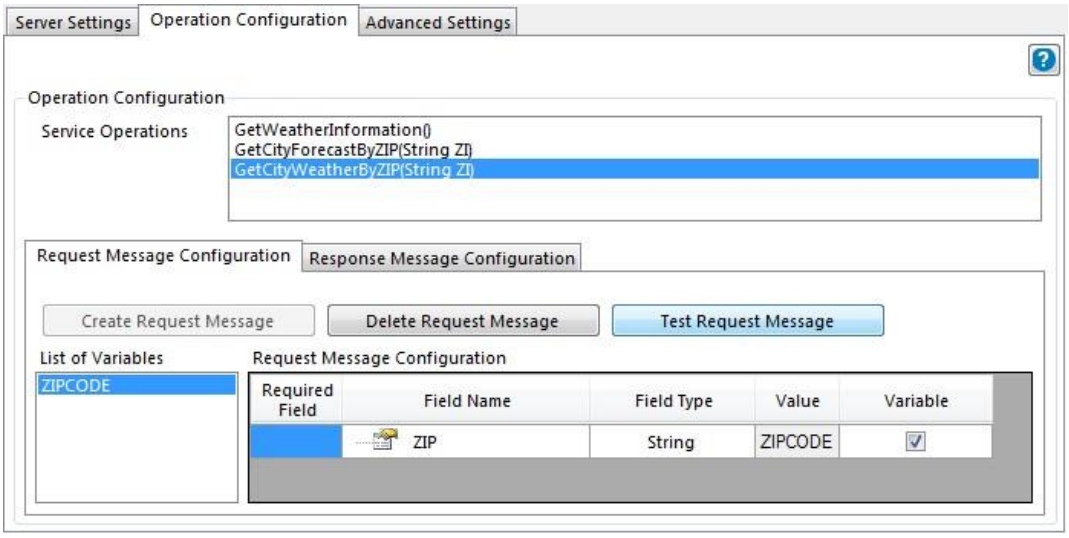


Figure 35: Web Service Destination Request Message Configuration

Before you do the response message configuration, you have to press *Test Request Message*. When you are prompted for a ZIP code, enter a valid United States ZIP code, such as 12345, and then press *OK*.

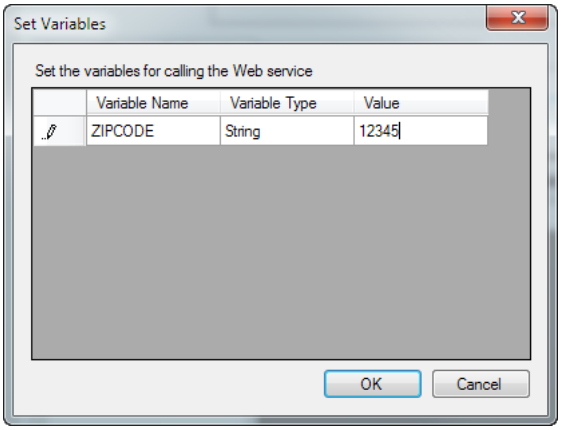


Figure 36: Test Your Request Message

You should now see a list of response message field names. Assign the destination variables `SUCCESS`,

`TEMP`, and `CITY` to the corresponding response field names. Check if these three destination variables appear in the list of variables.

