# Contents

## Introduction

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What’s in this manual</td>
<td>2</td>
</tr>
<tr>
<td>What’s new in this release?</td>
<td>2</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>3</td>
</tr>
<tr>
<td>Symbols and conventions</td>
<td>3</td>
</tr>
<tr>
<td>Product support</td>
<td>4</td>
</tr>
</tbody>
</table>

## Part 1: Web Service Basics

<table>
<thead>
<tr>
<th>Chapter 1: Dynamics GP Web Service Overview</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is a web service?</td>
<td>7</td>
</tr>
<tr>
<td>Web service benefits</td>
<td>7</td>
</tr>
<tr>
<td>What the Dynamics GP Web Service provides</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2: Web Service Architecture</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web service foundation</td>
<td>9</td>
</tr>
<tr>
<td>Configurations</td>
<td>10</td>
</tr>
<tr>
<td>Security for web services</td>
<td>11</td>
</tr>
<tr>
<td>Policy for web services</td>
<td>12</td>
</tr>
<tr>
<td>Exception logging</td>
<td>12</td>
</tr>
</tbody>
</table>

## Part 2: Getting Started

<table>
<thead>
<tr>
<th>Chapter 3: Endpoints</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing the endpoint to use</td>
<td>15</td>
</tr>
<tr>
<td>Endpoint URLs</td>
<td>15</td>
</tr>
<tr>
<td>WSDL</td>
<td>17</td>
</tr>
<tr>
<td>WSDL generation</td>
<td>17</td>
</tr>
<tr>
<td>Proxies</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4: Connecting to the Legacy Endpoint</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio 2005</td>
<td>21</td>
</tr>
<tr>
<td>Visual Studio 2008</td>
<td>22</td>
</tr>
<tr>
<td>Web service namespace</td>
<td>24</td>
</tr>
<tr>
<td>Creating a service instance</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5: Connecting to the Native Endpoint</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio 2008</td>
<td>25</td>
</tr>
<tr>
<td>Application configuration file</td>
<td>27</td>
</tr>
<tr>
<td>Using SvcUtil.exe</td>
<td>29</td>
</tr>
<tr>
<td>Service namespace</td>
<td>29</td>
</tr>
<tr>
<td>Creating a service instance</td>
<td>30</td>
</tr>
<tr>
<td>Closing the service instance</td>
<td>30</td>
</tr>
</tbody>
</table>
CONTENTS

Chapter 6: Dynamics GP Service Object Model ................................................. 31
  Object categories .......................................................................................... 31
  Inheritance ..................................................................................................... 32
  Learning the object model ........................................................................... 33
  Endpoint differences .................................................................................... 33
  Dynamics GP Web Service Reference ......................................................... 34

Chapter 7: Methods ......................................................................................... 35
  Method categories ........................................................................................ 35
  Framework methods .................................................................................... 36

Chapter 8: Data types ...................................................................................... 37
  Standard data types .................................................................................... 37
  Enumerations .............................................................................................. 38
  Account numbers ....................................................................................... 38
  Choice types .............................................................................................. 39

Chapter 9: Context .......................................................................................... 41
  Organization key ........................................................................................ 41
  Culture ........................................................................................................ 41
  Currency ..................................................................................................... 41
  Role key ..................................................................................................... 42
  WorkOnBehalfOf ....................................................................................... 42

Part 3: Using the Web Service ........................................................................ 44

Chapter 10: Creating Objects ......................................................................... 45
  Create methods .......................................................................................... 45
  Create policy ............................................................................................. 47

Chapter 11: Retrieving Objects ..................................................................... 53
  Retrieve methods ....................................................................................... 53
  Individual objects ..................................................................................... 53
  Lists of objects .......................................................................................... 55
  Restriction reference ................................................................................ 61

Chapter 12: Updating Objects ........................................................................ 65
  Update methods ........................................................................................ 65
  Update policy ............................................................................................ 69
  Concurrency ............................................................................................... 71

Chapter 13: Deleting or Voiding Objects ....................................................... 77
  Delete or void methods ............................................................................. 77
  Delete or void policy ................................................................................ 79

Chapter 14: Exceptions .................................................................................. 83
  Exceptions console .................................................................................. 83
  System exceptions .................................................................................... 85
  Validation exceptions ................................................................................ 86
  Handling exceptions in code ..................................................................... 86
Chapter 15: Policy ................................................................. 93
  Policy overview ........................................................................ 93
  Configuring from the Dynamics Security Console ...................... 94
  Configuring from code .............................................................. 94
  Using policy in a web service application ................................... 95

Chapter 16: Other Programming Techniques .............................. 99
  Calling methods asynchronously ............................................... 99
  Working on behalf of another user ............................................ 101

Chapter 17: Entity ID Filtering .................................................. 103
  Setting up entity ID filtering .................................................... 103
  Security for entity ID filtering .................................................. 104
  Implementing entity ID filtering ............................................... 105
  Filtering limitations ................................................................. 106
  Entity ID filtering example ..................................................... 106

Chapter 18: Entity Action Tracking .......................................... 111
  Overview of entity action tracking ........................................... 111
  Enabling entity action tracking ................................................. 111
  Retrieving action tracking information ..................................... 113
  Entity action tracking example ............................................... 115
  Considerations for entity action tracking ................................ 118

Chapter 19: Optimized Proxy .................................................... 119
  Using a predefined optimized interface .................................... 119
  Available interfaces ............................................................... 120
  Configuring the endpoint ....................................................... 120
  Using the optimized interface ................................................ 121

Chapter 17: Entity ID Filtering .................................................. 103
  Setting up entity ID filtering .................................................... 103
  Security for entity ID filtering .................................................. 104
  Implementing entity ID filtering ............................................... 105
  Filtering limitations ................................................................. 106
  Entity ID filtering example ..................................................... 106

Part 4: Extending the Web Service ............................................. 124

Chapter 20: Overview ............................................................. 125
  How objects can be extended .................................................. 125
  Parts of an object extension .................................................... 126
  Creating an object extension .................................................. 126

Chapter 21: Dynamics GP Service Events ................................. 127
  How events are processed ...................................................... 127
  Event list ................................................................................ 127

Chapter 22: Extension Assembly .............................................. 131
  Creating an extension assembly .............................................. 131
  Event handler methods .......................................................... 132
  Sample: Retrieved event handler method .................................... 134
  Sample: ValidatingForCreate event handler method .................. 135
  Sample: Creating event handler method .................................... 138
  Sample: ValidatingForUpdate event handler method ................. 140
  Sample: Updating event handler method ..................................... 142
  Sample: Deleting event handler method .................................... 144
Chapter 23: Registering Events .......................................................... 145
  Business object configuration file ......................................................... 145
  Modifying the configuration file ............................................................. 146
  Event registration example .................................................................... 147

Chapter 24: Using Service Extensions ................................................ 149
  Retrieving extension data ...................................................................... 149
  Creating or updating extension data ....................................................... 152

Part 5: Creating a New Web Service .................................................... 158

Chapter 25: Designing a Service ......................................................... 159
  Why create a service? ........................................................................... 159
  Using the Dynamics GP Service framework ......................................... 160
  Service design patterns ......................................................................... 160
  Creating a service document type ......................................................... 166
  Using eConnect ................................................................................... 167
  Document conversions ......................................................................... 168
  Securing the service ............................................................................ 168
  Using policy and behaviors ................................................................... 169

Chapter 26: Defining eConnect Operations ........................................ 171
  Overview of eConnect for services ....................................................... 171
  Updating the Transaction Requester for retrieve operations .......... 171
  Adding a SQL stored procedure for create and update operations .... 174
  Adding a SQL stored procedure for a delete operation ..................... 185
  Adding Error Codes ............................................................................ 188

Chapter 27: Creating a Service Document Type ............................... 191
  Overview of a document type ............................................................... 191
  Creating the project for the document type ......................................... 192
  Defining the document type ............................................................... 193
  Defining the key .................................................................................. 198
  Defining the summary ........................................................................ 200
  Defining the summary document list .................................................. 203
  Adding criteria .................................................................................... 204
  Adding an enumeration ...................................................................... 208
  Building the assembly ........................................................................ 208

Chapter 28: Adding Security Metadata ............................................. 209
  Overview of security metadata ............................................................ 209
  Creating the project for the security helper ....................................... 210
  Creating the security helper application ............................................. 211
  Adding operations, tasks, and roles .................................................... 215
  Removing roles, tasks, and operations ................................................. 220
  Building the application ..................................................................... 222
Chapter 29: Adding Policy Metadata ............................................................... 223
  Overview of policy metadata ............................................................................ 223
  Creating the project for the policy helper ......................................................... 224
  Creating the policy helper application ............................................................ 225
  Creating behaviors ......................................................................................... 229
  Creating policies .......................................................................................... 233
  Removing policies and behaviors .................................................................... 234
  Completing the policy helper application ....................................................... 236
  Adding a policy resource assembly ................................................................. 237
  Building the application and resource assembly ............................................. 239

Chapter 30: Creating the Service Interface ...................................................... 241
  Overview of a service interface ...................................................................... 241
  Creating the Visual Studio project .................................................................. 242
  Adding an interface for the native endpoint ................................................... 244
  Adding an interface for the legacy endpoint .................................................. 246
  Implementing a service contract for the native endpoint ............................... 248
  Adding Dynamics GP Service methods .......................................................... 252
  Implementing a contract for the legacy endpoint .......................................... 254
  Adding configuration information ................................................................. 256
  Building the service ...................................................................................... 258

Chapter 31: Creating XSLT Files ...................................................................... 259
  Overview of service transforms ....................................................................... 259
  Creating XSLT for the GetByKey operation ................................................... 261
  Creating XSLT for the GetList operation ....................................................... 267
  Creating XSLT for the Create operation ....................................................... 271
  Creating XSLT for the Update operation ....................................................... 277
  Creating XSLT for the Delete operation ....................................................... 279

Chapter 32: Deploying the Service ................................................................. 283
  Overview of service deployment ..................................................................... 283
  Installing the eConnect components .............................................................. 284
  Installing the service components ................................................................ 285

Part 6: Web Service Samples .......................................................................... 290

Chapter 33: Sales Documents (Legacy) ........................................................... 291
  Overview of Sales Documents sample ......................................................... 291
  Running the Sales Documents sample ......................................................... 291
  How the Sales Documents sample works ..................................................... 293
  Dynamics GP service use for Sales Documents .......................................... 293

Chapter 34: Sales Documents (Native) ............................................................ 295
  Overview of Sales Documents sample ......................................................... 295
  Running the Sales Documents sample ......................................................... 295
  How the Sales Documents sample works ..................................................... 297
  Dynamics GP service use for Sales Documents .......................................... 297
### CONTENTS

**Appendix A: Culture Codes** .................................................. 331
*Overview of the Lead Service* .............................................. 331
*Lead Service sample files* .................................................. 331
*Installing the sample application* ......................................... 332
*Installing the Lead Service* ............................................... 332
*Testing the native service* ................................................... 333
*Testing the legacy service* ................................................ 334
*Uninstalling the Lead Service* ........................................... 334

**Appendix B: ISO Currency Codes** ....................................... 333

**Appendix C: Troubleshooting** ............................................. 335
*Exceptions* ................................................................. 335
*The stored data differs from what was submitted* ............... 337
*Performance* ............................................................... 338
*Dynamics GP service does not respond* ............................... 339
*Security* ................................................................. 340
*Policy* ................................................................. 340

**Glossary** ................................................................. 341

**Index** ................................................................. 343
Introduction

Welcome to Web Services for Microsoft Dynamics® GP. This documentation explains how you can use these services from within your own applications. It also describes how you can extend the services and create your own services. Before you begin using the services for Microsoft Dynamics GP, take a few moments to review the information presented here. Understanding the information provided here will help you learn about what the web services provide.

What’s in this manual

The Microsoft Dynamics GP Web Services Programmer’s Guide is designed to give you an in-depth understanding of how to use these services in your application development. Even if you are familiar with using web services, you will find it helpful to browse the material presented here. Information is divided into the following parts:

- **Part 1, Web Service Basics**, explains what is provided by the web services for Microsoft Dynamics GP and describes the architecture.
- **Part 2, Getting Started**, describes how to connect to the services, introduces the object model, methods, and data types.
- **Part 3, Using the Web Service**, explains how to perform various actions through the Dynamics GP service, handle exceptions that occur, and use policies to configure the service behavior.
- **Part 4, Extending the Web Service**, describes how to extend objects in the service to provide access to additional data.
- **Part 5, Creating a New Web Service**, describes how to create a new service based on the Microsoft Dynamics GP Service framework.
- **Part 6, Web Service Samples**, provides extended example applications that demonstrate how to use the Dynamics GP service. Another comprehensive example shows how to create a new service.

In addition, the Microsoft Dynamics GP Web Service Reference is an online help file that contains detailed information about the methods, classes, enumerations, policies, and interfaces provided by the Dynamics GP service. You will use this reference frequently as you develop your application that uses the Dynamics GP service.

What’s new in this release?

Refer to the following list for more information about the new features added in this release of Web Services for Microsoft Dynamics GP and where you can learn more about each:

1. **Windows Communication Foundation (WCF)**
   Web Services for Microsoft Dynamics GP has been re-architected and is now built using Windows Communication Foundation (WCF). WCF is the preferred way to create services for the Microsoft platform. It provides more configuration options and better performance for the services.
2. **Additional classes**
   Several classes have been added to the Dynamics GP service for this release. Most of the new classes provide access to objects for the Human Resources and Manufacturing modules in Microsoft Dynamics GP. Refer to the Dynamics GP Web Service Reference online help file for a complete list of the new classes.

3. **New methods**
   Several new methods have been added to the Dynamics GP service. Most of these support the new Human Resources and Manufacturing classes. Refer to the Dynamics GP Web Service Reference online help file for a complete list of the new methods.

**Prerequisites**

Since you will be working with information from Microsoft Dynamics GP, knowledge of the accounting system will be helpful. Consult the Microsoft Dynamics GP documentation resources to learn more about the product.

It is assumed that you have already installed and configured the Web Services for Microsoft Dynamics GP. If you haven’t done so, consult the Web Services for Microsoft Dynamics GP Installation and Administration Guide for information about performing the installation.

The information provided in this documentation will help you understand how to use the services for Microsoft Dynamics GP. It is assumed that you are familiar with the development system that you will be using to create your application that accesses the services.

This guide uses Visual Studio 2008 to demonstrate the various techniques for web service development. If you’re using another development tool, the general techniques will be similar. Refer to the documentation for your development tool for details about accessing web services.

**Symbols and conventions**

To help you use this documentation more effectively, we’ve used the following symbols and conventions to make specific types of information stand out.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🧠</td>
<td>The light bulb symbol indicates helpful tips, shortcuts, and suggestions.</td>
</tr>
<tr>
<td>⚠️</td>
<td>Warnings indicate situations you should be aware of when completing tasks.</td>
</tr>
</tbody>
</table>

*Margin notes summarize important information.*

Margin notes call attention to critical information and direct you to other areas of the documentation where a topic is explained.
Code examples for the Dynamics GP service are included for both the legacy endpoint and the native endpoint.

- **Legacy endpoint**
  This symbol is used to indicate the example is for the legacy endpoint.

- **Native endpoint**
  This symbol is used to indicate the example is for the native endpoint.

If a code example does not have either of these symbols, then that example can be used for either type of endpoint.

**Product support**

Technical support for Web Services for Microsoft Dynamics GP can be accessed using the following methods.

- **Telephone support** – Technical Support at (888) 477-7877 between 8:00 a.m. and 5:00 p.m. Central Time, Monday through Friday. International users can contact Technical Support at (701) 281-0555.

- **Internet** – Technical support is also available online through CustomerSource or PartnerSource. Go to [www.microsoft.com/Dynamics/GP](http://www.microsoft.com/Dynamics/GP) and click the CustomerSource or PartnerSource link.
Part 1: Web Service Basics

This portion of the documentation contains basic information you should know before developing applications that use Web Services for Microsoft Dynamics GP. The following information is discussed:

- **Chapter 1, “Dynamics GP Web Service Overview,”** provides an overview of web services and what the Dynamics GP web service provides.
- **Chapter 2, “Web Service Architecture,”** describes the parts that make up the Web Services for Microsoft Dynamics GP, and how these parts work together.
Chapter 1: Dynamics GP Web Service Overview

The Web Services for Microsoft Dynamics GP provide an ideal way for external applications to integrate with the data contained in the Microsoft Dynamics GP accounting system. The following topics introduce Web Services for Microsoft Dynamics GP:

- What is a web service?
- Web service benefits
- What the Dynamics GP Web Service provides

What is a web service?

In the most general terms, a web service is defined as a software system that is designed to support machine-to-machine interaction over a network. More specifically, web services are software systems that provide data and services to other applications. Web services use standard Internet transport protocols such as Hypertext Transfer Protocol (HTTP) and standard XML-based document formats such as Simple Object Access Protocol (SOAP) to exchange information.

Windows Communication Foundation (WCF) is used as the foundation to implement the Web Services for Microsoft Dynamics GP. WCF became part of the .NET Framework beginning with version 3. WFC provides support for many standard protocols that can be used for web services.

Web service benefits

In general terms, web services provide several key benefits for software developers:

1. Based on industry standards
   Once a software developer learns how to use a web service, the learning curve is greatly reduced for other web services that follows the standards.

2. Development tool independence
   Any development tool that supports the web service standard should be able to access the data and services provided by the web service.

3. Insulation from future changes
   Web services attempt to keep the web service interface unchanged, even though the data and code behind the web service may change in future versions of a product. This helps applications that use the web service to continue working properly, even though the application behind the web service has changed.
4. **Secure access to data.**

Web services can tightly control access to the data and services they make available to other applications.

**What the Dynamics GP Web Service provides**

The web service for Microsoft Dynamics GP provides access to the primary documents in the accounting system. Some of the document types include:

- Customers
- Vendors
- Sales documents
- Purchase documents
- Receivables transactions
- Payables transactions
- General ledger transactions
- Accounts

Through the web service, integrating applications can retrieve documents, create new documents, update existing documents, and delete or void documents.

The web service for Microsoft Dynamics GP is fully integrated with the Dynamics Security Service. The administrator of the web service can configure security so only specified users are allowed to perform actions like creating or updating sales documents.
Chapter 2: Web Service Architecture

When developing applications that use the Web Services for Microsoft Dynamics GP, it will be helpful to understand the architecture used to implement them. Information about the architecture is divided into the following sections:

- Web service foundation
- Configurations
- Security for web services
- Policy for web services
- Exception logging

Web service foundation

Web Services for Microsoft Dynamics GP is constructed on a base of Windows Communication Foundation (WCF) and eConnect. The architecture is shown in the following illustration.

**Windows Communication Foundation**

The preferred foundation for web services on the Microsoft Windows Server platform is the Windows Communication Foundation. WCF provides a versatile framework that can be used to implement several types of web services. WCF is used to implement the Microsoft Dynamics GP Service Host. This is a Windows service that can host several WCF services for Microsoft Dynamics GP. One of these is the Dynamics GP service. The Dynamics GP service provides a legacy endpoint and native endpoint. External applications use these web service endpoints to access data in Microsoft Dynamics GP.

**Legacy endpoint**

The legacy web service endpoint uses the BasicHttpBinding. This endpoint has the characteristics of a standard ASMX-based web service, just like a web service that was created with ASP.NET. Releases 9 and 10 of Web Services for Microsoft Dynamics GP were ASMX-based web services that were implemented using ASP.NET. Applications can use the legacy endpoint of the Dynamics GP service just like they had used the ASP.NET-based web service from the previous releases.
Native endpoint  The native web service endpoint uses the WSHttpBinding. This endpoint is similar to legacy endpoint, but has better performance. The native endpoint can also use additional web service features such as reliable messaging. The code that applications use to connect to the native endpoint of the Dynamics GP service is different from the code to connect to the legacy endpoint.

eConnect
The Dynamics GP web service uses eConnect to provide access to the data managed by the accounting system. eConnect is a set of SQL stored procedures and supporting code used by integrating applications to access data in Microsoft Dynamics GP. Data validation logic is built into eConnect, helping ensure the integrity of any data written to the database through the web services.

Though eConnect provides the data access for the Dynamics GP web service, no knowledge of eConnect is required to use the web service. The Dynamics GP web service interface completely isolates the web service developer from eConnect. The eConnect interfaces can still be used when the Dynamics GP web service is installed.

Configurations
Two common configurations are used with Web Services for Microsoft Dynamics GP. In the basic configuration, Windows Communication Foundation (WCF) and the Web Services for Microsoft Dynamics GP are installed on the same server that is hosting SQL Server and managing Microsoft Dynamics GP data. This is shown in the following illustration:

The following illustration shows the second common configuration for the Web Services for Microsoft Dynamics GP. In this configuration, the web services are installed on a separate server, and access the SQL Server that manages Microsoft Dynamics GP data over the local network.
Which configuration you choose will depend on how extensively you will be using the Web Services for Microsoft Dynamics GP, and what server resources you have available. The two-server configuration will provide better performance if the web service will be heavily used.

**Security for web services**

Security for the Dynamics GP service is controlled by the Dynamics Security service. The Dynamics Security service is installed onto the same server as the Dynamics GP service.

Through web service, the administrator will configure which users and groups are able to execute the methods (operations) provided by the Dynamics GP service. If an application attempts to run a method for which the current user doesn’t have access, a security exception will be raised and the action will be prevented. Security is controlled through the Dynamics Security Administration console, which is a snap-in for Microsoft Management Console (MMC). The console is shown in the following illustration.
Policy for web services

Policy is another security-related feature for the Dynamics GP service. The policy system allows the administrator to control how business objects are created, updated, or deleted through the Dynamics GP service.

Each create, update, and delete or void method has a policy object that is passed with the operation. This policy object specifies the set of behaviors for the operation. Each behavior controls one characteristic for the operation being performed. For instance, the policy for the CreateCustomer method has the behavior named “Create Active Behavior”. This behavior controls whether the customer being created is set to the active or inactive state.

Behaviors are classified as internal or external. An internal behavior is one that can be specified by only the Web Services for Microsoft Dynamics GP administrator. An external behavior is one that can be specified by the application that is calling the method and passing in the policy object. Policy is configured using the Dynamics Security console. You will learn more about specifying behaviors from within your web service application in Chapter 15, “Policy.”

Exception logging

The Dynamics GP service maintains a record of all exceptions (errors) that occur for service operations. The administrator will use this information to help diagnose and resolve any issues for applications that use the service.

You can use the Dynamics GP Web Services Exceptions console to view the exception information. This is a snap-in for Microsoft Management Console (MMC) that retrieves and displays the exceptions logged by the Dynamics GP service.

The console is shown in the following illustration.

The exception information can also be queried by applications that access the Dynamics GP service. Retrieving exception information allows the client applications to display helpful error messages for the user, or to respond appropriately to exceptions that occur. You will learn more about working with exceptions in Chapter 14, “Exceptions.”
Part 2: Getting Started

This portion of the documentation explains how to start developing an application that accesses the Dynamics GP service. The following information is discussed:

- **Chapter 3, “Endpoints,”** describes the two endpoints available for the Dynamics GP service. It also explains how to choose which endpoint to use.

- **Chapter 4, “Connecting to the Legacy Endpoint,”** explains how to create a connection to and a proxy for the legacy endpoint.

- **Chapter 5, “Connecting to the Native Endpoint,”** explains how to create a connection to and a proxy for the native endpoint.

- **Chapter 6, “Dynamics GP Service Object Model,”** describes the object model for the Dynamics GP service.

- **Chapter 7, “Methods,”** describes the methods that are provided in the Dynamics GP service.

- **Chapter 8, “Data types,”** explains how to work with the various types of data used in the objects for the Dynamics GP service.

- **Chapter 9, “Context,”** describes the context object, and how it is used when calling methods in the Dynamics GP service.
Chapter 3: Endpoints

Two endpoints are available to use with the Dynamics GP service. To use the service, you must first create a connection to one of those endpoints. Information about endpoints is divided into the following sections:

- Choosing the endpoint to use
- Endpoint URLs
- WSDL
- WSDL generation
- Proxies

Choosing the endpoint to use

The Dynamics GP service provides two endpoints that applications can connect to. Several factors will determine whether you use the legacy endpoint or the native endpoint for your application.

Legacy endpoint

The legacy endpoint uses the BasicHttpBinding. Consider using the legacy endpoint in the following situations:

- You are upgrading an application that had integrated with the Dynamics GP web service from an earlier release of Microsoft Dynamics GP.

- You are using a development tool that supports only standard ASMX-based web services (such as those created with ASP.NET). For example, applications created with Visual Studio 2005 can connect only to ASMX-based web services, so they must use the legacy endpoint.

Native endpoint

The native endpoint uses the WSHttpBinding. Consider using the native endpoint in the following situations:

- You are creating a new application that will integrate with the Dynamics GP web service, and your development tool supports the WSHttpBinding used for the native endpoint.

- You need the improved performance or features of the native endpoint. The development tool you use must support the WSHttpBinding. Visual Studio 2008 and later support connections to the native endpoint.

- You need the improved security provided by the native endpoint. By default, the data exchanged with the native endpoint is encrypted.

Endpoint URLs

The endpoints for the Dynamics GP web service are accessible through a standard web browser. If you enter the URL for the endpoint into a browser, you can see details about the service.

Legacy URL

By default, the URL for the legacy endpoint is:

http://machine_name:port/DynamicsGPWebServices
The port value is typically 48620. You can find the full URL of the legacy endpoint, including the port number, in the DynamicsGPLegacy.config file on the server that is running web services. This file is typically found in this location:

C:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs\n
When you view this URL in a web browser, you will see some basic examples of code that connects to the service. This is shown in the following illustration.

**Native URL**

By default, the URL for the native endpoint is:

http://machine_name:port/Dynamics/GPService

The port value is typically 48620. You can find the full URL of the native endpoint, including the port number, in the DynamicsGP.config file on the server that is running web services. This file is typically found in this location:

C:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs\n
When you view this URL in a web browser, you will see some basic examples of code that connects to the service. This is shown in the following illustration.
WSDL

When viewing the endpoints in a web browser, you can click the link that appears as a parameter for svcutil.exe to see the Web Service Description Language (WSDL) file that completely describes the web service. The WSDL file is in XML format.

The WSDL documents for the legacy endpoint and the native endpoint are not the same. For the legacy endpoint, you will see that the target namespace in the WSDL file is:

http://schemas.microsoft.com/dynamics/gp/2006/01

For the native endpoint, you will see that the target namespace in the WSDL file is:

http://schemas.microsoft.com/dynamics/gp/2010/01

Typically, the WSDL file isn’t used directly by the developer. It is designed to be read by development tools so they can learn about the definitions of the operations, objects, and enumerations provided by the service.

WSDL generation

In typical WCF services, the WSDL documents are generated dynamically. The WSDL documents for the native endpoint and the legacy endpoint are very large. It can take a few minutes for them to be generated. For this reason, static versions of these two WSDL documents are used for the Dynamics GP service. Instead of allowing WCF to generate the WSDL documents, the request for them is redirected to another service that returns the static versions of the documents. This allows the WSDL documents to be returned quickly when they are requested.

Using a WSDL document

Applications that access the endpoints for the Dynamics GP service typically don’t retrieve the WSDL documents, so they are not affected by the static WSDL document. Development tools will need to retrieve the WSDL documents when connecting to the Dynamics GP service. Some development tools may not be able to retrieve a static WSDL document. If a development tool cannot retrieve the static WSDL document, do one of the following:

- Enter the redirected URL directly. This URL is found at the top of the web page that is displayed when you access the endpoint with a web browser.

- Change the endpoint configuration of the Dynamics GP service to use the dynamically-generated WSDL.

Switching to a dynamically-generated WSDL

The configuration file for the endpoint specifies whether the static WSDL or the dynamically-generated WSDL are used. The static WSDL documents are used for the default installation of Web Services for Microsoft Dynamics GP. You can change the configuration to use the dynamically-generated WSDL documents. To do this, complete the following procedure:
1. **Edit the configuration file for the endpoint.**
   Using a text editor, open the configuration file for the endpoint you want to change. The configuration files are typically found in this location:

   C:\Program Files\Microsoft Dynamic\GPWebServices\ServiceConfigs\  

   The configuration file for the native endpoint is named **DynamicsGP.config**. The configuration file for the legacy endpoint is named **DynamicsGPLegacy.config**.

2. **Change the behavior configuration for the endpoint.**
   Locate the `<services>` element in the configuration file. The `<service>` element specifies the contract for the endpoint and the behavior for the WSDL generation.

   **Legacy endpoint** The following value for the behaviorConfiguration attribute specifies that the static WSDL is used:

   ```xml
   <service
   name="Microsoft.Dynamics.GP.WebServices.LegacyContract.DynamicsGP"
   behaviorConfiguration="GPLegacyStaticMetadataBehavior">
   ```

   The following value specifies that the dynamic WSDL is used:

   ```xml
   <service
   name="Microsoft.Dynamics.GP.WebServices.LegacyContract.DynamicsGP"
   behaviorConfiguration="GPDynamicMetadataBehavior">
   ```

   **Native endpoint** The following value for the behaviorConfiguration attribute specifies that the static WSDL is used:

   ```xml
   <service name="Microsoft.Dynamics.GP.WebServices.Contract.DynamicsGP"
   behaviorConfiguration="GPWCFStaticMetadataBehavior">
   ```

   The following value specifies that the dynamic WSDL is used:

   ```xml
   <service name="Microsoft.Dynamics.GP.WebServices.Contract.DynamicsGP"
   behaviorConfiguration="GPDynamicMetadataBehavior">
   ```

3. **Save the configuration changes.**

4. **Restart the Microsoft Dynamics GP Service Host.**
   The Microsoft Dynamics GP Service Host must be restarted for this change to take effect. From the Start menu, locate the Administrative Tools group and choose Services. Find the entry for the Microsoft Dynamics GP Service Host. Right-click the entry and choose Restart.
Proxies

While it is possible to directly interact with a service by exchanging messages in SOAP format, most interaction happens through a proxy. A proxy is a special set of classes that will act as a wrapper for the operations, objects, and enumerations defined by the service. Your code will interact with the proxy classes, rather than the service directly. This greatly simplifies the code to access the service.

In most development tools, you create the proxy for a service by adding a web reference or a service reference to your project. This creates the proxy and the code to connect to the service.

How you create a reference to the Dynamics GP service will depend on whether you are using the legacy endpoint or the native endpoint. Refer to Chapter 4, “Connecting to the Legacy Endpoint,” and Chapter 5, “Connecting to the Native Endpoint,” for details.

Most development tools will have a browsing tool that allows you to view the details of the proxy created for the service. In Visual Studio you can view the proxy with the Object Browser. The Object Browser shows the details of the classes in the proxy. It’s a good way to see the details of the proxy generated for the endpoint you are using to interact with the Dynamics GP service.
Chapter 4: Connecting to the Legacy Endpoint

The legacy endpoint has the behavior of a standard ASMX-based web service. To connect to it, you will add a web reference for your project. After the development tool has been given the URL of the legacy endpoint, it will retrieve the WSDL file that describes the service. It will use this information to build the proxy classes for the service.

The Dynamics GP web service is large, and uses some web service standards that are not fully supported by some development tools and applications that can access ASMX-based web services. You may not be able to create web references to the legacy endpoint of the Dynamics GP service from some of these tools.

The following topics discuss connecting to and using the legacy endpoint:

- Visual Studio 2005
- Visual Studio 2008
- Web service namespace
- Creating a service instance

Visual Studio 2005

To connect to the legacy endpoint of the Dynamics GP service with Visual Studio 2005, you need to add a web reference to your project. The following procedure explains how to do this for a C# project.

1. **Choose to add a web reference.**
   In the Project menu, choose Add Web Reference. The Add Web Reference window will be displayed.

2. **Supply the URL for the legacy endpoint.**
   Refer to Endpoint URLs on page 15 to determine the URL to use. After the URL has been entered, click Go to search for the legacy endpoint. When the service is found, its information will be displayed.

3. **Name the web service reference.**
   The web service will be referenced in your code, so it must be given a name. The Web reference name control provides a place to enter the name.

Throughout this documentation, the web reference to the legacy endpoint of the Dynamics GP service is named “DynamicsGPService”.

WEB SERVICE PROGRAMMER’S GUIDE 21
4. **Add the web reference.**
   Click Add Reference to create the proxy for the web service.

The following illustration shows the web reference in a Visual C# project that connects to the legacy endpoint of the Dynamics GP service.

![Visual Studio 2008](image)

**Visual Studio 2008**

With Visual Studio 2008, you have two options to connect to the legacy endpoint of the Dynamics GP service. You can either target the .NET Framework 2.0 or you can add a backward-compatible web reference.

**Targeting the .NET Framework 2.0**

In the properties of your Visual Studio project, you can choose .NET Framework 2.0 as the target framework.

![Set the Target Framework to .NET Framework 2.0 for the Visual Studio project.](image)

When your project targets this version of the .NET Framework, it behaves just like a Visual Studio 2005 project. This means it can connect only to ASMX-based web services. You can connect to the legacy endpoint of the Dynamics GP service using the same procedure described in *Visual Studio 2005* on page 21.

Be aware that targeting the .NET Framework 2.0 prevents your application from accessing features available in the newer versions of the .NET Framework.
Adding a backward-compatible web reference
In Visual Studio 2008, you can add a backward-compatible web reference to your Visual Studio project that connects to the legacy endpoint of the Dynamics GP service. The following procedure explains how to do this for a C# project.

1. **Choose to add a service reference.**
   In the Project menu, choose Add Service Reference. The Add Service Reference window will be displayed.

2. **Display the advanced options.**
   Click Advanced to display the Service Reference Settings.

3. **Choose to add a web reference.**
   Click Add Web Reference to open the Add Web Reference window.

4. **Supply the URL for the legacy endpoint.**
   Refer to [Endpoint URLs](#) on page 15 to determine the URL to use. After the URL has been entered, click Go to search for the web service endpoint. When the web service is found, its information will be displayed.

5. **Name the web service reference.**
   The web service will be referenced in your code, so it must be given a name. The Web reference name control provides a place to enter the name.

![Service Reference Settings](DynamicsGPService.png)

Throughout this documentation, the web reference to the legacy endpoint of the Dynamics GP service is named “DynamicsGPService”.

6. **Add the web reference.**
   Click Add Reference to create the proxy for the web service.
Web service namespace

In Visual Studio, the generated proxy classes will be added to a separate namespace in the project. The name of this namespace is the same as the name you assigned to the web reference. For example, if the web reference is named DynamicsGPService, the namespace will also have this name.

To make it easier to reference the classes, methods, and enumerations from the web service, you will want to add this namespace to your application code. For instance, the following C# statement will add this namespace to the current application (named WebServiceApplication).

```csharp
using WebServiceApplication.DynamicsGPService;
```

Adding the `using` statement will keep you from having to fully-qualify the classes, methods, and enumerations you refer to in the web service proxy.

Creating a service instance

After you have made a web reference to connect to the legacy endpoint of Dynamics GP service, you will create an instance of the service so you can access the service methods. The “DynamicsGP” class in the generated proxy represents the base service. You will create an instance of this class that provides access to the service methods.

The following example shows the C# code required to create an instance of the Dynamics GP service for the legacy endpoint.

```csharp
// Create an instance of the web service
DynamicsGP wsDynamicsGP = new DynamicsGP();
```

Throughout the code examples for the legacy endpoint in this documentation, the service instance is named “wsDynamicsGP”.

The service instance also provides access to properties that control how the service is called. For instance, when accessing the Dynamics GP service you can specify that the current user’s login credentials will be used for the service call. The following C# code shows how this is done for a project created in Visual Studio 2005 or Visual Studio 2008.

```csharp
// Be sure that default credentials are being used
wsDynamicsGP.UseDefaultCredentials = true;
```

The Timeout property is another important property of the service instance. It specifies how long the client application will wait for a service request to be completed. If your web service application is working with a large number of documents, it may encounter timeout errors. You can use the Timeout property to increase the timeout value. The following C# example sets the timeout to infinite.

```csharp
wsDynamicsGP.Timeout = System.Threading.Timeout.Infinite;
```

Refer to the troubleshooting information in the Web Services Installation and Administration Guide for more information about timeout issues.
Chapter 5: Connecting to the Native Endpoint

The native endpoint uses the wsHttpBinding. To connect to it, you will add a service reference for your project. After the development tool has been given the URL of the native endpoint, it will retrieve the WSDL file that describes the service. It will use this information to build the proxy classes for the service.

To connect to the native endpoint, you must be using a development tool that allows connections to services that use the wsHttpBinding. Visual Studio 2008 and later can create references that connect to the native endpoint.

The following topics discuss connecting to and using the native endpoint:

- Visual Studio 2008
- Application configuration file
- Using SvcUtil.exe
- Service namespace
- Creating a service instance
- Closing the service instance

Visual Studio 2008

To connect to the native endpoint of the Dynamics GP service with Visual Studio 2008, you need to add a service reference to your project. The following procedure explains how to do this for a C# project.

1. Choose to add a service reference.
   In the Project menu, choose Add Service Reference. The Add Service Reference window will be displayed.

2. Supply the URL for the native endpoint.
   Refer to Endpoint URLs on page 15 to determine the URL to use. After the URL has been entered, click Go to search for the native endpoint. When the service is found, its information will be displayed.

3. Supply the namespace.
   The items in the service proxy will be added to a namespace. The Namespace control provides a place to enter the namespace you want to use.
Throughout this documentation, the proxy for the native endpoint is added to the namespace “DynamicsGPService”.

4. **Specify the advanced options for the service reference (optional).**
   Click Advanced to display the advanced options for the service reference. In most cases, you won’t need to change any of these settings.

   One advanced setting you may need to change is the Generate asynchronous operations option. If you will be making asynchronous calls to the service, you must mark this option so that the proper asynchronous methods are added to the proxy. Refer to Calling methods asynchronously on page 99 for details about making asynchronous calls.

   The proxy generated for the web service reference to the legacy endpoint always contains the asynchronous operations.

   Click OK to save the advance settings.

5. **Create the service reference.**
   Click OK to create the proxy for the service reference.

   The Dynamics GP service is very large. Creating the service reference in Visual Studio can take a few minutes. During that time, Visual Studio will not be responsive.

   The following illustration shows the service reference in a Visual C# project that connects to the native endpoint of the Dynamics GP service.

6. **Adjust the buffer sizes for the service reference.**
   Due to the large document sizes and the quantity of data that can be sent to and retrieved from the Dynamics GP service, some of the default buffer size values must be adjusted. Refer to Application configuration file on page 27 for details about the app.config file and the buffer values that need to be adjusted.

   Due to the large size of the Dynamics GP service, the connection settings for the app.config are not always generated when a service reference is added to the project. If this situation occurs, you will need to manually add entries to the app.config file. A sample app.config file can be found later in Application configuration file on page 27. You can also generate the configuration settings using the svcutil.exe, which is described in Using SvcUtil.exe on page 29.
CHAPTER 5  CONNECTING TO THE NATIVE ENDPOINT

Application configuration file

The application configuration file (app.config) contains settings that control how the application will communicate with the native endpoint for the Dynamics GP service. The settings in the app.config file are added when you create the service reference.

Due to the large size of the Dynamics GP service, the connection settings for the app.config are not always generated when a service reference is added to the project. If this situation occurs, you will need to manually add entries to the app.config file. A sample app.config file can be found later in this section. You can also generate the configuration settings using the svcutil.exe, which is described in Using SvcUtil.exe on page 29.

Buffer sizes

Due to the large document sizes and the quantity of data that can be sent to and retrieved from the Dynamics GP service, some of the default buffer size values must be increased to their maximum values. Make the following adjustments in the app.config file:

In the <binding> node:

maxBufferPoolSize="2147483647"
maxReceivedMessageSize="2147483647"

In the <readerQuotas> node:

maxNameTableCharCount="2147483647"

Client settings

The <client> node specifies the URL of the native endpoint that your application is accessing. When your application is installed, this URL will need to be adjusted to point to the native endpoint of the Dynamics GP service.

Within the <identity> node you will find the <userPrincipalName> node. This node is important when configuring security for the native endpoint. To allow the most secure access to the endpoint, the <userPrincipalName> node must be set to the fully-qualified domain name of the user that is running the Microsoft Dynamics GP Service Host on the server that is running Web Services for Microsoft Dynamics GP.

If both the client machine accessing web services and the user running the Dynamics GP Service Host have been registered as Service Principal Names (SPNs) by the domain administrator, then Kerberos authentication will be used. If SPNs have not been registered, or the value in the <userPrincipalName> node isn’t a valid fully-qualified domain name, then authentication will fall back to the less secure NTLM.

The <userPrincipalName> must contain a value that is in the format “machine\user”, even if Kerberos authentication isn’t being used. If the user principal name is missing or is formatted incorrectly, your application will not be able to access the native endpoint of the Dynamics GP service.
Sample app.config file
The following is a sample app.config file that contains the settings for a service reference to the native endpoint of the Dynamics GP service. You can use this sample when you need to manually add the connection information to the app.config file. Be sure to set the URL in the <endpoint> node. You should also set the <userPrincipalName> node if you plan to use Kerberos authentication.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <wsHttpBinding>
        <binding name="GPWebService" closeTimeout="00:01:00" openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00" bypassProxyOnLocal="false" transactionFlow="false" hostNameComparisonMode="StrongWildcard" maxBufferSize="2147483647" maxReceivedMessageSize="2147483647" messageEncoding="Text" textEncoding="utf-8" useDefaultWebProxy="true" allowCookies="false">
          <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384" maxBytesPerRead="4096" maxNameTableCharCount="2147483647"/>
          <reliableSession ordered="true" inactivityTimeout="00:10:00" enabled="false"/>
        </binding>
        <security mode="Message">
          <transport clientCredentialType="Windows" proxyCredentialType="None" realm=""/>
          <message clientCredentialType="Windows" negotiateServiceCredential="true" algorithmSuite="Default" establishSecurityContext="true"/>
        </security>
      </wsHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://server:48620/Dynamics/GPService/GPService" binding="wsHttpBinding" bindingConfiguration="GPWebService" contract="DynamicsGPService.DynamicsGP" name="GPWebService">
        <identity>
          <userPrincipalName value="machine\user"/>
        </identity>
      </endpoint>
    </client>
  </system.serviceModel>
</configuration>
```
CHAPTER 5  CONNECTING TO THE NATIVE ENDPOINT

Using SvcUtil.exe

You can use SvcUtil.exe to generate a proxy and the configuration settings that define how to connect to the native endpoint for the Dynamics GP service. This may be necessary if Visual Studio cannot create the appropriate entries in the app.config for the service reference you add to your project. Complete the following procedure to use SvcUtil.exe to generate a proxy and configuration settings.

1. **Open a Visual Studio Command Prompt window.**
   Click the Start menu, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt.

2. **Set the current folder.**
   In the command prompt, set the current folder to the location where you want the generated proxy and configuration files to be created.

3. **Run SvcUtil.exe to generate the output files.**
   To use svcutil.exe you need to provide the URL of the native endpoint for the Dynamics GP service and the namespace of the service reference you added to your Visual Studio project. Typically, this is “DynamicsGPService”.

   Enter the following on a single line and then press Enter.

   ```cmd
   svcutil.exe http://<machine>:<port>/Dynamics/GPService
   /n:*,DynamicsGPService
   ```

4. **Use the generated files.**
   If you choose to, you can use the generated proxy in your project. You can also use the generated configuration settings in the app.config for a Visual Studio project. In a text editor, open the output.config file created by SvcUtil.exe. Copy the `<system.serviceModel>` node from the output.config file and add it to the app.config file of your Visual Studio project.

**Service namespace**

In Visual Studio, the generated proxy classes will be added to the separate namespace in the project. This is the namespace you supplied when you created the service reference.

To make it easier to reference the classes, methods, and enumerations from the service, you will want to add this namespace to your application code. For instance, the following C# statement will add this namespace to the current application (named WebServiceApplication).