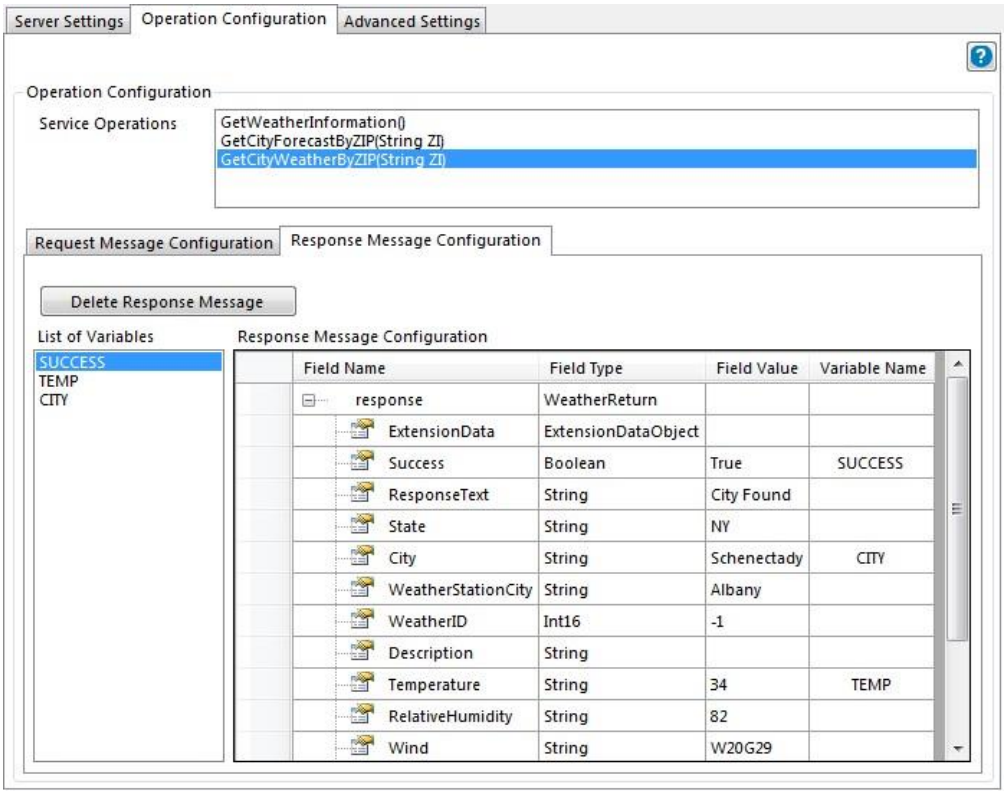


Figure 37: Web Service Destination Response Message Configuration

Now the destination system configuration is done. Save the destination system before you proceed.

9. Create an agent instance for the source system. On the *Host* tab, make the required settings for the execution of the agent service. Enter your own service user name and password to ensure that the output weather file is written into your Desktop folder. Set the *Log Level* to **`Verbose`**.

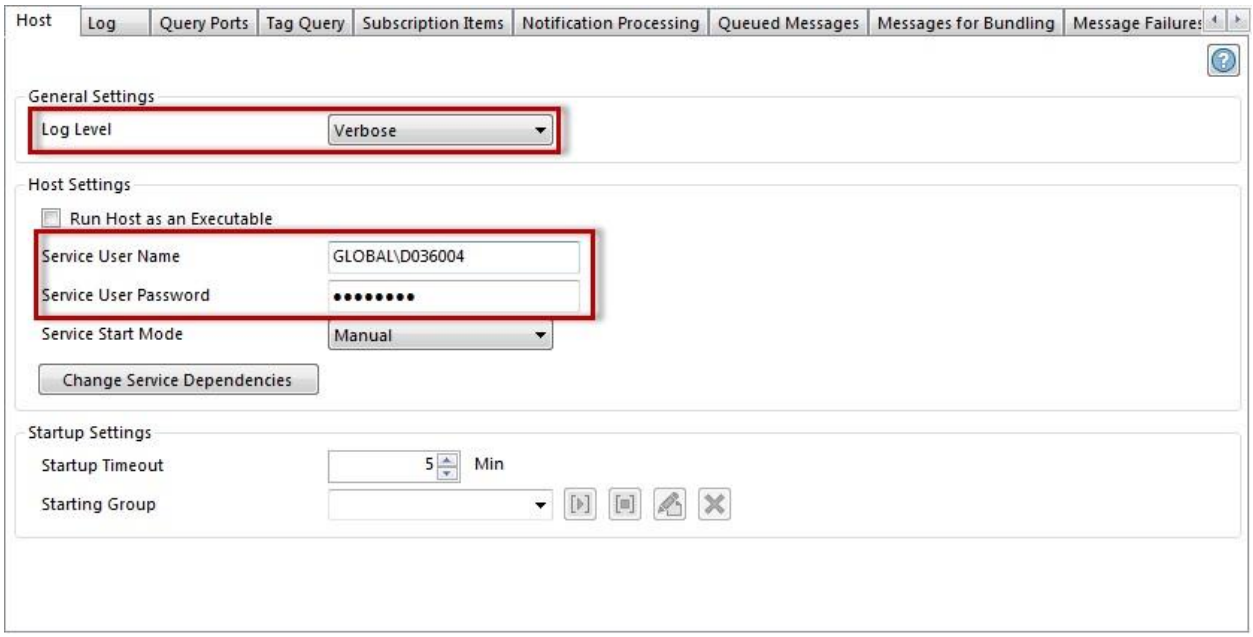


Figure 38: Agent Configuration

10. Go to the *Subscription Items* tab of your agent instance. Add the subscription item `ReceiveDataFileContent` that is provided by the File Monitor Agent.