```csharp
using WebServiceApplication.DynamicsGPService;
```

Adding the `using` statement will keep you from having to fully-qualify the classes, methods, and enumerations you refer to in the web service proxy.
Creating a service instance

After you have made a service reference to connect to the native endpoint of Dynamics GP service, you will create an instance of the service so you can access the service methods. The “DynamicsGPClient” class in the generated proxy represents the base service. You will create an instance of this class that provides access to the service methods.

The following example shows the C# code required to create an instance of the Dynamics GP service for the native endpoint.

```csharp
// Create an instance of the service
DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();
```

Throughout the code examples for the legacy endpoint in this documentation, the service instance is named “wsDynamicsGP”.

Closing the service instance

After your application has finished using the service instance for the native endpoint, it is important that you close the instance. This allows the Dynamics GP service to release the resources used for the connection. The following C# code shows how to close the service instance.

```csharp
// Close the service
if (wsDynamicsGP.State != CommunicationState.Faulted)
{
    wsDynamicsGP.Close();
}
```

If applications do not close the service instance for the native endpoint when they are finished using it, the native endpoint can be left in a condition where it is unable to make new connections.
Chapter 6: Dynamics GP Service Object Model

The Dynamics GP service contains numerous objects that you can work with through the service. To effectively use the service, it’s helpful to understand how these objects work together to represent data in the system. Information about the object model is divided into the following sections:

- Object categories
- Inheritance
- Learning the object model
- Endpoint differences
- Dynamics GP Web Service Reference

Object categories

The number of objects available through the Dynamics GP web service is quite large. To understand how all of these objects are used, it helps to divide them into three categories.

Business document objects
These are the objects that represent business documents in Microsoft Dynamics GP. You will recognize them because their names closely match the documents they correspond to in the accounting system. Examples include:

- Customer
- Vendor
- CreditLimit
- Account
- SalesInvoice
- Applicant

Other objects are parts of business documents or define a set of properties for a business document. Example of this type of object include:

- Address
- SalesLine
- Tax

The vast majority of objects in the Dynamics GP service are business documents or some part of a business document.

Base objects
Many objects in the Dynamics GP service share common characteristics. Rather than repeat these characteristics for every individual object, a single base object is used. This object serves as the basis for each specific type of object. The following are some common base objects:

- BusinessObject
- Key
- Criteria
- Restriction

For instance, all business documents such as Customer and SalesInvoice are based on BusinessObject. Every key object used to uniquely identify an object is based on the Key object.

You will never work with the base objects directly. They are used only by the service to implement the other objects.
**Helper objects**

Some objects are included simply to help you work with the Dynamics GP service. They perform actions such as controlling how the service is called, or dealing with errors that occur during processing. The following are some of the helper objects:

- Context
- Policy
- ExceptionInformation
- ValidationItem

**Inheritance**

As described in *Object categories* on page 31, many of the business document objects you will use as you work with the Dynamics GP service inherit characteristics from other objects. For instance, the CustomerAddress object, which completely describes the address information for a customer, inherits from a series of other objects. The inheritance hierarchy for the CustomerAddress object is shown in the following illustration:

![Inheritance Diagram](image)

The CustomerAddress object has all of the characteristics (properties) of each of the objects it inherits from. Notice that BusinessObject serves as the base object for the inheritance hierarchy.
CHAPTER 6 DYNAMICS GP SERVICE OBJECT MODEL

Learning the object model

At first, the large number of objects in the Dynamics GP service object model can seem overwhelming. As you work with the Dynamics GP service, you will learn about each category of object. Soon you will be able to quickly decide what category an object is in, and how it should be used.

One approach to learning the object model is to start with a specific method. Each method requires a set of objects to perform its action. Begin with those objects, and branch out to the related objects.

For example, assume you wanted to execute the CreateCustomer method, which creates a customer in Microsoft Dynamics GP. This method uses the following objects:

- Customer
- Context
- Policy

As you work with the Customer object, you will see that it references some additional objects including:

- CustomerKey
- CustomerClassKey
- CustomerAddressKey
- CustomerCreditLimit

As you learn about these related objects, you will expand your understanding of the object model for the Customer object you are creating.

Endpoint differences

The object models for the legacy and native endpoints for the Dynamics GP service are almost identical. However, there are some differences that you need to be aware of when you use each endpoint.

Data type naming differences

The base data types, such as string, integer, and boolean use names that are different in the legacy and native endpoints. The following table lists the base data type name used in the legacy endpoint and the corresponding name used in the native endpoint:

<table>
<thead>
<tr>
<th>Legacy endpoint</th>
<th>Native endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>DateTime</td>
<td>dateTime</td>
</tr>
<tr>
<td>Int32</td>
<td>int</td>
</tr>
<tr>
<td>Decimal</td>
<td>decimal</td>
</tr>
<tr>
<td>String</td>
<td>string</td>
</tr>
</tbody>
</table>

You will see these naming differences where the base names are used, such as restriction criteria. The datetime restriction type is a good example. In the legacy endpoint, the datetime restriction type has this name:

BetweenRestrictionOfNullableOfDateTime
In the native endpoint, the datetime restriction type has this name:

BetweenRestrictionOfNullableOfDateTime

Notice that the base data type (in italic text) has a slightly different name, due to a difference in the casing.

**Object name differences**
Collection objects for enumeration values use different names in the legacy and native endpoints. In the native endpoint, these collection objects are nullable. The the legacy endpoint, they are not.

For example, in the legacy endpoint the object representing a collection of ApplicationStatus objects has the following name:

ArrayOfApplicationStatus

In the native endpoint, the object representing a collection of ApplicationStatus objects is nullable. It has the following name:

ArrayOfNullableOfApplicationStatus

**Dynamics GP Web Service Reference**

The Dynamics GP Web Service Reference is an online help file that provides detailed information about the classes (objects) available in the Dynamics GP service. Use this comprehensive reference as you learn about the object model for the web service. The links in the help file make it easy to browse through the properties of an object and see the other objects related to it.
Chapter 7: Methods

To perform tasks with the Dynamics GP service, you must call the methods provided. The following sections introduce the methods available in the service:

- Method categories
- Framework methods

Method categories

The methods for the Dynamics GP service can be divided into the following categories: Get, GetList, Create, Update, and Delete or Void.

Get

Get operations retrieve individual documents from Microsoft Dynamics GP. To retrieve a document, you specify the document’s key (unique identifier). If the document can be found, it will be returned from the Dynamics GP service. You will learn more about Get operations in Chapter 11, “Retrieving Objects.”

GetList

GetList operations retrieve collections of summary objects that meet the specified set of search criteria. A summary object is a special version of a business document, containing only the most important properties of the document. For example, the CustomerSummary object contains only the most important properties of the Customer object. A small number of GetList operations retrieve entire documents, rather than summary documents. You will learn more about criteria and GetList operations in Chapter 11, “Retrieving Objects.”

Create

Create operations create new documents in Microsoft Dynamics GP. To create a new document, you will first create a new instance of the document object. After you specify the details for the object, you will call the Create method to create the new object. You will learn more about creating new documents in Chapter 10, “Creating Objects.”

Update

Update operations modify existing documents in Microsoft Dynamics GP. For most documents, you can create an instance of the document type, specify the properties to update, and supply the key value to indicate which document will be updated. Other types of documents, specifically sales documents, require you to first retrieve an instance of the document before you can update it. You will learn more about updating documents in Chapter 12, “Updating Objects.”

Delete or void

Delete operations permanently delete existing documents from Microsoft Dynamics GP. Some documents cannot be deleted, but can be voided instead. Void operations perform this action. Chapter 13, “Deleting or Voiding Objects,” describes how to do this in detail.
Framework methods

The Dynamics GP service also provides several methods that help you work with the it.

Exceptions

Code that accesses the Dynamics GP service must be able to handle any exceptions that occur while methods are processed. Several methods are included in the Dynamics GP service that your application can use to retrieve information about an exception that occurred.

The exception information can be used to present a helpful error message to the user, explaining an issue such as a data validation exception. The information might also be used directly by the code to handle an error, such as retrying the web service call after a timeout exception. You will learn more about exceptions in Chapter 14, “Exceptions.”

Policy

Policy allows the administrator of the Dynamics GP service to control how create, update, and delete or void operations are performed. The individual characteristics that can be controlled are called behaviors. For instance, the policy for the UpdateSalesOrder method has a behavior that can specify how quantity shortages are to be handled for the sales order document.

Your application that uses the Dynamics GP service must retrieve and use the appropriate policies when it calls methods that require them. The Dynamics GP service contains special web methods that retrieve policy information for the specified web methods. You will learn more about policy in Chapter 15, “Policy.”
Chapter 8: Data types

Several types of data are represented in the properties defined in the objects for the Dynamics GP service. This portion of the documentation provides details you will need to know as you work with these different data types in your own applications. The following topics are discussed:

- Standard data types
- Enumerations
- Account numbers
- Choice types

Standard data types

The following is a list of the standard data types used in the Dynamics GP service, along with some details about each type.

String values

String values in the Dynamics GP service must be handled carefully to avoid truncation issues. String properties for the business documents in the Dynamics GP service have a defined length that is detailed in the Dynamics GP Web Service Reference online documentation. It’s essential that you don’t exceed the stated string length for these properties. Any strings that exceed the specified length will be truncated, resulting in data corruption.

⚠️ No exception will be logged when a string value is truncated by the Dynamics GP service. ⚠️

The empty string (“”) is considered the empty value for a string property.

Integer values

The value 0 (zero) is considered the empty value for an integer property.

Decimal values

The value 0.0 is considered the empty value for a decimal property.

DateTime values

The value 1/1/1900 12:00:00am is considered the empty value for a DateTime property.

MoneyAmount values

The MoneyAmount data type maintains three pieces of data:

- The currency used for the amount
- The value of the amount (as a decimal)
- The number of decimal digits in the amount

The currency corresponds to the ISO code for the currency represented. Refer to Appendix B, “ISO Currency Codes,” for a list of the standard ISO codes used for currencies in Microsoft Dynamics GP.

The value is considered empty when when the currency hasn’t been specified, the decimal value is 0.0, and the number of digits is 0.
Percent values
The Percent data type maintains two pieces of data:

- The value of the amount (as a decimal)
- The number of decimal digits in the amount

The value is considered empty when the value amount is 0.0 and the number of digits is 0.

Enumerations
To make using the Dynamics GP service easier, several properties have enumerations (a predefined set of values) defined for them. For example, the Customer class has the **StatementCycle** property that defines when the customer will be billed. This property has the type StatementCycle, which is an enumeration defined by the service. It has the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Statement</td>
<td>No statements sent</td>
</tr>
<tr>
<td>Weekly</td>
<td>Statements sent weekly</td>
</tr>
<tr>
<td>Bi-Weekly</td>
<td>Statements sent once every two weeks</td>
</tr>
<tr>
<td>Semi-Monthly</td>
<td>Statements sent twice each month</td>
</tr>
<tr>
<td>Monthly</td>
<td>Statements sent once each month</td>
</tr>
<tr>
<td>Bi-Monthly</td>
<td>Statements sent once every two months</td>
</tr>
<tr>
<td>Quarterly</td>
<td>Statements sent once each quarter</td>
</tr>
</tbody>
</table>

When you create a web reference to the legacy endpoint or a service reference for the native endpoint, the proxy generated will contain all of the enumerations. These simplify working with these properties. For example, the following C# statement shows how the StatementCycle property can be set for a customer object.

```csharp
```

For some properties that use an enumeration, the value of the property isn’t a single value from the enumeration. Instead, it is the summation of zero or more of the values from the enumeration. For instance, the Company class has the **CompanyOptions** property. The CompanyOptions enumeration is used to set the value of this property. The value of the property is the sum of the enumeration items that apply to the company.

Account numbers
The format for account numbers is defined at the time Microsoft Dynamics GP is installed. The Dynamics GP service uses the account number format that was defined. The Dynamics GP service will return account numbers in this format. When your web service application uses account numbers, it must supply them in the format that matches the current Microsoft Dynamics GP installation.

Since the account format may not be known at the time you write your application, the Dynamics GP web service includes the **GetGLAccountFormatList** method to retrieve information about the account number format. This method retrieves a collection of **GLAccountFormat** objects, each of which describes one segment of the account number. Examine each of these objects to determine the length of the account segment.
The separator character that appears between the segments of the account number is available from the AccountSegmentSeparator property of the Company object. Use the GetCompanyByKey method to retrieve this object.

With this information, you can construct the format needed for account numbers to be processed by the Dynamics GP service.

**Choice types**

Of the data types used for the Dynamics GP service, choice types require the most additional work to reference from your web service application. The “choice” type allows a single parameter to contain one of several different types of data. For instance, the FinanceCharge property for a customer is of type MoneyPercentChoice. It can contain either a money value or a percent value.

When you look at the MoneyPercentChoice in the generated proxy, you will see that it appears with a single Item property, which has the type Object.

![The MoneyPercentChoice type has only one property, which is named Item.](image)

When your code reads the value of a parameter that uses the “choice” type, it must determine what type of value to read. When your code writes a choice parameter, it must specify which type of data it is writing. You can get this type information from the generated proxy code, or from the WSDL file for the Dynamics GP service.

In Visual Studio, you can double-click the Item property in the Object Browser to view its implementation details in the generated proxy. The following is the generated code for the MoneyPercentChoice type in the proxy for the legacy endpoint of the Dynamics GP service:

```csharp
public partial class MoneyPercentChoice {

    private object itemField;

    [System.Xml.Serialization.XmlElementAttribute("Amount", typeof(MoneyAmount))]
    [System.Xml.Serialization.XmlElementAttribute("Percent", typeof(Percent))]
    public object Item {
        get {
            return this.itemField;
        }
        set {
            this.itemField = value;
        }
    }
}
```

Notice in the XML serialization that the type for the Item property can be MoneyAmount or Percent, which are standard types for the Dynamics GP service.
Reading a choice type
The following C# example shows how you would retrieve the value of the FinanceCharge property for a customer object. The example will find out the type of the Item property for the MoneyAmount type, and then retrieve the underlying type.

DynamicsGPService.MoneyPercentChoice financeCharge;
DynamicsGPService.MoneyAmount financeChargeAmount;
DynamicsGPService.Percent financeChargePercent;

// Examine the finance charge for the customer
financeCharge = customer.FinanceCharge;

if (financeCharge.Item.GetType() == typeof(DynamicsGPService.MoneyAmount))
{
    financeChargeAmount = (DynamicsGPService.MoneyAmount)financeCharge.Item;
    MessageBox.Show("Finance charge is a money amount of: " +
    financeChargeAmount.Value.ToString());
}

if (financeCharge.Item.GetType() == typeof(DynamicsGPService.Percent))
{
    financeChargePercent = (DynamicsGPService.Percent)financeCharge.Item;
    MessageBox.Show("Finance charge is a percentage: " +
    financeChargePercent.Value.ToString());
}

Writing a choice type
The following C# example shows how you would write the value of the FinanceCharge property for a customer object. The value being written has the type MoneyAmount.

MoneyAmount financeCharge = new MoneyAmount();

financeCharge.Currency = "USD";
financeCharge.DecimalDigits = 2;
financeCharge.Value = 5000.00;
customer.FinanceCharge.Item = financeCharge;
Chapter 9: Context

The Context object is used for every method in the Dynamics GP service. It provides important details that specify how the method call should be performed. Information about the Context object is divided into the following sections:

- **Organization key**
- **Culture**
- **Currency**
- **Role key**
- **WorkOnBehalfOf**

**Organization key**

The organization key property of the Context object specifies the company in Microsoft Dynamics GP that the method should be performed in. Specifying the company allows the Dynamics GP service to perform requests in the correct company database in SQL Server. The value corresponds to the numeric Company ID value in Microsoft Dynamics GP. The following C# example demonstrates setting the OrganizationKey property.

```csharp
// Create a context with which to call the web service
context = new Context();

// Specify which company
companyKey = new CompanyKey();
companyKey.Id = 1;

// Set up the context
context.OrganizationKey = companyKey;

If the web method is to be performed in the context of the system (DYNAMICS database) then the OrganizationKey should be set to null.

**Culture**

The Culture property of the Context object is optional. It is a string property that specifies the culture (locale) of the user who is making the method call. Based on the locale specified, the Dynamics GP service could return values in a format specific to that locale. Refer to Appendix A, “Culture Codes,” for a list of the culture codes.

**Currency**

The Currency property of the Context object should be specified only when information is being retrieved from the Dynamics GP service. This property indicates how currency values are retrieved for a document. The following options are available:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactional</td>
<td>The transaction currency (originating) amount will be retrieved.</td>
</tr>
<tr>
<td>Local</td>
<td>The local currency (functional) amount will be retrieved.</td>
</tr>
</tbody>
</table>

This property is used only for documents that support multicurrency. If multicurrency isn’t supported for a document, the Local (functional) currency amount will be used.
Role key

The role key property of the Context object is optional. If supplied, the role specified will be used to choose the policy instance and appropriate behavior options for the method call. If you don’t supply a role key, the Dynamics GP service will attempt to find a role for the user and company specified in the context object. If only one role can be found, that role will be used. If more than one role is found, or no roles are found, the default role will be used.

WorkOnBehalfOf

The WorkOnBehalfOf property of the Context object is used in the special cases where one user is running the service methods, but the security privileges for a second user should be used. The WorkOnBehalfOf property will be set to the login name for this second user.

For typical web service calls, you shouldn’t set this property. It should be set only in the special cases where one web service user is performing actions on behalf of a different user.

The value for the property will have the following form:

DOMAIN\USERNAME

or

COMPUTERNAME\USERNAME

Refer to Working on behalf of another user on page 101 for more details about the special situations where the property is used.
Part 3: Using the Web Service

This portion of the documentation provided detailed information about performing actions with the Dynamics GP service. The following items are discussed:

- Chapter 10, “Creating Objects,” explains techniques for creating objects through the service.
- Chapter 11, “Retrieving Objects,” describes techniques for retrieving objects from the service.
- Chapter 12, “Updating Objects,” describes how to use the Dynamics GP service to update objects.
- Chapter 13, “Deleting or Voiding Objects,” explains how to use the Dynamics GP service to delete or void objects.
- Chapter 14, “Exceptions,” describes how to handle exceptions that occur during service calls.
- Chapter 15, “Policy,” explains how to use policy to control the service operations.
- Chapter 16, “Other Programming Techniques,” describes some additional programming techniques that will be useful when working with the Dynamics GP service.
- Chapter 17, “Entity ID Filtering,” explains how the identity information contained in objects can be used to filter them as they are retrieved from the Dynamics GP service.
- Chapter 18, “Entity Action Tracking,” describes how changes to objects in the Dynamics GP service can be tracked.
- Chapter 19, “Optimized Proxy,” explains how to create an optimized proxy for your application that accesses the Dynamics GP service.
Chapter 10: Creating Objects

The Dynamics GP service allows new objects to be created in Dynamics GP. Information about creating objects is divided into the following sections:

- Create methods
- Create policy

Create methods

The “Create” methods add new data to Dynamics GP. Create methods use objects to specify the type of data being created and the values to be saved. To create an object, use the following basic steps:

- Instantiate an object of the type to be created.
- Populate the object’s properties with values. All required properties must be assigned a value. All non-required properties that are not populated will receive a default value when the Create method is run.

The required properties for each object are indicated in bold in the Microsoft Dynamics GP Web Service Reference online help file.

- Create a policy object for the create operation. A policy object provides additional control for the create operation. You will learn more about the policies in Create policy on page 47.

- Use the object as a parameter for the Create method.

The following example demonstrates creating a new vendor. The vendor object requires a vendor key to uniquely identify this vendor. Notice how a vendor key object is created and used to populate the required property. The Name property is also populated. The vendor object’s remaining properties will use default values.

The vendor object is then passed as the first parameter to the CreateVendor method. The CreateVendor method also requires context and policy objects to be created and passed as the second and third parameters.

Notice how the GetPolicyByOperation method instantiates the vendorPolicy object and retrieves the policy for the operation. The “CreateVendor” parameter specifies the default create policy for a vendor object. GetPolicyByOperation also requires a context object for its second parameter.

Legacy endpoint

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
```
Context context;
Vendor vendor;
VendorKey vendorKey;
Policy vendorPolicy;

// Create an instance of the service
DynamicsGP wsDynamicsGP = new DynamicsGP();

// Be sure the default credential are used
wsDynamicsGP.UseDefaultCredentials = true;

// Create a context object with which to call the web service
context = new Context();

// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context
context.OrganizationKey = companyKey;

// Create a new vendor key
vendorKey = new VendorKey();
vendorKey.Id = "TstVndr0001";

// Populate the vendor object
vendor = new Vendor();
vendor.Key = vendorKey;
vendor.Name = "TestVendor0001";

// Get the create policy for the vendor
vendorPolicy = wsDynamicsGP.GetPolicyByOperation("CreateVendor", context);

// Create the vendor
wsDynamicsGP.CreateVendor(vendor, context, vendorPolicy);
}

namespace DynamicsGPWebServiceSample
{:}
class Program
{:}
static void Main(string[] args)
{:}
    CompanyKey companyKey;
    Context context;
    Vendor vendor;
VendorKey vendorKey;
Policy vendorPolicy;

// Create an instance of the service
DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

// Create a context object with which to call the web service
context = new Context();

// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context
context.OrganizationKey = companyKey;

// Create a new vendor key
vendorKey = new VendorKey();
vendorKey.Id = "TstVndr0001";

// Populate the vendor object
vendor = new Vendor();
vendor.Key = vendorKey;
vendor.Name = "TestVendor0001";

// Get the create policy for the vendor
vendorPolicy = wsDynamicsGP.GetPolicyByOperation("CreateVendor", context);

// Create the vendor
wsDynamicsGP.CreateVendor(vendor, context, vendorPolicy);

// Close the service
if (wsDynamicsGP.State != CommunicationState.Faulted)
{
    wsDynamicsGP.Close();
}


Create policy

A Create method requires an instance of the “create” policy for the object being created. The create policy contains a set of behaviors that control various aspects of the create operation, such as how certain properties will be set, or whether additional actions will be performed. The web service administrator controls all of the internal behavior options through the Dynamics Security console. The external behaviors can be set by the application that is calling the create method.

When you call a Create method for the Dynamics GP service, you must retrieve an instance of the policy used for that method. Do this using either the GetPolicyByOperation web method or the GetPolicyByKey web method. If the context object you pass to either method specifies a valid role for the user, the policy that is configured for that role will be returned. Otherwise, the default policy for the operation will be returned.
Once the policy for the create operation has been retrieved, any external behaviors for the policy can be changed, if needed. The policy object will be passed to the web service Create method.

In the following C# example, a new GL transaction is created. The policy for the CreateGLTransaction web method is retrieved. One behavior for this policy controls whether a reversing transaction is also created. The ID (GUID) for this behavior is looked up from the Dynamics GP Web Service Reference and used to locate the behavior. The behavior option specifying that a reversing transaction should be created is also looked up in the reference. The parameter for this behavior option is set to the reversing transaction date. Finally, the selected option for the behavior is set to the option to create the reversing transaction.

```csharp
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            BatchKey batchKey;
            GLTransactionKey transactionKey;
            GLTransaction transaction;
            GLTransactionLineKey glTransactionLineKey;
            GLTransactionLine transactionDebitLine;
            GLTransactionLine transactionCreditLine;
            GLAccountNumberKey debitAccountKey;
            GLAccountNumberKey creditAccountKey;
            MoneyAmount debitAmount;
            MoneyAmount creditAmount;
            MoneyAmount zeroAmount;
            Policy transactionCreatePolicy;
            Behavior reversingTrxBehavior;
            BehaviorOption reversingTrxBehaviorOption;
            Guid behaviorGUID;
            int behaviorOptionID;

            // Create an instance of the web service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure the default credentials are used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the web service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);
```
// Set up the context object
context.OrganizationKey = (OrganizationKey)companyKey;
context.CultureName = "en-US";

// Create a batch key object to specify the batch
batchKey = new BatchKey();
batchKey.Id = "RMCSH000000011";

// Create a GL transaction key object to identify the transaction
transactionKey = new GLTransactionKey();
transactionKey.JournalId = 3372;
transactionKey.Date = new DateTime(2017, 4, 12);

// Create the transaction object
transaction = new GLTransaction();

// Populate the GL transaction object's key property
transaction.Key = transactionKey;

// Populate the batch key and reference properties
transaction.BatchKey = batchKey;
transaction.Reference = "Receivables Cash Receipts";

// Create a GL transaction line key
glTransactionLineKey = new GLTransactionLineKey();

// Create two GL transaction lines:
// Create the debit transaction line
transactionDebitLine = new GLTransactionLine();

// Populate the transaction line key
transactionDebitLine.Key = glTransactionLineKey;

// Create a GL account number key object to specify the account
debitAccountKey = new GLAccountNumberKey();
debitAccountKey.Id = "000-1100-00";

// Populate the debit line GL account key property
transactionDebitLine.GLAccountKey = debitAccountKey;

// Create a money object
zeroAmount = new MoneyAmount();
zeroAmount.Value = 0m;
zeroAmount.Currency = "USD";

// Create a money object for the debit amount
debitAmount = new MoneyAmount();
debitAmount.Value = 500m;
debitAmount.Currency = "USD";

// Populate the transaction line debit and credit amounts
transactionDebitLine.DebitAmount = debitAmount;
transactionDebitLine.CreditAmount = zeroAmount;

// Create the credit transaction line
transactionCreditLine = new GLTransactionLine();
// Populate the transaction line key
transactionCreditLine.Key = glTransactionLineKey;

// Create a GL account number key object to specify the account
creditAccountKey = new GLAccountNumberKey();
creditAccountKey.Id = "000-1200-00";

// Populate the credit line GL account key property
transactionCreditLine.GLAccountKey = creditAccountKey;

// Create a money amount object for the credit
creditAmount = new MoneyAmount();
creditAmount.Value = 500m;
creditAmount.Currency = "USD";

// Populate the transaction line debit and credit amounts
transactionCreditLine.DebitAmount = zeroAmount;
transactionCreditLine.CreditAmount = creditAmount;

// Create an array to hold the two GL transaction line objects
GLTransactionLine[] lines = { transactionDebitLine, transactionCreditLine };
// Set the reversing date for the behavior option
reversingTrxBehaviorOption.Parameters[0].Value = reversingTrxDate.ToShortDateString();

// Set selected behavior option to create a reversing transaction
reversingTrxBehaviorSelectedOption = reversingTrxBehaviorOption;

// Create the GL transaction
wsDynamicsGP.CreateGLTransaction(transaction, context, transactionCreatePolicy);
Chapter 11: Retrieving Objects

An individual object or list of objects can be retrieved through the Dynamics GP service. The objects returned by the service will be populated with data from Microsoft Dynamics GP. Information about retrieving objects is divided into the following sections:

- **Retrieve methods**
- **Individual objects**
- **Lists of objects**
- **Restriction reference**

Retrieve methods

There are two ways to retrieve objects from the Dynamics GP service. Objects can be retrieved individually or as a list.

When objects are retrieved individually, you must specify the unique key value for the object you want to retrieve. The “GetByKey” methods in the Dynamics GP service are used to retrieve individual objects.

When objects are returned as a list, you will supply a set of criteria that indicate which objects you want to retrieve. The Dynamics GP service has criteria objects that you will use to specify criteria. The “GetList” methods in the Dynamics GP service are used to retrieve lists of objects.

Individual objects in the Dynamics GP service can contain a significant amount of data. When a list of objects is retrieved from the service, a large amount of data would need to be returned. A summary object contains the most important details of the main object. For example, the CustomerSummary object contains the important details of a Customer object.

To speed the request for a list of objects, the summary version of the object is returned by most of the “GetList” methods in the Dynamics GP service. For some of the smaller objects, the corresponding “GetList” methods return the actual objects in the list.

Individual objects

A “GetByKey” method returns a single object specified by a key. You will create an instance of the key object for the type of object you want to retrieve. The key object’s identification property must be set to the value that uniquely identifies the object to be retrieved.

If the object specified cannot be found, the Dynamics GP service will raise a “Business object not found.” exception. You may want to trap for this exception in your code and display an appropriate message for the user.

The following example demonstrates retrieving a customer object. Notice how the customer key object is created and its Id property populated with a specific customer ID value. The customer key object is then passed as the first parameter to the GetCustomerByKey method. The GetCustomerByKey method also requires a context object to be created and passed as the method’s second parameter.
**Legacy endpoint**

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            Customer customer;
            CustomerKey customerKey;

            // Create an instance of the service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure that default credentials are being used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the web service
            context = new Context();

            // Specify which company to use (lesson company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a customer key
            customerKey = new CustomerKey();
            customerKey.Id = "AARONFIT0001";

            // Retrieve the customer object
            customer = wsDynamicsGP.GetCustomerByKey(customerKey, context);
            MessageBox.Show(customer.Name);
        }
    }
}
```

**Native endpoint**

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
    }
}
```
CHAPTER 11  RETRIEVING OBJECTS

{  
    static void Main(string[] args)  
    {  
        CompanyKey companyKey;  
        Context context;  
        Customer customer;  
        CustomerKey customerKey;  

        // Create an instance of the service  
        DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();  

        // Create a context with which to call the web service  
        context = new Context();  

        // Specify which company to use (lesson company)  
        companyKey = new CompanyKey();  
        companyKey.Id = (-1);  

        // Set up the context  
        context.OrganizationKey = (OrganizationKey)companyKey;  

        // Create a customer key  
        customerKey = new CustomerKey();  
        customerKey.Id = "AARONFIT0001";  

        // Retrieve the customer object  
        customer = wsDynamicsGP.GetCustomerByKey(customerKey, context);  

        MessageBox.Show(customer.Name);  

        // Close the service  
        if (wsDynamicsGP.State != CommunicationState.Faulted)  
        {  
            wsDynamicsGP.Close();  
        }  
    }  
}  

Lists of objects

A “GetList” method returns a collection of summary objects, or in certain special cases, the collection of objects themselves. The content of the list is controlled by a criteria object. The list will contain only summary objects that meet the restrictions set by the criteria object.

By default, only the first 1000 items that meet the specified criteria will be returned in the list. This limit prevents the service from being overloaded by a query that returns an extremely large collection.

Creating a criteria object

To specify which objects you want to retrieve in a list, you will first create a criteria object for that type. For instance, you will create a CustomerCriteria object to specify which customer objects to return from the GetCustomerList method.
The criteria object has restriction properties that correspond to specific properties of the object. For example, the Name property of the CustomerCriteria object is a string restriction that corresponds to the Name property of the Customer object. When you set the Name property of the CustomerCriteria object, customer summary objects for customers with names that meet the restriction will be returned. Refer to Restriction reference on page 61 for more information about the types of restrictions used for criteria.

List examples
The following example demonstrates how to retrieve a list of vendor summaries. The vendor summary list produced by the example contains only vendors with the specified vendor class property.

Notice how the object named classIdRestriction of the LikeRestrictionOfString class is used to specify the vendor class name. The classIdRestriction object is then used to populate the ClassId property of the vendor criteria object named vendorCriteria.

The vendorCriteria object is passed as the first parameter in the GetVendorList method. The GetVendorList method also requires a context object to be created and passed as the method’s second parameter.

```csharp
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            VendorCriteria vendorCriteria;
            VendorSummary[] vendorSummaryList;
            LikeRestrictionOfString classIdRestriction;

            // Create an instance of the web service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure that default credentials are being used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context object with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Specify the criteria for the vendor summaries to retrieve
            classIdRestriction = new LikeRestrictionOfString();
            classIdRestriction.EqualValue = "USA-US-I";
        }
    }
}
```
// Create a vendor criteria object and set the restriction
vendorCriteria = new VendorCriteria();
vendorCriteria.ClassId = classIdRestriction;

// Retrieve the list of vendor summaries
vendorSummaryList = wsDynamicsGP.GetVendorList(vendorCriteria, context);

MessageBox.Show("Total vendors in class: " + vendorSummaryList.Length);
}
}
}

Native endpoint
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            VendorCriteria vendorCriteria;
            VendorSummary[] vendorSummaryList;
            LikeRestrictionOfstring classIdRestriction;

            // Create an instance of the web service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context object with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;
            context.CultureName = "en-US";

            // Specify the criteria for the vendor summaries to retrieve
            classIdRestriction = new LikeRestrictionOfstring();
            classIdRestriction.EqualValue = "USA-US-I";

            // Create a vendor criteria object and set the restriction
            vendorCriteria = new VendorCriteria();
            vendorCriteria.ClassId = classIdRestriction;

            // Retrieve the list of vendor summaries
            vendorSummaryList = wsDynamicsGP.GetVendorList(vendorCriteria,

context);

MessageBox.Show("Total vendors in class: " +
vendorSummaryList.Length);

// Close the service
if (wsDynamicsGP.State != CommunicationState.Faulted)
{
    wsDynamicsGP.Close();
}
}
}

Example 2

The following example demonstrates how to retrieve a list of vendor summaries that were modified during a specific time period. Notice how the object named modifiedDateRestriction of the BetweenRestrictionOfNullableOfDateTime class is used to specify the time period to be examined. The ModifiedDate property of the criteria object requires that both the date and time portion of the value be specified. The values for this criteria property must also be converted to Universal Coordinated Time (UTC). The modifiedDateRestriction object is then used to populate the ModifiedDate property of the vendor criteria object named vendorCriteria.

The vendorCriteria object is passed as the first parameter in the GetVendorList method. The GetVendorList method also requires a context object to be created and passed as the method’s second parameter.

---

**GetList method**

**Example 2**

Legacy endpoint

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            VendorCriteria vendorCriteria;
            VendorSummary[] vendorSummaryList;
            BetweenRestrictionOfNullableOfDateTime modifiedDateRestriction;

            // Create an instance of the web service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Make sure that default credentials are being used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context object with which to call the service
            context = new Context();
        }
    }
}
```
// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context object
context.OrganizationKey = (OrganizationKey)companyKey;

// Specify the criteria for the vendor summaries to retrieve
modifiedDateRestriction = new
BetweenRestrictionOfNullableOfDateTime();
DateTime startDate = new
DateTime(2017, 4, 9, 8, 0, 0).ToUniversalTime();
DateTime endDate = new
DateTime(2017, 4, 9, 17, 0, 0).ToUniversalTime();
modifiedDateRestriction.From = startDate;
modifiedDateRestriction.To = endDate;

vendorCriteria = new VendorCriteria();
vendorCriteria.ModifiedDate = modifiedDateRestriction;

// Retrieve the list of vendor summaries
vendorSummaryList = wsDynamicsGP.GetVendorList(vendorCriteria, context);

MessageBox.Show("Total vendors modified today: " +
vendorSummaryList.Length);

Native endpoint
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            VendorCriteria vendorCriteria;
            VendorSummary[] vendorSummaryList;
            BetweenRestrictionOfNullableOfDateTime modifiedDateRestriction;

            // Create an instance of the web service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context object with which to call the service
            context = new Context();
// Specify which company to use (sample company)
cpanyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context object
context.OrganizationKey = (OrganizationKey)companyKey;
context.CultureName = "en-US";

// Specify the criteria for the vendor summaries to retrieve
modifiedDateRestriction = new
    BetweenRestrictionOfNullableOfdateTime();
DateTime startDate = new
    DateTime(2017, 4, 9, 8, 0, 0).ToUniversalTime();
DateTime endDate = new
    DateTime(2017, 4, 9, 17, 0, 0).ToUniversalTime();
modifiedDateRestriction.From = startDate;
modifiedDateRestriction.To = endDate;

vendorCriteria = new VendorCriteria();
vendorCriteria.ModifiedDate = modifiedDateRestriction;

// Retrieve the list of vendor summaries
vendorSummaryList = wsDynamicsGP.GetVendorList(vendorCriteria,
    context);

MessageBox.Show("Total vendors modified today: " +
    vendorSummaryList.Length);

// Close the service
if (wsDynamicsGP.State != CommunicationState.Faulted)
{
    wsDynamicsGP.Close();
}
}

Using the list
The objects returned in the collection won’t be editable. If the objects need to be
edited, or the necessary information isn’t contained in the summary object, you will
need to use the GetByKey method to retrieve the corresponding complete object.

Optimizing list operations
For optimal performance when retrieving lists of object from the Dynamics GP
service, it’s important that you structure the criteria carefully. Use the following
guidelines when setting up criteria:

• Use the simplest criteria that can perform the search. The more properties you
  specify for the criteria object, the longer the search will take.

• Use properties in the criteria object that correspond to indexed columns in the
database. In the Dynamics GP Web Service Reference, the properties of the
criteria objects that correspond to indexed columns are marked with a dagger
(†).
Restriction reference

Several types of restrictions used with criteria objects. Some restriction types correspond to the various types of data, such as string values, integers, and booleans. Other restriction types are based on how the restriction is structured. For instance a “list” restriction specifies a collection of possible values for a property.

The restriction types are part of a hierarchy, with more complex restriction types being constructed from the simpler types. All restriction objects derive from the Restriction class in the Dynamics GP service.

Basic restrictions
The most basic restriction contains a single value of a specific type. These restrictions have a name with the form:

RestrictionOfNullableOfType

Some restrictions of this type are for primitive types. For example the restriction RestrictionOfNullableOfInt32 represents a single integer value for the legacy endpoint. For the native endpoint, this restriction type is named RestrictionOfNullableOfint. Other restrictions of this type represent a single instance of a more complex piece of information. For instance, the restriction RestrictionOfNullableOfSalesDocumentType represents a single type of sales document.

The following C# example shows how to create and use a basic restriction for a criteria object. This example creates a boolean restriction for the IsOnHold property of the customer criteria object.

Legacy endpoint
CustomerCriteria customerCriteria;
RestrictionOfNullableOfBoolean onHoldRestriction;
onHoldRestriction = new RestrictionOfNullableOfBoolean();
onHoldRestriction.EqualValue = true;
customerCriteria = new CustomerCriteria();
customerCriteria.IsOnHold = onHoldRestriction;

Native endpoint
CustomerCriteria customerCriteria;
RestrictionOfNullableOfboolean onHoldRestriction;
onHoldRestriction = new RestrictionOfNullableOfboolean();
onHoldRestriction.EqualValue = true;
customerCriteria = new CustomerCriteria();
customerCriteria.IsOnHold = onHoldRestriction;

List restrictions
The “list” restrictions specify a distinct set of values. These restrictions have a name with the form:

ListRestrictionOfNullableOfType

Like the basic restrictions, some of these list restrictions are for primitive types and others for more complex types. List restrictions represent a distinct set of values of the specific types.
The following C# example shows how to create and use a list restriction for a criteria object. This example creates a list restriction for the Type property of the item criteria object. The restriction contains a list of two item types, for Service and SalesItem. Notice how an array is created that contains the two possible values for the list restriction.

```csharp
ItemCriteria itemCriteria;
ListRestrictionOfNullableOfItemType typeRestriction;
ItemType?[] itemTypes;

typeRestriction = new ListRestrictionOfNullableOfItemType();
itemTypes = new ItemType?[2];
itemTypes[0] = ItemType.Service;
itemTypes[1] = ItemType.SalesItem;
typeRestriction.Items = itemTypes;
itemCriteria = new ItemCriteria();
itemCriteria.Type = typeRestriction;
```

### Between restrictions

The “between” restrictions specify a range of values. You can specify a GreaterThan value, a LessThan value, or From and To values to specify a range. These restrictions have a name with the form:

`BetweenRestrictionOfNullableOf>Type`

The between restrictions are available only for primitive types, like date/time or string values.

The following C# example shows how to create and use a between restriction for a criteria object. This example creates a between restriction for the Date property of a GL transaction criteria object. The property is set to a range between April 25, 2007 and April 26, 2007.

#### Legacy endpoint

```csharp
GLTransactionCriteria transactionCriteria;
BetweenRestrictionOfNullableOfDateTime dateRestriction;

dateRestriction = new BetweenRestrictionOfNullableOfDateTime();
DateTime startDate = new DateTime(2007, 4, 25);
DateTime endDate = new DateTime(2007, 4, 26);
dateRestriction.From = startDate;
dateRestriction.To = endDate;
transactionCriteria = new GLTransactionCriteria();
transactionCriteria.Date = dateRestriction;
```

#### Native endpoint

```csharp
GLTransactionCriteria transactionCriteria;
BetweenRestrictionOfNullableOfDateTime dateRestriction;

dateRestriction = new BetweenRestrictionOfNullableOfDateTime();
DateTime startDate = new DateTime(2007, 4, 25);
DateTime endDate = new DateTime(2007, 4, 26);
dateRestriction.From = startDate;
dateRestriction.To = endDate;
transactionCriteria = new GLTransactionCriteria();
transactionCriteria.Date = dateRestriction;
```
Like restriction
String properties for criteria object have unique behavior. A “Like” expression, similar to those used in Transact-SQL can be used to define the restriction. The LikeRestrictionOfString restriction inherits from the “between” and “list” restriction types, so restrictions of this type can also be used as well.

The following C# example shows how to create and use a “Like” restriction for a string property of a criteria object. This example creates a restriction for the State property of the customer criteria object. Notice how the “Like” clause is used to specify customers from the state “ND”.

Legacy endpoint
```csharp
CustomerCriteria customerCriteria;
LikeRestrictionOfString stateRestriction;

stateRestriction = new LikeRestrictionOfString();
stateRestriction.Like = "%ND%";
customerCriteria = new CustomerCriteria();
customerCriteria.State = stateRestriction;
```

Native endpoint
```csharp
CustomerCriteria customerCriteria;
LikeRestrictionOfString stateRestriction;

stateRestriction = new LikeRestrictionOfString();
stateRestriction.Like = "%ND%";
customerCriteria = new CustomerCriteria();
customerCriteria.State = stateRestriction;
```
Chapter 12: Updating Objects

The Dynamics GP service can be used to change the values of individual properties for existing objects in Dynamics GP. Information about updating objects is divided into the following sections:

- **Update methods**
- **Update policy**
- **Concurrency**

**Update methods**

The “Update” methods are used to change the values of an existing object’s properties. Most updates can be performed using the following steps:

- Instantiate an object of the type to be updated.
- Create and set the key value for the object being updated.
- Populate only the properties that are to be updated.
- Create a policy object for the update operation. A policy object provides additional control for the update operation. You will learn more about the policies in **Update policy** on page 69.
- Use the object as a parameter for the Update method.

The following example demonstrates updating a vendor object. Notice how the vendor key object’s Id property is populated with the ID of an existing vendor. A new vendor object will use this vendor key to specify which vendor to update. A value is supplied for the vendor object’s Name property.

The vendor object is then passed as the first parameter of the **UpdateVendor** method. The **UpdateVendor** method also requires context and policy objects to be created and passed as the second and third parameters.

Notice how the **GetPolicyByOperation** method instantiates the vendorPolicy object. The “UpdateVendor” parameter specifies the default update policy for a vendor object. **GetPolicyByOperation** also requires a context object for its second parameter.

```csharp
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
```
VendorKey vendorKey;
Vendor vendor;
Policy vendorPolicy;

// Create an instance of the service
DynamicsGP wsDynamicsGP = new DynamicsGP();

// Make sure the default credentials are being used
wsDynamicsGP.UseDefaultCredentials = true;

// Create a context object with which to call the web service
context = new Context();

// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context
colorize.OrganizationalKey = companyKey;

// Specify the vendor to be updated
vendorKey = new VendorKey();
vendorKey.Id = "ACETRAVE0001";

// Create the vendor object and set the value of
// the field to be updated
vendor = new Vendor();
vendor.Name = "A Travel Company, Inc.";
vendor.Key = vendorKey;

// Get the update policy for vendor
vendorPolicy = wsDynamicsGP.GetPolicyByOperation("UpdateVendor", context);

// Update the vendor information
wsDynamicsGP.UpdateVendor(vendor, context, vendorPolicy);

Native endpoint
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            VendorKey vendorKey;
            Vendor vendor;
CHAPTER 12  UPDATING OBJECTS

Policy vendorPolicy;

// Create an instance of the service
DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

// Create a context object with which to call the web service
context = new Context();

// Specify which company to use {sample company}
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context
context.OrganizationKey = companyKey;

// Specify the vendor to be updated
vendorKey = new VendorKey();
vendorKey.Id = "ACETRAVE0001";

// Create the vendor object and set the value of
// the field to be updated
vendor = new Vendor();
vendor.Name = "A Travel Company, Inc.";
vendor.Key = vendorKey;

// Get the update policy for vendor
vendorPolicy = wsDynamicsGP.GetPolicyByOperation("UpdateVendor", context);

// Update the vendor information
wsDynamicsGP.UpdateVendor(vendor, context, vendorPolicy);

// Close the service
if (wsDynamicsGP.State != CommunicationState.Faulted)
{
    wsDynamicsGP.Close();
}
}
}

When performing an update on a sales document (backorder, fulfillment order, invoice, order, quote, or return), the object must first be populated by a retrieve method. Retrieving the object ensures no data fields will be lost when the update is completed.

The following example demonstrates how to update a sales return object. Notice how the sales document key object specifies the sales document to be updated. The GetSalesReturnByKey method uses the key object along with a context object to return the specified sales return object. The sales return object’s Comment property is populated with a new value.

The update occurs when the sales return object is passed as the first parameter to the UpdateSalesReturn method. The UpdateSalesReturn method also requires context and policy objects to be created and passed as the second and third parameters.

Notice how the GetPolicyByOperation method instantiates the salesReturnPolicy object with the default policy for updating sales returns.
using System;
using System.Collections.Generic;
using System.Text;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;

            // Create an instance of the web service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure that default credentials are used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Set up a policy object
            Policy salesReturnPolicy = wsDynamicsGP.GetPolicyByOperation
            ("UpdateSalesReturn", context);

            // Create a Sales Return key
            SalesDocumentKey salesReturnKey = new SalesDocumentKey();
            salesReturnKey.Id = "RTN10010";

            // Retrieve the sales return object
            SalesReturn salesReturn = wsDynamicsGP.GetSalesReturnByKey
            (salesReturnKey, context);

            // Update the comment property
            salesReturn.Comment = "Update test comment";

            // Update the sales return object
            wsDynamicsGP.UpdateSalesReturn(salesReturn, context, salesReturnPolicy);
        }
    }
}
Update policy

An Update method requires an instance of the “update” policy for the object being updated. The update policy contains a set of behaviors that control various aspects of the update operation, such as how certain properties will be set, or whether additional actions will be performed. The web service administrator controls all of the internal behavior options through the Dynamics Security console. The external behaviors can be set by the application that is calling the update method.

When you call an Update method for the Dynamics GP service, you must retrieve an instance of the policy used for that method. Do this using either the GetPolicyByOperation web method or the GetPolicyByKey web method. If the context object you pass to either method specifies a valid role for the user, the policy that is configured for that role will be returned. Otherwise, the default policy for the operation will be returned.

Once the policy for the update operation has been retrieved, any external behaviors for the policy can be changed, if needed. The policy object will be passed to the web service Update method.

In the following C# example, a quantity for a line item in a sales order document is updated. The policy for the UpdateSalesOrder web method is retrieved. One behavior for this policy controls whether promotions are to be used. The ID (GUID) for this behavior is looked up from the Dynamics GP Web Service Reference and used to locate the behavior. The behavior option that indicates that promotions should be used is located, based on the ID from the Dynamics GP Web Service Reference. The selected behavior option for the behavior is set to this specific behavior option.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;
using System.Xml;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            SalesDocumentKey salesOrderKey;
            SalesOrder salesOrder;
            Policy salesOrderUpdatePolicy;
            Behavior promotionBehavior;
            BehaviorOption includePromotionsBehaviorOption;
            Guid behaviorGUID;
            int behaviorOptionID;

            // Create an instance of the web service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure the default credentials are used
            wsDynamicsGP.UseDefaultCredentials = true;
        }
    }
}
```
// Create a context with which to call the service
context = new Context();

// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context object
customizarOrganizationKey = (OrganizationKey)companyKey;

// Create a sales document key
salesOrderKey = new SalesDocumentKey();
salesOrderKey.Id = "ORDST2227";

// Retrieve the sales order
salesOrder = wsDynamicsGP.GetSalesOrderByKey(salesOrderKey, context);

// Update the first line of the sales order
salesOrder.Lines[0].Quantity.Value = 2;

// Retrieve the update policy for sales orders
salesOrderUpdatePolicy = wsDynamicsGP.GetPolicyByOperation("UpdateSalesOrder", context);

// Find the behavior in the list (from Web Service Reference)
behaviorGUID = new Guid("2e58b57c-8fe1-4e98-bf04-371621f8e8b7");
promotionBehavior = new Behavior();
foreach (Behavior b in salesOrderUpdatePolicy.Behaviors)
{
    if (b.Key.Id == behaviorGUID)
    {
        // Behavior was found
        promotionBehavior = b;
        break;
    }
}

// Find the behavior option (from Web Service Reference)
behaviorOptionID = 1;
includePromotionsBehaviorOption = new BehaviorOption();
foreach (BehaviorOption bo in promotionBehavior.Options)
{
    if (bo.Key.Id == behaviorOptionID)
    {
        // Behavior option was found
        includePromotionsBehaviorOption = bo;
        break;
    }
}

// Set the selected behavior option
promotionBehavior.SelectedOption = includePromotionsBehaviorOption;
// Update the sales order object
wsDynamicsGP.UpdateSalesOrder(salesOrder, context,
salesOrderUpdatePolicy);
}
}
}

Concurrency

When update operations are performed, the Dynamics GP service must handle any concurrency issues that occur. These issues happen because other users have locked the same rows in the database that the update operation is trying to change. Documents for which the Microsoft Dynamics GP client application actively locks rows, such as sales document, can produce these situations.

Typically, other users in Microsoft Dynamics GP won’t see concurrency issues caused by Dynamics GP service operations. The update operations occur quickly, and hold row locks for only the time necessary to perform the update. If a concurrency issue is caused by a Dynamics GP service update operation, the other user affected will see the standard message displayed that indicates another user has updated a record.

The most likely scenario in which a Dynamics GP service update operations will encounter concurrency issues occurs when the web service application is updating a large number of rows that the Microsoft Dynamics GP client can actively lock. For example, if a web service application was updating all of the sales orders created on a specific date, and other users are accessing the Microsoft Dynamics GP system, it is possible the web service application will encounter one of the sales documents being locked (in use). If this occurs, a validation exception will occur, indicating the sales order being updated was in use.

Your code must be able to handle concurrency exceptions that occur. The following C# example demonstrates one way this can be done. This example updates the comment for all sales orders created on 3/17/10. If a concurrency exception occurs because a sales order is being edited by another user, the following message is displayed. The user can choose to try updating the sales order again, or canceling the update and moving to the next document.

In this example, a list of the sales orders for the specific date is retrieved. The comment for each of these order is updated. Note the try...catch block that handles the validation error if a concurrency issue occurs. If the user chooses to retry the update operation, the goto statement causes the update operation to be run again.
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            BetweenRestrictionOfNullableOfDateTime transactionDateRestriction;
            SalesOrderCriteria salesOrderCriteria;
            SalesOrderSummary[] salesOrderSummary;
            DateTime dateValue;
            SalesDocumentKey salesOrderKey;
            SalesOrder salesOrder;
            Policy salesOrderUpdatePolicy;
            ValidationResult validationResult;

            // Create an instance of the service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure the default credentials are used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a date restriction object
            transactionDateRestriction = new
            BetweenRestrictionOfNullableOfDateTime();
            dateValue = new DateTime(2010, 3, 17);
            transactionDateRestriction.EqualValue = dateValue;

            // Create a sales order criteria object
            salesOrderCriteria = new SalesOrderCriteria();
            salesOrderCriteria.Date = transactionDateRestriction;

            // Retrieve the sales order summaries specified by the criteria
            salesOrderSummary = wsDynamicsGP.GetSalesOrderList(salesOrderCriteria, context);
// Retrieve the update policy for sales orders
salesOrderUpdatePolicy = wsDynamicsGP.GetPolicyByOperation("UpdateSalesOrder", context);

// Update the comment for each of the documents found
foreach (SalesOrderSummary a in salesOrderSummary)
{
    // Create a sales document key
    salesOrderKey = new SalesDocumentKey();
    salesOrderKey = a.Key;

    // Retrieve the sales order
    salesOrder = wsDynamicsGP.GetSalesOrderByKey(salesOrderKey, context);

    // Set the comment property
    salesOrder.Comment = "Customer notified - March 17";

    // Update the sales order object
    try
    {
        wsDynamicsGP.UpdateSalesOrder(salesOrder, context, salesOrderUpdatePolicy);
    }
    catch (SoapException soapErr)
    {
        // Try retrieving the Message node
        XmlDocument doc = new XmlDocument();
        XmlNamespaceManager nsManager = new
            XmlNamespaceManager(doc.NameTable);
        nsManager.AddNamespace("sm",
        doc.LoadXml(soapErr.Detail.InnerXml);
        XmlNode node = doc.SelectSingleNode("//sm:InnerException//sm:Message", nsManager);
        if (node != null)
        {
            // Use the GUID to load the validation details
            Guid LogId = new Guid(node.InnerText.Trim());
            validationResult = wsDynamicsGP.GetLoggedValidationResultByKey(LogId, context);

            if (validationResult.Errors[0].Id == "EConnectError-2079")
            {
                // It is a concurrency error, with record in use
                if (MessageBox.Show(null, "Document: " + a.Key.Id + ". " +
                    validationResult.Errors[0].Message + ".", "Sales Document Update",
                    MessageBoxButtons.RetryCancel,
                    MessageBoxIcon.Warning,
                    MessageBoxIcon.Warning, MessageBoxDefaultButton.Button2));
            }
        }
    }
}
Native endpoint

using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WebServiceSample.DynamicsGPService;
using System.ServiceModel;
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            BetweenRestrictionOfNullableOfDateTime transactionDateRestriction;
            SalesOrderCriteria salesOrderCriteria;
            SalesOrderSummary[] salesOrderSummary;
            DateTime dateValue;
            SalesDocumentKey salesOrderKey;
            SalesOrder salesOrder;
            Policy salesOrderUpdatePolicy;
            ValidationResult validationResult;

            // Create an instance of the service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the service

            MessageBoxDefaultButton.Button1) == DialogResult.Retry
            {
                goto UpdateSalesDocument;
            }
            else
            {
                // It is a different error
                MessageBox.Show(null, validationResult.Errors[0].Message, "Sales Document Update");
            }
            else
            {
                // Display the system exception message
                MessageBox.Show(soapErr.Message, "Sample", MessageBoxButtons.OK, MessageBoxIcon.Stop);
            }
        }
    }
}
context = new Context();

// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context object
context.OrganizationKey = (OrganizationKey)companyKey;

// Create a date restriction object
transactionDateRestriction = new
   BetweenRestrictionOfNullableOfdateTime();
dateValue = new DateTime(2010, 3, 19);
transactionDateRestriction.EqualValue = dateValue;

// Create a sales order criteria object
salesOrderCriteria = new SalesOrderCriteria();
salesOrderCriteria.Date = transactionDateRestriction;

// Retrieve the sales order summaries specified by the criteria
salesOrderSummary = wsDynamicsGP.GetSalesOrderList(
salesOrderCriteria, context);

// Retrieve the update policy for sales orders
salesOrderUpdatePolicy = wsDynamicsGP.GetPolicyByOperation(
"UpdateSalesOrder", context);

// Update the comment for each of the documents found
foreach (SalesOrderSummary a in salesOrderSummary)
{
   // Create a sales document key
   salesOrderKey = new SalesDocumentKey();
   salesOrderKey = a.Key;

   // Retrieve the sales order
   salesOrder = wsDynamicsGP.GetSalesOrderByKey(salesOrderKey, context);

   // Set the comment property
   salesOrder.Comment = "Customer notified - March 17";

   // Update the sales order object
   UpdateSalesDocument:
   try
   {
      wsDynamicsGP.UpdateSalesOrder(salesOrder, context,
         salesOrderUpdatePolicy);

      // Close the service
      if (wsDynamicsGP.State != CommunicationState.Faulted)
      {
         wsDynamicsGP.Close();
      }
   }
   catch (FaultException<System.ServiceModel.ExceptionDetail> ex)
   {
      if (ex.Detail.InnerException != null)
      {
         // Use the GUID to load the validation details
         
      }
   }
}
Guid LogId = new Guid(
    ex.Detail.InnerException.Message.Trim());

// Get the validation result object
validationResult = wsDynamicsGP.
GetLoggedValidationResultByKey(LogId, context);

if (validationResult.Errors[0].Id ==
"EConnectError-2079")
{
    // It is a concurrency error, with record in use
    if (MessageBox.Show(null,
        "Document: " + a.Key.Id + ". " +
        validationResult.Errors[0].Message + ".",
        "Sales Document Update",
        MessageBoxButtons.RetryCancel,
        MessageBoxIcon.Warning,
        MessageBoxIcon.DefaultButton.Button1) ==
        DialogResult.Retry)
    {
        goto UpdateSalesDocument;
    }
    else
    {
        if (wsDynamicsGP.State !=
            CommunicationState.Faulted)
        {
            wsDynamicsGP.Close();
        }
    }
}
else
{
    // It is a different error
    MessageBox.Show(null,
        validationResult.Errors[0].Message,
        "Sales Document Update");
}
else
{
    // Display the system exception message
    MessageBox.Show(ex.Message, "Sample",
        MessageBoxButtons.OK, MessageBoxIcon.Stop);

    if (wsDynamicsGP.State != CommunicationState.Faulted)
    {
        wsDynamicsGP.Close();
    }
}
}
Chapter 13: Deleting or Voiding Objects

An individual object can be deleted or voided through the Dynamics GP service. Deleted objects will be removed from the database. Voided object will remain in the database, but be in the voided state. Information about deleting or voiding objects is divided into the following sections:

- **Delete or void methods**
- **Delete or void policy**

**Delete or void methods**

The “Delete” or “Void” methods remove an object from or void an object in the database. To delete or void an object, use the following basic steps:

- Create a key object for the type of document you want to delete or void.
- Populate the key object with the unique ID of the document.
- Create a policy object for the delete or void operation. A policy object provides additional control for the delete or void operation. You will learn more about the policies in **Delete or void policy** on page 79.
- Pass the key object to the Delete or Void method.

The following C# example demonstrates deleting a customer. Notice how a customer key object is created and populated with the unique ID value “CONTOSO” for the customer. The DeleteCustomer policy is retrieved to populate the policy object. The customer key object is passed as the first parameter to the DeleteCustomer method. The DeleteCustomer method also requires the context and policy objects to be passed as the second and third parameters.

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            CustomerKey customerKey;
            Policy customerPolicy;

            // Create an instance of the service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure that default credentials are being used
            wsDynamicsGP.UseDefaultCredentials = true;
        }
    }
}
```
Native endpoint

```csharp
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            CustomerKey customerKey;
            Policy customerPolicy;

            // Create an instance of the service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (lesson company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a customer key
            customerKey = new CustomerKey();
            customerKey.Id = "CONTOSO";

            // Get the delete policy for the customer
            customerPolicy = wsDynamicsGP.GetPolicyByOperation
                ("DeleteCustomer", context);

            // Delete the customer
            wsDynamicsGP.DeleteCustomer(customerKey, context, customerPolicy);
        }
    }
}
CHAPTER 13  DELETING OR VOIDING OBJECTS

// Get the delete policy for the customer
customerPolicy = wsDynamicsGP.GetPolicyByOperation("DeleteCustomer", context);

// Delete the customer
wsDynamicsGP.DeleteCustomer(customerKey, context, customerPolicy);

// Close the service
if (wsDynamicsGP.State != CommunicationState.Faulted)
{
    wsDynamicsGP.Close();
}

Refer to Chapter 15, "Policy," for more information about using policies.

Delete or void policy

A Delete or Void method requires an instance of the “delete” policy for the object being deleted or the “void” policy for the object being voided. The delete or void policy contains a set of behaviors that control various aspects of how the operation will be performed. The web service administrator controls all of the internal behavior options through the Dynamics Security console. The external behaviors can be set by the application that is calling the delete or void method.

When you call a Delete or Void method for the Dynamics GP service, you must retrieve an instance of the policy used for that method. Do this using either the GetPolicyByOperation web method or the GetPolicyByKey web method. If the context object you pass to either method specifies a valid role for the user, the policy that is configured for that role will be returned. Otherwise, the default policy for the operation will be returned.

Once the policy for the delete or void operation has been retrieved, any external behaviors for the policy can be changed, if needed. The policy object will be passed to the web service Delete or Void method.

In the following C# example, a sales order is deleted. The policy for the DeleteSalesOrder web method is retrieved. One behavior for this policy controls whether posted deposits will be removed. The ID (GUID) for this behavior is looked up from the Dynamics GP Web Service Reference and used to locate the behavior. The behavior option specifying that posted deposits should not be removed is also looked up in the reference. Finally, the selected option for the behavior is set to the option to not remove posted deposits.

```csharp
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        using System;
        using System.Collections.Generic;
        using System.Text;
        using System.Windows.Forms;
        using DynamicsGPWebServiceSample.DynamicsGPService;
        using System.Xml;

        namespace DynamicsGPWebServiceSample
        {
            class Program
            {
```
static void Main(string[] args)
{
    CompanyKey companyKey;
    Context context;
    SalesDocumentKey salesOrderKey;
    Policy salesOrderDeletePolicy;
    Behavior removePostedDepositsBehavior;
    BehaviorOption doNotRemoveBehaviorOption;
    Guid behaviorGUID;
    int behaviorOptionID;

    // Create an instance of the web service
    DynamicsGP wsDynamicsGP = new DynamicsGP();

    // Be sure the default credentials are used
    wsDynamicsGP.UseDefaultCredentials = true;

    // Create a context with which to call the web service
    context = new Context();

    // Specify which company to use (sample company)
    companyKey = new CompanyKey();
    companyKey.Id = (-1);

    // Set up the context object
    context.OrganizationKey = (OrganizationKey)companyKey;
    context.CultureName = "en-US";

    // Create a sales document key
    salesOrderKey = new SalesDocumentKey();
    salesOrderKey.Id = "ORDST2226";

    // Get the sales order delete policy
    salesOrderDeletePolicy = wsDynamicsGP.GetPolicyByOperation("DeleteSalesOrder", context);

    // Find the behavior in the list (from Web Service Reference)
    behaviorGUID = new Guid("0f2c414b-fd12-4d83-940b-2bc0d65c3b77");
    removePostedDepositsBehavior = new Behavior();
    foreach (Behavior b in salesOrderDeletePolicy.Behaviors)
    {
        if (b.Key.Id == behaviorGUID)
        {
            // Behavior was found
            removePostedDepositsBehavior = b;
            break;
        }
    }

    // Find the behavior option (from Web Service Reference)
    behaviorOptionID = 0;
    doNotRemoveBehaviorOption = new BehaviorOption();
    foreach (BehaviorOption bo in
        removePostedDepositsBehavior.Options)
    {
        if (bo.Key.Id == behaviorOptionID)
{  
    // Behavior option was found  
    doNotRemoveBehaviorOption = bo;  
    break;  
  }
}

// Set selected behavior option to create a reversing transaction
removePostedDepositsBehavior.SelectedOption = doNotRemoveBehaviorOption;

// Delete the specified sales order
wsDynamicsGP.DeleteSalesOrder(salesOrderKey, context, salesOrderDeletePolicy);
}
Chapter 14: Exceptions

The Dynamics GP service returns errors as SOAP exceptions. The SOAP exceptions are classified as either system exceptions or validation exceptions. System exceptions are events that prevent normal completion of Dynamics GP service methods. Validation exceptions occur when data passed into a method violates the data validation rules for an object.

The Dynamics GP service maintains an exception log that contains detailed information for each exception that occurs. You can view the exception log entries to help you troubleshoot problems encountered by the Dynamics GP service methods. Methods in the Dynamics GP service also provide access to validation exception log entries so they can be handled or displayed by the application calling the service.

Information about exceptions is divided into the following sections:

- **Exceptions console**
- **System exceptions**
- **Validation exceptions**
- **Handling exceptions in code**

Exceptions console

To view exceptions that have been logged, you can use the Dynamics GP Web Services Exceptions console. This is a snap-in for Microsoft Management Console (MMC) that retrieves and displays the exceptions logged by the Dynamics GP service. The console is shown in the following illustration.

![Exceptions console screenshot](image)

Opening the console

To open the Dynamics GP Web Services Exceptions, choose the item from the Administrative Tools menu accessed from the Start menu.

Selecting exceptions to display

The console can display system exceptions, validation exceptions, or both. Use the pane on the left to select which type of exceptions you want to view. The date controls on the top of the console specify the date range of the exceptions you want to display. By default, the current day’s exceptions are displayed.
System exception details

To view detailed information for a system exception, select it in the list and choose Properties from the Action menu. Detailed properties for the system exception will be displayed.

Some exceptions will have additional details that you can view when you click the View Exception Detail button. Use the additional information provided, such as the messages and stack trace, to help you determine what caused the exception to occur.

There may be several levels of inner-exception details that you can drill down into. Be sure to examine all of the levels for useful troubleshooting details.

You can click View Request XML to see the complete XML message that was sent to the web service to be processed.
Validation exception details
To view detailed information for a validation exception, select it in the list and choose Properties from the Action menu. Detailed properties for the validation exception will be displayed.

Click View Validation Results to see further details about the validation error and validation warnings that were logged. This additional detail can help you determine what validation rules have been broken.

System exceptions
A system exception occurs when an unexpected event prevents the normal completion of a web method. The web service returns a SOAP exception to indicate the error occurred, and logs the details in the exception log. To improve security, a generic system exception message will be returned, telling the user to contact the system administrator for more details about the error. The details of the exception must be examined by the Dynamics GP service administrator.
A logged data exception object represents a system exception. The `GetLoggedExceptionDataByKey` method uses the exception’s log ID to retrieve a logged data exception object.

The `GetLoggedExceptionDataList` method returns a list of logged exception summary objects. The method returns the list of exceptions logged during a specified time period.

**Validation exceptions**

A validation exception occurs when an object contains data that violates the method’s data validation rules. A validation result object contains the detailed exception information from the exception log. Your application that uses the Dynamics GP service can trap these validation exceptions, and display an appropriate message describing the validation issues.

When a validation exception occurs, the exception contains an ID which identifies the exception in the exception log. The `GetLoggedValidationResultByKey` method retrieves the validation result object for the specified log ID. The `GetLoggedValidationResultByKey` method also requires a context object to be passed as the second parameter.

**Handling exceptions in code**

You should write your code so that it can handle the system exceptions and validation exceptions that occur when you call methods from the Dynamics GP service. Typically, this means using a try/catch structure. In the try block, you place your code that calls the Dynamics GP service. The catch block handles the exceptions that might occur. This technique is used for both the legacy endpoint and the native endpoint, though the details of handling exceptions are different for the two endpoints.

**Legacy endpoint**

For the legacy endpoint, exceptions are returned as `SoapException` objects. Your code must reference the `System.Web.Services.Protocols` namespace so that this `SoapException` class is available.

When an exception occurs, your code has to determine whether it is a system exception or a validation exception. If the exception is a system exception, you can simply display the message included in the `SoapException` object. If the exception is a validation exception, you can retrieve and display the additional details about what was not valid.

One way to find out what type of exception occurred is to examine the `InnerXml` data of the `Detail` property for the `SoapException` object. If the `InnerXml` data has a `<Message>` node, then the exception is a validation exception. The `<Message>` node value is the GUID that is used to retrieve the validation exception details. If there is no `<Message>` node in the `InnerXml` data, the exception is a system exception.

The following C# example shows exception handling code for the legacy endpoint. The try/catch block traps and handles the SOAP exception. The code in the catch block determines the type of exception. If it is a system exception, the exception message is displayed. If it is a validation exception, the validation details are retrieved and displayed. Notice how the vendor object and the `UpdateVendor` method creates data that causes two validation exceptions.
namespace DynamicsGPWebServiceSample
{

class Program
{
    static void Main(string[] args)
    {
        CompanyKey companyKey;
        Context context;
        VendorKey vendorKey;
        Vendor vendor;
        Policy policy;
        ValidationResult validationResult;

        // Create an instance of the service
        DynamicsGP wsDynamicsGP = new DynamicsGP();

        // Make sure that default credentials are being used
        wsDynamicsGP.UseDefaultCredentials = true;

        // Create a context with which to call the service
        context = new Context();
        context.OrganizationKey = (OrganizationKey)companyKey;

        // Specify which company to use (sample company)
        companyKey = new CompanyKey();
        companyKey.Id = (-1);
        context.OrganizationKey = (OrganizationKey)companyKey;

        // Create a vendor key of the vendor to be updated
        vendorKey = new VendorKey();
        vendorKey.Id = "ACETRAVE0001";

        // Create a vendor object
        vendor = new Vendor();
        vendor.Key = vendorKey;

        // Set a pair of properties to create two validation errors
        vendor.DiscountGracePeriod = 100;
        vendor.DueDateGracePeriod = 100;

        try
        {
            // Create a policy object
            policy = wsDynamicsGP.GetPolicyByOperation
                          ("UpdateVendor", context);

            // Update the vendor
            validationResult = vendor.UpdateVendor();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

Legacy endpoint
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using System.Xml;
using DynamicsGPWebServiceSample.DynamicsGPService;
// Attempt to update the vendor
wsDynamicsGP.UpdateVendor(vendor, context, policy);
}
catch (SoapException soapErr)
{
  // Examine the InnerXml document to find whether the
  // /InnerException/Message node exists. If it does, then
  // this is a validation exception. If it does
  // not, then this is a system exception.

  // Try retrieving the Message node
  XmlDocument doc = new XmlDocument();
  XmlNamespaceManager nsManager = new
    XmlNamespaceManager(doc.NameTable);
  nsManager.AddNamespace("sm",
  doc.LoadXml(soapErr.Detail.InnerXml);
  XmlNode node = doc.SelectSingleNode(
    "//sm:InnerException//sm:Message", nsManager);
  if (node != null)
  {
    // This is a validation exception

    // Use the GUID retrieved to load the validation details
    Guid LogId = new Guid(node.InnerText.Trim());

    // Get the validation result object
    validationResult = wsDynamicsGP.
      GetLoggedValidationResultByKey(LogId, context);

    // Display the validation exceptions
    StringBuilder validationList = new StringBuilder();
    foreach(ValidationError ve in validationResult.Errors)
    {
      validationList.AppendLine(ve.Message);
    }
    foreach(ValidationWarning vw in validationResult.Warnings)
    {
      validationList.AppendLine(vw.Message);
    }
    MessageBox.Show(validationList.ToString(), "Sample", 
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
  else
  {
    // This is a system exception

    // Display the message
    MessageBox.Show(soapErr.Message, "Sample", 
      MessageBoxButtons.OK, MessageBoxIcon.Stop);
  }
}
}
Native endpoint

For the native endpoint, exceptions are returned as FaultException objects. Your code must reference the System.ServiceModel namespace so that this FaultException class is available.

When an exception occurs, your code has to determine whether it is a system exception or a validation exception. If the exception is a system exception, you can simply display the message included in the SoapException object. If the exception is a validation exception, you can retrieve and display the additional details about what was not valid.

One way to find out what type of exception occurred is to examine the Detail.InnerException property for the FaultException object. If this property is not null, then the exception is a validation exception. The property Detail.InnerException.Message will contain the GUID that can be used to retrieve the validation exception details. If the Detail.InnerException property for the FaultException object is null, the exception is a system exception.

The following C# example shows exception handling code for the native endpoint. The try/catch block traps and handles the service fault exception. The code in the catch block determines the type of exception. If it is a system exception, the exception message is displayed. If it is a validation exception, the validation details are retrieved and displayed. Notice how the vendor object and the UpdateVendor method creates data that causes two validation exceptions.

```
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            VendorKey vendorKey;
            Vendor vendor;
            Policy policy;
            ValidationResult validationResult;

            // Create an instance of the service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);
```
// Set up the context
context.OrganizationKey = (OrganizationKey)companyKey;

// Create a vendor key of the vendor to be updated
vendorKey = new VendorKey();
vendorKey.Id = "ACETRAVE0001";

// Create a vendor object
vendor = new Vendor();
vendor.Key = vendorKey;

// Set a pair of properties to create two validation errors
vendor.DiscountGracePeriod = 100;
vendor.DueDateGracePeriod = 100;

try
{
    // Create a policy object
    policy = wsDynamicsGP.GetPolicyByOperation("UpdateVendor", context);

    // Attempt to update the vendor
    wsDynamicsGP.UpdateVendor(vendor, context, policy);
}
catch (FaultException<System.ServiceModel.ExceptionDetail> ex)
{
    // Examine the InnerException property to find whether it
    // is null. If it is, then this is a system exception. If
    // it is not, then this is a validation exception.

    if (ex.Detail.InnerException != null)
    {
        // This is a validation exception

        // Retrieve the GUID to load the validation details
        Guid LogId = new Guid(
            ex.Detail.InnerException.Message.Trim());

        // Get the validation result object
        validationResult = wsDynamicsGP.
                        GetLoggedValidationResultByKey(LogId, context);

        // Display the validation exceptions
        StringBuilder validationList = new StringBuilder();
        foreach (ValidationError ve in validationResult.Errors)
        {
            validationList.AppendLine(ve.Message);
        }
        foreach(ValidationWarning vw in validationResult.Warnings)
        {
            validationList.AppendLine(vw.Message);
        }
        MessageBox.Show(validationList.ToString(), "Sample",
                        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
}
else
{
    // This is a system exception

    // Display the message
    MessageBox.Show(ex.Message, "Sample",
        MessageBoxButtons.OK, MessageBoxIcon.Stop);
}
}
Chapter 15: Policy

Policy is a security-related feature for the Dynamics GP service. The policy system allows the Dynamics GP service administrator and the application using the Dynamics GP service to control how business objects are created, updated, or deleted. Information about policy is divided into the following sections:

- Policy overview
- Configuring from the Dynamics Security Console
- Configuring from code
- Using policy in a web service application

Policy overview

Each create, update, and delete operation has a policy object that is passed with the operation. This policy object specifies the set of behaviors for the operation. Each behavior controls one characteristic for the operation being performed. For instance, the policy for the CreateCustomer method has the behavior named “Create Active Behavior”. This behavior controls whether the customer being created is set to the active or inactive state.

Behaviors are classified as internal or external. An internal behavior is one that can be specified by only the Dynamics GP service administrator. An external behavior is one that can be specified by the application that is calling the method and passing in the policy object.

Some policies do not have any behaviors, but the policy object must still be passed with the call to the Dynamics GP service method.

Policy instances

Each company has a set of default policies that are always available. There is one default policy for each Dynamics GP service operation that requires a policy. Within a company, additional versions of the policy (with different behavior settings) can be created for each role defined in the Dynamics Security Service. Each of these is called a policy instance. When a web service application retrieves a policy to use, the Dynamics GP service applies logic to ensure the appropriate policy instance is returned. You will learn more about this in Using policy in a web service application on page 95.

Policy reference

The Dynamics GP Web Service Reference help file contains a complete list of the default policies that are defined for the operations in the Dynamics GP service. All of the behaviors (including internal and external) are listed, including a description of what the behavior does.
Configuring from the Dynamics Security Console

Most of the configuration for web service policy is performed using the Dynamics Security Console. The Policy node in the left-pane provides access to all of the policies that have been defined for the web service operations.

Information about configuring policy using this console can be found in the Web Services for Microsoft Dynamics GP Installation and Administration Guide.

Configuring from code

Several methods are available in the Dynamics GP service that can be used to configure policy settings from within a web service application. Details about these methods can be found in the Dynamics GP Web Service Reference online help file. Each of the policy-related methods provides a detailed example that demonstrates its use.

To use these methods, a user must be assigned to a role that can perform the operations in the Manage Policies task. The Policy Administrator role provides these capabilities.

Listing available policies

The following methods are used to get a list of available policies:

- GetPolicyList
- GetPolicyListByRoleKey

Listing roles for policy instances

The following method lists the roles for the various policy instances that have been created for a policy.

- GetPolicyRoles
Retrieving policy instances
The following methods are used to retrieve policy instances for use in a web service application:

- GetPolicyByKey
- GetCompletePolicyByKey
- GetPolicyByOperation

Creating, updating, and deleting policy instances
The following methods are used to create, update, and delete policy instances for specific policies.

- CreatePolicy
- UpdatePolicy
- DeletePolicy

Typically, policy instances for the various roles are managed using the Policy node in the Dynamics Security Console.

Using policy in a web service application
The create, update, and delete or void methods in the Dynamics GP service require the appropriate policy object to be passed to them. Your web service application must retrieve the appropriate policy instance to use. Your application can also control the external behaviors for the policy.

Retrieving a policy instance
To retrieve a policy instance to use for a method, use one of the following:

- GetPolicyByKey
- GetPolicyByOperation

Which you choose depends on whether you prefer to have GUIDs in your code that identify policies, or string values for operation names.

When a web service application retrieves the policy instance, the appropriate one will be returned, based on the context object that was used. The policy information will come from the company specified by the context object. If the context object has a role specified, the policy instance defined for that role will be returned. If no policy instance exists for that role within the specified company, the default policy for that operation will be returned.

If you don’t supply a role key, the Dynamics GP service will attempt to find a role for the user and company specified in the context object. If only one role can be found, that role will be used. If more than one role is found, or no roles are found, the default role will be used.

If you want to use a policy instance for a specific role, be sure that you specify the role key for the context object. Otherwise, it’s best to leave the role key property empty. This allows the Dynamics GP service administrator to control which policy instances are used based on the role a user is assigned to.
Setting external behaviors

After your web service application has retrieved a policy instance, you can decide whether to change the values for any external behaviors. This process involves locating the external behavior in the policy instance, finding the behavior option you want to use, and setting the selected behavior option.

You will need to use the Dynamics GP Web Service Reference to find the GUID value that identifies the behavior you want to set. The reference will also provide the ID of the behavior option you want to use. For example, the following multipart C# example demonstrates how the “Create Reversing Transaction” behavior for the Create GL Transaction policy is used. First, the behavior is found within the policy instance:

```
Policy transactionCreatePolicy;
Behavior reversingTrxBehavior;
Guid behaviorGUID;
BehaviorOption reversingTrxBehaviorOption;
int behaviorOptionID;

// Get the create policy for GL transactions
transactionCreatePolicy = wsDynamicsGP.GetPolicyByOperation("CreateGLTransaction", context);

// Find the behavior in the list (from Dynamics GP Service Reference)
behaviorGUID = new Guid("b0c9fe57-c4f0-4d6a-b0aa-ab4d6d7596c8");
reversingTrxBehavior = new Behavior();
foreach (Behavior b in transactionCreatePolicy.Behaviors)
{
    if (b.Key.Id == behaviorGUID)
    {
        // Behavior was found
        reversingTrxBehavior = b;
        break;
    }
}
```

The next portion of the C# code demonstrates how the behavior option to create a reversing transaction is found:

```
// Find the behavior option (from Dynamics GP Service Reference)
behaviorOptionID = 2;
reversingTrxBehaviorOption = new BehaviorOption();
foreach (BehaviorOption bo in reversingTrxBehavior.Options)
{
    if (bo.Key.Id == behaviorOptionID)
    {
        // Behavior option was found
        reversingTrxBehaviorOption = bo;
        break;
    }
}
```
Some behavior options have a parameter that can be set. In this case, the date of the reversing transaction can be set. The next portion of the C# code shows how the parameter for the reversing date is set:

```csharp
// Set the reversing date for the behavior option
reversingTrxBehaviorOption.Parameters[0].Value = reversingTrxDate.ToShortDateString();
```

Finally, the selected behavior option must be set for the behavior. The remainder of the C# code shows how the selected behavior is set for the “Create Reversing Transaction” behavior:

```csharp
// Set selected behavior option to create a reversing transaction
reversingTrxBehavior.SelectedOption = reversingTrxBehaviorOption;
```

The policy instance for the Create GL Transaction policy is ready to be passed to the `CreateGLTransaction` web method.
Chapter 16: Other Programming Techniques

The information presented here describes some additional programming techniques that you may find useful as you create applications that use the Dynamics GP service. The following items are discussed:

- Calling methods asynchronously
- Working on behalf of another user

Calling methods asynchronously

Some methods can take considerable time to process, especially when working with large data sets in Microsoft Dynamics GP. Rather than making the user wait for a lengthy service call to finish before returning control back to them, your application can call the method asynchronously. Control will be returned back to the application while the Dynamics GP service processes the request. When results are available, the event handler in your application will be notified and can act on them.

The “GetList” methods in the Dynamics GP service can have longer processing times, especially when the data set being searched is large, and the criteria is complex.

The proxy generated for the legacy endpoint always contains the asynchronous methods. The proxy generated for the native endpoint does not contain the asynchronous methods. If you want to use the asynchronous methods for the native endpoint, you must specify that they be generated when you create the service reference to connect to the native endpoint. Do this by clicking Advanced when you create the service reference, and then marking the Generate asynchronous operations option. Refer to Visual Studio 2008 on page 25 for details about creating a service reference.

The following C# example demonstrates how the GetCustomerList method can be called asynchronously. After setting up the criteria for the method call, the example creates an event handler to receive the response of the asynchronous call. Then the asynchronous call is made. The event handler at the end of the example receives the result when the call has finished. It examines the result for any errors, and if there are none, displays the number of customers that match the criteria.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            CustomerCriteria customerCriteria;
            LikeRestrictionOfString stateRestriction;
            LikeRestrictionOfString nameRestriction;
            RestrictionOfNullableOfBoolean activeRestriction;
```
// Create an instance of the service
DynamicsGP wsDynamicsGP = new DynamicsGP();

// Make sure that default credentials are being used
wsDynamicsGP.UseDefaultCredentials = true;

// Create a context with which to call the service
context = new Context();

// Specify which company to use (sample company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context
context.OrganizationKey = (OrganizationKey)companyKey;

// Create the criteria
customerCriteria = new CustomerCriteria();

// Specify the criteria for the customer summaries to retrieve
stateRestriction = new LikeRestrictionOfString();
stateRestriction.Like = "%IL%";
customerCriteria.State = stateRestriction;

nameRestriction = new LikeRestrictionOfString();
nameRestriction.Like = "A%";
customerCriteria.Name = nameRestriction;

activeRestriction = new RestrictionOfNullableOfBoolean();
activeRestriction.EqualValue = true;
customerCriteria.IsActive = activeRestriction;

// Use an event handler for the asynchronous web method
wsDynamicsGP.GetCustomerListCompleted += new
GetCustomerListCompletedEventHandler
(wsDynamicsGP_GetCustomerListCompleted);

// Retrieve the list of customer summaries (asynchronously)
wsDynamicsGP.GetCustomerListAsync(customerCriteria, context);

Console.WriteLine("Do additional work while waiting for result.");
Console.ReadLine();

// The event handler that responds to the completed event
static private void wsDynamicsGP_GetCustomerListCompleted(object
sender, GetCustomerListCompletedEventArgs e)
{
    if (e.Error == null)
    {
        // Display the number of customers matching the criteria
        MessageBox.Show("Active IL customers beginning with 'A': "+
e.Result.Length.ToString());
    }
}
}
Working on behalf of another user

An application that uses the Dynamics GP service may be running as a specific user, but being accessed by other different users. For example, a web-based application running on a server may have this situation. The application is running as a specific user (often a local user account defined on the server), but other users are accessing the application with their own logon credentials.

The various users of the application may have different security access for the Dynamics GP service. Some users of the application may be able to access methods in the Dynamics GP service that other users cannot. To provide proper security for the application, you will want to use the security settings for the user that is currently accessing the application. Since that user isn’t actually running the application, the application needs a way to perform work on behalf of that user. The `WorkOnBehalfOf` property of the Context object allows this capability when accessing the Dynamics GP service.

To use the `WorkOnBehalfOf` property, you need to create a role and assign the “Work on Behalf of Other Users” task to it. The user actually running the application (such as the local user on the server) must be assigned to this role. This allows that user to perform work on behalf of the other users who will be accessing the application. The following illustration shows this new role being created.

The “Work on Behalf of Other Users” task must be added to the new role you are creating.
When your application calls methods in the Dynamics GP service, you will set the default credentials on the service instance to be the local user that is actually running the application. For the Context object, you will set the WorkOnBehalfOf property to the login name for the user that is accessing the application. The security settings for that user will be used by the Dynamics GP service.
Chapter 17: Entity ID Filtering

Several objects in Dynamics GP have identity information associated with them. Examples include the Customer ID for customer objects, or the Salesperson ID for sales document objects. The Dynamics GP service can use entity ID filtering to restrict access to these objects, based on the Windows User ID of the person accessing the web service. Information about entity ID filtering is divided into the following sections:

- Setting up entity ID filtering
- Security for entity ID filtering
- Implementing entity ID filtering
- Filtering limitations
- Entity ID filtering example

Setting up entity ID filtering

Windows User IDs can be associated with the following objects in Microsoft Dynamics GP:

- Back Office User
- Customer
- Employee
- Salesperson
- Sales Territory
- Vendor

These objects are referred to as user-assignable business objects. Administrators of the Dynamics GP service use the Microsoft Dynamics Security Administration Console to create entity ID assignments. These assign Windows User IDs to specific objects in Microsoft Dynamics GP. For example, the following illustration shows the entity ID assignment that assigns the Windows user “CORPORATE\imarsh” to the salesperson object in Microsoft Dynamics GP with the ID value “IAN M.”.

Applications that access the Dynamics GP service can use these entity ID assignments when retrieving data. When applied, the user will be able retrieve only those objects for which their Windows User ID is associated. For instance, the user can be restricted to retrieve sales documents where the primary Salesperson ID for the document is associated with the user’s Windows User ID.
Security for entity ID filtering

Additional security roles, operations, and tasks are defined for the Dynamics GP service to support entity ID filtering. These roles, operations, and tasks indicate whether entity ID filtering will be applied for the current web service user.