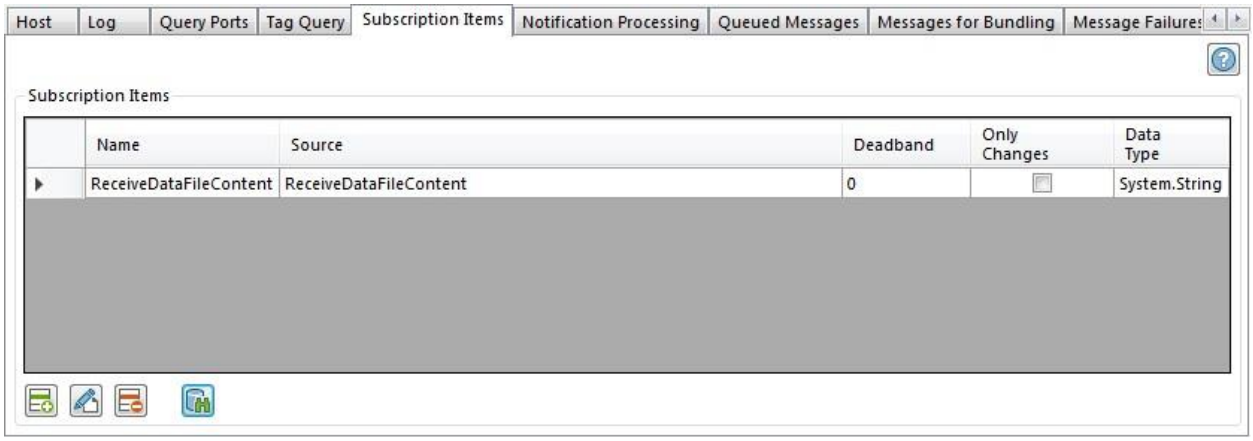


Figure 39: Add a Subscription Item to the Agent

11. Go to the *Notification Processing* tab of your agent instance. Select the *Process Notification Messages Exactly Once in Order* option. Select *Customer-Owned Enhancement*, and then enter the path to your ENP DLL in the *Dynamic Link Library* field. After doing that, the `Weather.Weather` class should be displayed in the *Class* field. If there is no class displayed, you probably chose a DLL that does not implement the `ICustomLogic` interface.

    Then press *Create Destination System*. A destination system is created from your DLL. The status should be *Destination system created.*