For example, granting access to a role that contains the operation *Query Sales Orders* allows the user to retrieve any sales order. Granting access to a role that contains the operation *Query Sales Orders Based On User* allows the user to retrieve only those sales orders that have an ID (such as the Salesperson ID) mapped to the current Windows User.

Roles that contain the tasks and operations that implement entity ID filtering have the word “Self” in their name. Users assigned to these roles will be able to see only objects that are associated to them based on the entity ID assignments. For example, the *Salesperson - Self* role provides access to customer, salesperson, and sales transaction information for the salesperson assigned to the current user.

The following diagram shows how the security settings for entity ID filtering control access for a “GetByKey” operation in the Dynamics GP service:
When entity ID filtering is used for GetList operations, the Scope property of the criteria used for the operation also affects what values are returned. You will learn more about using the Scope property in *Implementing entity ID filtering* on page 105. The following diagram shows how the Scope property and security settings for entity ID filtering control access for a “GetList” operation in the Dynamics GP service:

![Diagram showing the flow of entity ID filtering](image)

### Implementing entity ID filtering

When using the “GetByKey” operations for the Dynamics GP service, be sure to check the Dynamics GP Web Service Reference help file to verify which operations can be affected by the entity ID filtering. The administrator for the Dynamics GP web service may have assigned roles that cause the entity ID assignments to be used. Your application must be able to handle any exceptions that occur when the web service administrator assigns one of the restricted operations. For instance, the application must handle cases where no entity ID assignment was found for the current user.

The “GetList” operations that support entity ID filtering have a Scope property for the selection criteria that specifies which objects are returned. The Scope property can be set to return all objects, which is the default behavior. It can also be set to return only those objects that match the specific type of entity ID assigned to the current user.
For example, the CustomerCriteria object supports entity ID filtering, so it contains the Scope property. This property can be set to “Return All” to return all customers. It can be set to “Return Based on Customer Id” to return the customer with an ID that matches the Customer ID assigned to the current user. It can also be set to “Return Based on Salesperson Id” to return the customers that have the Salesperson ID that matches the Salesperson ID assigned to the current user.

If a Windows User ID is assigned to multiple entity IDs in Microsoft Dynamics GP, the GetList operation will return values for each of the entities that are mapped.

If you use the entity ID filtering for GetList operations, be sure to handle the “access denied” exceptions that can occur if the Dynamics GP service administrator hasn’t given access to the proper roles. Be sure to document the restricted operations that are required for your application.

It’s a good idea to test your web service applications with the roles that provide full access to service operations and also the restricted roles that implement the entity ID filtering.

If your application uses the WorkOnBehalfOf property for the Context object, be aware that this changes the user for entity ID filtering. The WorkOnBehalfOf user, not the user running the application, will be used for the entity ID filtering.

Filtering limitations

The entity ID filtering restricts which documents will be returned based on a primary ID associated with a document in Microsoft Dynamics GP. The entity ID filtering does not restrict documents based on the additional data they contain, such as line items.

For example, when sales documents are returned based on the Windows User ID that is mapped to a salesperson ID, the documents are filtered based on the salesperson ID in the document header. The line items for the sales document are not filtered based on the salesperson ID. The document may contain line items that were added by other salespeople. These line items could contain information you don’t want the restricted user to see.

If your web service application relies on entity ID filtering, be aware of situations when your application could retrieve documents that contain sensitive data in the document. Consider using your application’s user interface to filter out the data, such as the commission information for sales document line items.

Entity ID filtering example

The following C# code example shows how the Scope property of the criteria object is used. This example retrieves the historical sales orders for the current user. The Scope property of the Sales Order Criteria object is set to ReturnBasedOnSalespersonID, so that only documents associated with the current user will be returned. To work, this example requires that the user has access to the “Query Sales Orders Based On User” operation for the Dynamics GP service. There must also be an entity ID assignment that maps the user to a Salesperson ID in Microsoft Dynamics GP.
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            ListRestrictionOfNullableOfSalesTransactionState transactionStateRestriction;
            SalesOrderCriteria salesOrderCriteria;
            SalesOrderSummary[] salesOrderSummary;

            // Create an instance of the service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure the default credentials are used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a transaction state restriction object
            transactionStateRestriction = new
            ListRestrictionOfNullableOfSalesTransactionState();
            transactionStateRestriction.EqualValue =
            SalesTransactionState.History;

            // Create a sales order criteria object
            // Retrieve summary objects for historical sales orders
            salesOrderCriteria = new SalesOrderCriteria();
            salesOrderCriteria.TransactionState =
            transactionStateRestriction;

            // Specify the scope so that transactions for only the current
            // user are retrieved
            salesOrderCriteria.Scope =
            SalesDocumentScope.ReturnBasedonSalespersonId;
PART 3 USING THE WEB SERVICE

try
{
    // Retrieve the sales order summaries specified
    salesOrderSummary =
        wsDynamicsGP.GetSalesOrderList(salesOrderCriteria, context);

    // Display the ID and amount of each summary object
    StringBuilder summaryList = new StringBuilder();
    foreach (SalesOrderSummary a in salesOrderSummary)
    {
        summaryList.AppendLine("Order number: "+a.Key.Id +
    }
    MessageBox.Show(summaryList.ToString());
}
catch (SoapException soapErr)
{
    MessageBox.Show(soapErr.Message);
}
}

Native endpoint
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WebServiceSample.DynamicsGPService;
using System.ServiceModel;
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            ListRestrictionOfNullableOfSalesTransactionState
                transactionStateRestriction;
            SalesOrderCriteria salesOrderCriteria;
            SalesOrderSummary[] salesOrderSummary;

            // Create an instance of the service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;
// Create a transaction state restriction object
transactionStateRestriction = new
ListRestrictionOfNullableOfSalesTransactionState();
transactionStateRestriction.EqualValue =
SalesTransactionState.History;

// Create a sales order criteria object
// Retrieve summary objects for historical sales orders
salesOrderCriteria = new SalesOrderCriteria();
salesOrderCriteria.TransactionState =
transactionStateRestriction;

// Specify the scope so that transactions for only the current
// user are retrieved
salesOrderCriteria.Scope =
SalesDocumentScope.ReturnBasedonSalespersonId;

try
{
    // Retrieve the sales order summaries specified
    salesOrderSummary =
    wsDynamicsGP.GetSalesOrderList(salesOrderCriteria, context);

    // Display the ID and amount of each summary object
    StringBuilder summaryList = new StringBuilder();
    foreach (SalesOrderSummary a in salesOrderSummary)
    {
        summaryList.AppendLine("Order number: " + a.Key.Id +
    }
    MessageBox.Show(summaryList.ToString());
}

catch (FaultException<System.ServiceModel.ExceptionDetail> ex)
{
    MessageBox.Show(ex.Message);
}
}
Chapter 18: Entity Action Tracking

The Dynamics GP service can be configured to track the actions performed on several of the core business objects. This is useful when you want to keep data in Microsoft Dynamics GP synchronized with an external system. Information about entity action tracking is divided into the following sections:

- Overview of entity action tracking
- Enabling entity action tracking
- Retrieving action tracking information
- Entity action tracking example
- Considerations for entity action tracking

Overview of entity action tracking

Actions for the following business objects can be tracked:

- Currency
- Customer
- CustomerAddress
- InternetAddress
- Item
- PriceLevel
- Pricing
- SalesInvoice
- SalesOrder
- Salesperson
- UofMSchedule

You can enable the tracking for the following actions:

- Creates
- Updates
- Deletes

You have full control over which actions are tracked for a business object. After tracking is enabled for a specific action of a business object, an event is logged each time that action is performed. It doesn’t matter where the action originates. It may be from a Microsoft Dynamics GP client, from the Dynamics GP service, or from an external application accessing the database. You will use the “GetChanged” methods in the Dynamics GP service to find out what actions have been performed.

Enabling entity action tracking

By default, no entity actions are tracked. To enable entity action tracking, you must use the Requester Setup Tool that is installed with the eConnect components used by Web Services for Microsoft Dynamics GP. Use the following procedure to enable entity action tracking.

1. **Open the Requester Setup tool.**
   In the Start menu, choose Microsoft Dynamics >> eConnect for Microsoft Dynamics GP 2010 >> Requester Setup >> Requester Setup.
2. **Define the connection settings.**
   If you have not specified the connection settings for this utility, choose Setup Connection in the Connection Settings menu.

3. **Specify the SQL instance and database.**
   In the Connection Setup window, specify the name of the SQL instance and the database of the company for which you are enabling entity action tracking.

   Because they are used by all companies, changes to currency objects are tracked in the DYNAMICS database.

4. **Specify the security to be used for the database connection.**
   Supply a SQL user name and password, or mark Integrated Security to use the credentials of the current user.

5. **Save the connection settings.**
   Click Save to save the connection settings.

6. **Connect to the database.**
   Click Connect to load the current Transaction Requester settings.
7. **Mark the business objects you want to track.**

On the Insert, Update, and Delete tabs, mark the specific actions that you want to track. The following table lists the Transaction Requester entries that you can mark to enable entity action tracking for the supported business objects.

<table>
<thead>
<tr>
<th>Business Object</th>
<th>DOCTYPE</th>
<th>ALIAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Currency*</td>
<td>ModifiedCurrency</td>
<td>Currency</td>
</tr>
<tr>
<td>Customer</td>
<td>ModifiedCustomer</td>
<td>Customer</td>
</tr>
<tr>
<td>CustomerAddress</td>
<td>ModifiedCustomerAddress</td>
<td>CustomerAddress</td>
</tr>
<tr>
<td>InternetAddress</td>
<td>ModifiedInternetAddress</td>
<td>Internet_Address</td>
</tr>
<tr>
<td>Item</td>
<td>ModifiedItem</td>
<td>Item</td>
</tr>
<tr>
<td>PriceLevel</td>
<td>ModifiedPriceLevel</td>
<td>PRICELEVEL</td>
</tr>
<tr>
<td>Pricing</td>
<td>ModifiedPricing</td>
<td>PRICING</td>
</tr>
<tr>
<td>SalesInvoice</td>
<td>ModifiedSales</td>
<td>SOHeader</td>
</tr>
<tr>
<td>SalesOrder</td>
<td>ModifiedSales</td>
<td>SOHeader</td>
</tr>
<tr>
<td>Salesperson</td>
<td>ModifiedSalesperson</td>
<td>RM_SalesPerson</td>
</tr>
<tr>
<td>UofMSchedule</td>
<td>ModifiedUOFM</td>
<td>UOFM</td>
</tr>
</tbody>
</table>

*Set in the DYNAMICS database

8. **Save the changes.**

Click Update to save the changes for the Transaction Requester. Entity action tracking will begin for the business objects you marked.

9. **Close the Requester Setup Tool.**

Click Exit to close the Requester Setup Tool.

### Retrieving action tracking information

To retrieve action tracking information, use the “GetChanged” methods in the Dynamics GP service. These methods take a criteria object that specifies the time period and other properties that specify the actions to examine. They return collections of “changed key” objects that represent the actions that occurred.

> **Entity action tracking must be enabled for the “GetChanged” web methods to return valid results.**

### Restriction criteria

To specify the actions you want to retrieve, you must create a “ChangedKeyCriteria” object. This object specifies the time range and other properties that indicate what actions you want to examine. For example, the following C# code shows how the criteria object is created to retrieve the actions performed on sales orders over the last week.

```csharp
SalesOrderChangedKeyCriteria salesOrderCriteria;
BetweenRestrictionOfNullableOfDateTime restriction;

// Create the restriction that defines the interval examined for actions
// The following restriction includes the last week
restriction = new BetweenRestrictionOfNullableOfDateTime();
restriction.From = DateTime.Today.Date.AddDays(-7);
restriction.To = DateTime.Today.Date.AddDays(1);
```
PART 3 USING THE WEB SERVICE

// Create the criteria to return sales order objects that were acted on
salesOrderCriteria = new SalesOrderChangedKeyCriteria();
salesOrderCriteria.LastModifiedDate = restriction;

Native endpoint
SalesOrderChangedKeyCriteria salesOrderCriteria;
BetweenRestrictionOfNullableOfdateTime restriction;

// Create the restriction that defines the interval examined for actions
// The following restriction includes the last week
restriction = new BetweenRestrictionOfNullableOfdateTime();
restriction.From = DateTime.Today.Date.AddDays(-7);
restriction.To = DateTime.Today.Date.AddDays(1);

// Create the criteria to return sales order objects that were acted on
salesOrderCriteria = new SalesOrderChangedKeyCriteria();
salesOrderCriteria.LastModifiedDate = restriction;

Restricting by action
By default, all of the create, update, and delete actions for the specified time period
will be returned. You can use the “data modification action” restriction to return
actions of a specific type. For example, the following C# code shows how to create
the criteria so that only the create actions for salesperson objects performed in the
last day are returned.

Legacy endpoint
SalespersonChangedKeyCriteria salespersonCriteria;
BetweenRestrictionOfNullableOfDateTime restriction;
RestrictionOfNullableOfDataModificationAction dataModificationRestriction;

// Create the restriction that defines the interval examined for actions
// The following restriction includes all of today
restriction = new BetweenRestrictionOfNullableOfDateTime();
restriction.From = DateTime.Today.Date;
restriction.To = DateTime.Today.Date.AddDays(1);

// Create the restriction so only created salespeople are returned
dataModificationRestriction = new
   RestrictionOfNullableOfDataModificationAction();
dataModificationRestriction.EqualValue = DataModificationAction.Created;

// Create the criteria to return salesperson objects that were created
salespersonCriteria = new SalespersonChangedKeyCriteria();
salespersonCriteria.LastModifiedDate = restriction;
salespersonCriteria.Action = dataModificationRestriction;

Native endpoint
SalespersonChangedKeyCriteria salespersonCriteria;
BetweenRestrictionOfNullableOfDateTime restriction;
RestrictionOfNullableOfDataModificationAction dataModificationRestriction;

// Create the restriction that defines the interval examined for actions
// The following restriction includes all of today
restriction = new BetweenRestrictionOfNullableOfDateTime();
restriction.From = DateTime.Today.Date;
restriction.To = DateTime.Today.Date.AddDays(1);
// Create the restriction so only created salespeople are returned
dataModificationRestriction = new
    RestrictionOfNullableOfDataModificationAction();
dataModificationRestriction.EqualValue = DataModificationAction.Created;

// Create the criteria to return salesperson objects that were created
salespersonCriteria = new SalespersonChangedKeyCriteria();
salespersonCriteria.LastModifiedDate = restriction;
salespersonCriteria.Action = dataModificationRestriction;

You will use the collection of the “changed key” objects returned by the
“GetChanged” web method as the basis for further processing. Each “changed key”
object contains the key value for the object for which the action was performed. You
can use the key values to retrieve the full object if needed.

Entity action tracking example

The following example retrieves the list of the changed item key objects for all of the
items that have had actions performed in the last week. The ID value of the changed
item key object is used to retrieve the full item object. Details about each item are
displayed in a console window.

Legacy endpoint

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            ItemChangedKeyCriteria itemCriteria;
            ChangedItemKey[] changedItemKeyObjects;
            BetweenRestrictionOfNullableOfDateTime restriction;
            ItemKey itemKey;
            Item item;

            // Create an instance of the service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Be sure the default credentials are used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);
```
// Set up the context object
context.OrganizationKey = (OrganizationKey)companyKey;

// Create the restriction that defines the interval examined
// The following restriction includes the last week
restriction = new BetweenRestrictionOfNullableOfDateTime();
restriction.From = DateTime.Today.Date.AddDays(-7);
restriction.To = DateTime.Today.Date.AddDays(1);

// Create the criteria to return item objects that were acted on
itemCriteria = new ItemChangedKeyCriteria();
itemCriteria.LastModifiedDate = restriction;

// Retrieve the changed item key objects
changedItemKeyObjects =
wsDynamicsGP.GetChangedItemKeyList(itemCriteria, context);

// Display the details about each item that acted on this week
foreach (ChangedItemKey key in changedItemKeyObjects)
{
    // Display the key
    Console.WriteLine("Item key: " + key.Id);

    // Display what the object knows about its changes
    Console.WriteLine("Last Modified Date: " +
key.LastModifiedDate.ToString());

    // Get the item
    itemKey = new ItemKey();
    itemKey.Id = key.Id;
    item = wsDynamicsGP.GetItemByKey(itemKey, context);

    // Display the item description
    Console.WriteLine("Description: " + item.Description);

    // Display the operation performed
    if (key.Action == DataModificationAction.Created)
        Console.WriteLine("Action: Created");
    if (key.Action == DataModificationAction.Updated)
        Console.WriteLine("Action: Updated");
    if (key.Action == DataModificationAction.Deleted)
        Console.WriteLine("Action: Deleted");

    Console.WriteLine();
}
Console.WriteLine("Press 'Enter' to continue...");
Console.ReadLine();
Native endpoint
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            ItemChangedKeyCriteria itemCriteria;
            ChangedItemKey[] changedItemKeyObjects;
            BetweenRestrictionOfNullableOfDateTime restriction;
            ItemKey itemKey;
            Item item;

            // Create an instance of the service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (sample company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context object
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create the restriction that defines the interval examined
            // The following restriction includes the last week
            restriction = new BetweenRestrictionOfNullableOfDateTime();
            restriction.From = DateTime.Today.Date.AddDays(-7);
            restriction.To = DateTime.Today.Date.AddDays(1);

            // Create the criteria to return item objects that were acted on
            itemCriteria = new ItemChangedKeyCriteria();
            itemCriteria.LastModifiedDate = restriction;

            // Retrieve the changed item key objects
            changedItemKeyObjects =
                wsDynamicsGP.GetChangedItemKeyList(itemCriteria, context);

            // Display the details about each item that acted on this week
            foreach (ChangedItemKey key in changedItemKeyObjects)
            {
                // Display the key
                Console.WriteLine("Item key: " + key.Id);

                // Display what the object knows about its changes
                Console.WriteLine("Last Modified Date: " +
                               key.LastModifiedDate.ToString());
            }
        }
    }
}
// Get the item
itemKey = new ItemKey();
itemKey.Id = key.Id;
item = wsDynamicsGP.GetItemByKey(itemKey, context);

// Display the item description
Console.WriteLine("Description: " + item.Description);

// Display the operation performed
if (key.Action == DataModificationAction.Created)
    Console.WriteLine("Action: Created");
if (key.Action == DataModificationAction.Updated)
    Console.WriteLine("Action: Updated");
if (key.Action == DataModificationAction.Deleted)
    Console.WriteLine("Action: Deleted");

Console.WriteLine();
}
Console.WriteLine("Press 'Enter' to continue...");
Console.ReadLine();

Considerations for entity action tracking

Keep in mind the following potential issues when using entity action tracking:

- Each action you track requires a record to be written to the eConnect_Out table in the database. If you track multiple actions for a business object that is used frequently, application performance could be impacted. Turn on entity action tracking only for the actions that you need to track.

- It is up to you to decide when the records in the eConnect_Out table are no longer needed and can be removed from the table.

- When accessing entity action tracking information, you may see actions that you didn’t expect. For instance, when an item is created in Microsoft Dynamics GP, you would expect a create action to be logged. However, an update operation is also logged. This occurs because Microsoft Dynamics GP first creates the item, and then updates it with the information the user entered.
Chapter 19: Optimized Proxy

The Dynamics GP service contains over 350 web methods. When you create a web reference or service reference for the Dynamics GP Service, the proxy generated contains code to access all of these methods. In many cases, your web service application will use just a small subset of the available methods. To optimize the proxy, you want to access a smaller set of methods. The following sections describe how to do this:

- Using a predefined optimized interface
- Available interfaces
- Configuring the endpoint
- Using the optimized interface

Using a predefined optimized interface

Both the legacy endpoint and native endpoint have additional optimized interfaces that contain methods that are specific to functional areas of Microsoft Dynamics GP. To use one of these interfaces for your integration, you will use these basic steps:

1. **Select the interface to use.**
   Choose one of the predefined optimized interfaces that contains the methods you need for your integration.

2. **Configure the endpoint to use the optimized interface.**
   In the configuration for the endpoint, you will specify which interface you want to use.

3. **Create the optimized proxy for your application.**
   In your application, you will create a connection to the endpoint. This will create an optimized version of the proxy.

4. **Code and test your application.**
   Slight changes are required for your code to use the optimized proxy. You will see significant improvement in the application performance. You should then thoroughly test your application.

5. **Configure the endpoint to use the full interface.**
   In the configuration for the endpoint, you will indicate that you want the full interface to be used.

6. **Test your application with the full interface.**
   You will retest your application with the full interface. You should continue to see the improved performance, even though the full interface is being used for the endpoint.
Available interfaces

The following is a list of the additional interfaces available:

- Common
- Field Service
- Financials
- Human Resources/Payroll
- Inventory
- Manufacturing
- Project Accounting
- Purchasing
- Sales

The Common interface contains methods that are used in most applications that integrate with the Dynamics GP service. The methods in Common are automatically included in each of the other interfaces.

Refer to the Microsoft Dynamics GP Web Service Reference help file for a complete list of the methods that are found in each interface. Select the interface that contains the methods you need for your integration. You will need to get the name of the interface you want to use from the Microsoft Dynamics GP Web Service Reference help file.

Configuring the endpoint

To configure the endpoint to use one of the optimized interfaces, you must edit the configuration file for that endpoint. In the configuration, you will switch the contract used for the endpoint from the full Dynamics GP interface to one of the optimized interfaces. To do this, complete the following procedure.

1. **Edit the configuration file for the endpoint.**
   Using a text editor, open the configuration file for the endpoint you want to change. The configuration files are typically found in this location:
   C:\Program Files\Microsoft Dynamic\GPWebServices\ServiceConfigs\DynamicsGP.config
   The configuration file for the legacy endpoint is named DynamicsGP.config. The configuration file for the legacy endpoint is named DynamicsGPLegacy.config.

2. **Change the WSDL generation for the endpoint.**
   When using an optimized interface, you must configure the Dynamics GP service to use dynamic WSDL generation. Refer to WSDL generation on page 17 for more information.

   Locate the <services> element in the configuration file. The <service> element specifies the contract for the endpoint and the behavior for the WSDL generation.

   **Legacy endpoint** The following value for the behaviorConfiguration attribute specifies that the dynamic WSDL is used:

   ```xml
   <service
     name="Microsoft.Dynamics.GP.WebServices.LegacyContract.DynamicsGP"
     behaviorConfiguration="GPDynamicMetadataBehavior">
   ```
**Native endpoint** The following value for the behaviorConfiguration attribute specifies that the dynamic WSDL is used:

```
```

3. **Specify the interface used for the endpoint.**
   Locate the `<services>` element in the configuration file. Within the `<service>` element you will find an `<endpoint>` element. The contract attribute of the `<endpoint>` element specifies the interface that defines the methods available for the endpoint.

You will change the last part of the contract attribute to the name of the interface that you want to use. For example, assume you wanted to use the Sales interface, which has the name ISales. You would replace the name of the full interface with the name of optimized interface. In this case, that is “ISales”.

**Legacy endpoint** The following value for the contract attribute specifies that the Sales interface is used:

```
<endpoint address="DynamicsGPService.asmx" name="LegacyDynamicsGP" binding="basicHttpBinding" bindingConfiguration="BasicHttpBindingTarget" contract="Microsoft.Dynamics.GP.WebServices.LegacyContract.ISales"/>
```

**Native endpoint** The following value for the contract attribute specifies that the Sales interface is used:

```
<endpoint address="GPService" name="GPWebService" binding="wsHttpBinding" bindingConfiguration="WSHttpBindingTarget" contract="Microsoft.Dynamics.GP.WebServices.Contract.ISales"/>
```

4. **Save the configuration changes.**

5. **Restart the Microsoft Dynamics GP Service Host.**
   The Microsoft Dynamics GP Service Host must be restarted for this change to take effect. From the Start menu, locate the Administrative Tools group and choose Services. Find the entry for the Microsoft Dynamics GP Service Host. Right-click the entry and choose Restart.

**Using the optimized interface**

To use the optimized interface, you need to create a web reference (legacy endpoint) or a service reference (native endpoint) that connects to the endpoint. The URLs for the legacy and native endpoints have not changed. Only the set of methods provided by each endpoint will have changed.

*The process of creating a new reference will create the new optimized proxy for your application.*

*If you are using the native endpoint, be sure that you adjust the settings in the app.config, as described in Application configuration file on page 27.*
When using the native endpoint, the proxy object created for the optimized interface will have a new name, so you will need to update your code. For the full contract, the proxy object generated by Visual Studio is named **DynamicsGPClient**. For the optimized interfaces, the main part of the object name is changed to the interface name. For instance, if you were using the ISales interface, the proxy object is named **SalesClient**.

After you compile your application with the optimized proxy, you should notice two things:

- The application size should be smaller, because of the reduced number of methods and classes in the proxy code.
- The application should create a proxy instance more quickly.

When you have thoroughly tested your application with the optimized proxy, switch the endpoints for the Dynamics GP service back to the full contract with static WSDL generation. Be sure that you restart the Microsoft Dynamics GP Service Host so your changes become effective. Then re-test your application to verify that it works properly with the full contract.
Part 4: Extending the Web Service

This portion of the documentation provided detailed information about extending the Dynamics GP service to handle additional data. The following items are discussed:

- Chapter 20, “Overview,” discusses how the Dynamics GP web service can be extended, and describes the steps involved.

- Chapter 21, “Dynamics GP Service Events,” describes the events that are used when extending the Dynamics GP service.

- Chapter 22, “Extension Assembly,” explains how to create an extension assembly containing the code for your Dynamics GP service extension.

- Chapter 23, “Registering Events,” describes how to register for the various events available in the Dynamics GP service.

- Chapter 24, “Using Service Extensions,” provides examples that demonstrate how to use a Dynamics GP service object that has been extended.
Chapter 20: Overview

Before extending objects for the Dynamics GP service, it’s helpful to understand how the objects can be extended and the steps involved. The following topics are discussed:

- How objects can be extended
- Parts of an object extension
- Creating an object extension

How objects can be extended

All classes in the Dynamics GP service that inherit from the BusinessObject class can be extended through a data extension mechanism and a collection of events. The standard business documents in Microsoft Dynamics GP, such as customers, vendors, sales documents, and so on that inherit from BusinessObject can all be extended.

Typically, Dynamics GP web service objects are extended to support storing and retrieving additional data that is related to the object. For instance, the InventoryItem object could be extended to store additional data for the item. Dynamics GP service objects can also be extended for other purposes, such as adding additional data validation.

The ExtensionList collection for each business object contains the extensions (additional data) that is being passed along with the object. Each extension in the collection has the following:

- **ExtensionId** A string that identifies the extension. This typically describes what type of data the extension contains.

- **DocExtension** An XML element that contains the additional data being passed with the object.

When an application uses a Dynamics GP service object that has been extended, the application is responsible for processing the XML element for each extension added to the object. It must process the XML element to extract the additional data when the object is retrieved. It must also create the properly-formed XML element for each extension when an object is being created or updated. Refer to Chapter 24, “Using Service Extensions,” for detailed examples of how web service consumers will access extensions.
Parts of an object extension

To extend objects in the Dynamics GP service, you will create the following:

- An extension assembly that contains the processing code for the web service extension. This assembly will be added to the main folder for the Dynamics GP web service installation.

- Entries in the BusinessObjectsFile.config that tell the Dynamics GP service what additional events are to be processed.

Creating an object extension

To create an object extension, complete the following steps:

1. Decide what objects to extend and which events to use.
   Objects that inherit from the BusinessObject class can be extended. Refer to Chapter 21, “Dynamics GP Service Events,” for information about the events available for the Dynamics GP service.

2. Determine the format for the XML element.
   If your extension will include additional data that will be passed along with the object, you must determine the format for the XML element that will contain this data. Remember that consumers of the service will need to process this XML element, so a simple structure is preferable.

   For example, the following XML element is used in an extension for the Customer object. It contains contact history information for the customer.

   ```xml
   <ContactHistory>
   <FirstContactDate>6/6/1999 12:00:00 AM</FirstContactDate>
   <FirstContactSalesperson>NANCY B.</FirstContactSalesperson>
   <LastContactDate>3/1/2006 12:00:00 AM</LastContactDate>
   <LastContactSalesperson>ERIN J.</LastContactSalesperson>
   </ContactHistory>
   ```

3. Create the extension assembly.
   This assembly contains the processing code for the events that you are extending in the Dynamics GP service. Refer to Chapter 22, “Extension Assembly,” for details about creating this assembly.

4. Register the events to process.
   Add entries to the BusinessObjectFile.config file so that the Dynamics GP service will process the additional events. Refer to Chapter 23, “Registering Events,” for more information.
Chapter 21: Dynamics GP Service Events

The Dynamics GP service makes several events available to extensions. An event occurs for each step as an object in the Dynamics GP service is retrieved, created, updated, or deleted. A Dynamics GP service extension will be notified of each event for which it has been registered. Information about Dynamics GP service events is divided into the following sections:

- How events are processed
- Event list

### How events are processed

When the Dynamics GP service is started, it reads the entries in the `BusinessObjectsFile.config` to determine which additional events it should process. As the Dynamics GP service processes requests, it makes the appropriate calls to the extension assemblies for which events are registered. These extension assemblies must have static methods with the specified names and proper event signatures so they can be called by the Dynamics GP service.

When called, the event handling code in the extension assembly processes the request. Information about the request, such as the business object being processed, is available in the arguments passed to the event handling code.

Depending on the event type, the response from the event handling code can influence further processing. For example, if the event handler for the event `ValidateForUpdate` encounters a validation error, the code in the event handler will add that error to the `ValidationResult` object passed back to the Dynamics GP service. After all of the `ValidateForUpdate` events have been run, the Dynamics GP service will examine whether any validation errors have been noted. If any have, the Dynamics GP service will log the validation exceptions and stop any further processing.

### Event list

The following table lists the events available for each extensible object in the Dynamics GP service. You will use this information when planning which events to use for your extension, and when you edit the `BusinessObjectFile.config` to register the actual events.

<table>
<thead>
<tr>
<th>Event</th>
<th>Occurs</th>
<th>Typical use</th>
<th>EventHandlerType</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieved</td>
<td>After the object has been retrieved from the database, but before it has been returned from the service.</td>
<td>Retrieve additional data for a service object.</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>DefaultingForCreate</td>
<td>Before the object has been saved to the database.</td>
<td>Apply defaulting logic for the extended data.</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>ValidatingForCreate</td>
<td>After the defaulting event, but before the object is saved to the database.</td>
<td>Apply validation rules for the extended data.</td>
<td>BusinessObjectValidateEvent Handler</td>
<td>All validation events run, allowing all validation exceptions to be logged. If validation errors have been logged, further processing stops.</td>
</tr>
<tr>
<td>Event</td>
<td>Occurs</td>
<td>Typical use</td>
<td>EventHandlerType</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------</td>
<td>------------------------------------------------</td>
<td>------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Creating</td>
<td>Immediately after the object is created in the database.</td>
<td>Save extended data to the appropriate tables in the database.</td>
<td>BusinessObjectEventHandler</td>
<td>This event is included in the database transaction that creates the base object.</td>
</tr>
<tr>
<td>Created</td>
<td>After the object has been created successfully in the database.</td>
<td>None</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>DefaultingForUpdate</td>
<td>Before the object has been updated in the database.</td>
<td>Apply defaulting logic for the extended data.</td>
<td>BusinessObjectUpdateEventHandler</td>
<td>Has access to the original version of the core business object.</td>
</tr>
<tr>
<td>ValidatingForUpdate</td>
<td>After the defaulting event, but before the object is updated in the database.</td>
<td>Apply validation rules for the extended data.</td>
<td>BusinessObjectValidateForUpdateEventHandler</td>
<td>Has access to the previous version (before update) of the core business object. All validation events run, allowing all validation exceptions to be logged. If validation errors have been logged, further processing stops.</td>
</tr>
<tr>
<td>Updating</td>
<td>Immediately after the object is updated in the database.</td>
<td>Update extended data in the appropriate tables in the database.</td>
<td>BusinessObjectUpdateEventHandler</td>
<td>Has access to the original version of the core business object. This event is included in the database transaction that updates the base object.</td>
</tr>
<tr>
<td>Updated</td>
<td>After the object has been updated successfully in the database.</td>
<td>None</td>
<td>BusinessObjectUpdateEventHandler</td>
<td>Has access to the original version of the core business object.</td>
</tr>
<tr>
<td>DefaultingForDelete</td>
<td>Before the object has been deleted from the database.</td>
<td>None</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>ValidatingForDelete</td>
<td>After the defaulting event, but before the object has been deleted from the database.</td>
<td>Verifies any preconditions required for the object to be deleted.</td>
<td>BusinessObjectValidateEventHandler</td>
<td>All validation events run, allowing all validation exceptions to be logged. If validation errors have been logged, further processing stops.</td>
</tr>
<tr>
<td>Deleting</td>
<td>Immediately after the object has been deleted from the database.</td>
<td>Delete extended data from the appropriate tables in the database.</td>
<td>BusinessObjectEventHandler</td>
<td>This event is included in the database transaction that deletes the base object.</td>
</tr>
<tr>
<td>Deleted</td>
<td>After the object has been deleted successfully from the database.</td>
<td>None</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>DefaultingForVoid</td>
<td>Before the object has been voided.</td>
<td>None</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>ValidatingForVoid</td>
<td>After the defaulting event, but before the object has been voided.</td>
<td>Verifies any preconditions required for the object to be voided.</td>
<td>BusinessObjectValidateEventHandler</td>
<td>All validation events run, allowing all validation exceptions to be logged. If validation errors have been logged, further processing stops.</td>
</tr>
<tr>
<td>Voiding</td>
<td>Immediately after the object has been voided.</td>
<td>Performs any necessary “void” action on the extended data.</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
<tr>
<td>Event</td>
<td>Occurs</td>
<td>Typical use</td>
<td>EventHandlerType</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Voided</td>
<td>After the object has been voided successfully.</td>
<td>None</td>
<td>BusinessObjectEventHandler</td>
<td>None</td>
</tr>
</tbody>
</table>
Chapter 22: Extension Assembly

The extension assembly contains the processing code for the events you register for your extension. It’s important that this assembly be created properly so that the Dynamics GP service can call the static methods in the assembly. Information about creating an extension assembly is divided into the following sections:

- Creating an extension assembly
- Event handler methods
- Sample: Retrieved event handler method
- Sample: ValidatingForCreate event handler method
- Sample: Creating event handler method
- Sample: ValidatingForUpdate event handler method
- Sample: Updating event handler method
- Sample: Deleting event handler method

Creating an extension assembly

To create an extension assembly, complete the following steps:

1. **Create a new class library.**
   If Visual Studio, create a new project that has the type Class Library.

2. **Add references to Dynamics GP web service assemblies.**
   Add references to the following assemblies from the main folder for the Dynamics GP web service.
   - Microsoft.Dynamics.Common
   - Microsoft.Dynamics.Common.Types
   - Microsoft.Dynamics.GP.BusinessLogic

   These assemblies are typically found in this location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

3. **Add namespace references.**
   Add using (C#) or Imports (Visual Basic) statements to provide convenient access to the classes and methods needed for the extension assembly. Include the following:

   - Microsoft.Dynamics.Common
   - Microsoft.Dynamics.GP
   - System.Data
   - System.Data.SqlClient
   - System.Xml

   *The Data, SqlClient, and Xml namespaces are added to provide access to resources you will need when you write your event handling methods.*

4. **Specify the name for the namespace.**
   In the code for the class library, specify the name for the namespace. Use a name that indicates who created the web service extension or what type of extended data is being made available.
5. Add a public static class.
This new class in the extension assembly will contain the static methods that respond to Dynamics GP service events. Be sure the class is marked public and static. The following C# code is an example of a public class for an extension assembly:

```csharp
using System;
using System.Data;
using System.Data.SqlClient;
using System.Xml;
using Microsoft.Dynamics.Common;
using Microsoft.Dynamics.GP;

namespace ExtensionExample
{
    public static class ContactHistory
    {
        // Event handler methods are added here
    }
}
```

6. Add the event handler methods.
These public static method will respond to the events from the Dynamics GP service. For more information about these methods, refer to Event handler methods on page 132.

7. Build and deploy the extension assembly.
Once the extension assembly has been built, copy it to the folder for the Dynamics GP web service installation. Typically this will be in the following location:

C:\Program Files\Microsoft Dynamics\GPWebServices

Event handler methods

The event handler methods are static public methods that you add to the public static class in your extension assembly. In most cases, you will have one method for each event from the web service that your extension assembly will handle events for.

Event handler argument types

The names you use for the static methods aren’t critical. The name should describe the action that is being performed by the event handler. The method signature for each static method is important. The parameters must match what the Dynamics GP service is expecting. Each event handler method has the following basic form:

```csharp
public static void method_name(object sender, EventArgsType e)
```

In place of `EventArgsType` you must specify the appropriate type for the type of event the static method is responding to. The following table lists the event types and the event arguments type that should be used.

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Arguments Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieved</td>
<td>BusinessObjectEventArgs</td>
</tr>
<tr>
<td>DefaultingForCreate</td>
<td>BusinessObjectEventArgs</td>
</tr>
</tbody>
</table>
For example, the static event handler method that responds to an Updating event from the Dynamics GP service would look like the following:

```csharp
public static void UpdateItem(object sender, BusinessObjectUpdateEventArgs e)
```

### Writing event handler methods

The event handler method must perform the action to respond to the web service event, such as retrieving extended data for an object from the database. Properties passed in with the event arguments help the event handler complete its work. The following table lists the properties available with the event arguments.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusinessObject</td>
<td>The complete business object for which the service event is occurring.</td>
</tr>
<tr>
<td>Context</td>
<td>The Context object passed with the service call. This object contains information used in processing the event, such as the OrganizationKey.</td>
</tr>
<tr>
<td>OriginalBusinessObject</td>
<td>For update events, the complete business object as it existed before it was updated.</td>
</tr>
<tr>
<td>Policy</td>
<td>The Policy object passed with the service call.</td>
</tr>
<tr>
<td>ValidationResult</td>
<td>For validation events, the collection of validation warnings and errors. If the event handler encounters validation issues, it must add them to this collection.</td>
</tr>
</tbody>
</table>

Event handler methods frequently need to access the Dynamics GP database to complete their actions. Objects available from the Microsoft.Dynamics.Common assembly are provided to help with these tasks. The examples provided in the following sections demonstrate how to perform these database actions.

The event handler methods also need to work with the Extension objects that contain extended data for business objects. The examples provided show the techniques needed to work with the ExtensionList collection, as well as the XML element included with each Extension object.
Sample: Retrieved event handler method

The following C# example shows the Retrieved event handler method used to retrieve contact history information associated with a customer object. The event handler retrieves the customer object from the event arguments passed in. After retrieving a connection to the current company’s database and storing it in the connection private variable, it executes a SQL statement to retrieve the contact history information for the customer from the IG003 table. Finally, it builds the XML element that will contain the contact history information. It adds the XML element to an Extension object in the Extensions collection the customer object.

```csharp
// Declare private variable of type Microsoft.Dynamics.Common.Connection private static Connection connection;

public static void GetContactHistory(object sender, BusinessObjectEventArgs e)
{
    string firstContactDate;
    string firstContactSalesperson;
    string lastContactDate;
    string lastContactSalesperson;

    XmlDocument doc;
    XmlElement contactHistoryXML;
    XmlElement firstContactDateXML;
    XmlElement firstContactSalespersonXML;
    XmlElement lastContactDateXML;
    XmlElement lastContactSalespersonXML;
    XmlText text;

    Customer customer;

    if (e.BusinessObject.GetType() == typeof(Customer))
    {
        customer = (Customer)e.BusinessObject;

        // Get the connection to the database for the current company
        connection = Connection.GetInstance();

        // The SQL command to retrieve contact history information
        string selectCommand = "SELECT FirstContactDate,
                                ContactSalespersonID1, LastContactDate, ContactSalespersonID2 FROM
                                IG003 WHERE CUSTNMBR='" + customer.Key.Id + ''';

        SqlDataAdapter adapter = new SqlDataAdapter(selectCommand,
                                                    (SqlConnection)connection.GetConnection(e.Context.OrganizationKey));

        DataTable table = new DataTable();
        adapter.Fill(table);

        if (table.Rows.Count > 0)
        {
            // Get the data from the SQL result
            firstContactDate = table.Rows[0].ItemArray[0].ToString();
            firstContactSalesperson = table.Rows[0].ItemArray[1].ToString();
            lastContactDate = table.Rows[0].ItemArray[2].ToString();
            lastContactSalesperson = table.Rows[0].ItemArray[3].ToString();
        }
    }
}
```
// Build the Extension object to return from the service
Extension ContactHistory = new Extension();
ContactHistory.ExtensionId = "ContactHistory";

// Make the XML extension document
doc = new XmlDocument();

contactHistoryXML = doc.CreateElement("ContactHistory");

// First Contact Date
firstContactDateXML = doc.CreateElement("FirstContactDate");
text = doc.CreateTextNode(firstContactDate);
firstContactDateXML.AppendChild(text);
contactHistoryXML.AppendChild(firstContactDateXML);

// First Contact Salesperson
firstContactSalespersonXML =
    doc.CreateElement("FirstContactSalesperson");
text = doc.CreateTextNode(firstContactSalesperson);
firstContactSalespersonXML.AppendChild(text);
contactHistoryXML.AppendChild(firstContactSalespersonXML);

// Last Contact Date
lastContactDateXML = doc.CreateElement("LastContactDate");
text = doc.CreateTextNode(lastContactDate);
lastContactDateXML.AppendChild(text);
contactHistoryXML.AppendChild(lastContactDateXML);

// Last Contact Salesperson
lastContactSalespersonXML =
    doc.CreateElement("LastContactSalesperson");
text = doc.CreateTextNode(lastContactSalesperson);
lastContactSalespersonXML.AppendChild(text);
contactHistoryXML.AppendChild(lastContactSalespersonXML);

// Add the extension to the Customer object
ContactHistory.DocExtension = contactHistoryXML;
e.BusinessObject.Extensions.Add(ContactHistory);

Sample: ValidatingForCreate event handler method

The following C# example shows the ValidatingForCreate event handler method used to validate contact history information that is being added as extended data for a customer object. The event handler examines the Extension objects passed along with the customer. If one of the Extension objects has the ExtensionId value "ContactHistory", the event handler will process it. After extracting the XML element from the Extension object, the two Salesperson ID values included in the extended data are validated. If the Salesperson ID values cannot be validated, validation errors are added to the ValidationResults object that was passed into the event handler.
private static Connection connection;

public static void ValidateCreateContactHistory(object sender,
BusinessObjectValidateEventArgs e)
{
    bool found;
    string firstContactDate;
    string firstContactSalesperson;
    string lastContactDate;
    string lastContactSalesperson;
    string selectStatement;
    Customer customer;
    Extension ContactHistoryExtension = new Extension();

    if (e.BusinessObject.GetType() == typeof(Customer))
    {
        customer = (Customer)e.BusinessObject;

        // Look at the Extension list passed along
        found = false;
        foreach (Extension ext in customer.Extensions)
        {
            if (ext.ExtensionId == "ContactHistory")
            {
                ContactHistoryExtension = ext;
                found = true;
                break;
            }
        }
    }
    if (found == true)
    {
        // Found an extension, so it should be processed
        XElement contactHistory;
        contactHistory = ContactHistoryExtension.DocExtension;

        XmlNodeList nodeList;
        nodeList = contactHistory.ChildNodes;

        // First contact date
        firstContactDate = nodeList[0].InnerText.ToString();

        // First contact salesperson
        firstContactSalesperson = nodeList[1].InnerText.ToString();

        // Last contact date
        lastContactDate = nodeList[2].InnerText.ToString();

        // Last contact salesperson
        lastContactSalesperson = nodeList[3].InnerText.ToString();

        // Get the connection to the database for the current company
        connection = Connection.GetInstance();
        SqlCommand command = new SqlCommand();
        command.Connection = (SqlConnection)connection.GetConnection
                           (e.Context.OrganizationKey);
SqlDataAdapter adapter = new SqlDataAdapter(command);
DataTable table = new DataTable();

// Verify that the First Contact Salesperson and Last Contact Salesperson are valid

// The SQL statement to verify the First Contact Salesperson
selectStatement = "SELECT SLPRSNID FROM RM00301 WHERE SLPRSNID = '" + firstContactSalesperson + '";
command.CommandText = selectStatement;
adapter.Fill(table);
if (table.Rows.Count < 1)
{
    // Add an exception, because the First Contact salesperson was not found
    ValidationError validationError = new ValidationError();
    validationError.Id = '1001';
    validationError.Message = "Invalid First Contact Salesperson specified in Contact History.";
    validationError.ObjectType = typeof(Customer).ToString();
    validationError.PropertyNames.Add("FirstContactSalesperson");
    e.ValidationResult.Errors.Add(validationError);
}

// The SQL statement to verify the Last Contact Salesperson
selectStatement = "SELECT SLPRSNID FROM RM00301 WHERE SLPRSNID = '" + lastContactSalesperson + '";
command.CommandText = selectStatement;
adapter.SelectCommand = command;
table.Clear();
adapter.Fill(table);
if (table.Rows.Count < 1)
{
    // Add an exception, because the Last Contact salesperson was not found
    ValidationResult validationResult = new ValidationResult();
    ValidationError validationError = new ValidationError();
    validationError.Id = '1002';
    validationError.Message = "Invalid Last Contact Salesperson specified in Contact History.";
    validationError.ObjectType = typeof(Customer).ToString();
    validationError.PropertyNames.Add("LastContactSalesperson");
    e.ValidationResult.Errors.Add(validationError);
}
Sample: Creating event handler method

The following C# example shows the Creating event handler method used to save the contact history information for a customer object. The event handler examines the Extension objects passed along with the customer. If one of the Extension objects has the ExtensionId value “ContactHistory”, the event handler will process it. The code extracts the contact history information from XML element of the Extension object. After retrieving a connection to the current company’s database and storing in the connection private variable, the event handler executes a SQL statement to save the contact history information for the customer into the IG003 table.

```csharp
// Declare private variable of type Microsoft.Dynamics.Common.Connection private static Connection connection;

public static void CreateContactHistory(object sender, BusinessObjectEventArgs e)
{
    bool found;
    int rowsAffected;
    string contact;
    string firstContactDate;
    string firstContactSalesperson;
    string lastContactDate;
    string lastContactSalesperson;
    string updateStatement;
    string insertStatement;
    Customer customer;
    Extension ContactHistoryExtension = new Extension();

    if (e.BusinessObject.GetType() == typeof(Customer))
    {
        customer = (Customer)e.BusinessObject;

        // Look at the Extension list passed along
        found = false;
        foreach (Extension ext in customer.Extensions)
        {
            if (ext.ExtensionId == "ContactHistory")
            {
                ContactHistoryExtension = ext;
                found = true;
                break;
            }
        }
        if (found == true)
        {
            // Found an extension, so it should be processed
            XmlElement contactHistory;
            contactHistory = ContactHistoryExtension.DocExtension;

            XmlNodeList nodeList;
            nodeList = contactHistory.ChildNodes;

            // First contact date
            firstContactDate = nodeList[0].InnerText.ToString();
```

138 WEB SERVICE PROGRAMMER’S GUIDE
// First contact salesperson
firstContactSalesperson = nodeList[1].InnerText.ToString();

// Last contact date
lastContactDate = nodeList[2].InnerText.ToString();

// Last contact salesperson
lastContactSalesperson = nodeList[3].InnerText.ToString();

// Get the connection to the database for the current company
connection = Connection.GetInstance();

// The SQL statement to update contact history information
updateStatement = "UPDATE IG003 SET ContactSalespersonID1='' + firstContactSalesperson
 + ''', FirstContactDate='' + firstContactDate
 + ''', ContactSalespersonID2='' + lastContactSalesperson
 + ''', LastContactDate='' + lastContactDate
 + '''
WHERE CUSTNMBR = '' + customer.Key.Id + ''";

// Create the SQL connection
SqlCommand command = new SqlCommand(updateStatement);
SqlConnection sqlConnection = new SqlConnection
(connection.GetConnectionString(e.Context.OrganizationKey));
command.Connection = sqlConnection;

// Open the SQL connection
sqlConnection.Open();

// Execute the SQL statement
rowsAffected = command.ExecuteNonQuery();
if (rowsAffected == 0)
{
    // The row did not exist, so try creating it.
    // Is the ContactPerson specified? If not, set it to empty.
    if (customer.Addresses.Count == 0)
    {
        contact = "";
    }
    else
    {
        contact = customer.Addresses[0].ContactPerson;
    }

    insertStatement = "INSERT IG003 VALUES (''
    + firstContactSalesperson + '''', '''
    + lastContactSalesperson + '''', '''
    + contact
    + '''', ''', lastContactDate + '''', '''
    + firstContactDate + '''', "" + customer.Name
    + '''', "" + customer.Key.Id + '''");
    command.CommandText = insertStatement;
    rowsAffected = command.ExecuteNonQuery();
}

// Close the SQL connection
sqlConnection.Close();
}
Sample: ValidatingForUpdate event handler method

The following C# example shows the ValidatingForUpdate event handler method used to validate contact history information that is being updated as extended data for a customer object. The event handler examines the Extension objects passed along with the customer. If one of the Extension objects has the ExtensionId value “ContactHistory”, the event handler will process it. After extracting the XML element from the Extension object, the two Salesperson ID values included in the extended data are validated. If the Salesperson ID values cannot be validated, validation errors are added to the ValidationResults object that was passed into the event handler.

```csharp
private static Connection connection;

public static void ValidateUpdateContactHistory(object sender,
BusinessObjectValidateForUpdateEventArgs e)
{
    bool found;
    string firstContactDate;
    string firstContactSalesperson;
    string lastContactDate;
    string lastContactSalesperson;
    string selectStatement;

    Customer customer;
    Extension ContactHistoryExtension = new Extension();

    if (e.BusinessObject.GetType() == typeof(Customer))
    {
        customer = (Customer)e.BusinessObject;

        // Look at the Extension list passed along
        found = false;
        foreach (Extension ext in customer.Extensions)
        {
            if (ext.ExtensionId == "ContactHistory")
            {
                ContactHistoryExtension = ext;
                found = true;
                break;
            }
        }
        if (found == true)
        {
            // Found an extension, so it should be processed
            XmlElement contactHistory;
            contactHistory = ContactHistoryExtension.DocExtension;

            XmlNodeList nodeList;
            nodeList = contactHistory.ChildNodes;

            // First contact date
            firstContactDate = nodeList[0].InnerText.ToString();

            // First contact salesperson
            firstContactSalesperson = nodeList[1].InnerText.ToString();

            // Last contact date
            lastContactDate = nodeList[2].InnerText.ToString();
        }
    }
}
```
// Last contact salesperson
lastContactSalesperson = nodeList[3].InnerText.ToString();

// Get the connection to the database for the current company
collection = Connection.GetInstance();
SqlCommand command = new SqlCommand();
command.Connection = (SqlConnection) connection.GetConnection(e.Context.OrganizationKey);
SqlDataAdapter adapter = new SqlDataAdapter(command);
DataTable table = new DataTable();

// Verify that the First Contact Salesperson and Last Contact
// Salesperson are valid

// The SQL statement to verify the First Contact Salesperson
selectStatement = "SELECT SLPRSNID FROM RM00301 WHERE SLPRSNID = " + firstContactSalesperson + "";
command.CommandText = selectStatement;
adapter.Fill(table);
if (table.Rows.Count < 1)
{
    // Add an exception, because the First Contact salesperson was
    // not found
    ValidationResult validationResult = new ValidationResult();
    validationResult.Id = '1001';
    validationResult.Message = "Invalid First Contact Salesperson
    specified in Contact History."
    validationResult.ObjectType = typeof(Customer).ToString();
    e.ValidationResult.Errors.Add(validationResult);
}

// The SQL statement to verify the Last Contact Salesperson
selectStatement = "SELECT SLPRSNID FROM RM00301 WHERE SLPRSNID = " + lastContactSalesperson + "";
command.CommandText = selectStatement;
adapter.SelectCommand = command;
table.Clear();
adapter.Fill(table);
if (table.Rows.Count < 1)
{
    // Add an exception, because the Last Contact salesperson was
    // not found
    ValidationResult validationResult = new ValidationResult();
    validationResult.Id = '1002';
    validationResult.Message = "Invalid Last Contact Salesperson
    specified in Contact History."
    validationResult.ObjectType = typeof(Customer).ToString();
    validationResult.PropertyNames.Add("LastContactSalesperson");
    e.ValidationResult.Errors.Add(validationResult);
}
Sample: Updating event handler method

The following C# example shows the Updating event handler method used to save the contact history information for a customer object. The event handler examines the Extension objects passed along with the customer. If one of the Extension objects has the ExtensionId value “ContactHistory”, the event handler will process it. The code extracts the contact history information from XML element of the Extension object. After retrieving a connection to the current company’s database and storing it in the connection private variable, the event handler executes a SQL statement to save the updated contact history information for the customer into the IG003 table.

```csharp
private static Connection connection;

public static void UpdateContactHistory(object sender,
BusinessObjectUpdateEventArgs e)
{
    bool found;
    int rowsAffected;
    string contact;
    string firstContactDate;
    string firstContactSalesperson;
    string lastContactDate;
    string lastContactSalesperson;
    string updateStatement;
    string insertStatement;

    Customer customer;
    Extension ContactHistoryExtension = new Extension();

    if (e.BusinessObject.GetType() == typeof(Customer))
    {
        customer = (Customer)e.BusinessObject;
        // Look at the Extension list passed along
        found = false;
        foreach (Extension ext in customer.Extensions)
        {
            if (ext.ExtensionId == "ContactHistory")
            {
                ContactHistoryExtension = ext;
                found = true;
                break;
            }
        }
        if (found == true)
        {
            // Found an extension, so it should be processed
            XmlDocument contactHistory;
            contactHistory = ContactHistoryExtension.DocExtension;

            XmlNodeList nodeList;
            nodeList = contactHistory.ChildNodes;

            // First contact date
            firstContactDate = nodeList[0].InnerText.ToString();

            // First contact salesperson
            firstContactSalesperson = nodeList[1].InnerText.ToString();
```
// Last contact date
lastContactDate = nodeList[2].InnerText.ToString();

// Last contact salesperson
lastContactSalesperson = nodeList[3].InnerText.ToString();

// Get the connection to the database for the current company
connection = Connection.GetInstance();

// The SQL statement to update contact history information
updateStatement = "UPDATE IG003 SET ContactSalespersonID1='' +
firstContactSalesperson
+ ''' , FirstContactDate='' + firstContactDate
+ ''' , ContactSalespersonID2='' + lastContactSalesperson
+ ''' , LastContactDate='' + lastContactDate
+ ''' WHERE CUSTNMBR = '' + customer.Key.Id + ''' ;

// Create the SQL connection
SqlCommand command = new SqlCommand(updateStatement);
SqlConnection sqlConnection = new SqlConnection
(connection.GetConnectionString(e.Context.OrganizationKey));
command.Connection = sqlConnection;

// Open the SQL connection
sqlConnection.Open();

// Execute the SQL statement
rowsAffected = command.ExecuteNonQuery();

if (rowsAffected == 0)
{
    // The row did not exist, so try creating it.
    // Has a ContactPerson been specified? If not, set it to empty.
    if (customer.Addresses.Count == 0)
        contact = "";
    else
        contact = customer.Addresses[0].ContactPerson;

    insertStatement = "INSERT IG003 VALUES (''
+ firstContactSalesperson + ''' , ''
+ lastContactSalesperson + ''' , ''
+ contact
+ ''' , '' + lastContactDate + ''' , ''
+ firstContactDate + ''' , '' + customer.Name
+ ''' , '' + customer.Key.Id + ''' )"
;
command.CommandText = insertStatement;
rowsAffected = command.ExecuteNonQuery();
}

// Close the SQL connection
sqlConnection.Close();
Sample: Deleting event handler method

The following C# example shows the Deleting event handler method used to delete contact history information for a customer object. The event handler examines the customer object to retrieve the customer ID value. After retrieving a connection to the current company’s database and storing it in the connection private variable, the event handler executes a SQL statement to delete the contact history information for the customer from the IG003 table.

```csharp
private static Connection connection;

public static void DeleteContactHistory(object sender, BusinessObjectEventArgs e)
{
    string deleteStatement;

    Customer customer;

    if (e.BusinessObject.GetType() == typeof(Customer))
    {
        customer = (Customer)e.BusinessObject;

        // Get the connection to the database for the current company
        connection = Connection.GetInstance();

        // The SQL statement to delete contact history information
        deleteStatement = "DELETE FROM IG003 WHERE CUSTNMBR = '" +
                          customer.Key.Id + "';";

        // Create the SQL connection
        SqlCommand command = new SqlCommand(deleteStatement);
        SqlConnection sqlConnection = new SqlConnection
            (connection.GetConnectionString(e.Context.OrganizationKey));
        command.Connection = sqlConnection;

        // Open the SQL connection
        sqlConnection.Open();

        // Execute the SQL statement
        command.ExecuteNonQuery();

        // Close the SQL connection
        sqlConnection.Close();
    }
}
```
Chapter 23: Registering Events

To have events for the Dynamics GP service be run, they must be registered. Information about registering events is divided into the following sections:

- Business object configuration file
- Modifying the configuration file
- Event registration example

Business object configuration file

The file named BusinessObjectFile.config contains the registrations for all of the additional events that are run by the Dynamics GP service. This file is in XML format. It is located in the ServiceConfigs folder of Dynamics GP web service, typically found in this location:

C:\Program Files\Microsoft Dynamics\GPWebServices

DictionaryEntry elements

The BusinessObjectFile.config contains one <DictionaryEntry> element for each object that can have events registered for it. The Dynamics GP service objects that can raise events are those that inherit from BusinessObject. For convenience, the entries are in alphabetical order so they can be found easily.

Not all objects have <DictionaryEntry> entries in this configuration file. For instance, the Customer object isn’t included in the default version of this file. If you want to register events for an object that is not in the file, you will need to add a <DictionaryEntry> element for it. The easiest way to do this is by copying an existing element.

The <Key> for the <DictionaryEntry> element is the complete name of the business object. The complete name has the prefix Microsoft.Dynamics.GP, followed by the name of the object. The name of the object corresponds to the class name you see in the Dynamics GP Web Service Reference. As an example, the complete name for the customer object is: Microsoft.Dynamics.GP.Customer.

Event elements

A <DictionaryEntry> element will have one or more <Event> elements, each describing an event for the business object. Each <Event> element has the following:

EventName The name of the event being registered. This corresponds to one of the events listed in Event list on page 127.

EventHandlerType Specifies the type of event handler needed for the event. The handler type for each event type is also listed in Event list on page 127.

SoftwareVendor Identifies who added the event.

Type The qualified name that indicates the namespace and static class containing the event handler method for the event.

StaticMethod The name of the static method that will be run for the event. The signature for this static method must be appropriate for the type of event. Refer to Event handler methods on page 132 for more information.
**Assembly**  The name of the extension assembly that contains the static method for the event.

**Execute**  A boolean value that allows an event to be turned on or off. The value must be set to true for an event to be processed.

## Modifying the configuration file

When modifying the configuration file, be aware of the following situations that can occur.

### Restart after making changes

The BusinessObjectsFile.config is read when the Microsoft Dynamics GP Service Host is started. If you make changes to this file while the service host is running, those changes will not be honored until the Microsoft Dynamics GP Service Host is restarted.

### Configuration errors

Any configuration errors in the file, such as invalid class or method names, can cause system exceptions when the Dynamics GP service is performing operations.

Be sure there are no extra spaces or line breaks within the elements of the BusinessObjectsFile.config file.

When events for a method are processed, system exceptions will be logged that describe any event problems that occurred. Use the Exception Management Console to view the exceptions logged.

### Other modifications

Be aware that other modifications may have been made by other applications that are extending the Dynamics GP service. The file may not be in the default state. Any automation you use to modify this file will need to account for this possibility.

### Web service repair operations

When the Dynamics GP service is repaired, the BusinessObjectsFile.config will be restored to its original state. Any changes made to it will be lost. If you make extensive changes to the file, you will want to make a backup after the changes are complete.
Event registration example

The following example shows the entries made in the BusinessObjectFile.config to register events for the extension assembly that manages contact history information for the customer object. The events registered here correspond to the six sample event handler methods described in Chapter 22, “Extension Assembly.”

The extension assembly in this example has the name “ExtensionExample”. The event handler static methods are located in the class named “ContactHistory”, in the namespace “ExtensionExample”. Each event specifies the appropriate event handler type for that event.

```xml
<DictionaryEntry>
  <Key xsi:type="xsd:string">Microsoft.Dynamics.GP.Customer</Key>
  <Value xsi:type="BusinessObjectConfiguration">
    <Event>
      <EventName>Retrieved</EventName>
      <EventHandlerType>
        <Type>Microsoft.Dynamics.Common.BusinessObjectEventHandler</Type>
        <Assembly>Microsoft.Dynamics.Common</Assembly>
      </EventHandlerType>
      <EventHandler>
        <SoftwareVendor>MicrosoftDocumentation</SoftwareVendor>
        <Type>ExtensionExample.ContactHistory</Type>
        <StaticMethod>GetContactHistory</StaticMethod>
        <Assembly>ExtensionExample</Assembly>
        <Execute>true</Execute>
      </EventHandler>
    </Event>
    <Event>
      <EventName>ValidatingForUpdate</EventName>
      <EventHandlerType>
        <Type>Microsoft.Dynamics.Common.BusinessObjectValidateForUpdateEventHandler</Type>
        <Assembly>Microsoft.Dynamics.Common</Assembly>
      </EventHandlerType>
      <EventHandler>
        <SoftwareVendor>MicrosoftDocumentation</SoftwareVendor>
        <Type>ExtensionExample.ContactHistory</Type>
        <StaticMethod>ValidateUpdateContactHistory</StaticMethod>
        <Assembly>ExtensionExample</Assembly>
        <Execute>true</Execute>
      </EventHandler>
    </Event>
    <Event>
      <EventName>Updating</EventName>
      <EventHandlerType>
        <Type>Microsoft.Dynamics.Common.BusinessObjectUpdateEventHandler</Type>
        <Assembly>Microsoft.Dynamics.Common</Assembly>
      </EventHandlerType>
      <EventHandler>
        <SoftwareVendor>MicrosoftDocumentation</SoftwareVendor>
        <Type>ExtensionExample.ContactHistory</Type>
        <StaticMethod>UpdateContactHistory</StaticMethod>
        <Assembly>ExtensionExample</Assembly>
        <Execute>true</Execute>
      </EventHandler>
    </Event>
  </Value>
</DictionaryEntry>
```
<Event>
  <EventName>ValidatingForCreate</EventName>
  <EventHandlerType>
    <Type>Microsoft.Dynamics.Common.BusinessObjectValidateEventHandler</Type>
    <Assembly>Microsoft.Dynamics.Common</Assembly>
  </EventHandlerType>
  <EventHandler>
    <SoftwareVendor>MicrosoftDocumentation</SoftwareVendor>
    <Type>ExtensionExample.ContactHistory</Type>
    <StaticMethod>ValidateCreateContactHistory</StaticMethod>
    <Assembly>ExtensionExample</Assembly>
    <Execute>true</Execute>
  </EventHandler>
</Event>

<Event>
  <EventName>Creating</EventName>
  <EventHandlerType>
    <Type>Microsoft.Dynamics.Common.BusinessObjectEventHandler</Type>
    <Assembly>Microsoft.Dynamics.Common</Assembly>
  </EventHandlerType>
  <EventHandler>
    <SoftwareVendor>MicrosoftDocumentation</SoftwareVendor>
    <Type>ExtensionExample.ContactHistory</Type>
    <StaticMethod>CreateContactHistory</StaticMethod>
    <Assembly>ExtensionExample</Assembly>
    <Execute>true</Execute>
  </EventHandler>
</Event>

<Event>
  <EventName>Deleting</EventName>
  <EventHandlerType>
    <Type>Microsoft.Dynamics.Common.BusinessObjectEventHandler</Type>
    <Assembly>Microsoft.Dynamics.Common</Assembly>
  </EventHandlerType>
  <EventHandler>
    <SoftwareVendor>MicrosoftDocumentation</SoftwareVendor>
    <Type>ExtensionExample.ContactHistory</Type>
    <StaticMethod>DeleteContactHistory</StaticMethod>
    <Assembly>ExtensionExample</Assembly>
    <Execute>true</Execute>
  </EventHandler>
</Event>

</Value>
</DictionaryEntry>
Chapter 24: Using Service Extensions

Application that access objects through the Dynamics GP service will have to work with the additional data that is included in the ExtensionList collection for each object. The information presented here provides examples that demonstrate how this is done. The following items are discussed:

- Retrieving extension data
- Creating or updating extension data

Retrieving extension data

The ExtensionList collection is available for every service object that inherits from the BusinessObject class. When an application retrieves an object from the Dynamics GP service, it should examine the ExtensionList collection to find whether any additional data is available in the Extension objects contained in the collection. The application should examine the ExtensionId parameter of each Extension object to find out what data is included in the extension.

For example, assume the Customer object was extended to include contact history information. An Extension object with the ExtensionId value “ContactHistory” could be included with the customer object when the object was retrieved. The application retrieving the customer object would look in the ExtensionList collection for this additional data.

Once the Extension object is found, the application will use the DocExtension property to retrieve the XML element that contains the additional data. In this example, the contact history XML element has the following format:

```xml
<ContactHistory>
    <FirstContactDate>6/6/1999 12:00:00 AM</FirstContactDate>
    <FirstContactSalesperson>NANCY B.</FirstContactSalesperson>
    <LastContactDate>3/1/2006 12:00:00 AM</LastContactDate>
    <LastContactSalesperson>ERIN J.</LastContactSalesperson>
</ContactHistory>
```

It is the responsibility of developer who extended the object to provide information about the format of the XML element containing the extended data.

The following C# example demonstrates retrieving the contact history information that is being included with the Customer object in the ExtensionList collection. Note how the ExtensionList collection is examined to determine whether the contact history information has been included. The XML element containing the contact history data is processed to retrieve the individual items.

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using DynamicsGPWebServiceSample.DynamicsGPService;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
```
(CompanyKey companyKey;
Context context;
Customer customer;
CustomerKey customerKey;
string firstContactDate;
string firstContactSalesperson;
string lastContactDate;
string lastContactSalesperson;

// Create an instance of the web service
DynamicsGP wsDynamicsGP = new DynamicsGP();

// Make sure that default credentials are being used
wsDynamicsGP.UseDefaultCredentials = true;

// Create a context with which to call the service
context = new Context();

// Specify which company to use (lesson company)
companyKey = new CompanyKey();
companyKey.Id = (-1);

// Set up the context
context.OrganizationKey = (OrganizationKey)companyKey;

// Create a customer key
customerKey = new CustomerKey();
customerKey.Id = "WORLDENT0001";

// Retrieve the customer object
customer = wsDynamicsGP.GetCustomerByKey(customerKey, context);

// Look for the contact history extension
foreach (Extension ext in customer.Extensions)
{
    if (ext.ExtensionId == "ContactHistory")
    {
        XmlNodeList nodeList;
        nodeList = contactHistory.ChildNodes;

        //First contact date
        firstContactDate = nodeList[0].InnerText.ToString();

        //First contact salesperson
        firstContactSalesperson = nodeList[1].InnerText.ToString();

        //Last contact date
        lastContactDate = nodeList[2].InnerText.ToString();

        //Last contact salesperson
        lastContactSalesperson = nodeList[3].InnerText.ToString();
    }
}
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            Customer customer;
            CustomerKey customerKey;
            string firstContactDate;
            string firstContactSalesperson;
            string lastContactDate;
            string lastContactSalesperson;

            // Create an instance of the web service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use (lesson company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a customer key
            customerKey = new CustomerKey();
            customerKey.Id = "AARONFIT0001";

            // Retrieve the customer object
            customer = wsDynamicsGP.GetCustomerByKey(customerKey, context);

            // Look for the contact history extension
            foreach (Extension ext in customer.Extensions)
            {
                if (ext.ExtensionId == "ContactHistory")
                {
                    // The extension is a LINQ element type
                    XElement contactHistory;
                    contactHistory = customer.Extensions[0].DocExtension;
Creating or updating extension data

When an application uses the Dynamics GP service to create a new object or update an existing object that has additional data included in the ExtensionList collection, it is the responsibility of the application to maintain this additional data. When creating new objects, the application must add the necessary Extension objects to the ExtensionList collection. When updating objects, the application must be sure the Extension objects in the collection contain the necessary information.

For instance, in the example discussed in Retrieving extension data on page 149, the Customer object has been extended to include contact history information in an extension with the ExtensionId value “ContactHistory”. The XML element containing this information has the following format:

```
<ContactHistory>
  <FirstContactDate>6/6/1999 12:00:00 AM</FirstContactDate>
  <FirstContactSalesperson>NANCY B.</FirstContactSalesperson>
  <LastContactDate>3/1/2006 12:00:00 AM</LastContactDate>
  <LastContactSalesperson>ERIN J.</LastContactSalesperson>
</ContactHistory>
```

The following C# example demonstrates creating the contact history information that is being included with the Customer object in the ExtensionList collection. An XML element is created that contains the contact history information. A new Extension object is created with the ExtensionId value “ContactHistory”. The XML element containing the contact history data is added as the DocExtension for the object. Finally, the new customer object is saved.

```
    // Use LINQ queries to retrieve the values
    // First contact date
    firstContactDate = (string)contactHistory
        .Element("FirstContactDate");

    // First contact salesperson
    firstContactSalesperson = (string)contactHistory
        .Element("FirstContactSalesperson");

    // Last contact date
    lastContactDate = (string)contactHistory
        .Element("LastContactDate");

    // Last contact salesperson
    lastContactSalesperson = (string)contactHistory
        .Element("LastContactSalesperson");
```

Legacy endpoint

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using DynamicsGPWebServiceSample.DynamicsGPSERVICE;
```
using System.Xml;
using System.Globalization;

namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            Customer customer;
            CustomerKey customerKey;
            Policy customerPolicy;
            DateTime today;

            XmlDocument doc;
           XmlElement contactHistoryXML;
            XmlElement firstContactDateXML;
            XmlElement firstContactSalespersonXML;
            XmlElement lastContactDateXML;
            XmlElement lastContactSalespersonXML;
            XmlText text;

            // Create an instance of the service
            DynamicsGP wsDynamicsGP = new DynamicsGP();

            // Make sure that default credentials are being used
            wsDynamicsGP.UseDefaultCredentials = true;

            // Create a context with which to call the service
            context = new Context();

            // Specify which company to use {lesson company}
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a new customer object
            customer = new Customer();

            // Create a customer key
            customerKey = new CustomerKey();
            customerKey.Id = "CONTOSO";
            customer.Key = customerKey;

            // Set properties for the new customer
            customer.Name = "Contoso, Ltd";

            // Customer Address Key
            CustomerAddressKey customerAddressKey = new CustomerAddressKey();
            customerAddressKey.CustomerKey = customerKey;
            customerAddressKey.Id = "PRIMARY";
        }
    }
}
// Customer Address List with Contact Person
CustomerAddress[] customerAddresses = new CustomerAddress[1];
customerAddresses[0] = new CustomerAddress();
customerAddresses[0].Key = customerAddressKey;
customerAddresses[0].ContactPerson = "Steve";
customer.DefaultAddressKey = customerAddressKey;
customer.Addresses = customerAddresses;

// Set today's date
today = DateTime.Today;

// Get the create policy for the customer
customerPolicy = wsDynamicsGP.GetPolicyByOperation("CreateCustomer", context);

// Add the Contact History (extended data) for the new customer

// Make a new Extension
Extension ext = new Extension();
ext.ExtensionId = "ContactHistory";

// Make the XML extension document
doc = new XmlDocument();

contactHistoryXML = doc.CreateElement("ContactHistory");

// First Contact Date
firstContactDateXML = doc.CreateElement("FirstContactDate");
firstContactDateXML.AppendChild(text);
contactHistoryXML.AppendChild(firstContactDateXML);

// First Contact Salesperson
firstContactSalespersonXML =
doc.CreateElement("FirstContactSalesperson");
text = doc.CreateTextNode("PAUL W.");
firstContactSalespersonXML.AppendChild(text);
contactHistoryXML.AppendChild(firstContactSalespersonXML);

// Last Contact Date
lastContactDateXML = doc.CreateElement("LastContactDate");
lastContactDateXML.AppendChild(text);
contactHistoryXML.AppendChild(lastContactDateXML);

// Last Contact Salesperson
lastContactSalespersonXML =
doc.CreateElement("LastContactSalesperson");
text = doc.CreateTextNode("PAUL W.");
lastContactSalespersonXML.AppendChild(text);
contactHistoryXML.AppendChild(lastContactSalespersonXML);

// Add the extension to the Customer object
ext.DocExtension = contactHistoryXML;
Extension[] extensionList = new Extension[1];
extensionList[0] = ext;
customer.Extensions = extensionList;

// Create the customer
wsDynamicsGP.CreateCustomer(customer, context, customerPolicy);

Native endpoint
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml.Linq;
using WebServiceSample.DynamicsGPService;
using System.Globalization;
namespace DynamicsGPWebServiceSample
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanyKey companyKey;
            Context context;
            Customer customer;
            CustomerKey customerKey;
            Policy customerPolicy;
            DateTime today;
            XElement contactHistoryXML;

            // Create an instance of the service
            DynamicsGPClient wsDynamicsGP = new DynamicsGPClient();

            // Create a context with which to call the web service
            context = new Context();

            // Specify which company to use (lesson company)
            companyKey = new CompanyKey();
            companyKey.Id = (-1);

            // Set up the context
            context.OrganizationKey = (OrganizationKey)companyKey;

            // Create a new customer object
            customer = new Customer();

            // Create a customer key
            customerKey = new CustomerKey();
            customerKey.Id = "CONTOSO";
            customer.Key = customerKey;
// Set properties for the new customer
customer.Name = "Contoso, Ltd";

// Customer Address Key
CustomerAddressKey customerAddressKey = new CustomerAddressKey();
customerAddressKey.CustomerKey = customerKey;
customerAddressKey.Id = "PRIMARY";

// Customer Address List with Contact Person
CustomerAddress[] customerAddresses = new CustomerAddress[1];
customerAddresses[0] = new CustomerAddress();
customerAddresses[0].Key = customerAddressKey;
customerAddresses[0].ContactPerson = "Steve";
customer.DefaultAddressKey = customerAddressKey;
customer.Addresses = customerAddresses;

// Set today's date
today = DateTime.Today;

// Get the create policy for the customer
customerPolicy = wsDynamicsGP.GetPolicyByOperation("CreateCustomer", context);

// Add the Contact History (extended data) for the new customer

// Make a new Extension
Extension ext = new Extension();
ext.ExtensionId = "ContactHistory";

// Make the XML extension document
contactHistoryXML = new XElement("ContactHistory",
    new XElement("FirstContactDate", today.ToString("G",
    DateTimeFormatInfo.InvariantInfo)),
    new XElement("FirstContactSalesperson", "PAUL W."),
    new XElement("LastContactDate", today.ToString("G",
    DateTimeFormatInfo.InvariantInfo)),
    new XElement("LastContactSalesperson", "PAUL W."));

// Add the extension to the Customer object
ext.DocExtension = contactHistoryXML;

ExtensionList extensionList = new ExtensionList();
extensionList.Add(ext);
customer.Extensions = extensionList;

// Create the customer
wsDynamicsGP.CreateCustomer(customer, context, customerPolicy);
Part 5: Creating a New Web Service

This portion of the documentation provides detailed information about creating a new service based on the Dynamics GP Service framework. The following items are discussed:

- **Chapter 25, “Designing a Service,”** describes key design elements to consider as you plan your new service.

- **Chapter 26, “Defining eConnect Operations,”** describes how to add the eConnect components that the Dynamics GP Service framework requires for your service.

- **Chapter 27, “Creating a Service Document Type,”** describes how to create a .NET assembly that defines the business document object used with your service.


- **Chapter 29, “Adding Policy Metadata,”** describes how to add new policies and behaviors to the Dynamics GP service.

- **Chapter 30, “Creating the Service Interface,”** describes how to create a service that uses the Microsoft Dynamics GP Service framework.

- **Chapter 31, “Creating XSLT Files,”** describes how to use XSLT to convert between the schema of the Dynamic GP Service framework documents and eConnect XML documents.

- **Chapter 32, “Deploying the Service,”** describes how to install your new service.
Chapter 25: Designing a Service

Several things must be considered when you are creating a new service. This portion of the documentation describes what you need to think about before you create a service. The following sections describe the process.

- Why create a service?
- Using the Dynamics GP Service framework
- Service design patterns
- Creating a service document type
- Using eConnect
- Document conversions
- Securing the service
- Using policy and behaviors

Why create a service?

Web Services for Microsoft Dynamics GP provides a way for external applications to integrate with data contained in Microsoft Dynamics GP. When you add a new type of document to Microsoft Dynamics GP, you should evaluate whether an external application will need to integrate with your document data. If you decide to allow external applications to use your document data, you will need to add a service for your document type.

When you add a service for a new type of document, the most important design decision is to define what operations your service provides to the external application. To define your service operations, you need to evaluate whether you want the external application to retrieve, create, update, or remove documents. The service operations you select have the following influence on your service design:

- You need to determine whether the Microsoft Dynamics GP Service framework supports the service operations you chose. The Dynamics GP Service framework supports service operations that retrieve a single document, retrieve a collection of documents, create a new document, and update, delete, or void an existing document.

- You need to add a service method for each operation that you chose. The methods allow the external application to perform operations with your document type.

- You need to add your document type to the Dynamics GP Service framework. Each service operation requires you to supply information about your document type to the Dynamics GP Service framework.

For example, the sample service included with the Web Services for Microsoft Dynamics GP SDK gives an external application the ability to retrieve, create, update, and delete lead documents. The sample creates a new service, adds a method for each operation, and adds leads as a new document type to the service framework.

For more information about Dynamics GP Services, see Chapter 1, “Dynamics GP Web Service Overview.”
Using the Dynamics GP Service framework

When you add a service for a new type of document, you need to use the Dynamics GP Service framework as your service foundation. The Dynamics GP Service framework allows you to add new document types.

The Dynamics GP Service framework defines a standard set of operations. The framework operations enable your service methods to perform actions on the Dynamics GP database. When you define the methods for your service, you need to select the framework operation that applies to each method. Your service methods can perform the following types of operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Inserts the data from a new document into the Microsoft Dynamics GP database. Requires you to supply the data fields for the complete document.</td>
</tr>
<tr>
<td>Delete</td>
<td>Removes an existing document from the Microsoft Dynamics GP database. Requires you to supply a key value that specifies the document to remove.</td>
</tr>
<tr>
<td>GetByKey</td>
<td>Retrieves a single document. Requires you to supply a key value that specifies the document to retrieve.</td>
</tr>
<tr>
<td>GetList</td>
<td>Retrieves a collection of one or more summary documents. Requires you to specify the criteria to use to query records for the collection.</td>
</tr>
<tr>
<td>Update</td>
<td>Modifies an existing document in the Dynamics GP database. Requires you to supply the data fields for the complete document.</td>
</tr>
<tr>
<td>Void</td>
<td>Marks an existing document as void but does not remove the document from the Microsoft Dynamics GP database. A Void operation begins like a Delete operation, you supply a key value that specifies the document to void. You complete the Void operation like an Update operation, you update the existing record and set a specified value in the field that identifies void records.</td>
</tr>
</tbody>
</table>

In addition, the Dynamics GP Service framework enables your service to use the native service endpoint, the legacy service endpoint, or both endpoints. The endpoints your service supports is based on the needs of the applications that use the service. The sample Leads service implements both endpoints. For more information about endpoints, see Chapter 3, “Endpoints.”

Service design patterns

The Dynamics GP Service framework supports the following types of service design patterns:

- A service that retrieves a single document or a collection of summary documents. This type of service requires you to use the GetByKey and GetList operations of the Dynamics GP Service framework.

- A service that creates, updates, and deletes or voids documents. This type of service requires you to use the Create, Update, Delete, or Void operations of the Dynamics GP Service framework.

The first step to design a new service requires you to determine what type of access you want to provide to an external application. The choice of design type determines the steps you need to complete to implement your service.
For example, a service that provides an external application the ability to retrieve an existing document requires you to implement the retrieve document design. However, a service that combines the ability to retrieve a document with the ability to create and update documents requires you to implement both the retrieve document and create and update document design patterns.

The following sections of the documentation describe the important elements of each service design pattern.
Retrieving documents
To enable an external application to retrieve a single document, the Dynamics GP Service completes the following process.

To retrieve more than one document, the Dynamics GP Service uses the Dynamics GP Service framework GetList operation. The process is similar except that GetList requires a criteria object instead of a key and returns a collection of summary documents.
To better understand how the service works, look at the role played by each element in the process.

- The Windows Communication Foundation (WCF) service provides a GetByKey method that the external application uses to retrieve a specified document. To use the service method, the external application supplies parameters that specify the document key, the type of the document, and a context object that provides important details of how the service call should be performed.

- The service method calls the GetByKey operation of the Dynamics GP Service framework.

- The Dynamics GP Service framework uses the Dynamics Security Service to verify that the user calling the service method is authorized to access the method. If the external application attempts to run a service method for which the current user does not have access, a security exception is raised and the operation ends.

- To retrieve the document data, the framework uses the document key with the eConnect Transaction Requester.

- The eConnect Transaction Requester performs the database query that retrieves the data for the specified document. The Transaction Requester uses metadata to determine the data fields for the specified type of document. The Transaction Requester returns the data in a Transaction Requester XML document.

- The framework uses XSLT (Extensible Stylesheet Language Transformation) to convert the XML document that the Transaction Requester produces into an XML document that the framework can deserialize into a document object.

- The service method returns the document object to the caller.

When you design a service that retrieves a new type of document, you need to supply the following elements.

- A .NET assembly that defines the document. The assembly contains classes that define the document data fields, the document key, the fields of the summary document, and the criteria and restrictions used to retrieve a collection of summary documents.

- A WCF service that contains the GetByKey and GetList methods for your document type.

- Transaction Requester metadata that describes the data fields to use in the query that retrieves the document or summary documents.

- An XSLT template that transforms the Transaction Requester XML document into a service XML document that the Dynamics GP Service framework serializes into a document object from your .NET document type assembly.

- Dynamics Security Service metadata that add GetByKey and GetList security operations for your document type. The Dynamics Security Service uses the security operation and the user role to determine whether the user is authorized to perform the GetByKey or GetList operation.
Creating, updating, and deleting or voiding documents

To enable an external application to create a document, the Dynamics GP Service uses the following process.

1. External application
2. Windows Communication Foundation service
3. Dynamics GP Service framework
4. Dynamics Security Service
5. "Create" policy
6. Service document object
7. Service and policy XML
8. XSLT
9. eConnect XML Document
10. "Create" stored procedure
11. Dynamics GP database
12. eConnect error information
13. eConnect return values
14. Service exceptions
To update, delete, or void a document, the Dynamics GP Service uses the Dynamics GP Service framework Update, Delete, or Void operations. The process is similar for each of these operations.

To better understand how the service works, look at role played by each element in the process.

- The WCF service provides a Create method that the external application uses to create a new document in Microsoft Dynamics GP. To use the method, the external application supplies parameters that include the document, the policy associated with the document, and a context object that provides important details of how the service call should be performed.

- The service method calls the Create operation of the Dynamics GP Service framework.

- The Dynamics GP Service framework uses the Dynamics Security Service to verify that the user calling the service method is authorized to create the document. If the external application attempts to run a method for which the current user does not have access, a security exception is raised and the operation ends.

- The Dynamics GP Service framework retrieves policy information for the document. The policy contains information that controls important characteristics of the new document.

- The framework serializes the document object and the policy information into a service XML document.

- The framework uses XSLT to convert the service XML document into an eConnect XML document that eConnect uses to run an eConnect SQL stored procedure. In addition, the conversion process is used to update the XML to include the characteristics specified by the policy XML.

- eConnect uses the XML document to create the document in Microsoft Dynamics GP. The eConnect stored procedure validates the data, implements business logic for the operation, and adds, updates or removes data in Microsoft Dynamics GP.

- The Dynamics GP Service framework checks the return values from eConnect to see whether an error occurred. If eConnect detected an error, the eConnect exception information is sent to the Dynamics GP Service exception system and is returned to the caller as a BusinessException.

When you design a service that creates, updates, deletes, or voids a document, you need to supply the following elements.

- A .NET assembly that defines the document. The assembly contains classes that define the document, and the document key.

- A WCF service that contains the Create, Update, and Delete or Void methods for your document type.

- An eConnect SQL stored procedure that validates the data, implements business logic for the operation, and adds, updates or removes data in Microsoft Dynamics GP.
• An eConnect XML document schema that works with your eConnect SQL stored procedure. The XML document schema needs to specify the name of the document type, the SQL stored procedure, and the data fields that represent the input parameters for the SQL stored procedure.

• An XSLT template that transforms the XML document from the service XML document schema to your eConnect XML document schema.

• Dynamics Security Service metadata that add Create, Update, and Delete or Void security operations for your document type. The Dynamics Security Service uses the security operation and the user role to determine whether the user is authorized to perform the specified operation.

• Policy metadata that add a policy and behaviors for each type of operation your service supports. The policy metadata defines the characteristics of the operation that can be configured for all documents of the specified type. In addition, the policy metadata allows the administrator to view and configure your policy in the Dynamics Security console.

Creating a service document type

To use the Dynamics GP Service framework, you need to add your document to the framework. To add a document type to the framework, there are several factors to consider.

First, select a type name for your document. The Dynamics GP Service framework uses the type name to identify components and resources related to your document. Select a type name that is unique and does not duplicate the name of an existing document type.

For example, the sample service included with the Web Services for Microsoft Dynamics GP SDK uses Lead as the type name of the service documents. If you look at the examples in the SDK, you will see the Lead type name used with classes, security metadata, policy metadata, and the XSLT files that the sample Leads service uses.

Next, define the structure and data members of the documents that the Dynamics GP Service framework requires.

• Specify the data members that represent a single document. You need to identify the name and data type for each data member of the document.

• Specify the most important data members of a document. Use these data members to define a summary document. You need to specify the name and data type for each data member of the summary document.

• Specify the types of restrictions used to retrieve a list of summary documents. Identify the data members you want to be able to query when retrieving a list of summary documents.

For information about how to create a document type assembly, see Chapter 27, “Creating a Service Document Type.”
Using eConnect

The Dynamics GP Service framework uses eConnect to retrieve, create, update, and delete data in the Microsoft Dynamics GP database. To add a document type to the service framework, you need to supply eConnect SQL stored procedures and Transaction Requester metadata for your document. The following sections describe what you need for each type of service design.

Using the Transaction Requester to retrieve documents

To retrieve a single record or a list of records, the Dynamics GP Service framework uses the eConnect Transaction Requester. When implementing a GetByKey or GetList operation, you need to identify the data fields for your document and summary document. You use the name of these fields when you add your document metadata to the Transaction Requester. The metadata provides the information the Transaction Requester uses to construct a query that returns the specified record or records.

The Transaction Requester returns an XML document that contains the document data. To enable the Dynamics GP Service framework to use the Transaction Requester, you need to map the data values from the Transaction Requester XML document to an instance of your service framework document or summary document.

For more information about how to use the Transaction Requester to retrieve documents, see Updating the Transaction Requester for retrieve operations on page 171.

Using eConnect SQL stored procedures to create, update, and delete documents

To implement service methods that create, update, and delete documents, you need to give the Dynamics GP Service framework access to the tables in the Microsoft Dynamics GP database that store the data for your document.

The Dynamics GP Service framework uses eConnect stored procedures to create, update, or delete data. To add your document to the service framework, you need to add a SQL stored procedure that works with eConnect for each operation. You must also define any business logic that is associated with each operation. When you create your SQL stored procedures, add the document business logic that you defined to each stored procedure.

To use your SQL stored procedures with eConnect, you must define eConnect XML documents for each operation. These eConnect XML documents must comply with the eConnect schema requirements. For example, your eConnect XML document must specify a document name, a SQL stored procedure name, and the name of each input variable.

To enable the Dynamics GP Service framework to use eConnect and your stored procedures, you need to map the data values from your service framework documents to your eConnect XML documents. The data fields of the eConnect XML document represent the input parameters of your eConnect stored procedures.

For more information about how to add SQL stored procedures to eConnect, see Adding a SQL stored procedure for create and update operations on page 174 and Adding a SQL stored procedure for a delete operation on page 185.
Document conversions

As described in the previous sections, the Dynamics GP Service framework converts a document from one XML document schema to another. The conversions require you to map the data members from one document to the corresponding data members of the other document.

To perform the conversion, the Dynamics GP Service framework uses XSLT. XSLT is a language that transforms an XML document into another document that is different in form or structure. When you add a new document to the service framework, you need to create an XSLT template that transforms the XML of the source document into the XML of the target document. The Dynamics GP Service framework requires you to supply XSLT for the following types of conversions:

- If your service includes a GetByKey method to retrieve a specified document, create an XSLT template that transforms the XML produced by the eConnect Transaction Requester into your service XML document.

- If your service includes a GetList method to retrieve a collection of summary documents, create an XSLT template that transforms the XML produced by eConnect Transaction Requester in your service XML summary document.

- If your service includes a Create, Update, Delete, or Void method, you need to create an XSLT template for each operation. In each case, you use the XSLT to transform the service XML document into an eConnect XML document that works with one of your eConnect SQL stored procedures.

For more information about creating XSLT templates, see Chapter 31, “Creating XSLT Files.”

Securing the service

The Dynamics GP Service framework uses the Dynamics Security Service to specify the users and groups that are able to execute a service method. You will use the Dynamics Security Service to secure access to the methods of your service. The Dynamics Security Service uses operations, tasks, and roles to control access to a service method.

To use the Dynamics Security Service, identify the methods you want to secure. Add metadata to the Dynamics Security Service that creates the security operations and tasks for your methods. After you add your security metadata, your service administrators can use the Dynamics Security Administration console to assign the operations or tasks to a role. For more information about how to add metadata to the Dynamics Security Service, see Chapter 28, “Adding Security Metadata.”
Using policy and behaviors

The Dynamics GP Service framework uses policies to enable a service administrator to specify important characteristics of a service operation. In the Dynamics GP Service framework, each create, update, delete, or void method requires a policy. You need to add a policy for each method you create that is one of these types.

When you create a policy, decide whether to add behaviors to the policy. Behaviors allow your service administrator to configure a characteristic for that method. For example, you could add a behavior that causes a create policy to use a specified value as the default value of a document data member. For more information about how to add policy metadata to the Dynamics GP Service, see Chapter 29, “Adding Policy Metadata.”

To add new policies and behaviors, add policy metadata to the Dynamics GP Service. Use the metadata to define the policy and the configurable behaviors for your methods. To configure your policy behaviors, use the Dynamics Security Console.

To implement a policy in the business logic for the service operation, you use the XSLT template for that operation. As a result, you need to identify the data members in your documents that are impacted by policy. For example, assume you defined a behavior to specify a default value for a data member when a document is created. In the XSLT template that transforms the service document being created into the eConnect XML that will be written to the Dynamics GP database, the XSLT code will examine the policy information and then set the appropriate data member to the default value.
Chapter 26: Defining eConnect Operations

Microsoft Dynamics GP eConnect is the layer in the service architecture that retrieves, creates, updates, and deletes data. The following sections describe how to add the eConnect support that the Dynamics GP Service framework needs for a new type of document.

- Overview of eConnect for services
- Updating the Transaction Requester for retrieve operations
- Adding a SQL stored procedure for create and update operations
- Adding a SQL stored procedure for a delete operation
- Adding Error Codes

Overview of eConnect for services

Web Services for Microsoft Dynamics GP uses eConnect to retrieve, create, update, and delete records in the Microsoft Dynamics GP database. The Dynamics GP Service framework uses the following eConnect components.

- The eConnect Transaction Requester that retrieves data from the Microsoft Dynamics GP database. The Transaction Requester uses metadata to specify the data to retrieve. For more information about using the Transaction Requester, see Updating the Transaction Requester for retrieve operations on page 171.

- An eConnect business object that inserts, updates, or deletes data in the Microsoft Dynamics GP database. In eConnect, a business object is implemented as a SQL stored procedure. For more information about creating eConnect SQL stored procedures, see Adding a SQL stored procedure for create and update operations on page 174.

For more information about business objects, eConnect XML documents, and the Transaction Requester, see the eConnect Programmer’s Guide.

Updating the Transaction Requester for retrieve operations

The Dynamics GP Service framework uses the eConnect Transaction Requester to retrieve the data for a service document. If your service uses the GetByKey or GetList operations of the Dynamics GP Service framework, you need to use the Transaction Requester to get your document data.

- For a GetByKey operation, the Transaction Requester retrieves a single document. To specify the document, you must supply the document ID.

- For a GetList operation, the Transaction Requester retrieves a collection of summary documents that match a specified criteria. The criteria represents a value or range of values that is shared by each document in the collection.

To specify the data to retrieve, the Transaction Requester uses metadata. The metadata describes the data members of your document and provides the information that eConnect uses to construct a query. The query retrieves the data for the specified document from the Dynamics GP database.
The Transaction Requester returns the data it retrieves as a Transaction Requester XML document. The document contains XML nodes that identify each data field and the current value of that data field.

To use the Transaction Requester with a new type of document, you need to provide metadata that describes the data members of your document and summary document. The following steps show how to add metadata for a new document type to the eConnect_Out_Setup table.

**Naming document types**
One of the key metadata elements is the name. To comply with naming conventions in eConnect and Dynamics GP Services, start the name with “WS”. Next, append the type name. Finally, if your are retrieving summary documents, append “GetList”. For example, the Transaction Requester metadata for lead documents uses the following names:

WSLead  
WSLeadGetList

When you add your document metadata to the eConnect_Out_Setup table, you must use the name for the DOCTYPE parameter.

**Adding Transaction Requester SQL stored procedures**
To add your document to the Transaction Requester, you need to run the eConnectOutCreate stored procedure. This stored procedure creates a new stored procedure that contains the query that retrieves the data for your document. The eConnectOutCreate stored procedure requires you to supply a name and an output type. Use the name that you created earlier in this section.

The output type specifies whether a single document (GetByKey) or a collection of summary documents (GetList) is being created.

- If you are adding a Transaction Requester operation for only a GetByKey operation, set the output type parameter to “1”.

- If you are adding Transaction Requester operations for both a GetByKey and a GetList operation, the GetList operation must be added first with the output type parameter “1”. Then the GetByKey operation can be added with the output type parameter “2”.

The following section contains a SQL script example that shows how to use the eConnectOutCreate stored procedure.

**Adding Transaction Requester metadata**
To add a new type of document to the Transaction Requester, complete the following procedure:

1. **Open the Visual Studio solution.**
   Use the same project you used for the business objects. Open the solution in Visual Studio.
2. **Add a SQL script file.**
   From the Project menu, choose Add SQL Script. In the Add New Item window, click SQL Script in the list of Templates. Enter a name for your script file, and then click Add.

3. **Add statements to the SQL script file.**
   Use the SQL script to add entries to the eConnect_Out_Setup table of the company database.

   The following sample script inserts metadata for leads to the eConnect_Out_Setup table. Notice how the first insert statement specifies all the data fields. This metadata describes how to retrieve a single document.

   The second insert statement specifies a smaller number of fields. This metadata describes how to retrieve a summary document. Dynamics GP Services uses summary documents to produce a list of documents.

   Also, notice the name used by the the DOCTYPE parameter of each insert.

   ```sql
   GO
   /* Specify the data to retrieve with a GetByKey request */
   INSERT INTO dbo.eConnect_Out_Setup
   (DOCTYPE, TABLENAME, ALIAS, MAIN, INDEX1, INDEXCNT, DATACNT, DATA1, DATA2, DATA3, DATA4, DATA5, DATA6, DATA7, DATA8, DATA9, DATA10, DATA11, DATA12, DATA13, DATA14, DATA15, DATA16, DATA17, DATA18, DATA19, DATA20, DATA21, DATA22, DATA23)
   VALUES('WSLead', 'IG001', 'Lead', 1, 'LeadID', 1, 23, 'LeadName', 'SLPRSNID', 'CITY', 'STATE', 'ZIP', 'ADDRESS1', 'ADDRESS2', 'PHONE1', 'PHONE2', 'FAX', 'LeadBusinessCategory', 'COUNTRY', 'CONTACT', 'PotentialRevenue', 'QualifiedLead', 'LeadSource', 'QualificationDate', 'LeadPassword', 'NOTEINDX', 'Workflow_Approval_Status', 'Workflow_Priority', 'Approved_Salesperson_ID', 'DEX_ROW_TS')
   /* Specify the data to retrieve with a GetList request */
   INSERT INTO dbo.eConnect_Out_Setup
   (DOCTYPE, TABLENAME, ALIAS, MAIN, INDEX1, INDEXCNT, DATACNT, DATA1, DATA2, DATA3, DATA4, DATA5, DATA6)
   VALUES('WSLeadGetList', 'IG001', 'Lead', 1, 'LeadID', 1, 6, 'LeadName', 'SLPRSNID', 'LeadBusinessCategory', 'QualifiedLead', 'LeadSource', 'DEX_ROW_TS')
   GO
   */