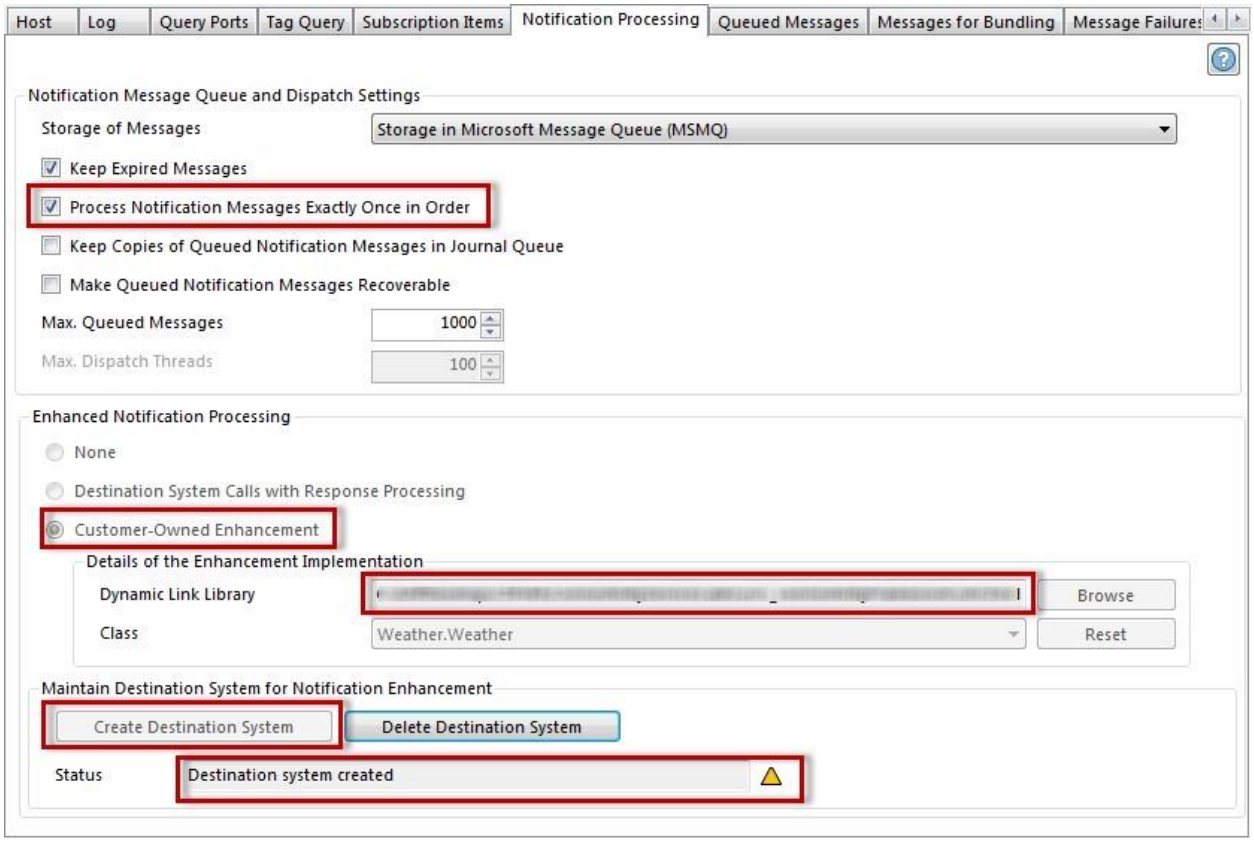


Figure 40: Link the ENP DLL

12. Add a static notification to your agent instance. Change to the *Output* tab of the notification, and add the subscription item to the expression list by pressing *Generate Expressions*.

Figure 41: Choose an Output Expression

Remember that we defined a constant for this expression in our implementation.

```
internal struct NotificationOutputExpressionNames
    {         internal const string NotifMsgContent =
"ReceiveDataFileContent";
```



```
}
```

Therefore, the output expression has to be named `ReceiveDataFileContent`, too.

13. Go to the *Destinations* tab of the notification. Add the destination system name **`/Enhanced Notification Processing`** to your notification. Then do the module and variable assignments. Here the enhanced notification processing module and variable names that you defined in your implementation as well as the destination system and agent instance come into play.

    First assign your Web Service destination system and your agent instance to the agent and destination system call modules, for example:

Figure 42: Assign Agent and Destination System Call Modules

Then do the assignment of destination variables to the enhancement variables that you defined in your implementation. You do not have to maintain source system tags for this scenario.

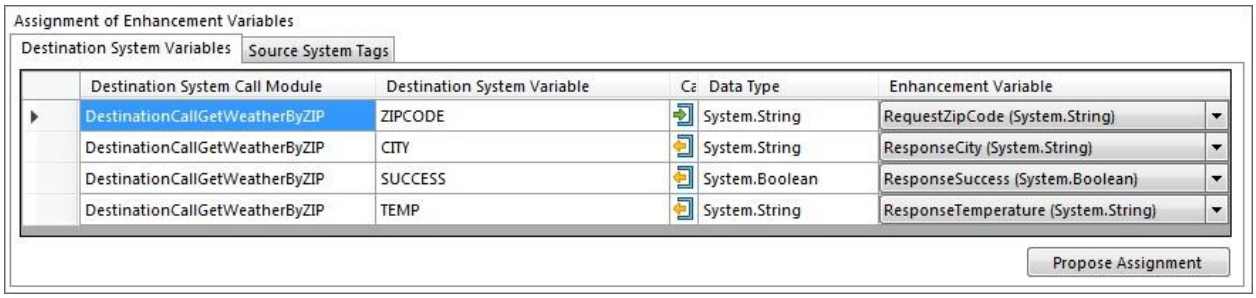


Figure 43: Assign Enhancement Variables

Finally save your agent configuration.