4. **Create the Transaction Requester SQL stored procedures.**
   To create the Transaction Requester SQL stored procedures for your document type, run the eConnectOutCreate stored procedure.

   The following script sample creates Transaction Requester SQL stored procedures for leads. Notice how the GetList operation is added first and with an output type of “1”. The following statement adds the GetByKey operations and uses an output type of “2”.

   ```sql
   /* Create procedures for LeadGetList and LeadGetByKey */
   exec eConnectOutCreate 'WSLeadGetList', 1
   exec eConnectOutCreate 'WSLead', 2
   ```
5. **Save the file.**
   In the File menu, choose Save. Once you have completed the script files for eConnect, close the Visual Studio project.

**Adding a SQL stored procedure for create and update operations**

The Dynamics GP Service framework uses eConnect SQL stored procedures to insert or update a document in the Dynamics GP database. If your service uses the Create, Update, or Void operations of the Dynamics GP Service framework, you need to add SQL stored procedures for that operation.

In addition, eConnect SQL stored procedures validate data and implement the business logic associated with the document. The following diagram shows the typical structure of an eConnect SQL stored procedure that creates or updates a document.

When you add your document type to eConnect, you need to create a SQL stored procedure that performs each of these steps.

- Specify the input parameters. The input parameters define the data fields that the SQL stored procedure uses to create, update, or delete a document. In addition, you need to add the output parameters required by the eConnect error reporting system.

- Validate the input parameters. Verify that the required data fields for your document contain data values.
• Check whether the specified document exists. Use the result to determine whether your SQL stored procedure can perform the specified operation.

• Perform business logic associated with the document. Business logic often includes validating input parameter values, retrieving additional data, and performing calculation to produce new data values.

• Determine whether to lock a record during an update. To obtain exclusive access to a record, implement Dynamics GP active locking in your SQL stored procedure. For more information about active locking, see the Microsoft Dexterity documentation.

• Use the eConnect error reporting system to report errors. If your SQL stored procedure encounters validation or business logic errors, use the eConnect error reporting system to notify the caller of the error. For more information about adding error codes and messages, see Adding Error Codes on page 188.

To complete a create, update, or delete operation, the Dynamics GP Service framework sends an XML document to eConnect. The XML document specifies the name of the SQL stored procedure and contains XML nodes that supply data for the input parameters of the stored procedure.

To add a new document type to the Dynamics GP Service platform, you need to add SQL stored procedures that create, and update documents in the Dynamics GP database. One SQL stored procedure can be used to perform both the create and update operations for your document type. To create a SQL stored procedure that performs the create and update operations, complete the following steps:

Create a Visual Studio solution
Open Visual Studio. In the File menu, point to New, and then click Project. In the New Project window, expand Other Project Types, and then select Database from the list of Project types.

Enter a name for your project. Review the Location and Solution Name, and then click OK.

If you are prompted to add a database reference, add a reference to your Dynamics GP data server, and then click OK.

Add a SQL script file
From the Project menu, choose Add SQL Script. In the Add New Item window, click SQL Script in the Templates list. Enter a name for your script file, and then click Add.
Add SQL statements to create and define the stored procedure

Add SQL statements that specify the name of the stored procedure, add it to the stored procedures of a Dynamics GP company database, and define the input and output parameters of the stored procedure.

The following sample shows how to create a stored procedure named sampleLeadCreateUpdate. This stored procedure is used when eConnect performs create or update operations for the IG_Lead_MSTR table (IG001).

Notice the naming convention used with the input parameters. The name of each input parameter specifies a data field from the IG001 table. In addition, each input parameter name starts with "@I_v". The use of this naming convention makes it easier to see how the input parameters are used in the stored procedure. While not required, you should adopt a similar convention that makes it easier to update and maintain your stored procedure.

Also notice how each parameter specifies the data type and size for that parameter. When you specify input parameters, use the same type and size as the data field in the Dynamics GP database.

In this example, the parameters designated as /*<Required>*/ identify parameters that must contain data. Required parameters do not have a default value. The other parameters are optional parameters. Optional parameters must have a default value. Set the default to a value that allows the document to be successfully created or updated in the Dynamics GP database.

The parameters also include two output parameters named @O_iErrorState and @oErrString. These parameters are used in eConnect error reporting and must be included with every eConnect business object. The type and size of each output parameter is also required.

```sql
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'sampleLeadCreateUpdate')
BEGIN
    DROP Procedure dbo.sampleLeadCreateUpdate
END
GO
CREATE Procedure dbo.sampleLeadCreateUpdate
(  @I_vLeadID char(15), /*<Required>*/
  @I_vLeadName char(31), /*<Required>*/
  @I_vSLPRSNID char(15), /*<Required>*/
  @I_vCITY char(35) = '',
  @I_vSTATE char(29) = '',
  @I_vZIP char(11) = ''
);```
CHAPTER 26  DEFINING ECONNECT OPERATIONS

Define and initialize local variables.

The stored procedure for a create/update operation requires four local variables that are used to manage status and error state information. The code in the stored procedure must initialize the local variables.

The following script sample shows how the required local variables for the create/update stored procedure are defined and initialized. The sample also shows how the @oErrString parameter is initialized.

```sql
/** Declare local variables **/
declare
    @iStatus int,
    @iErrorCodeErrState int,
    @exists tinyint,
    @O_oErrorState int,

/** Initialize local variables **/
select @O_oErrorState = 0,
    @iStatus    = 0,
    @exists     = 0,
    @O_oErrorState = 0,

if (@oErrString is NULL)
    begin
        select @oErrString = ''
    end
```
Add SQL statements to validate the input parameters

The stored procedures should contain code to verify that the input parameters contain valid data.

The following script sample shows how to validate input parameters. Notice how a value is assigned to the @O_iErrorState output parameter after each validation error.

The eConnect error reporting system uses this parameter value to retrieve the error message from the taErrorCode table of the the DYNAMICS database.

For more information about how to add error codes, see *Adding Error Codes* on page 188.

```sql
/** Verify there are no null required parameters **/
if(@I_vLeadId is null or
   @I_vLeadName is null or
   @I_vSLPRSNID is null or
   @I_vLeadBusinessCategory is null)
begin
    select @O_iErrorState = 61061 /* A required parameter was null */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @oErrString,
        @oErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end

/** Verify there are no invalid negative amounts **/
if (@I_vPotentialRevenue < 0)
begin
    select @O_iErrorState = 61062 /* Potential Revenue value is not valid */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @oErrString,
        @oErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end
```

Determine whether the record exists

Determine whether the SQL stored procedure should create a new record or update an existing record. The following script sample shows how to use a query to identify whether the specified record exists.

```sql
/** Determine whether the specified record exists **/
/* If the specified lead exists, set the @exists local variable to 1 */
if(exists(select top 1 LeadID from IG001 (nolock) where LeadID = @I_vLeadID))
begin
    select @exists = 1
end
```
Implement the business logic
Implement the business logic for the document and the operation. Business logic ranges from simply validating parameters to performing additional calculations required by the document.

The following script sample implements business logic associated with a lead document. In this example, the business logic validates several of the data values for the lead being created or updated. If a validation error occurs, the appropriate error code is assigned to the @O_iErrorState output parameter, the stored procedure ends, and the output parameters are returned to the caller.

/* Verify the required ID field is populated */
if(@I_vLeadID = '')
begin
    select @O_iErrorState = 61063 /* A Lead ID value was not supplied */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @ErrString,
        @ErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end
/* Verify the required Name field is populated */
if(@I_vLeadName = '')
begin
    select @O_iErrorState = 61064 /* The Name value was not supplied */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @ErrString,
        @ErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end
/* Verify the required Salesperson ID field is populated */
if(@I_vSLPRSNID = '')
begin
    select @O_iErrorState = 61065 /* The Salesperson was not specified */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @ErrString,
        @ErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end
/* Validate that the salesperson ID exists */
/* Verify the salesperson ID is in the RM00301 table of the company */
if(not exists(select top 1 SLPRSNID from RM00301 (nolock) where SLPRSNID = @I_vSLPRSNID))
begin
    select @O_iErrorState = 61066 /* The salesperson does not exist */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @ErrString,
        @ErrString output,
        @O_oErrorState output
end
return (@O_iErrorState)
end

/* Validate the LeadBusinessCategory value */
if((@I_vLeadBusinessCategory < 1) or (@I_vLeadBusinessCategory > 8))
begin
    select @O_iErrorState = 61067 /* The Business Category is not valid */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @oErrString,
        @oErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end

/* Validate the qualified lead value */
if((@I_vQualifiedLead < 1) or (@I_vQualifiedLead >2))
begin
    select @O_iErrorState = 61068 /* Qualified property is not valid */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @oErrString,
        @oErrString output,
        @O_oErrorState output
    return (@O_iErrorState)
end

/* Validate that Lead Source and Qualification date are only supplied */
/* when the Qualified Lead value is true */
if(@I_vQualifiedLead = 1)
begin
    if((@I_vLeadSource <> '') or (@I_vQualificationDate <> ''))
    begin
        select @O_iErrorState = 61069 /* Source and Qualification */
            /* cannot be set when the Qualified field is false */
        exec @iStatus = taUpdateString
            @O_iErrorState,
            @oErrString,
            @oErrString output,
            @O_oErrorState output
        return (@O_iErrorState)
    end
end
Add SQL statements to create a new record

The stored procedure must contain code to create a new record in the Dynamics GP database. Use the value of the @exists variable that you set earlier to determine whether to create a record in the Dynamics GP database. If @exists is false (0), insert a new record in the database. Use the input parameters and variables to supply data values for each data field of the new record.

The following script sample shows how to check the value of the @exists variable and how to add a SQL insert statement that creates a lead.

```sql
/** If the specified record does not exist, insert the record in IG001 **/
if(@exists = 0)
begin
    insert IG001
    {
        LeadID,
        LeadName,
        SLPRSNID,
        CITY,
        STATE,
        ZIP,
        ADDRESS1,
        ADDRESS2,
        PHONE1,
        PHONE2,
        FAX,
        LeadBusinessCategory,
        COUNTRY,
        CONTACT,
        PotentialRevenue,
        QualifiedLead,
        LeadSource,
        QualificationDate,
        Workflow_Approval_Status,
        Workflow_Priority,
        Approved_Salesperson_ID
    }
    select
        @I_vLeadID,
        @I_vLeadName,
        @I_vSLPRSNID,
        @I_vCITY,
        @I_vSTATE,
        @I_vZIP,
        @I_vADDRESS1,
        @I_vADDRESS2,
        @I_vPHONE1,
        @I_vPHONE2,
        @I_vFAX,
```
@I_vLeadBusinessCategory,
@I_vCOUNTRY,
@I_vCONTACT,
@I_vPotentialRevenue,
@I_vQualifiedLead,
@I_vLeadSource,
@I_vQualificationDate,
@WorkflowApprovalStatus,
@WorkflowPriority,
@ApprovedSalesperson

if(@error <> 0)
begin
    select @O_iErrorState = 61070/* An insert error occurred */
    exec @iStatus = taUpdateString
        @O_iErrorState,
        @oErrString,
        @oErrString output,
        @iAddCodeErrState output
    return (@O_iErrorState)
end
end

**Lock the record (optional)**

If your business object updates a record in a table that supports active locking, use a lock to obtain exclusive access to the record. To use locking, add a local variable and initialize it to zero.

@lock tinyint

@lock = 0

The eConnect business object implement active locking by calling a stored procedure named taDEXLOCKS. To use taDEXLOCKS, set the @vOperation parameter to “1”, specify the table name, and row ID of the record to lock. The following script sample locks a purchase order.

/* Get the table and row data for taDEXLOCKS */
select @POPHdrTbl = rtrim(db_name()) + '.dbo.POP10100'
select @DexRowID = DEX_ROW_ID from POP10100 (nolock) where PONUMBER = @I_vPONUMBER

/* Check for an existing lock */
if (not exists(select 1 from tempdb..DEX_LOCK (nolock) where table_path_name = @POPHdrTbl and row_id = @DexRowID and session_id = @@spid))
begin
    /* If the record is not locked, set a new lock */
    select @lock = 1
    exec @iStatus = DYNAMICS..taDEXLOCKS
        @I_vOperation = 1,
        @I_vtable_path_name = @POPHdrTbl,
        @I_vrow_id = @DexRowID,
        @O_oExists = @oExists output,
        @O_oInsertStatus = @OInsStatus output,
        @O_iErrorState = @DexLockErrorState output
/* Handle locking errors */
if ((@OInsStatus <> 1) or (@DexLockErrorState <> 0) or (@iError <> 0))
begin
  if (@DexLockErrorState <> 0)
  begin
    select @oErrString = rtrim(@oErrString) + ' ' + @DexLockErrorState
  end
  select @O_iErrorState = 9184
  exec @iStatus = taUpdateString
  @O_iErrorState,
  @oErrString,
  @oErrString output,
  @O_oErrorState output
  return (@O_iErrorState)
end

Add SQL statements to update an existing record

The stored procedure must contain code to update an existing record in the Dynamics GP database. Use the value of the @exists variable that you set earlier to determine whether to update a record. If @exists is true (1), update the existing record in the database. Use the input parameters and variables to supply data values for each data field of the specified record.

The following script sample shows how to check the value of the @exists variable and how to add a SQL statement that updates a lead.

/** If the specified record exists, update the existing record in IG001 **/
if(@exists <> 0)
begin
  update IG001 set
    LeadName = @I_vLeadName,
    SLPRSNID = @I_vSLPRSNID,
    CITY = @I_vCITY,
    STATE = @I_vSTATE,
    ZIP = @I_vZIP,
    ADDRESS1 = @I_vADDRESS1,
    ADDRESS2 = @I_vADDRESS2,
    PHONE1 = @I_vPHONE1,
    PHONE2 = @I_vPHONE2,
    FAX = @I_vFAX,
    LeadBusinessCategory = @I_vLeadBusinessCategory,
    COUNTRY = @I_vCOUNTRY,
    CONTACT = @I_vCONTACT,
    PotentialRevenue = @I_vPotentialRevenue,
    QualifiedLead = @I_vQualifiedLead,
    LeadSource = @I_vLeadSource,
    QualificationDate = @I_vQualificationDate
  where LeadID = @I_vLeadID
if(@@error <> 0)
begin
  select @O_iErrorState = 61071/* An update error occurred */
  exec @iStatus = taUpdateString
  @O_iErrorState,
Remove the lock (optional)

If you set a lock on the record, you must release the lock. The following sample shows how to use taDEXLOCKS to release a lock on a purchase order. Notice how the value of the @_vOperation parameter is now set to “3”.

if (@lock = 1)
begin
exec @iStatus = DYNAMICS..taDEXLOCKS
@I_vOperation = 3,
@I_vtable_path_name = @POPHdrTbl,
@I_vrow_id = @DexRowID,
@O_oExists = @oExists output,
@O_oInsertStatus = @OInsStatus output,
@O_iErrorState = @DexLockErrorState output
select @iError = @@error
/* Handle locking errors */
if ((@iStatus <> 0) or (@DexLockErrorState <> 0) or (@iError <> 0))
begin
if (@DexLockErrorState <> 0)
begin
select @oErrString = rtrim(@oErrString) + ' ' +
@DexLockErrorState
end
select @O_iErrorState = 9222
exec @iStatus = taUpdateString
@O_iErrorState,
@oErrString,
@oErrString output,
@O_oErrorState output
end
end

Specify security permissions for the stored procedure

To allow access to the stored procedure, specify the security permissions required to run this stored procedure. Dynamics GP eConnect grants execute permissions to DYNGRP.

GRANT EXEC ON dbo.sampleLeadCreateUpdate TO DYNGRP
GO

Save the file

From the File menu, choose Save.
Adding a SQL stored procedure for a delete operation

To use the Dynamics GP Service framework to delete a document, you need to add a SQL stored procedure that removes the specified document from the Dynamics GP database. To create the SQL stored procedure, complete the following steps:

Open the Visual Studio solution
Use the same project you used for the Create/Update business object. Open the solution in Visual Studio.

Add a SQL script file
From the Project menu, choose Add SQL Script. In the Add New Item window, click SQL Script in the list of Templates. Enter a name for your script file, and then click Add.

Create a SQL stored procedure
Specify a name for the stored procedure and add the parameters. The following sample script creates a stored procedure named sampleLeadDelete. Notice that the only input parameter is the ID of the record to be deleted. The parameters also include the two required output parameters.

```sql
IF EXISTS (SELECT * FROM dbo.sysobjects WHERE type = 'P'
    AND name = 'sampleLeadDelete')
BEGIN
    DROP Procedure dbo.sampleLeadDelete
END
GO

CREATE Procedure dbo.sampleLeadDelete
(
    @I_vLeadID char(15),
    @O_iErrorState int OUTPUT,
    @oErrString varchar(255) OUTPUT
)
AS
set transaction isolation level read uncommitted
set nocount on
```
Declare and initialize local variables
Add variables you use in your stored procedure and supply a default value for each. The following script sample declares and initializes local variables. The sample also initializes the @O_iErrorState, and @ErrString output parameter.

```sql
declare
  @iStatus int,
  @iAddCodeErrState int

select @O_iErrorState = 0,
      @iStatus = 0

if (@oErrString is NULL)
begin
  select @oErrString = ''
end
```

Validate the input parameters
Verify that the input parameter contains valid data. The following script sample shows how to validate the ID parameter.

```sql
if (@I_vLeadID is NULL)
begin
  select @O_iErrorState = 61050/* The Lead ID value cannot be null */
  exec @iStatus = taUpdateString
    @O_iErrorState,
    @oErrString,
    @oErrString output,
    @iAddCodeErrState output
  return (@O_iErrorState)
end

if (@I_vLeadID = '')
begin
  select @O_iErrorState = 61051/* The Lead ID values is not valid */
  exec @iStatus = taUpdateString
    @O_iErrorState,
    @oErrString,
    @oErrString output,
    @iAddCodeErrState output
  return (@O_iErrorState)
end
```
**Determine whether the record exists**

Verify the input parameters contains a valid ID. The following script sample uses a query to identify whether the LeadID value is an existing lead. If the lead does not exist, an error is assigned to @O_iErrorState, and the stored procedure ends.

```sql
select @I_vLeadID = UPPER(@I_vLeadID)
if (@I_vLeadID <> '')
begin
    if not exists (select 1 from IG001 (nolock) where LEADID = @I_vLeadID)
    begin
        select @O_iErrorState = 61052/* The lead does not exist */
        exec @iStatus = taUpdateString
        @O_iErrorState,
        @oErrString,
        @oErrString output,
        @iAddCodeErrState output
    end
end
```

**Delete the specified record**

Use the SQL delete statement to remove the specified record. The following script sample shows how to use the SQL delete statement to remove a lead.

```sql
delete IG001 where LEADID = @I_vLeadID
if @@error <> 0
begin
    select @O_iErrorState = 61053
    exec @iStatus = taUpdateString
    @O_iErrorState,
    @oErrString,
    @oErrString output,
    @iAddCodeErrState output
    return (@O_iErrorState)
end
return (@O_iErrorState)
GO
```

**Specify security permissions for the stored procedure**

To allow access to the stored procedure, specify the security permissions required to run the stored procedure. Dynamics GP eConnect grants execute permissions to DYNGRP.

```sql
GRANT EXEC ON dbo.sampleLeadDelete TO DYNGRP
GO
```

**Save the file**

From the File menu, choose Save.
Adding Error Codes

When you use an eConnect SQL stored procedure to validate input parameters and perform business logic, you need to alert the caller when an error occurs. The eConnect error reporting system allows you to report these types of errors.

To use the eConnect error reporting system, you need to add error codes and messages for each validation or business logic check that you perform. The error code uniquely identifies your error message. The error message contains a brief description of the error that was detected.

The Dynamics GP Service framework uses the eConnect error reporting system to log errors identified by your SQL stored procedure in the Dynamics GP Web Services Exceptions console.

To provide error messages for your business objects errors, add your error codes and messages to the taErrorCodes table of the DYNAMICS database. To add custom error information, complete the following procedure:

Open the Visual Studio solution
Use the same project you used for the SQL stored procedures. Open the solution in Visual Studio.

Add a SQL script file
From the Project menu, choose Add SQL Script. In the Add New Item window, click SQL Script in the list of Templates. Enter a name for your script file, and then click Add.

Create a SQL script
Use the SQL insert statement to add your error code and error message entries to the taErrorCodes table of the DYNAMICS database. The eConnect error reporting system requires you to specify an error code, the name of the stored procedure where the error occurs, and the error message to display. You can also supply the name of the field, and the parameter, but these are optional.

The following script sample shows how to add error information. Notice how each error code corresponds to an error code in the create/update or delete stored procedure samples.

```sql
USE DYNAMICS
GO

/* Add entries for the lead delete operation */
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61050','sampleLeadDelete','The Lead ID value cannot be null','LeadID','')
```
/* Add entries for the lead create/update operation */
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61061','sampleLeadCreateUpdate','A required parameter was set to null','','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61062','sampleLeadCreateUpdate','The specified Potential Revenue or Business Category is not valid','','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61063','sampleLeadCreateUpdate','A Lead ID value was not supplied','LeadID','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61064','sampleLeadCreateUpdate','The Name value was not supplied','Name','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61065','sampleLeadCreateUpdate','The Salesperson was not specified','SLSPRSNID','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61066','sampleLeadCreateUpdate','The specified salesperson does not exist','SLSPRSNID','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61067','sampleLeadCreateUpdate','The Business Category value cannot be empty','BusCategory','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61068','sampleLeadCreateUpdate','The specified Business Category value is not a valid entry','BusCategory','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61069','sampleLeadCreateUpdate','The value supplied for the Qualified property is not a valid entry','Qualified','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61070','sampleLeadCreateUpdate','Source and Qualification values cannot be set when the Qualified field is false','Qualified','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61071','sampleLeadCreateUpdate','An error occurred during the insert operation the Lead was not created','','')
INSERT INTO
dbo.taErrorCode(ErrorCode,SourceProc,ErrorDesc,ErrorKeyFields,ErrorParms)
VALUES('61072','sampleLeadCreateUpdate','An error occurred during the update operation','','')

Save the file.
From the File menu, choose Save.
Chapter 27: Creating a Service Document Type

To add a new type of document to Web Services for Microsoft Dynamics GP, create a .NET assembly that describes the document type. The steps to create a .NET document type assembly are described in the following sections:

- Overview of a document type
- Creating the project for the document type
- Defining the document type
- Defining the key
- Defining the summary
- Defining the summary document list
- Adding criteria
- Adding an enumeration
- Building the assembly

Overview of a document type

You need to choose a name for your document type. Use this type name to identify your document type class and its related classes. For example, the code samples that follow use a type name of Lead.

You will encounter a number of services naming conventions that require you to include your type name.

When you create a document type assembly, you need to specify a namespace that includes all your document classes. Choose a namespace that ensures your class and enumeration names are unique.

To add a new type of document to the Dynamics GP service, you must create a .NET assembly that defines the document type. The Dynamics GP Service framework requires the assembly to include classes for the following:

<table>
<thead>
<tr>
<th>Class use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document type</td>
<td>A class that defines the business document. The class must inherit from the Microsoft.Dynamics.Common.BusinessObject class. The class must be serializable and uniquely identifiable.</td>
</tr>
<tr>
<td>Document key</td>
<td>A class with a data property that uniquely identifies a single document. The class must inherit from the Microsoft.Dynamics.GP.ReferenceKey class. The class must be serializable.</td>
</tr>
<tr>
<td>Summary</td>
<td>A class that defines a limited number of the most important data fields from a business document. The class must inherit from the Microsoft.Dynamics.Common.BusinessObject class. The class must be serializable.</td>
</tr>
<tr>
<td>Criteria</td>
<td>A class that contains the restrictions that define what is returned by a GetList operation in the Dynamics GP service. The class must inherit from the Microsoft.Dynamics.Common.Criteria class. The class must be serializable.</td>
</tr>
<tr>
<td>Summary array</td>
<td>A class that enables the service to retrieve a collection of summary documents. The class must inherit from the System.Collections.Generic.List class.</td>
</tr>
</tbody>
</table>
When you create a new document type, you need to specify the data members for your document. The data members in your document type correspond to the data fields that are supported by your eConnect business objects and Transaction Requester metadata that you defined. For more information about adding eConnect support, see Chapter 26, “Defining eConnect Operations.”

Creating the project for the document type

To build a .NET assembly for your document type, use Visual Studio to create a library project. To create the project, complete the following procedure:

1. Create a Visual Studio class library project.
   Open Visual Studio. In the File menu, point to New, and then click Project. In the New Project window, select Visual C# in the Project types list. In Templates, select Class Library from the list of Visual Studio installed templates.

   Enter a name for your project. A typical project name includes the company name, the product name, and the document type. For example, the Leads service uses the following project name:

   Microsoft.Dynamics.GP.Samples.Leads

   The name you specify will be the namespace for all the classes in your document type.

   Review the Location and Solution Name, and then click OK.

2. Delete the Class1.cs file from the project.
   By default, Visual Studio creates a file named Class1.cs. The Class1.cs file is not needed. To delete the file, open the View menu, and choose Solution Explorer. Use Solution Explorer to find Class1.cs, and delete it from the project.

   If a dialog window opens asking whether to delete the file, click OK.

3. Add references.
   In the Project menu, choose Add References. The Add References window opens. Click the Browse tab and navigate to the Dynamics GP “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

   Select the following assemblies, and then click OK:

   - Microsoft.Dynamics.Common.dll
   - Microsoft.Dynamics.Common.Types.dll
   - Microsoft.Dynamics.GP.BusinessLogic.dll

4. Save the project.
   In the File menu, choose Save.
Defining the document type

Create a class that defines the data members of your document. To add the class to your class library, complete the following steps:

1. **Create a class.**
   In the Visual Studio Project menu, choose Add New Item. Select Class from the list of Templates in the Add New Item window. Enter your document type in the Name box, and then click Add. Visual Studio adds the class to the project.

   For example, you would enter Lead.cs in the Name box to create the Lead class for the Lead document type. The following sample code shows the Lead class. Notice how the namespace and class name specify the Lead document type.

   ```csharp
   using System;
   using System.Collections.Generic;
   using System.Text
   namespace Microsoft.Dynamics.GP.Samples.Leads
   {
       class Lead
       {
           
       }
   }
   ```

2. **Add namespace references.**
   To simplify the use of several classes and methods, add using statements for the following namespaces:

   - System.Runtime.InteropServices
   - System.Xml.Serialization
   - Microsoft.Dynamics.Common
   - Microsoft.Dynamics.GP

3. **Add class attributes.**
   Use attributes to mark your class as serializable and a data contract. Use a GUID attribute to make your class uniquely identifiable.

   To generate a GUID for the attribute, open the Tools menu in Visual Studio and choose Create GUID. In the GUID Format list, mark Registry Format. Click New GUID and then copy the GUID to your class attribute.

   The following sample code shows the attributes for the Lead class. Notice how the GuidAttribute does not include the braces for the GUID.

   ```csharp
   [Serializable, DataContract]
   [GuidAttribute("2852BB26-3BA8-4663-9613-033327D7F3C2")]
   class Lead
   {
   
   }
   ```
4. **Inherit from the BusinessObject class.**

Add Microsoft.Dynamics.Common.BusinessObject as the base class for your document class. Also, specify the class as public.

The following sample code shows how the Lead class inherits from the BusinessObject class.

```csharp
public class Lead : BusinessObject
{
}
```

5. **Add the class data members.**

Add private data members to your document class for each data field in your document. Each data member must be a data type that is used in the Dynamics GP service. For more information on data types, see Chapter 8, “Data types.”

The following sample code adds the data members for lead documents to the Lead class. While most of the data members use standard data types, a lead document also includes the following data types:

- The key data member uses the LeadKey class to uniquely identify the lead document. For more information about how to create a key, see *Defining the key* on page 198.

- The phone1, phone2, and fax data members are PhoneNumber data types from the the Microsoft.Dynamics.Common namespace.

- The leadBusinessCategory data member uses an enumeration to identify the business category for the lead. For more information about how to create an enumeration, see *Adding an enumeration* on page 208.

- The potentialRevenue data member is a MoneyAmount data type from the Microsoft.Dynamics.Common namespace.

  The potentialRevenue data member is not private. To allow updates to the Currency and DecimalDigit properties of a MoneyAmount data member, the data member cannot be private.

```csharp
private LeadKey key;
private string name;
private string salespersonID;
private string city;
private string state;
private string zip;
private string address1;
private string address2;
private PhoneNumber phone1;
private PhoneNumber phone2;
private PhoneNumber fax;
private LeadCategory leadBusinessCategory;
private string country;
private string contact;
MoneyAmount potentialRevenue;
private bool qualifiedLead;
private string leadSource;
```
private DateTime qualificationDate;
private string workflowApprovalStatus;
private string workflowPriority;
private string approvedSalespersonID;
private DateTime modifiedDate;

6. **Add a class property for each data member.**

After you define the data members of the class, you must add a class property that provides access to each data member. The property allows you to set or retrieve the value of each data member. Add the properties to your document class immediately after the data member definitions.

Use the DataMember attribute to identify each property as a data member. In addition, set the EmitDefaultValue property to false for all reference types and non-nullable enumeration types.

The following sample code adds class properties for leads:

```csharp
[DataMember(EmitDefaultValue = false)]
public LeadKey Key
{
    get{return key;}
    set{key = value;
        BaseKey = value;
    }
}

[DataMember(EmitDefaultValue = false)]
public string Name
{
    get{return name;}
    set{name = value;}
}

[DataMember(EmitDefaultValue = false)]
public string SalespersonID
{
    get{return salespersonID;}
    set{salespersonID = value;}
}

[DataMember(EmitDefaultValue = false)]
public string City
{
    get{return city;}
    set{city = value;}
}

[DataMember(EmitDefaultValue = false)]
public string State
{
    get{return state;}
    set{state = value;}
}

[DataMember(EmitDefaultValue = false)]
public string Zip
{
    get{return zip;}
    set{zip = value;}
}
```
[DataMember(EmitDefaultValue = false)]
public string Address1
{
    get { return address1; }
    set { address1 = value; }
}
[DataMember(EmitDefaultValue = false)]
public string Address2
{
    get { return address2; }
    set { address2 = value; }
}
[DataMember(EmitDefaultValue = false)]
public PhoneNumber Phone1
{
    get { return phone1; }
    set { phone1 = value; }
}
[DataMember(EmitDefaultValue = false)]
public PhoneNumber Phone2
{
    get { return phone2; }
    set { phone2 = value; }
}
[DataMember(EmitDefaultValue = false)]
public PhoneNumber Fax
{
    get { return fax; }
    set { fax = value; }
}
[DataMember(EmitDefaultValue = false)]
public LeadCategory LeadBusinessCategory
{
    get { return leadBusinessCategory; }
    set { leadBusinessCategory = value; }
}
[DataMember(EmitDefaultValue = false)]
public string Country
{
    get { return country; }
    set { country = value; }
}
[DataMember(EmitDefaultValue = false)]
public string Contact
{
    get { return contact; }
    set { contact = value; }
}
[DataMember(EmitDefaultValue = false)]
public MoneyAmount PotentialRevenue
{
    get { return potentialRevenue; }
    set { potentialRevenue = value; }
}
[DataMember(EmitDefaultValue = false)]
public bool QualifiedLead
CHAPTER 27  CREATING A SERVICE DOCUMENT TYPE

private void Initialize()
{
    get { return qualifiedLead; }
    set { qualifiedLead = value; }
}

[DataMember]
public string LeadSource
{
    get { return leadSource; }
    set { leadSource = value; }
}

[DataMember]
public DateTime QualificationDate
{
    get { return qualificationDate; }
    set { qualificationDate = value; }
}

[DataMember(EmitDefaultValue = false)]
public string WorkflowApprovalStatus
{
    get { return workflowApprovalStatus; }
    set { workflowApprovalStatus = value; }
}

[DataMember(EmitDefaultValue = false)]
public string WorkflowPriority
{
    get { return workflowPriority; }
    set { workflowPriority = value; }
}

[DataMember(EmitDefaultValue = false)]
public string ApprovedSalespersonID
{
    get { return approvedSalespersonID; }
    set { approvedSalespersonID = value; }
}

[DataMember]
public DateTime ModifiedDate
{
    get { return modifiedDate; }
    set { modifiedDate = value; }
}

7. Add a private method to initialize data fields.
Add a method to specify the initial value of specified data field. Typically, the initialize method provides values for the data fields that cannot be null. The constructor and OnDeserializing methods use this method.

The following sample code shows the Initialize method for the Lead class. Notice how the Key and PotentialRevenue properties are initialized. The Key is an object that identifies the document. For more information about how to create a key, see Defining the key on page 198.

The GetInstance method of the MoneyAmount class creates a new MoneyAmount object. In this example, the object has a value of zero, a currency of US dollars, and supports two decimal places.

private void Initialize()
8. **Add a constructor and OnDeserializing method.**
Create a class constructor and OnDeserializing method. The OnDeserializing method enables your class to be serialized by the Dynamics GP Service framework. Call the Initialize method from both of these methods.

The following sample code shows the constructor and OnDeserializing method for the Lead class. Notice how the OnDeserializing attribute is added to the OnDeserializing method.

```csharp
public Lead()
{
    Initialize();
}
[OnDeserializing]
public new void OnDeserializing(StreamingContext context)
{
    Initialize();
}
```

9. **Save the class.**
From the File menu, choose Save.

**Defining the key**

Web Services for Microsoft Dynamics GP uses key objects to uniquely identify documents. To add a class that defines the key for a new document type, complete the following steps:

1. **Add a class for the key.**
Add a new public class for your document key. Add the key class to the same file and namespace as your document type class.

   The following sample code adds the LeadKey class. Notice how the LeadKey class is added to the Microsoft.Dynamics.GP.Samples.Leads namespace.

   ```csharp
   namespace Microsoft.Dynamics.GP.Samples.Leads
   {
       public class LeadKey
       {
        
       }
   }
   ```

2. **Mark the class as serializable.**
Add the Serializable and DataContract attributes.

   ```csharp
   [Serializable, DataContract]
   public class LeadKey
   {
   ```
3. **Inherit from the ReferenceKey class.**
The key class must inherit from the Microsoft.Dynamics.GP.ReferenceKey class.

```csharp
public class LeadKey : ReferenceKey
{

}
```

4. **Add a private method to initialize data fields.**
The Reference Key class uses a list object to store the document ID value. To initialize the list, create an empty Property object and add it to the properties collection of the base class.

The following sample code shows the Initialize method for the LeadKey class. Notice how a Property object is created, initialized, and added to the ReferenceKey properties collection.

```csharp
private void Initialize()
{
    Property id = String.Empty;
    properties.Add(id);
}
```

5. **Add a constructor and OnDeserializing method.**
Create a class constructor and OnDeserializing method. The OnDeserializing method enables your class to be serialized by the Dynamics GP Service framework. Call the Initialize method from both of these methods.

The following sample code shows the constructor and OnDeserializing method for the LeadKey class. Notice how the OnDeserializing attribute is added to the OnDeserializing method.

```csharp
public LeadKey()
{
    Initialize();
}
[OnDeserializing]
public new void OnDeserializing(StreamingContext context)
{
    Initialize();
}
```

6. **Add a property for the key ID.**
Implement a class property to set or retrieve the value of the ID in the properties collection of the base class. Add the DataMember attribute to the property. Set the EmitDefaultValue property of the attribute to false.

*If your document requires a multi-segment key, add a property for each segment of the key.*

The following sample code shows the ID property of the LeadKey class. Notice how the property sets or retrieves the value of the first element in the properties collection of the base class.
[DataMember(EmitDefaultValue = false)]
public string Id {
    get { return properties[0]; }
    set { properties[0] = value; }
}

7. **Add a GetInstance method.**
   Add a static GetInstance method that returns an instance of your key class.

   ```csharp
   public static LeadKey GetInstance()
   {
       return new LeadKey();
   }
   ```

8. **Save the class.**
   From the File menu, choose Save.

### Defining the summary

Web Services for Microsoft Dynamics GP uses a summary document when a GetList method retrieves a list of documents. To add a class that defines the data members of your summary document, complete the following steps:

1. **Create a summary class.**
   From the Project menu, choose Add Class. Select Class from the List of Templates in the Add New Item window. Enter a name for your class and click Add. Visual Studio adds a class to your namespace and a file to your project.

   Your summary document class should follow the standard naming convention. Typically, the summary document name begins with document type name followed by “Summary”.

   ```csharp
   class LeadSummary
   {
   }
   ```

2. **Verify the namespace reference.**
   If your class file does not already include it, add a `using` statement for the following namespace:
   ```csharp
   using Microsoft.Dynamics.Common
   ```

3. **Mark the class as serializable.**
   Add the Serializable and DataContract attributes.
CHAPTER 27  CREATING A SERVICE DOCUMENT TYPE

[Serializable, DataContract]
public class LeadSummary
{
}

4. **Inherit from the BusinessObject class.**

The following sample code shows how the LeadSummary class inherits from the BusinessObject class:

```csharp
public class LeadSummary : BusinessObject
{
}
```

5. **Add the data members.**
Add a private data member for each data field you want to include in the summary document. Add each data member to the summary document class that you created. The following sample code defines the data members of a lead summary:

```csharp
private LeadKey key;
private string name;
private string salespersonID;
private bool qualifiedLead;
private string leadSource;
private DateTime modifiedDate;
```

6. **Add a property for each data member.**
Add a property that provides access to each data member. The property allows you to set or retrieve the value of each data member. Add the properties to the summary document class immediately after the data members.

Use the DataMember attribute to identify each property as a data member. In addition, set the EmitDefaultValue property to false for all reference types and non-nullable enumeration types.

The following sample code shows the properties of the LeadSummary class:

```csharp
[DataMember(EmitDefaultValue = false)]
public LeadKey Key
{
    get{return key;}
    set{key = value;}
}
[DataMember(EmitDefaultValue = false)]
public string Name
{
    get{return name;}
    set{name = value;}
}
[DataMember(EmitDefaultValue = false)]
public string SalespersonID
```
7. **Add a private method to initialize data fields.**

Add a method to specify the initial value of the Key data field. Typically, the initialize method provides default values for data fields that cannot be null. The constructor and OnDeserializing methods use this method.

The following sample code initializes the Key property of the LeadSummary class. Notice how an empty LeadKey is used to initialize the Key property.

```csharp
public LeadSummary()
{
    Key = new LeadKey();
    Key.Id = "";
}
```

8. **Add a constructor and OnDeserializing method.**

Create a class constructor and OnDeserializing method. The OnDeserializing method enables your class to be serialized by the Dynamics GP Service framework. Call the Initialize method from both of these methods.

The following sample code shows the constructor and OnDeserializing method for the LeadKey class. Notice how the OnDeserializing attribute is added to the OnDeserializing method.

```csharp
public LeadSummary()
{
    Initialize();
}

[OnDeserializing]
public new void OnDeserializing(StreamingContext context)
{
    Initialize();
}
9. **Add a GetInstance method.**

Add a static GetInstance method that returns an instance of your summary object.

The following sample code shows the GetInstance method of the LeadSummary class.

```csharp
public static LeadSummary GetInstance()
{
    return new LeadSummary();
}
```

10. **Save the class.**

From the File menu, choose Save.

### Defining the summary document list

Web Services for Microsoft Dynamics GP uses a GetList method to retrieve a list of summary documents that match a specified criteria. You must add a class that defines the structure of the summary document list.

Your summary document list class must follow the required naming conventions. The list class name must begin with “ArrayOf” and include the name of your document summary class.

\[ \text{ArrayOf} \text{DocumentSummaryName} \]

The following example shows the name of the summary document list class for Leads:

\[ \text{ArrayOfLeadSummary} \]

Add your list class to the same file and namespace as your summary class. Also, add `System.Collections.Generic.List<>` as the base class of your list class. When you add `List<>` as the base class, use the name of your summary document class to specify the type of the list. The following example shows the class declaration for a list of lead summary documents:

```csharp
public class ArrayOfLeadSummary : List<LeadSummary>
{
    public ArrayOfLeadSummary()
    {
    }
}
```
Adding criteria

In Web Services for Microsoft Dynamics GP, you use criteria objects to specify the summary documents that a GetList method will retrieve in a list. To support a GetList operation, add a criteria class for your document type.

To create a criteria class, complete the following steps:

1. **Create a criteria class.**
   From the Visual Studio Project menu, choose Add Class. Select Class from the List of Templates in the Add New Item window. Enter a name for your class and click Add. Visual Studio adds a class to your namespace and a file to your project.

   Your criteria class should follow the standard naming convention. Typically, the criteria name begins with document type name followed by “Criteria”.

   ```
   DocumentTypeCriteria
   ```

   The following example shows the name of the criteria class for Leads:

   ```
   LeadCriteria
   ```

   The following code sample shows the declaration of the LeadCriteria class.

   ```
   class LeadCriteria
   {
   }
   ```

2. **Verify the namespace reference.**
   If your class file does not already include it, add a using statement for the following namespace:

   ```
   Microsoft.Dynamics.Common
   ```

3. **Mark the class as serializable.**
   Add the Serializable and DataContract attributes.

   ```
   [Serializable, DataContract]
   public class LeadCriteria
   ```

4. **Inherit from the Criteria class.**
   The criteria class must inherit from the Microsoft.Dynamics.Common.Criteria class.

   The following code sample adds Criteria as the base class for LeadCriteria.

   ```
   public class LeadCriteria : Criteria
   ```
5. **Add the class data members.**

Add a private data member for each queryable data field. Typically, the criteria data members include the same data fields as the summary class. The type of each data member is a Restriction class. For more information about how to use Restrictions, see [Restriction reference](#) on page 61.

The following code sample shows the data members of the LeadCriteria class. Notice how each restriction object specifies the type of the data field.

```csharp
private LikeRestriction<string> id;
private LikeRestriction<string> name;
private LikeRestriction<string> salespersonID;
private RestrictionOfNullable<bool> qualifiedLead;
private LikeRestriction<string> leadSource;
private BetweenRestrictionOfNullable<DateTime> modifiedDate;
```

6. **Add a property for each data member.**

Add a property that provides access to each data member. The property allows you to set or retrieve the value of each data member.

Use theDataMember attribute to identify each property as a data member. In addition, set the EmitDefaultValue property to false for all reference types and non-nullable enumeration types.

The following sample code shows the properties of the LeadCriteria class:

```csharp
[DataMember(EmitDefaultValue = false)]
public LikeRestriction<string> Id
{
    get{return id;}
    set{id = value;}
}
[DataMember(EmitDefaultValue = false)]
public LikeRestriction<string> Name
{
    get{return name;}
    set{name = value;}
}
[DataMember(EmitDefaultValue = false)]
public LikeRestriction<string> SalespersonID
{
    get{return salespersonID;}
    set{salespersonID = value;}
}
[DataMember(EmitDefaultValue = false)]
public RestrictionOfNullable<bool> QualifiedLead
{
    get{return qualifiedLead;}
    set{qualifiedLead = value;}
}
[DataMember(EmitDefaultValue = false)]
public LikeRestriction<string> LeadSource
{
    get{return leadSource;}
    set{leadSource = value;}
}
```
public BetweenRestrictionOfNullable<DateTime> ModifiedDate
{
    get { return modifiedDate; }
    set { modifiedDate = value; }
}

7. **Add a constructor (if required).**
   If the data members of your criteria class require initial values, use the
   constructor to set the value of each data member. The LeadCriteria did not
   require the constructor to complete any initialization.

8. **Override the BuildArrays method.**
   Add a method that overrides the BuildArrays method of the base class. Use the
   method to create an array that specifies the name of each column in the
   database table where the criteria data is stored. Also, create two arrays that
   specify the name of the properties you added to your criteria class.

The following sample code shows the BuildArrays method of the LeadCriteria
class. Notice the following:

- The columns array identifies each column using the Physical Name from
  the Dexterity dictionary. The DEX_ROW_TS column is included because
  the Use Row Timestamp option was marked in the table definition.

- The restriction arrays specify the property names from the criteria class.
  Notice how ModifiedDate is cast to BetweenRestriction<DateTime?>. Res-
  trictions require this type for all datetime values. The “?” indicates that the
  datetime can accept null as a value.

- The order of the properties in the restriction arrays must match the order of
  the columns in the columns array. For example, the ID property must be the
  first member of the restriction arrays because the LeadID is the first mem-
  ber of the columns array.

protected override void BuildArrays()
{
    columns = new string[] { "LeadID", "LeadName", "SLPRSNIID",
                             "QualifiedLead", "LeadSource", "DEX_ROW_TS" };

    restrictions = new Restriction[] { Id, Name, SalespersonID,
                                       QualifiedLead, LeadSource,
                                       (BetweenRestriction<DateTime?>)ModifiedDate };
9. **Override the GetWhereClause method (if required).**
You might find that the values of the criteria class properties do not correspond to the values stored in the data fields of the Dynamics GP database. For example, the database may use values other than “0” and “1” for a boolean field. To adjust for these types of data inconsistencies, you can override the GetWhereClause method of the base class. When you override the GetWhereClause method, you can specify the data values the criteria object uses to query the Dynamics GP database.

For example, the QualifiedLead field is a boolean field but the values stored in Dynamics GP are “1” for false and “2” for true instead of “0” and “1”. The following sample code shows how to override the GetWhereClause method so the LeadCriteria class correctly identifies qualified leads.

```csharp
public override string GetWhereClause()
{
    StringBuilder whereClause = new StringBuilder(base.GetWhereClause());
    if (QualifiedLead != null)
    {
        if (QualifiedLead.EqualValue == true)
        {
            whereClause.Replace("QualifiedLead = 1", "QualifiedLead = 2");
            whereClause.Replace("QualifiedLead = 0", "QualifiedLead = 1");
        }
        if (QualifiedLead.EqualValue == false)
        {
            // Replace the value of QualifiedLead when the query
            // retrieves records marked as False
            whereClause.Replace("QualifiedLead = 0", "QualifiedLead = 1");
            // Replace the value of QualifiedLead when the query
            // retrieves records that are not marked as True.
            whereClause.Replace("QualifiedLead <> 1", "QualifiedLead <> 2");
        }
    }
    return whereClause.ToString();
}
```

10. **Save the class.**
    From the File menu, choose Save.
Adding an enumeration

If your document type uses an enumeration, you must add that enumeration to your project. To work with services, the enumeration must be serializable, and it must specify how to represent the enumeration in XML.

The enumeration must be in the same namespace as your document type. Typically, enumerations are added to the same file as the document type and document key classes.

The following code sample shows the LeadCategory enumeration. Notice how the enumeration includes the Serializable and DataContract attributes. In addition, each member has an XmlEnum and EnumMember attribute that specifies the value to use in XML.

```csharp
[Serializable, DataContract]
public enum LeadCategory
{
    [XmlEnum("RealEstate")]
    [EnumMember(Value = "RealEstate")]
    RealEstate = 1,
    [XmlEnum("Wholesale")]
    [EnumMember(Value = "Wholesale")]
    Wholesale,
    [XmlEnum("Retail")]
    [EnumMember(Value = "Retail")]
    Retail,
    [XmlEnum("Contractor")]
    [EnumMember(Value = "Contractor")]
    Contractor,
    [XmlEnum("Educational")]
    [EnumMember(Value = "Educational")]
    Educational,
    [XmlEnum("Media")]
    [EnumMember(Value = "Media")]
    Media,
    [XmlEnum("Software")]
    [EnumMember(Value = "Software")]
    Software,
    [XmlEnum("Restaurant")]
    [EnumMember(Value = "Restaurant")]
    Restaurant
}
```

Building the assembly

Chapter 28: Adding Security Metadata

To secure a new service, add metadata to the Dynamics Security Service that describes the security operations and tasks of the service. The following sections describe how to add security metadata to the Dynamics Security Service:

- Overview of security metadata
- Creating the project for the security helper
- Creating the security helper application
- Adding operations, tasks, and roles
- Removing roles, tasks, and operations
- Building the application

Overview of security metadata

The Dynamics Security Service controls access to service methods. To secure a new service, create a “security helper” application that adds security metadata for the new service methods. Use the “security helper” application to add the following types of metadata.

- Add a Dynamics Security Service operation for each method in your service. An operation is a specific action that the Dynamics Security Service secures.
- Add Dynamics Security Service tasks for your service. A task is a collection of related operations. Use tasks to simplify the administration of security for your service.
- Add operations and tasks to Dynamics Security Service roles. To provide administrative access to your service, assign your security operations to the Dynamics Security Service Superuser role.

To update the Dynamics Security Service, your “security helper” application requires configuration parameters from the Web Services for Microsoft Dynamics GP installation. One way to retrieve the required parameters is to import the contents of an existing Dynamics GP Service configuration file into the configuration file of your “security helper” application. The code for the sample Leads service “security helper” application demonstrates one technique for importing the required configuration information.

When you uninstall your service, you should also remove the security metadata associated with your service from the Dynamics Security Service. To remove your security metadata, add code to your “security helper” application that removes the security operations and tasks that you created. The “security helper” application for the sample Leads service shows how to remove security metadata from the Dynamics Security Service.
Creating the project for the security helper

To build a “security helper” application, use Visual Studio to create a console application project. The following steps show how to create the project.

1. **Create a Visual Studio console application project.**
   Open Visual Studio. From the File menu, point to New, and then click Project. In the New Project window, choose Visual C# from the Project types list. In Templates, select Console Application from the list of Visual Studio installed templates.

   Enter a name for your project. Review the Location and Solution Name and click OK.

2. **Add assembly references.**
   In the Project menu, choose Add References. The Add References window opens. Click the .NET tab. Select the following assemblies and click OK:
   - System.Configuration
   - System.EnterpriseServices
   - System.Web.Services

   In the Project menu, choose Add References. The Add References window opens. Click the Browse tab and navigate to the “GPWebServices” folder, typically found in the following location:
   C:\Program Files\Microsoft Dynamics\GPWebServices

   Select the following assembly, and then click OK:
   - Microsoft.Dynamics.Security.dll

3. **Add an application configuration file.**
   In the Project menu, choose Add New Item. Select Application Configuration File from the list of Templates. Leave the default name of App.config, and then click Add.

   Use Solution Explorer to verify that your project includes a file named App.config.

4. **Save the project.**
   In the File menu, choose Save.
Creating the security helper application

Open the file that contains the method named Main and complete the following steps:

1. **Add namespace references.**
   To simplify the use of several classes and methods, add using statements for the following namespaces:
   - System.Configuration
   - Microsoft.Dynamics.Security

2. **Add data members.**
   Add private data members to your class. To use service context information, add a SecurityContext object and an ApplicationKey object. Add an OperationKey, TaskKey, or RoleKey for each operation, task, or role you are adding to the security service.

   The following code example shows the objects used to add security for the sample Leads service.

   ```csharp
   private SecurityContext securityContext;
   private ApplicationKey appKey;
   private OperationKey getByKeyOpKey;
   private OperationKey getListOpKey;
   private OperationKey deleteOpKey;
   private OperationKey createOpKey;
   private OperationKey updateOpKey;
   private TaskKey viewTaskKey;
   private TaskKey manageTaskKey;
   private RoleKey roleKey;
   ```

3. **Initialize the data members.**
   To populate the security context and object keys, add a private method named InitializeComponent to the class.

   ```csharp
   private void InitializeComponent()
   {
   }
   ```

   Use the InitializeComponent method to instantiate a security context object. Set the ID property of the application key object to the GUID that identifies the Web Services for Microsoft Dynamics GP application. To identify Web Service for Microsoft Dynamics GP, always use the following GUID:

   25cc1a21-2cc4-4b13-a1c8-eea186fb688a
The following code sample shows how to populate the ApplicationKey property of the security context. Notice how the application key ID is populated with the GUID that specifies the Web Services for Microsoft Dynamics GP application.

\[
\text{securityContext} = \text{new SecurityContext();}
\]
\[
\text{appKey} = \text{new ApplicationKey();}
\]
\[
\text{appKey.Id} = "25cc1a21-2cc4-4b13-a1c8-eea186fb888a";
\]
\[
\text{securityContext.ApplicationKey} = \text{appKey};
\]

Next, create an operation key for each security operation. To populate the ID of each operation key, you must follow the required naming convention. Each security key ID must contain the GUID that identifies your document class followed by the operation type. The GUID is the same GUID you used in the GuidAttribute of your document type class. For more information about setting the GuidAttribute, see *Defining the document type* on page 193.

The following table shows the required format for each type of operation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation key ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetByKey</td>
<td>&lt;GUID&gt;GetByKey</td>
</tr>
<tr>
<td>GetList</td>
<td>&lt;GUID&gt;GetList</td>
</tr>
<tr>
<td>Create</td>
<td>&lt;GUID&gt;Create</td>
</tr>
<tr>
<td>Delete</td>
<td>&lt;GUID&gt;Delete</td>
</tr>
<tr>
<td>Update</td>
<td>&lt;GUID&gt;Update</td>
</tr>
</tbody>
</table>

The service requires the operation key IDs to contain both the GUID and operation type. If you do not include both, you will not be able to access your data.

The following code example shows how to create the security operation keys for Leads. Notice how each ID contains the following GUID value:

\[
2852BB26-3BA8-4663-9613-033327D7F3C2
\]

This is the same GUID that was used in the GuidAttribute of the Lead class. To view the GuidAttribute of the Lead class, see *Defining the document type* on page 193.

\[
\text{getByKeyOpKey} = \text{new OperationKey();}
\]
\[
\text{getByKeyOpKey.Id} = "2852BB26-3BA8-4663-9613-033327D7F3C2GetByKey";
\]
\[
\text{getListOpKey} = \text{new OperationKey();}
\]
\[
\text{getListOpKey.Id} = "2852BB26-3BA8-4663-9613-033327D7F3C2GetList";
\]
\[
\text{deleteOpKey} = \text{new OperationKey();}
\]
\[
\text{deleteOpKey.Id} = "2852BB26-3BA8-4663-9613-033327D7F3C2Delete";
\]
\[
\text{createOpKey} = \text{new OperationKey();}
\]
\[
\text{createOpKey.Id} = "2852BB26-3BA8-4663-9613-033327D7F3C2Create";
\]
\[
\text{updateOpKey} = \text{new OperationKey();}
\]
\[
\text{updateOpKey.Id} = "2852BB26-3BA8-4663-9613-033327D7F3C2Update";
\]
Next, create a task key for each task. Populate the ID property of each task with a GUID that uniquely identifies the task.

The following code example creates two task keys for leads. The GUID values were generated using the Create GUID tool from Visual Studio. Notice how the braces were removed from each GUID.

```csharp
viewTaskKey = new TaskKey();
viewTaskKey.Id = "A33871C6-8393-4040-803D-FA1EF7306136";

manageTaskKey = new TaskKey();
manageTaskKey.Id = "7029D575-7FB4-4194-A905-4B0276C5D890";
```

Finally, create a role key. Populate the ID property with a GUID that specifies a Dynamics Security Service role. To specify the Dynamics Security Service Superuser role use the following GUID:

```csharp
e18b321a-9548-48fb-b75a-dee0a618ddaa
```

To retrieve the ID of other roles, use the GetRoles method of the Dynamics Security Service. For more information about how to retrieve role information, see the Microsoft Dynamics Security Service Reference.

The following code example creates a role key. The GUID value specifies the Superuser role in the Dynamics Security Service.

```csharp
roleKey = new RoleKey();
roleKey.Id = "e18b321a-9548-48fb-b75a-dee0a618ddaa";
```

4. **Add a constructor.**

Create a constructor method for the class. Use the constructor to run the initialization method.

The following code example shows the constructor. Notice how the constructor calls the initialization method that was created earlier.

```csharp
public Program()
{
    InitializeComponent();
}
```
5. Implement command line parameters.
In the Main method, add support for command line parameters. Use the parameter "/load" to add security metadata. Use the parameter "/remove" to delete security metadata. The following code example shows how to implement the command line parameters.

```csharp
static void Main(string[] args)
{
    if ((args.Length == 1) && ((args[0].ToLowerInvariant() == "/load") ||
    (args[0].ToLowerInvariant() == "/remove")))
    {
    }
    else
    {
        Console.WriteLine("InstallLeadSecurityMetadata.exe requires
     a single parameter");
        Console.WriteLine("Valid parameters are: /load or /remove");
        Console.WriteLine();
    }
    Console.WriteLine("Press any key to quit.");
    Console.ReadKey();
}
```

6. Create an instance of the class.
Add a statement to the Main method that instantiates the class.

```csharp
Program addLeads = new Program();
```

7. Retrieve configuration information.
In your app.config file, add a reference to the WSInstallAppSettings.config file. The file reference provides configuration settings for your application that enable you helper application to update the Dynamics Security Service. The WSInstallAppSetting.config file is typically found in the following folder:

c:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs

The following code example shows how to add a file reference to the WSInstallAppSetting.config file. Notice how the filepath specifies the expected location of that configuration file.

```xml
<configuration>
    <appSettings file="ServiceConfigs\WSInstallAppSettings.config"/>
</configuration>
```
Adding operations, tasks, and roles

Add a method that creates operations and tasks, and then adds these operations and tasks to roles in the service.

1. Add a method.
   Add a private method named Load to the class.

   ```csharp
   private void Load()
   {
   }
   ```

2. Add an operation for each method.
   Create a Microsoft.Dynamics.Security.Operation object for each of your service methods. Populate the Key, Name, and Description property of each operation. Use a Microsoft.Dynamics.Security.OperationService object to add or update the Dynamics Security Service with your operation metadata.

   The following code example adds operation metadata to the Dynamics Security Service for the sample Leads service. Notice how the Key property of each operation uses the key that was created during initialization.

   ```csharp
   // Instantiate an OperationService object.
   OperationService opService = OperationService.GetInstance();

   // Create a array for the security operation objects.
   Microsoft.Dynamics.Security.Operation[] leadOps = new Operation[5];

   // Populate a security object that enables a GetByKey operation for leads.
   leadOps[0].Key = getByKeyOpKey;
   leadOps[0].Name = "View Leads";
   leadOps[0].Description = "Privilege to view a lead";

   // Populate a security object that enables a GetList operation for leads.
   leadOps[1].Key = getListOpKey;
   leadOps[1].Name = "Query Leads";
   leadOps[1].Description = "Privilege to view leads";

   // Populate a security object that enables a Delete operation for leads.
   leadOps[2].Key = deleteOpKey;
   leadOps[2].Name = "Delete Leads";
   leadOps[2].Description = "Privilege to delete a lead";
   ```
// Populate a security object that enables a Create operation for leads.
leadOps[3].Key = createOpKey;
leadOps[3].Name = "Create Leads";
leadOps[3].Description = "Privilege to create a lead";

// Populate a security object that enables an Update operation for leads.
leadOps[4].Key = updateOpKey;
leadOps[4].Name = "Modify Leads";
leadOps[4].Description = "Privilege to modify a lead";

foreach (Operation op in leadOps)
{
    try
    {
        // If the operation exists, update the existing operation.
        opService.UpdateOperation(securityContext, op);
    }
    catch (NonExistentSecurityObjectException)
    {
        // If the operation does not exist, add the operation to the
        // security service.
        opService.CreateOperation(securityContext, op);
    }
}

3. Add tasks that group the operations.
Tasks help organize and group the related operations. Typically tasks are created for “Manage” and “View”.

- “Manage” tasks contain operations like GetByKey, GetList, Create, Delete, and Update.
- “View” tasks contain the GetByKey and GetList operations.

Create a Microsoft.Dynamics.Security.Task object for each task. Populate the Key, Name, and Description property of the task. To specify the operations for the task, add your operation keys to the Operation collection of the task. Use a Microsoft.Dynamics.Security.TaskService object to add or update the Dynamics GP Security Service with your task metadata.
The following code example adds two tasks to the metadata of the Dynamics Security Service for the sample Leads service. Notice how the operation keys for the operations that were created earlier populate the operation collection of each task.

```csharp
// Instantiate the TaskService object.
TaskService taskService = TaskService.GetInstance();

// Create a view task object for leads.
Task viewTask = Task.GetInstance();
viewTask.Key = viewTaskKey;
viewTask.Name = "View Leads";
viewTask.Description = "View Leads";
viewTask.Operations.Add(getByKeyOpKey);
viewTask.Operations.Add(getListOpKey);
try
{
    // If the task already exists, update the existing task object.
    taskService.UpdateTask(securityContext, viewTask);
}
catch (NonExistentSecurityObjectException)
{
    // If the task does not exist, add the task object.
    taskService.CreateTask(securityContext, viewTask);
}

// Create a manage task object for leads.
Task manageTask = Task.GetInstance();
manageTask.Key = manageTaskKey;
manageTask.Name = "Manage Leads";
manageTask.Description = "Manage Leads";
manageTask.Operations.Add(getByKeyOpKey);
manageTask.Operations.Add(getListOpKey);
manageTask.Operations.Add(deleteOpKey);
manageTask.Operations.Add(createOpKey);
manageTask.Operations.Add(updateOpKey);
try
{
    // If the task already exists, update the existing task object.
    taskService.UpdateTask(securityContext, manageTask);
}
catch (NonExistentSecurityObjectException)
{
    // If the task does not exist, add the task object.
    taskService.CreateTask(securityContext, manageTask);
}
```
4. **Add new roles (optional).**
To provide additional flexibility in the security administration of your service, you might want to add one or more new security roles to the Dynamics Security Service. The sample Leads service does not include code to add a new role. To add role metadata to the Dynamics Security Service, follow the pattern used to create task and operation metadata. To add role metadata to the Dynamics Security Service, you use the following classes from the Microsoft.Dynamics.Security namespace.

- To begin, use the RoleKey class to create a new role key. To uniquely identify your role, use a GUID to populate the ID property of the role key. If you use the Visual Studio Create GUID application to generate the GUID, be sure to remove the braces from the GUID.

- Next, create a new role object. To instantiate the role object, use the GetInstance method of the Role class. Populate the Name and Description properties of the role. Use the role key you created earlier to populate the Key property of the role.

- Finally, use a RoleService object to add the metadata for the role to the Dynamics Security Service. To instantiate the role service, use the GetInstance method of the RoleService class. To add the role metadata to the Dynamics Security Service, use the CreateRole method of the role service. The CreateRole method requires a security context object parameter and a role object parameter.

5. **Add the operations and tasks to the roles.**
To secure access to your service, add your security operations and tasks to roles. To add an operation or task to a role, use the GetRole method of the Microsoft.Dynamics.Security.RoleService object to retrieve the role. Use the role keys you created earlier to specify the role. After you retrieve the role, you can add security operations and tasks to the role.

- To associate a security operation with a role, add the operation key to the Operations collection of the role. To allow administrative access to your service, you must add all of the security operations you created to the Dynamics Security Service Superuser role.

- To associate a security task with a role, add the task key to the Tasks collection of the role. To allow any role other than the Superuser role to access your service, add the appropriate security task to that role.

To save your changes to the role metadata, use the UpdateRole method of the Microsoft.Dynamics.Security.RoleService object.
The following code example adds operation metadata to the Superuser role of the Dynamics Security Service. Notice how the operation keys for the operations that were created earlier populate the Operations collection of the role.

```csharp
// Instantiate the RoleService object
RoleService roleService = RoleService.GetInstance();

// Retrieve a role object that represents the superuser role
Role role = roleService.GetRole(securityContext, roleKey);

// Add the operation object to the role if that object is not already
// assigned to that role.
bool newOpAdded = false;

foreach (Operation newOp in leadOps)
{
    {
        role.Operations.Add(newOp.Key);
        newOpAdded = true;
    }
}

// Only update the security service when an operation is
// added to the role object.
if (newOpAdded == true)
{
    roleService.UpdateRole(securityContext, role);
}
```

6. **Add the method to Main.**

In the Main method, add statements that run the Load method when the command line parameter is “/load”.

The following code example shows how the load method is called.

```csharp
if (args[0].ToLowerInvariant() == "/load")
{
    addLeads.Load();
    Console.WriteLine("The Lead operations have been added
to Dynamics GP security.");
    Console.WriteLine("\n");
}
```
Removing roles, tasks, and operations

To allow your service to be uninstalled, add a method to the “security helper” application that removes your security metadata from the Dynamics Security Service. To prevent your “security helper” application from creating inconsistencies in the security metadata, remove your security metadata in the following sequence:

- Remove operations and tasks from roles.
- Remove roles you added for your service.
- Remove tasks you added for your operations.
- Remove operations you added for your service.

The following steps describe how to use the “security helper” application to remove security metadata from the Dynamics Security Service.

1. **Add a method.**
   Add a private method named Remove to the class.

   ```csharp
   private void Remove()
   {
   }
   ```

2. **Specify the operation keys in the method.**
   To simplify removal of operation metadata, create an array that contains the your security operation keys and task keys.

   The following code example creates an array of the operations used by the sample Leads service. Notice how the array includes the operation keys that were created by the initialization method.

   ```csharp
   OperationKey[] opKeys = new OperationKey[5];
   opKeys[0] = getByKeyOpKey;
   opKeys[1] = getListOpKey;
   opKeys[2] = deleteOpKey;
   opKeys[3] = createOpKey;
   ```

3. **Remove tasks and operations that were added to the role.**
   Retrieve the roles where you added security operations or tasks. Use the Microsoft.Dynamics.Security.RoleService object to retrieve the role. Use a role key object to specify the role.

   - For the Superuser role, remove the operation keys from the Operations collection of the role.

   - For other roles, remove the task keys from the Tasks collection of the role.

The following code example removes operation metadata from the Superuser role of the the Dynamics Security Service. Notice how the array of operation keys specify the operations to remove.

```csharp
// Instantiate the RoleService object
RoleService roleService = RoleService.GetInstance();

Role role = roleService.GetRole(securityContext, roleKey);

// Remove the security operation from the role if it has been assigned to
// the Superuser role.
bool opRemoved = false;

foreach (OperationKey key in opKeys)
{
    if (role.Operations.BinarySearch(key) >= 0)
    {
        role.Operations.Remove(key);
        opRemoved = true;
    }
}

// Only update the security service when an operation is
// removed from the role object.
if (opRemoved == true)
{
    roleService.UpdateRole(securityContext, role);
}
```

4. **Remove roles (optional).**
   To remove a role, use the DeleteRole method of the Microsoft.Dynamics.Security.RoleService. Use a RoleKey object to specify the role to remove. The “security helper” application for the sample Leads service does not delete any roles, so no code sample is provided for this step.

5. **Remove tasks.**
   Use the DeleteTask method of the Microsoft.Dynamics.Security.TaskService to remove any tasks that you added. Use a TaskKey object to specify the task to remove.

   The following code example shows how to remove task metadata from the Dynamics Security Service.

   ```csharp
   // Instantiate the TaskService object.
   TaskService taskService = TaskService.GetInstance();
   try
   {
       // Delete the specified tasks
       taskService.DeleteTask(securityContext, viewTaskKey);
       taskService.DeleteTask(securityContext, manageTaskKey);
   }
   catch (NonExistentSecurityObjectException)
   {
       // If the task does not exist, no action is needed.
       // Trap the error and continue.
   }
   ```
6. **Remove operations.**

   Use DeleteOperation of the Microsoft.Dynamics.Security.OperationService to remove the security operations that you added. Use an OperationKey object to specify the operation.

   The following code example shows how to remove operation metadata from the Dynamics Security Service.

   ```csharp
   // Instantiate an OperationService object.
   OperationService opService = OperationService.GetInstance();
   foreach (OperationKey key in opKeys)
   {
     try
     {
       opService.DeleteOperation(securityContext, key);
     }
     catch (NonExistentSecurityObjectException)
     {
       // If the operation does not exist, no action is needed.
       // Trap the error and continue.
     }
   }
   
   7. **Add the method to Main.**

   In the Main method, add statements that run the Remove method when the command line parameter is "/remove".

   The following code example shows how the remove method is called.

   ```csharp
   if (args[0].ToLowerInvariant() == "/remove")
   {
     addLeads.Remove();
     Console.WriteLine("The Lead operations have been removed from Dynamics GP security.");
     Console.WriteLine("\n");
   }
   
   **Building the application**

   Open the Visual Studio Build menu, and choose Build. Visual Studio builds your console application.
Chapter 29: Adding Policy Metadata

To provide policy control over how your service creates, updates, or deletes a business document, add custom policy metadata to the Dynamics GP Service. The following sections describe how to add policy metadata.

- **Overview of policy metadata**
- **Creating the project for the policy helper**
- **Creating the policy helper application**
- **Creating behaviors**
- **Creating policies**
- **Removing policies and behaviors**
- **Completing the policy helper application**
- **Adding a policy resource assembly**
- **Building the application and resource assembly**

Overview of policy metadata

Web Services for Microsoft Dynamics GP requires a policy object for each service method that creates, updates, or deletes a business document. When adding a service for a new type of document, you must add policy metadata for the create, update, and delete operations of the service. For more information about how policies are used, see Chapter 15, “Policy.” Policy metadata includes the following components:

**Policies** A policy is a collection of configured behaviors that can control important characteristics of the create, update, and delete operations of a service.

**Behaviors** A behavior is a single, configurable characteristic of a policy. A policy can include one or more behaviors. A policy is not required to have behaviors.

**Behavior options** A behavior option represents the configuration options that are available for a behavior. Each behavior option represents a configuration choice the service administrator can select.

**Parameters** A parameter stores the configuration information entered by the service administrator. Behavior options use parameters to enable customization of the behavior.

To add policy metadata, you need to create a “policy helper” application. The “policy helper” application adds policy metadata to the Dynamics GP Service. You must run the “policy helper” application when you install your service. When your “policy helper” application installs policy metadata, you must first add the metadata for the behaviors and then add the metadata for the policies.

You also use the “policy helper” application to remove policy metadata when you uninstall your service. To remove policy metadata, use the “policy helper” application to first remove the metadata for the policies and then remove the metadata for the behaviors.
The service administrator uses the Dynamics Security Console to view and configure policies for services. When you add policy, you must create a .NET assembly that contains the text and messages that the Dynamics Security Console displays for your policy.

The “policy helper” application you create will require the same configuration parameters that were used to install policy metadata for the Dynamics GP Service. To supply the required parameters, create an application configuration file that your “policy helper” application can use to retrieve configuration information.

This section of the documentation describes how to add policy metadata for a service. To implement the configuration options that a policy supports, you will need to add XSLT for each policy behavior. For more information about how to use XSLT to implement a policy, see Creating XSLT for the Create operation on page 271.

Creating the project for the policy helper

To create a “policy helper” application, use Visual Studio to create a console application project. The following steps show how to create the project:

1. **Create a Visual Studio console application project.**
   
   Open Visual Studio. In the File menu, point to New, and then click Project. In the New Project window, choose Visual C# from the Project types list. In Templates, select Console Application from the list of Visual Studio installed templates.

   Enter a name for your project. Typically, the name identifies the document type and the application that installs policy metadata. For example, the “policy helper” application for the sample Leads service uses the following name:

   InstallLeadPolicyMetadata

   Review the Location and Solution Name, and click OK.

2. **Add assembly references.**
   
   Open the Project menu, and choose Add References. The Add References window opens. Click the .NET tab. Select the following assembly, and then click OK:

   - System.Configuration

   Open the Project menu, and choose Add References. The Add References window opens. Click the Browse tab and navigate to the Dynamics GP Services “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

   Select the following assemblies, and then click OK:

   - Microsoft.Dynamics.GP.InstallData.dll
   - Microsoft.Dynamics.Common.dll
   - Microsoft.Dynamics.Common.Types.dll
   - Microsoft.Dynamics.GP.DataAccess.dll
3. **Add an application configuration file.**
   In the Project menu, choose Add New Item. Select Application Configuration File from the list of Templates. Leave the default name of App.config, and then click Add.

   Use Solution Explorer to verify that your project includes a file named App.config.

4. **Save the project.**
   From the File menu, choose Save.

**Creating the policy helper application**

Open the file that contains the method named Main and complete the following steps:

1. **Add namespace references.**
   To simplify the use of several classes and methods, add using statements for the following namespaces:
   
   - System.Configuration
   - Microsoft.Dynamics.Common

2. **Add data members.**
   Add private data members to your class. To use service context information, add a Context object. Add a GUID for each policy and behavior. Also add a GUID for each parameter required by a behavior. Add a dictionary object that stores the collection of behaviors for the policies.

   The following code example shows the objects used to add policies for the sample Leads service. Notice how a GUID is used to uniquely identify the create, delete, and update policies. The create policy includes one behavior so a GUID is used to uniquely identify the behavior. Also, the create behavior includes one behavior option parameter that stores the configuration information the service administrator enters. A GUID is used to uniquely identify the behavior option parameter.

   ```csharp
   private Context context;
   private Guid deleteLeadPolicyId;
   private Guid createLeadPolicyId;
   private Guid updateLeadPolicyId;
   private Guid createQualifiedLeadBehaviorId;
   private int QualifiedLeadParameterId;
   private Dictionary<Guid, Behavior> behaviors;
   ```
3. **Initialize the data members.**

To populate the private data members, add a private method to the class. Use a parameter to specify the company that requires the new policy. The company ID will be obtained from the command line parameters of the “policy helper” application.

```csharp
private void InitializeComponent(int cmpIdNumber)
{
}
```

Use the ` InitializeComponent` method to instantiate a context object.

```csharp
// Instantiate a context object
context = new Context();

// Specify the company to use
// Set the ID using the company ID supplied by the command line parameter
CompanyKey companyKey = new CompanyKey();
companyKey.Id = cmpIdNumber;

// Set up the context object
context.OrganizationKey = (OrganizationKey)companyKey;
context.CultureName = "en-US";
```

Create the keys that uniquely identify each policy. The following code sample shows the policies for the sample Leads service. The GUIDs were generated using the Create GUID utility accessed from the Visual Studio Tools menu. Notice how the braces were removed from each GUID.

*The GUID value you assign to the key uniquely identifies each policy or behavior. When you work with policies and behaviors, you must use the GUID value to specify an individual policy or behavior.*

```csharp
createLeadPolicyId = new Guid("D3E303CC-CCB5-4945-9DD9-4DA409264518");
updateLeadPolicyId = new Guid("443E9AFD-50E0-44e2-A2F1-8FFFC2DCB596");
deleteLeadPolicyId = new Guid("3B7922CE-A7C7-41fc-879E-C749A246B5D1");
```

Create keys that uniquely identify each behavior. The following code sample creates a behavior key for the create policy of the sample Leads service.

```csharp
createQualifiedLeadBehaviorId =
    new Guid("10099A58-64BF-4fa3-B35F-ED43D1B6FF9C");
```

Specify a value that identifies each behavior option parameter. Parameter are typically numbered sequentially starting with “1”. The following code example specifies the ID of the behavior option parameter of the create policy for the sample Leads service.

```csharp
QualifiedLeadParameterId = 1;
```

Create a dictionary that will be used to store behaviors. The following code example uses a dictionary collection to store behaviors. The dictionary allows you to use the ID of a behavior to retrieve that behavior.

```csharp
behaviors = new Dictionary<Guid, Behavior>();
```
4. Implement command line parameters.
In the Main method, add support for command line parameters. Use "/load" to add policy metadata. Use "/remove" to delete your policy metadata. Use an integer to specify the ID of the company where the policies need to be installed.