Now you have concluded all the implementation and configuration tasks. Start the agent instance and see if it runs. Of course, there is no notification message sent to the enhancement implementation, since you have not yet moved an XML file to the File Monitor Agent folder. But a running agent at this point in time indicates that you carried out the configuration correctly.

14. Create an XML file that contains the ZIP code. First create the file outside the `FileMonitorFolder` folder. Use a common text editor, such as Microsoft Notepad, and create a text file with an arbitrary name (for example, `ZIP.txt)`. Copy the following text to the file:

    ```
    <ZIP>12345</ZIP>
    ```

    As you can see, the ZIP code 12345 is enclosed in two tags `<ZIP>` and `</ZIP>`. Remember that we defined a constant for the tag `ZIP` in our implementation.

    ```
    internal struct XMLTags
        {         internal const string ZIP =
    "ZIP";
        }
    ```

    So we have to use the `ZIP` tag in our XML file, too. Save the file.

15. Now move the XML to the `FileMonitorFolder` folder. The File Monitor Agent should now start processing the file and move it to the `FileMonitorFolderProcessed` folder afterwards. The agent then passes the contents of the XML file to the ENP as a notification message where it is processed according to the enhancement implementation. If everything went well, you should find a `weather.txt` file on your Windows Desktop after a few seconds. Open the file and check if it contains the weather information, for example:

    ```
    09.06.2015 06:17:16: Weather for 12345 Schenectady: Temperature 45F
    ```

6.3 Sample Coding

This is the complete coding of the `Class1.cs` file that is used for the sample implementation:

```
using System; using System.Collections.Generic; using
System.Text; using System.Diagnostics; using
SAP.Manufacturing.CustomLogic; using System.Xml; using
System.IO;

namespace Weather
{
  #region Internal constants for modules, variables, and others
  /// <summary>
  /// Definition of agent and destination system call module names. These  /// are the modules that
appear in the "Module and Variable Assignment"  /// view inside the notification destination screen.
  /// </summary>  internal struct CustLogicModuleNames
  {
    internal const string DestinationCall = "DestinationCallGetWeatherByZIP";  internal const string
AgentCallback = "AgentCallbackCdyneWeather";
  }

  /// <summary>
  /// Definition of enhancement variables. These are the variables that appear
  /// in the "Module and Variable Assignment" view inside the notification  /// destination screen.
/// </summary>  internal struct CustLogicVariableNames
  {
    internal const string RequestZipCode = "RequestZipCode";  internal const string
ResponseSuccess = "ResponseSuccess";  internal const string ResponseTemperature =
"ResponseTemperature";  internal const string ResponseCity = "ResponseCity";
  }

  /// <summary>
  /// Constants for the destination system variables. Use the same values
  /// in the "Module and Variable Assignment" view inside the notification
  /// destination screen, and in the web service destination request and  /// response message configuration screen.
  /// </summary>  internal struct DestinationVariableNames
  {
    internal const string RequestZipCode = "ZIPCODE";  internal const string ResponseSuccess =
"SUCCESS";  internal const string ResponseTemperature = "TEMP";  internal const string
ResponseCity = "CITY";
  }
```

```
  /// <summary>
  /// Constants that represent the output expression that are shown in the    /// notification output tab.    ///
</summary>    internal struct NotificationOutputExpressionNames
  {
    internal const string NotifMsgContent = "ReceiveDataFileContent";
  }

  /// <summary>
  /// Constants for the tags in the input XML file
  /// </summary>    internal struct XMLTags
  {
    internal const string ZIP = "ZIP";
  }

  #endregion

  #region Customer-owned implementation of the ICustomLogic interface
  /// <summary>
  /// Customer-owned implementation of the ICustomLogic interface.
  /// </summary>    class Weather : ICustomLogic
  {
    private IAgentCallback agentCallback;       private IDestinationCallback destinationCallback;
private ICustomLogicConfiguration enhancementConfig;       private CustomLogicTracer
enhancementTracer;

    private Dictionary<string, ModuleType> modules;       private
Dictionary<string, Type> enhancementVariables;

    // ===================================================================

    #region Public interface methods to be defined
    /// <summary>
    /// The default constructor required for the instantiation.
    /// </summary>       public
Weather()
    {
      this.modules = this.initLogicModulesSet();          this.enhancementVariables =
this.initVariablesSet();
    }

    /// <summary>
    /// Initialize the enhancement with a reference to the agent callback,       /// destination
callback, enhancement variable configuration, and a tracer       /// (logging) instance.
```



```
    /// </summary>
    /// <param name="agentCallback">Agent callback</param>
    /// <param name="destinationCallback">Destination callback</param>
    /// <param name="enhancementConfig">Enhancement configuration</param>        /// <param
name="enhancementTracer">Tracer instance for logging</param>        void ICustomLogic.Initialize(
      IAgentCallback agentCallback,
      IDestinationCallback destinationCallback,
      ICustomLogicConfiguration enhancementConfig,
      CustomLogicTracer enhancementTracer)
    {
      this.agentCallback = agentCallback;           this.destinationCallback =
destinationCallback;           this.enhancementConfig = enhancementConfig;
this.enhancementTracer = enhancementTracer;
    }

    /// <summary>
    /// Returns the module names which identify used logic modules (agents
    /// or destinations). This method is called during the design time by the
    /// configuration dialog of the Management Console.
    /// </summary>
    /// <returns>Map of module names to their types</returns>
    Dictionary<string, ModuleType> ICustomLogic.GetLogicModules()
    {
      return this.modules;
    }

    /// <summary>
    /// Returns a map of enhancement variables to their types. Each variable
    /// can be mapped to destination system variables for the destination call.
    /// The key in the map is the name of the enhancement variable, and the        /// value is the data type of the
variable.
    /// </summary>
    /// <param name="notificationID">Notification ID. Should be evaluated if the
    /// scenario contains more than one notification. In such a case, only the
    /// variables that belong to that specific notification should be
    /// returned</param>
    /// <returns>Map of enhancement variables to their types</returns>
    Dictionary<string, Type> ICustomLogic.GetCustomVariables(Guid notificationID)
    {
      return this.enhancementVariables;
    }

    /// <summary>
```

```
    /// Synchronuous method which processes the notification message raised        /// by the agent. The user's code
resides within this implemented method.
    /// </summary>
    /// <param name="notification">Notification message</param>        /// <returns>True if the processing
was successful</returns>        bool ICustomLogic.Send(CustomLogicNotificationMessage notification)
    {

        Dictionary<string, object> enhancementVariableValues =                    new Dictionary<string,
object>();

        // ------------------------------------------------------
        // Extract the ZIP code from the notification message
        // The notification message content has to have the
        // format
        //
```



```
// <ZIP>[ZIP Code]</ZIP>, e.g. <ZIP>12345</ZIP>
//
// The ZIP code has to be a valid US postal code.
// ----------------------------------------------------          try
{
  this.enhancementTracer.TraceMessage(
    TraceEventType.Verbose, "Parse the notification message");

  string inputXML =                notification.DataItems[
      NotificationOutputExpressionNames.NotifMsgContent
        ].Value.ToString();

  XmlReader reader = XmlReader.Create(new StringReader(inputXML));          bool startZip = false;

  while (reader.Read())
  {
    switch (reader.NodeType)
    {
      case XmlNodeType.Element:
        if (reader.Name.Equals(XMLTags.ZIP))
          startZip = true;                    break;

      case XmlNodeType.Text:                    if (startZip == true)
          enhancementVariableValues.Add(
            CustLogicVariableNames.RequestZipCode,
reader.Value.ToString());                    break;

      case XmlNodeType.EndElement:
        if (reader.Name.Equals(XMLTags.ZIP))
          startZip = false;                    break;
    }
  }
}
catch (Exception x)
```



```
{
  this.enhancementTracer.TraceException(
TraceEventType.Error, x.Message, x);
this.enhancementTracer.TraceMessage(TraceEventType.Error,
    "Parsing failed. Check if the output value 'ReceiveDataFileContent' is spelled correctly, and if the input file is
well formatted.");          throw new CustomLogicException("Parsing failed");
```

```
        }

        // ---------------------------------------------
        // Call the destination system (= web service)
        // ---------------------------------------------
Dictionary<string, object> result = null;

try
        {
          this.enhancementTracer.TraceMessage(TraceEventType.Verbose,
            "Call web service");

          // Get the destination system variables and their value for parametric
          ///web service call
          Dictionary<string, object> destinationSystemVariableValues
            = new Dictionary<string, object>();
          Dictionary<string, string> variableMap
            = enhancementConfig.GetModuleVariables(
notification.NotificationDestinationID,
              CustLogicModuleNames.DestinationCall);

          foreach (string destinationSystemVariable in variableMap.Keys)
          {
            if (destinationSystemVariable ==
                DestinationVariableNames.RequestZipCode)
            {
              destinationSystemVariableValues.Add(
variableMap[destinationSystemVariable],                enhancementVariableValues[
variableMap[destinationSystemVariable]]);
            }
          }

          result = this.destinationCallback.CallModule(
            CustLogicModuleNames.DestinationCall,
            notification,
            destinationSystemVariableValues);
        }
        catch
        {
          result = null;
        }

        if (result == null)
        {
```

```
        this.enhancementTracer.TraceMessage(TraceEventType.Verbose,
          "Web service call failed");          throw new CustomLogicException("Web service call failed");
      }

      // ------------------------------------------------------------------
      // Evaluate the destination system response and create the output file
      // ------------------------------------------------------------------          List<QueryMessage> queryMsg;
      string path =
        Environment.GetFolderPath(Environment.SpecialFolder.Desktop)
        + "\\weather.txt";

      bool success = false;        string city = "";        string
temp = "";
       try
      {
                        city = result[CustLogicVariableNames.ResponseCity].ToString();          temp =
                              result[CustLogicVariableNames.ResponseTemperature].
          ToString();
        success = (bool)result[CustLogicVariableNames.ResponseSuccess];
      }
      catch
      {
        success = false;
      }

      string outtext = "";
```



```
      if (success == true)          outtext =
String.Format(
          "{0}: Weather for {1} {2}: Temperature {3}F\n",
DateTime.UtcNow.ToString(),          enhancementVariableValues[
            CustLogicVariableNames.RequestZipCode],
city,          temp);        else
        outtext = String.Format(
          "{0}: Invalid ZIP code {1}.\n",          DateTime.UtcNow.ToString(),
enhancementVariableValues[
            CustLogicVariableNames.RequestZipCode]);

      Dictionary<string, LogicModuleStruct> custLogicModuleConfig =
this.enhancementConfig.GetLogicModuleConfiguration(          notification.NotificationDestinationID);
```

```
        this.agentCallback.DeleteFileExecute(              custLogicModuleConfig[
            CustLogicModuleNames.AgentCallback].agentName,              path,
out queryMsg);          this.agentCallback.AppendFileExecute(
custLogicModuleConfig[
            CustLogicModuleNames.AgentCallback].agentName,              path,
outtext,            out queryMsg);

        return true;
    }

    #endregion

    // ===================================================================

    #region Private helper methods
    /// <summary>
    /// Map module names to their types.
    /// </summary>
    /// <returns>Map of modules names to their types.</returns>        private Dictionary<string,
ModuleType> initLogicModulesSet()
    {
        Dictionary<string, ModuleType> modules =              new Dictionary<string,
ModuleType>();

        modules.Add(
          CustLogicModuleNames.DestinationCall,
ModuleType.Destination);          modules.Add(
          CustLogicModuleNames.AgentCallback,
          ModuleType.Agent);

        return modules;
    }

    /// <summary>
    /// Map custom-logic variables to their types.
    /// </summary>
    /// <returns>Map of enhancement variables to their types.</returns>        private Dictionary<string, Type>
initVariablesSet()
    {
        Dictionary<string, Type> enhancementVariables =              new Dictionary<string,
Type>();

        enhancementVariables.Add(CustLogicVariableNames.RequestZipCode,              typeof(string));
```

```
        enhancementVariables.Add(CustLogicVariableNames.ResponseSuccess,        typeof(bool));
        enhancementVariables.Add(CustLogicVariableNames.ResponseTemperature,    typeof(string));
        enhancementVariables.Add(CustLogicVariableNames.ResponseCity,           typeof(string));
return enhancementVariables;
      }
      #endregion
   }
   #endregion
}
```

www.sap.com/contactsap

Material Number