The following code example shows one technique for obtaining command line information. Notice how the number of parameters are verified and how the first parameter is checked to be sure it contains "/load" or "/remove".

```csharp
static void Main(string[] args)
{
    if (args.Length == 2)
    {
        if ((args[0].ToLowerInvariant() == "/load") ||
            (args[0].ToLowerInvariant() == "/remove"))
        {

        }
        else
        {
            Console.WriteLine("InstallLeadPolicyMetadata.exe requires
            the first parameter to be: /load or /remove");
            Console.WriteLine("Valid parameters are: /load or /remove");
            Console.WriteLine();
        }
    }
    else
    {
        Console.WriteLine("InstallLeadPolicyMetadata.exe requires
        two parameters");
        Console.WriteLine("Valid parameters are: /load or /remove and
        a company number");
        Console.WriteLine();
    }

    Console.WriteLine("Press any key to quit.");
    Console.ReadKey();
}
```

5. Create an instance of the class.
Add a statement to the Main method that instantiates the class.

```csharp
Program managePolicies = new Program();
```
6. **Get the company ID and initialize the class.**
Add statements to the Main method that retrieve the company ID from the command line parameters and initialize the class.

The following code example shows how to retrieve the company ID from the command line parameters. Notice how the company ID is converted to an integer and used as an input parameter for the InitializeComponent method.

```csharp
int companyId = Int32.Parse(args[1]);
managePolicies.InitializeComponent(companyId);
```

7. **Retrieve configuration information.**
In your app.config file, add caching configuration and application setting information. To specify configuration settings, add references to the WSCachingConfiguration.config and WSInstallAppSettings.config files typically found in the following folder:

```
c:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs
```

The following code example shows how to add caching configuration informations and file references to the WSCachingConfiguration.config and WSInstallAppSettings.config files. Notice how the filepath specifies the expected location of the configuration files.

```xml
<configuration>
  <configSections>
    <section name="cachingConfiguration"
        Configuration.CacheManagerSettings,
        Microsoft.Dynamics.EnterpriseLibrary.Caching,
        Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"/>
  </configSections>
  <cachingConfiguration
    configSource="ServiceConfigs\WSCachingConfiguration.config"/>
  <appSettings
    file="ServiceConfigs\WSInstallAppSettings.config"/>
</configuration>
```
Creating behaviors

Add a method that creates the behavior, behavior option, and behavior parameter metadata for your policies. The following is the behavior information that the Dynamics Security Service console displays for the Create Lead Policy of the sample Leads service.

![Create Lead Policy](image)

To add a behavior to a policy, complete the following steps:

1. **Define behavior options.**
   Create an enumeration that specifies a value for each behavior option that the behavior supports. The following code example shows the behavior options enumeration for the CreateLead policy. Notice how the enumeration specifies the value of each data member.

   ```csharp
   private enum QualifiedLeadBehaviorOption : int
   {
       Qualified = 1,
       NotQualified = 2
   }
   
   2. **Add a method to create behaviors.**
   Add a method to create behavior metadata and add it to the Dynamics GP Service.

   ```csharp
   private void LoadBehaviors()
   {
   
   }
3. **Create behaviors.**

In the `LoadBehavior` method, add the behaviors, behavior options, and behavior parameters for your policies. To begin, create a behavior object for each behavior. To create a behavior, supply the behaviors constructor with the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>behaviorId</code></td>
<td>A GUID that uniquely identifies the behavior. This is the GUID that you created earlier.</td>
</tr>
<tr>
<td><code>resXAssemblyName</code></td>
<td>A string that specifies the name of the resource assembly that contains the text resources for this behavior.</td>
</tr>
<tr>
<td><code>descriptionResXId</code></td>
<td>A string that specifies the ID of the text resource that provides the description of the behavior.</td>
</tr>
<tr>
<td><code>nameResXId</code></td>
<td>A string that specifies the ID of the text resource that provides the name of the behavior.</td>
</tr>
</tbody>
</table>

For more information about how to create a resource assembly that contains the name and description information that display in the Dynamics Security Console for a behavior, see *Adding a policy resource assembly* on page 237. The resource assembly also provides name and description information for the behavior options, behavior parameters, and policies.

The following code example creates a behavior object for the Create Lead Policy. Notice how the first parameter uses the GUID that was created earlier to uniquely identify this behavior.

```
Behavior createQualifiedLeadBehavior =
    new Behavior(createQualifiedLeadBehaviorId,
                 "Sample.PolicyResources",
                 "CreateQualifiedLeadBehaviorDescription",
                 "CreateQualifiedLeadBehaviorName");
```

4. **Create behavior options.**

Create behavior options for each behavior object. To create a behavior option, supply the behavior option constructor with the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>behaviorId</code></td>
<td>A GUID that identifies the behavior that this option is part of.</td>
</tr>
<tr>
<td><code>behaviorOptionId</code></td>
<td>An integer that identifies this option for the behavior.</td>
</tr>
<tr>
<td><code>resXAssemblyName</code></td>
<td>A string that specifies the name of the resource assembly that contains the text resources for this behavior option.</td>
</tr>
<tr>
<td><code>descriptionResXId</code></td>
<td>A string that specifies the ID of the text resource that provides the description for this behavior option.</td>
</tr>
<tr>
<td><code>nameResXId</code></td>
<td>A string that specifies the ID of the text resource that provides the name for this behavior option.</td>
</tr>
</tbody>
</table>
The following code example creates two behavior options. Notice how the second parameter uses the enumeration that was created earlier to uniquely identify each option for the behavior.

```java
BehaviorOption notQualified =
    new BehaviorOption(createQualifiedLeadBehaviorId,
                       (short)QualifiedLeadBehaviorOption.NotQualified,
                       "Sample.PolicyResources",
                       "NotQualifiedLeadOptionDescription",
                       "NotQualifiedLeadOptionName");

BehaviorOption qualified =
    new BehaviorOption(createQualifiedLeadBehaviorId,
                       (short)QualifiedLeadBehaviorOption.Qualified,
                       "Sample.PolicyResources",
                       "QualifiedLeadOptionDescription",
                       "QualifiedLeadOptionName");
```

5. **Create behavior parameters.**

If a behavior option requires input from the user, create a behavior parameter to store what the user enters.

To create a behavior parameter, supply the behavior parameter constructor with the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
The following code example creates a parameter for the Qualified behavior option. Notice how the second parameter specifies the behavior option and the third parameter specifies the ID of the behavior parameter. Also, notice how the value of the parameter is set to default to an empty string.

The Value property of a behavior property can only accept values that are a string data type.

```csharp
Parameter qualifiedParam = new Parameter(createQualifiedLeadBehaviorId,
                                          (short)QualifiedLeadBehaviorOption.Qualified,
                                          (short)QualifiedLeadParameterId,
                                          "Sample.PolicyResources",
                                          "QualifiedLeadParameterDescription",
                                          "QualifiedLeadParameterName");
qualifiedParam.Value = string.Empty;
```

6. **Add the parameters and options to the behaviors.**
   To begin, add the parameter to the Parameters collection of the behavior option.

   ```csharp
   qualified.Parameters.Add(qualifiedParam);
   ```
   Then add the behavior options to the Options collection of the behavior.

   ```csharp
   createQualifiedLeadBehavior.Options.Add(notQualified);
   createQualifiedLeadBehavior.Options.Add(qualified);
   ```

7. **Configure the behaviors.**
   Set the default values of the behaviors and specify whether the behavior is 'internal' or 'external'.

   ```csharp
   createQualifiedLeadBehavior.SelectedOption =
       createQualifiedLeadBehavior.GetBehaviorOption(
                                                   (short)QualifiedLeadBehaviorOption.NotQualified);
   createQualifiedLeadBehavior.Internal = true;
   ```

8. **Add behavior metadata to the Dynamics GP Service.**
   Use the ServiceFactory object to instantiate a PolicyBusinessService object.

   ```csharp
   PolicyBusinessService policyBusinessService =
       (PolicyBusinessService)ServiceFactory.GetServiceInstance(
                                                   CommonConstants.PolicyBusinessService);
   ```
   Use the PolicyBusinessService object to add the behavior to the Dynamics GP Service metadata.

   ```csharp
   policyBusinessService.CreateBehavior(context,
                                          createQualifiedLeadBehavior);
   ```
9. **Add behaviors to a collection.**
Add your behaviors to the dictionary collection that you created earlier. You can use the dictionary collection object to retrieve individual behaviors using the behavior Id. You will use the dictionary collection to add behaviors to a policy.

The following code example adds the behavior to the behaviors collection. Notice how dictionary collection object uses the behavior ID to identify the behavior object that is being added to the collection.

```csharp
behaviors.Add(createQualifiedLeadBehaviorId, createQualifiedLeadBehavior);
```

### Creating policies

To add policy metadata for a new service, complete the following steps:

1. **Add a method to create policies.**
Add a method that will add policy metadata for your methods.

```csharp
private void LoadPolicies()
{
}
```

2. **Create policies.**
In the LoadPolicies method, create a policy object for each of your service operations. To create a policy, supply the policy constructor with the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>policyId</td>
<td>A GUID that uniquely identifies the policy.</td>
</tr>
<tr>
<td>resXAssemblyName</td>
<td>A string that specifies the name of the resource assembly that contains the text resources for this policy.</td>
</tr>
<tr>
<td>nameResXId</td>
<td>A string that specifies the ID of the text resource that provides the name of the policy.</td>
</tr>
<tr>
<td>rootBusinessObjectName</td>
<td>A string that specifies the ID of the text resource that provides the name of the document type.</td>
</tr>
<tr>
<td>ResXId</td>
<td></td>
</tr>
</tbody>
</table>

The following code example creates the policy objects for the create, update, and delete operations of the sample Leads service. Notice how the first parameter uses the GUID value that was created earlier to identify the policy.

```csharp
Policy createLeadPolicy =
    new Policy(createLeadPolicyId,
        "Sample.PolicyResources",
        "CreateLeadPolicyName",
        "LeadRootObjectName");
Policy updateLeadPolicy =
    new Policy(updateLeadPolicyId,
        "Sample.PolicyResources",
        "UpdateLeadPolicyName",
        "LeadRootObjectName");
```
Policy deleteLeadPolicy =
    new Policy(deleteLeadPolicyId,
                "Sample.PolicyResources",
                "DeleteLeadPolicyName",
                "LeadRootObjectName");

3. Add behaviors to the policy.
   For policies with behaviors, add behavior objects to the policy. Use the behavior
   key to retrieve the specified behavior object from the collection you created to
   store behaviors.

   The following code example shows how to use the behaviors dictionary
   collection object to retrieve a behavior from the behavior collection and add it to
   the createLeadPolicy object.

   createLeadPolicy.Behaviors.Add(behaviors[createQualifiedLeadBehaviorId]);

4. Add policies to the Dynamics GP Service metadata.
   Use the ServiceFactory to instantiate a PolicyBusinessService object.

   PolicyBusinessService policyBusinessService =
       (PolicyBusinessService)ServiceFactory.GetServiceInstance(
                   CommonConstants.PolicyBusinessService);

   Use the PolicyBusinessService object to add your policy metadata to the
   Dynamics GP Service metadata.

   The following code example create policies for the sample Leads service. Notice
   how the CreatePolicy method requires you to supply a context and policy
   object.

   policyBusinessService.CreatePolicy(context, createLeadPolicy);
   policyBusinessService.CreatePolicy(context, updateLeadPolicy);
   policyBusinessService.CreatePolicy(context, deleteLeadPolicy);

Removing policies and behaviors

   To allow your service to be uninstalled, add methods that remove your policy
   metadata from the Dynamics GP Service metadata. When you remove policy
   metadata for a service, you must first remove the policies and then remove the
   behaviors.

1. Add methods that create keys.
   Implement a method that takes a GUID and returns a policy key. Use the static
   GetInstance method of the PolicyKey class to create a key. Use the GUID
   parameter to populate the ID property of the key.

   private PolicyKey CreatePolicyKey(Guid guid)
   {
       PolicyKey policyKey = PolicyKey.GetInstance();
       policyKey.Id = guid;

       return policyKey;
   }
Implement a method that takes a GUID and returns a behavior key. Use the static GetInstance method of the BehaviorKey class to create a key. Use the GUID parameter to populate the ID property of the key.

```csharp
private BehaviorKey CreateBehaviorKey(Guid guid)
{
    BehaviorKey behaviorKey = BehaviorKey.GetInstance();
    behaviorKey.Id = guid;

    return behaviorKey;
}
```

2. **Add a method to remove policies.**

In the method, use the ServiceFactory to instantiate a PolicyDataService object. Then, use the DeletePolicy method of the PolicyDataService to remove policy metadata.

The following code example removes the policy metadata for the sample Leads service operations. Notice how the key values that were defined during initialization are used to specify the policies.

```csharp
private void RemovePolicies()
{
    PolicyDataService policyDataService =
        (PolicyDataService)ServiceFactory.GetServiceInstance(
            CommonConstants.PolicyDataService);

    policyDataService.DeletePolicy(context,
        CreatePolicyKey(deleteLeadPolicyId));
    policyDataService.DeletePolicy(context,
        CreatePolicyKey(createLeadPolicyId));
    policyDataService.DeletePolicy(context,
        CreatePolicyKey(updateLeadPolicyId));
}
```

3. **Add a method to remove behaviors.**

In the method, use the ServiceFactory to instantiate a PolicyDataService object. Use the DeleteBehavior method of the PolicyDataService to remove behavior metadata.

The following code example removes the behavior metadata for the sample Leads service. Notice how the key values that were defined during initialization are used to specify the behavior.

```csharp
private void RemoveBehaviors()
{
    PolicyDataService policyDataService =
        (PolicyDataService)ServiceFactory.GetServiceInstance(
            CommonConstants.PolicyDataService);

    policyDataService.DeleteBehavior(context,
        CreateBehaviorKey(createQualifiedLeadBehaviorId));
}
```
Completing the policy helper application

To make the “policy helper” application load and remove policy metadata, complete the following procedure:

1. **Add a method that loads behavior and policy metadata.**
   Add a method that controls the loading of behavior and policy metadata. This method should call the LoadBehaviors and LoadPolicies methods that you created earlier.

   ```csharp
   You must always add the behavior metadata before you add the policy metadata.
   
   private void Load()
   {
       LoadBehaviors();
       LoadPolicies();
   }
   ```

2. **Add a method that removes policy and behavior metadata.**
   Add a method that controls the removal of policy and behavior metadata. This method calls the RemovePolicies and RemoveBehaviors methods that you created earlier.

   ```csharp
   You must always remove your policy metadata before you remove your behavior metadata.
   
   private void Remove()
   {
       RemovePolicies();
       RemoveBehaviors();
   }
   ```

3. **Add the Load and Remove methods to the Main method.**
   In the Main method of your application, associate the “/load” and “/remove” command line parameters with the Load and Remove methods you created earlier.

   The following code example shows how to use the command line parameters to add or remove policy metadata. The example also shows how to display an error for an invalid command line parameter.

   ```csharp
   if (args[0].ToLowerInvariant() == "/load")
   {
       managePolicies.Load();
       Console.WriteLine("Lead policies have been added to
           the Dynamics GP Service metadata.");
   }
   ```
else if (args[0].ToLowerInvariant() == "/remove")
{
    managePolicies.Remove();
    Console.WriteLine("Lead policies have been removed from
    the Dynamics GP Service metadata.");
    Console.WriteLine();
}
else
{
    Console.WriteLine("Error: Unknown parameter encountered.");
    Console.WriteLine("Error: The parameter was {0}",
        args[0].ToLowerInvariant());
    Console.WriteLine("Valid parameters are: /load or /remove");
    Console.WriteLine();
}

4. **Save the project.**
   From the file menu, choose Save All.

**Adding a policy resource assembly**

You must create an assembly that contains the text that the Dynamics Security Service console displays for your policies and behaviors. To add a resource assembly to the current Visual Studio solution, complete the following steps:

1. **Create a Visual Studio class library project.**
   In the File menu, point to Add, and then click New Project. In the Add New Project window, choose Visual C# from the Project types list. In Templates, select Class Library from the list of Visual Studio installed templates.
   
   Enter a name for your project. Review the Location and Solution Name and click OK.

   *Use the same assembly name as the one you used for the resXAssemblyName parameter for the policy, behavior, behavior option, and behavior parameter constructors that you added to your “policy helper” application.*

2. **Add a resource file to the project.**
   In the Project menu, choose Add New Item. In the Add New Item window, choose Resources File from the list of Templates. Enter the same name that you entered for the project and click Add.
3. Add names and values to the resource file.

From the Visual Studio Solution Explorer, open the resource file. To add names and values to the resource file, click the Strings button. In the Name column, enter a name that uniquely identifies the resource. The name should describe the resource and should contain no spaces or special characters.

You will use these names as the resource name parameters of the policy, behavior, behavior option, and behavior parameter constructors that you used in your “policy helper” application. The constructor in the “policy helper” application adds the specified text resource to the metadata of the your policies and behaviors.

In the Value column, enter the text that you want the Dynamics Security Console to display. The Comment column is optional.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateLeadPolicyName</td>
<td>Create Lead Policy</td>
</tr>
<tr>
<td>DeleteLeadPolicyName</td>
<td>Delete Lead Policy</td>
</tr>
<tr>
<td>UpdateLeadPolicyName</td>
<td>Update Lead Policy</td>
</tr>
</tbody>
</table>

To begin, specify the root object name for your document type. The following example shows the root object name for leads.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LeadRootObjectName</td>
<td>Leads</td>
<td></td>
</tr>
</tbody>
</table>

Add the names for your policies. The following example shows the names for each of the lead policies.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateLeadPolicyName</td>
<td>Create Lead Policy</td>
</tr>
<tr>
<td>DeleteLeadPolicyName</td>
<td>Delete Lead Policy</td>
</tr>
<tr>
<td>UpdateLeadPolicyName</td>
<td>Update Lead Policy</td>
</tr>
</tbody>
</table>
Add a name and description for each of your behaviors. The following example shows the names of the behaviors associated with the Create Lead Policy.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateQualifiedLeadBehaviorName</td>
<td>Create qualified lead</td>
</tr>
<tr>
<td>CreateQualifiedLeadBehaviorDescription</td>
<td>Specify the lead source for a qualified lead</td>
</tr>
</tbody>
</table>

Add a name and description for each of the behavior options for your behaviors. The following example shows the behavior options for the behavior associated with the Create Lead Policy.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotQualifiedLeadOptionName</td>
<td>Not Qualified</td>
</tr>
<tr>
<td>NotQualifiedLeadOptionDescription</td>
<td>Lead is not qualified</td>
</tr>
<tr>
<td>QualifiedLeadOptionName</td>
<td>Qualified</td>
</tr>
<tr>
<td>QualifiedLeadOptionDescription</td>
<td>Lead is qualified</td>
</tr>
</tbody>
</table>

Add a name and description for each behavior parameter. The following example shows the parameter name and description for the Create Lead Policy.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>QualifiedLeadParameterName</td>
<td>Lead source</td>
</tr>
<tr>
<td>QualifiedLeadParameterDescription</td>
<td>Source of the lead</td>
</tr>
</tbody>
</table>

4. **Save the project.**
   From the file menu, choose Save All.

**Building the application and resource assembly**

In the Visual Studio Build menu, choose Build Solution. Visual Studio builds your console application and the policy resource assembly.
Chapter 30: Creating the Service Interface

To provide web access to your documents, create a service interface that defines the methods to create, update, delete, and retrieve business documents. The following sections describe how to create the service interface.

- Overview of a service interface
- Creating the Visual Studio project
- Adding an interface for the native endpoint
- Adding an interface for the legacy endpoint
- Implementing a service contract for the native endpoint
- Adding Dynamics GP Service methods
- Implementing a contract for the legacy endpoint
- Adding configuration information
- Building the service

Overview of a service interface

To give developers access to a new type of business document, create a service interface with methods that access the document. A service interface for Microsoft Dynamics GP requires the following references:

- A reference to an assembly that defines the business document type. The document type includes the document, the document key, the summary document, and the document criteria.

- A reference to the Microsoft.Dynamics.GP.BusinessLogic assembly. The assembly includes the GreatPlainsBusinessService class. This class performs the following operations for any properly-defined business document type.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Adds a new document to the Dynamics GP database.</td>
</tr>
<tr>
<td>Delete</td>
<td>Removes an existing document from the Dynamics GP database.</td>
</tr>
<tr>
<td>GetByKey</td>
<td>Retrieves a single document.</td>
</tr>
<tr>
<td>GetList</td>
<td>Retrieves a list of summary documents.</td>
</tr>
<tr>
<td>Update</td>
<td>Modifies an existing document in the Dynamics GP database.</td>
</tr>
<tr>
<td>Void</td>
<td>Marks an existing document as void but does not remove it from the Dynamics GP database.</td>
</tr>
</tbody>
</table>

To enable the Dynamics Security Service to control access to your documents, add a Dynamics GP context object as a parameter for your methods. The Dynamics Security Service uses context information to determine whether the user calling the method is authorized to access that method.

The create, delete, update, and void methods of the GreatPlainsBusinessService require a policy object. To obtain policy information, add policy as a parameter for your create, delete, update, and void methods.

To simplify the retrieval of policy information, add the GetPolicyByKey method from the Dynamics GP Service to your service. The GetPolicyByKey method gives developers a way to retrieve policy information using a single service reference. If you do not include GetPolicyByKey, you force developers to use the Dynamics GP Service to retrieve policy information for your service methods.
You might find that adding methods from the Dynamics GP Service makes your service interface easier to use. To use an existing Dynamics GP method, add a method to your service that calls methods on the Microsoft Dynamics GP Service. For more information about how to add methods from the Dynamics GP Service to your service, see Adding Dynamics GP Service methods on page 252.

To return information about errors identified by the GreatPlainsBusinessService, convert the error to a message that can be viewed by the service user. Errors that originate in a Dynamics GP Service method that you add to your service do not require conversion. For more information about how to handle errors in a service method, see the example code in Implementing a service contract for the native endpoint on page 248.

Creating the Visual Studio project

To create the project, complete the following procedure:

1. Create a Windows Communication Foundation (WCF) Service Library project.

Open Visual Studio. In the File menu, point to New, and then click Project. In the New Project window, click Visual C# from the Project types list. Select WCF in the Project types list. Select WCF Service Library from the Templates list. Enter a Name for your service, review the Location and Solution Name, and then click OK.

When the project is created, you have two .cs file.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;projectname&gt;.cs</code></td>
<td>Use this file to define the interface for your service. If your service support more than a single endpoint, add a namespace for each endpoint. For example, the Leads service includes a file named ILeadService.cs.</td>
</tr>
<tr>
<td><code>&lt;projectname&gt;.cs</code></td>
<td>Use this file to implement the methods described by the interface. If your interface includes namespaces for more than one endpoint, add the same namespaces to this file. For example, the Leads service includes a file named LeadService.cs.</td>
</tr>
</tbody>
</table>

2. Add references.

In the Project menu, choose Add References. The Add References window opens. Click the Browse tab and browse to the “GPWebServices” folder, typically found in the following location:

C:\Program Files\Microsoft Dynamics\GPWebServices

Select the following assemblies and click OK:

- Microsoft.Dynamics.Common.dll
- Microsoft.Dynamics.Common.Types.dll
- Microsoft.Dynamics.GP.BusinessLogic.dll
- Microsoft.Dynamics.Security.dll

Also, add a reference to the document assembly you created for your document type. For example, the Leads service includes a reference to the following assembly.
3. **Add namespace references.**
   From Solution Explorer, right-click the I<projectname>.cs file and choose View Code. Add **using** statements for the following namespaces:
   - Microsoft.Dynamics.Common
   - Microsoft.Dynamics.GP

   Include a **using** statement for your document type namespace. For example, the Leads service adds the following namespace:
   - Microsoft.Dynamics.GP.Samples.Leads

   From Solution Explorer, right-click the <projectname>.cs file and choose View Code. Add **using** statements for the following namespaces:
   - Microsoft.Dynamics.Common
   - Microsoft.Dynamics.Security
   - Microsoft.Dynamics.GP

   Include a **using** statement for your document type namespace. For example, the Leads service adds the following namespace:
   - Microsoft.Dynamics.GP.Samples.Leads

4. **Update the interface file.**
   Open the interface file. In the sample Leads service, this is the ILeadService.cs file.

   Remove any interface sample code and data contract code that Visual Studio placed in your namespace.

   Your interface should resemble the following code example from the Leads service.

   ```csharp
   using System;
   using System.Collections.Generic;
   using System.Linq;
   using System.Runtime.Serialization;
   using System.ServiceModel;
   using System.Text;
   using Microsoft.Dynamics.Common;
   using Microsoft.Dynamics.GP;
   using Microsoft.Dynamics.GP.Samples.Leads;

   // Specify the service contract for the native endpoint
   namespace SampleLeadService.Contract
   {
       [ServiceContract]
       public interface ILeads
       {
       }
   }
   ```
5. **Update the contract file**

Open the contract file. In the sample Leads service, this is the LeadSample.cs file.

*Remove any sample code that Visual Studio placed in your contact class.*

The contents of the contract file should resemble the following code example form the Leads service. Notice how the Leads class implements the ILeads interface.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using Microsoft.Dynamics.Common;
using Microsoft.Dynamics.Security;
using Microsoft.Dynamics.GP;
using Microsoft.Dynamics.GP.Samples.Leads;

// Implement the service contract for the native endpoint
namespace SampleLeadService.Contract
{
    public class Leads : ILeads
    {
    }
}
```

**Adding an interface for the native endpoint**

To add a service to the native endpoint, create an interface that specifies the operations of that service. Typically, the interface is in the file I<projectname>.cs file of your Visual Studio project.

To create an interface for the Dynamics GP Service native endpoint, complete the following steps.

1. **Specify the namespace.**
   The interface that uses the native endpoint has a namespace that contains the contract interface and implementation.

   The following code example shows the namespace that the sample Leads service uses with the native endpoint.

   ```csharp
   namespace SampleLeadService.Contract
   {
   }
   ```

2. **Add an interface to the namespace.**
   Add an interface for your service that supports the native endpoint of the Dynamics GP Service. The name of your interface must begin with I.

   The following example code from the sample Leads service adds an interface named ILeads to the namespace used for the native endpoint.
CHAPTER 30  CREATING THE SERVICE INTERFACE

namespace SampleLeadService.Contract
{
    public interface ILeads
    {
    
    }
}

3. **Add the ServiceContract attribute.**
   Add the ServiceContract attribute to the interface. Use the Name property to
   specify a name for the service. Use the Namespace property to specify the XML
   namespace of the native endpoint of the Dynamics GP Service. Set the
   SessionMode property to NotAllowed.

   The following example code from the Leads service add the ServiceContract
   attribute to the ILeads interface.

   namespace SampleLeadService.Contract
   {
       [ServiceContract(Name = "Leads",
                       SessionMode = SessionMode.NotAllowed)]
       public interface ILeads
       {
       
       }
   }

4. **Specify the service operations.**
   Add a method signature for each operation that your service supports. You
   must specify the method name, return type, and parameters.

   Also, add the OperationContract attribute to each operation of your interface.

   The following code example shows the operations for the ILeads interface of the
   sample Leads service. Notice how the OperationContract attribute is added to
   each operation.

   namespace SampleLeadService.Contract
   {
       [ServiceContract(Name = "Leads",
                        SessionMode = SessionMode.NotAllowed)]
       public interface ILeads
       {
           [OperationContract]
           Lead GetLeadByKey(LeadKey leadKey, Context context);

           [OperationContract]
           List<LeadSummary> GetLeadList(LeadCriteria criteria,
                   Context context);

           [OperationContract]
           void CreateLead(Lead lead, Context context, Policy policy);
       }
   }
Adding an interface for the legacy endpoint

To specify operations for the legacy endpoint, add a separate interface to your project. Typically, the interface is in the file I<projectname>.cs file of your Visual Studio project.

To create an interface for the Dynamics GP Service legacy endpoint, complete the following steps.

1. **Specify the namespace.**
   Add a namespace for the legacy endpoint interface and implementation.

   The following code example shows the namespace that the sample Leads service uses with the legacy endpoint.

   namespace SampleLeadService.LegacyContract
   {
   
   }

2. **Add an interface to the namespace**
   Add an interface to the namespace. The name of your interface must begin with I.

   The following example code from the sample Leads service adds an interface named ILeadsLegacy to the namespace used for the legacy endpoint.

   namespace SampleLeadService.LegacyContract
   {
     public interface ILeadsLegacy
     {
     
     }
   }

   void UpdateLead(Lead lead, Context context, Policy policy);

   [OperationContract]
   void DeleteLead(LeadKey leadKey, Context context, Policy policy);

   [OperationContract]
   Salesperson GetSalespersonByKey(SalespersonKey salespersonKey,
       Context context);

   [OperationContract]
   List<SalespersonSummary> GetSalespersonList(
       SalespersonCriteria salespersonCriteria, Context context);

   [OperationContract]
   Policy GetPolicyByKey(PolicyKey policyKey, Context context);
3. **Add the ServiceContract and XmlSerializerFormat attributes.**
Add the ServiceContract attribute to the interface. Use the Name property to specify a name for the service. Use the Namespace property to specify the XML namespace of the legacy endpoint of the Dynamics GP Service.

Add the XmlSerializerFormat attribute to the interface. The XmlSerializerFormat instructs the Dynamics GP Service framework to use the XmlSerializer.

The following example code from the sample Leads service add the ServiceContract and XmlSerializerFormat attributes to the ILeadsLegacy interface.

```csharp
namespace SampleLeadService.LegacyContract
{
[ServiceContract(Name = "Leads",
XmlSerializerFormat]
public interface ILeadsLegacy
{

}
}
```

4. **Specify the service operations.**
Add a method signature for each operation that your service supports. You must specify the method name, return type, and parameters.

Also, add the OperationContract attribute to each operation of your interface. Use the action property of the attribute to specify the name of the contract method that implements each operation.

The following code example shows the operations for the ILeads interface of the sample Leads service. Notice how the OperationContract attribute is added to each operation and the action property specifies the name of the contract method that implements the operation. In the action property, http://www.microsoft.com/DynamicsGP/Samples/2006/ specifies the namespace for the legacy endpoint of the Dynamics GP Service.

```csharp
namespace SampleLeadService.LegacyContract
{
[ServiceContract(Name = "Leads",
XmlSerializerFormat]
public interface ILeadsLegacy
{
    Lead GetLeadByKey(LeadKey leadKey, Context context);

    List<LeadSummary> GetLeadList(LeadCriteria criteria, Context context);

    void UpdateLead(Lead lead, Context context);
}
```
void CreateLead(Lead lead, Context context, Policy policy);

void UpdateLead(Lead lead, Context context, Policy policy);

void DeleteLead(LeadKey leadKey, Context context, Policy policy);

Salesperson GetSalespersonByKey(SalespersonKey salespersonKey, Context context);

List<SalespersonSummary> GetSalespersonList(SalespersonCriteria salespersonCriteria, Context context);

Policy GetPolicyByKey(PolicyKey policyKey, Context context);

Implementing a service contract for the native endpoint

To enable service access to your document type, add a class that implements each of the methods in your native endpoint interface.

Typically, the class that implements the interface is in a separate file. For example, the sample Leads service implements the ILeads interface in the LeadService.cs file.

The following steps show how to use the Dynamics GP Service framework to implement the service methods for the native endpoint:

1. **Verify the namespace.**
   The class that implements the interface must be in the same namespace as the interface. Verify that your <projectname>.cs file includes your namespace or add the namespace to the file.

   For example, the sample Leads service uses the SampleLeadService.Contract namespace for the native endpoint.

2. **Add a class to the namespace.**
   Add a class to the namespace that implements the interface you specified.

   The following code example adds the Leads class to the SampleLeadService.Contract namespace. Notice how the Leads class implements the ILeads interface.

   // Implement the service contract for the native endpoint namespace SampleLeadService.Contract
3. **Add the ServiceBehavior attribute.**

Use the properties of the ServiceBehavior attribute to specify the name and namespace of the service. Use the same namespace as the native endpoint of the Dynamics GP Service.

The following code example adds the ServiceBehavior attribute to the Leads class. Notice how the Name property specifies a name for the service and the Namespace property specifies the namespace for the native endpoint of the Dynamics GP Service.

```csharp
// Implement the service contract for the native endpoint
namespace SampleLeadService.Contract
{
    [ServiceBehavior(Name = "Leads",
    public class Leads : ILeads
    {
    }
}
```

4. **Add a method to retrieve a single document.**

This method retrieves a single document. Add method parameters that require a document key and a Dynamics GP service context object.

Use the GetByKey method of the GreatPlainsBusinessService to retrieve the specified document. The GetByKey method requires you to supply the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>A Dynamics GP Service key object that specifies a document.</td>
</tr>
<tr>
<td>Context</td>
<td>A Dynamics GP Service context object.</td>
</tr>
<tr>
<td>Type</td>
<td>Specifies the type of the document to retrieve.</td>
</tr>
</tbody>
</table>

The following code example shows the GetLeadByKey method for the sample Leads service. Notice the use of the `typeof()` method to specify the document type. Also, notice how the ExceptionManager converts error information to a SOAP exception.

```csharp
public Lead GetLeadByKey(LeadKey leadKey, Context context)
{
    try
    {
        GreatPlainsBusinessService service =
            GreatPlainsBusinessService.GetInstance();
        return (Lead)service.GetByKey(leadKey, context, typeof(Lead));
    }
    catch (BusinessException err)
5. **Add a method to retrieve a list of summary documents.**

This method retrieves a list of summary documents. Add method parameters that require a document criteria and a Dynamics GP Service context object.

Use the GetList method of the GreatPlainsBusinessService to retrieve the specified documents. The GetList method requires you to supply type information for the list. In addition, the GetList method requires you to supply the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Criteria</td>
<td>A document criteria object that specifies the characteristics of the documents to include in the list.</td>
</tr>
<tr>
<td>Context</td>
<td>A Dynamics GP Service context object.</td>
</tr>
</tbody>
</table>

The following code example shows the GetLeadList method for the sample Leads service. Notice how the GetList method of the GreatPlainsBusinessService requires you to specify the document type and the summary type.

```csharp
public List<LeadSummary> GetLeadList(LeadCriteria criteria,
                                      Context context)
{
    try
    {
        GreatPlainsBusinessService service =
            GreatPlainsBusinessService.GetInstance();
        return service.GetList<Lead, LeadSummary>(criteria, context);
    }
    catch (BusinessException err)
    {
        throw ExceptionManager.ConvertBusinessToSoapException(err);
    }
}
```

6. **Add a method to create a document.**

This method creates a document. Add method parameters that require a document, a Dynamics GP Service context object, and a document create policy.

Use the Create method of the GreatPlainsBusinessService to insert the document in the Dynamics GP database. The Create method requires you to supply the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusinessObject</td>
<td>A Dynamics GP Service business document.</td>
</tr>
<tr>
<td>Context</td>
<td>A Dynamics GP Service context object.</td>
</tr>
<tr>
<td>Policy</td>
<td>A Dynamics GP Service policy object.</td>
</tr>
</tbody>
</table>

The following code example shows the CreateLead method for the sample Leads service. Notice how the document, context, and policy method parameters are used with the Create method of the GreatPlainsBusinessService.
public void CreateLead(Lead lead, Context context, Policy policy) {
    try {
        GreatPlainsBusinessService service = GreatPlainsBusinessService.GetInstance();
        service.Create(lead, context, policy);
    } catch (BusinessException err) {
        throw ExceptionManager.ConvertBusinessToSoapException(err);
    }
}

7. **Add a method to update a document.**
This method updates a document. Add method parameters that require a document, a Dynamics GP Service context object, and a document update policy.

Use the Update method of the GreatPlainsBusinessService to update the document in the Dynamics GP database. The Update method requires you to supply the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusinessObject</td>
<td>A Dynamics GP Service business document.</td>
</tr>
<tr>
<td>Context</td>
<td>A Dynamics GP Service context object.</td>
</tr>
<tr>
<td>Policy</td>
<td>A Dynamics GP Service policy object.</td>
</tr>
</tbody>
</table>

The following code example shows the UpdateLead method for the sample Leads service. Notice how the document, context, and policy parameters are used with the Update method of the GreatPlainsBusinessService.

public void UpdateLead(Lead lead, Context context, Policy policy) {
    try {
        GreatPlainsBusinessService service = GreatPlainsBusinessService.GetInstance();
        service.Update(lead, context, policy);
    } catch (BusinessException err) {
        throw ExceptionManager.ConvertBusinessToSoapException(err);
    }
}

8. **Add a method to delete a document.**
This method deletes a document. Add method parameters that require a document key, a Dynamics GP Service context object, and a document delete policy.
Use the Delete method of the GreatPlainsBusinessService to remove the document from the Dynamics GP database. The Delete method requires you to supply the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>A Dynamics GP Service key object that specifies a document.</td>
</tr>
<tr>
<td>Context</td>
<td>A Dynamics GP Service context object.</td>
</tr>
<tr>
<td>Policy</td>
<td>A Dynamics GP Service policy object.</td>
</tr>
<tr>
<td>Type</td>
<td>Specifies the type of the document to delete.</td>
</tr>
</tbody>
</table>

The following code example shows the DeleteLead method for the sample Leads service. Notice how the document key, context, and policy parameters of the method are used with the Delete method of the GreatPlainsBusinessService. Also, notice the use of the the `typeof()` method to specify the document type for the Delete method of the GreatPlainsBusinessService.

```java
public void DeleteLead(LeadKey leadKey, Context context, Policy policy) {
    try {
        GreatPlainsBusinessService service =
        GreatPlainsBusinessService.GetInstance();
        service.Delete(leadKey, context, policy, typeof(Lead));
    }
    catch (BusinessException err) {
        throw ExceptionManager.ConvertBusinessToSoapException(err);
    }
}
```

### Adding Dynamics GP Service methods

You might find that your service interface requires access to one or more methods of the Dynamics GP Service. To add a method from the Dynamics GP Service, you first add that method signature to your interface. Adding Dynamics GP Service methods to your service is optional.

To add methods from the Dynamics GP Service to your service, complete the following steps:

1. **Identify the Dynamics GP Service methods you need.**
   Determine what methods your service needs from the Dynamics GP Service. Use the Microsoft Dynamics GP Service Reference documentation to identify the parameters and return value of each method.

   For example, you might want to add the GetPolicyByKey method to provide developers with the ability to get policy information. You would add GetPolicyByKey to retrieve the policy information required by your create, update, or delete method. The Microsoft Dynamics GP Service Reference documentation shows the GetPolicyByKey method requires a policy key object parameter, a context object parameter, and returns a Policy object.

2. **Add the method to your service interface.**
   For each Dynamics GP Service method you want to use, add that method to your service interface. The method must have the same parameters and return
value as the Dynamics GP Service method you want to use. You must also add the OperationContract attribute.

The following code example adds GetPolicyByKey, GetSalespersonByKey, and GetSalespersonList to the ILeads interface. Notice how the parameters and return type of each interface method match the parameters and return type of the Dynamics GP Service method.

```csharp
[OperationContract]
Policy GetPolicyByKey(PolicyKey policyKey, Context context);

[OperationContract]
Salesperson GetSalespersonByKey(SalespersonKey salespersonKey,
                                   Context context);

[OperationContract]
List<SalespersonSummary> GetSalespersonList(
                                       SalespersonCriteria salespersonCriteria, Context context);
```

3. **Add the methods to your contract class.**

In the class you use to implement your service interface, add a method for each of the operations you added to the interface. In the method, instantiate a Microsoft.Dynamics.GP.WebServices.Contract.DynamicsGP object.

To complete the operation, use your method to call the corresponding method from the DynamicsGP service object. Use the input parameters from your method to supply the parameters for the Dynamics GP Service method.

The following code example shows how to use the GetPolicyByKey, GetSalespersonByKey and GetSalespersonList methods from the Leads class of the sample Leads service. Notice how each method passes the input parameters to the methods of the Dynamics GP Service.

```
public Policy GetPolicyByKey(PolicyKey policyKey, Context context)
{
    try
    {
        return ws.GetPolicyByKey(policyKey, context);
    }
    catch
    {
        throw;
    }
}

public Salesperson GetSalespersonByKey(SalespersonKey salespersonKey,
                                        Context context)
{
    try
    {
        
```
Implementing a contract for the legacy endpoint

To provide access to your document type from a legacy endpoint, add a class that implements your legacy endpoint interface.

Typically, the class that implements the interface is in the <projectname>.cs file. For example, the sample Leads service implements the ILeadsLegacy interface in the LeadService.cs file.

The following steps show how to use the Dynamics GP Service framework to implement the service methods for the legacy endpoint:

1. **Verify the namespace.**
   The class that implements the interface must be in the same namespace as the interface. Verify your <projectname>.cs file include your namespace or add the namespace to the file.

   For example, the sample Leads service uses the SampleLeadService.LegacyContract namespace for the legacy endpoint.

2. **Add a class to the namespace.**
   Add a class to the namespace that implements your legacy interface. Add a method to the class for each operation specified by your legacy interface.
The following code example adds the Leads class to the SampleLeadService.LegacyContract namespace. Notice how the Leads class implements the ILeadsLegacy interface.

```csharp
namespace SampleLeadService.LegacyContract
{
    public class Leads : ILeadsLegacy
    {
    }
}
```

3. **Add the ServiceBehavior attribute.**
Use the properties of the ServiceBehavior to specify the name and namespace of the service. Use the same namespace as the legacy endpoint of the Dynamics GP Service.

The following code example adds the ServiceBehavior attribute to the Leads class. Notice how the Name property specifies a name for the service and the Namespace property specifies the namespace for the legacy endpoint of the Dynamics GP Service.

```csharp
namespace SampleLeadService.LegacyContract
{
    [ServiceBehavior(Name = "Leads",
    public class Leads : ILeadsLegacy
    {
    }
}
```

4. **Add the interface methods to the class.**
Add a method to the class for each service method specified by the legacy interface. The steps to add methods are the same as those described for the native interface.

If you implemented the native interface, you can use that class as a base class for your legacy class. By inheriting from the native class, you reuse the existing methods to implement your legacy interface.

The following code example adds SampleLeadService.Contracts.Leads as the base class for the Leads class used with the legacy endpoint.

```csharp
namespace SampleLeadService.LegacyContract
{
    [ServiceBehavior(Name = "Leads",
    public class Leads : SampleLeadService.Contract.Leads, ILeadsLegacy
    {
    }
}
```
Adding configuration information

To add a service to the Dynamics GP Service framework, you must configure your service to work with the framework. Typically, configuration information is added to a configuration file for your service.

To add a configuration file for your service, complete the following steps:

1. **Add a configuration file to your service project.**
   In the Visual Studio, click Project, and then click Add New Item. From the Add New Item window, click General in the Categories list. In the list of Templates, click Application Configuration File. In the Name box, enter a name that uniquely identifies your service configuration file. Click Add. The file is added to your service project.

   For example, the sample Leads service includes a file named SampleLeadService.config.

2. **Add cache settings for the service.**
   Add a section to the configuration file that specifies cache settings for your service. Typically, you use the same cache configuration as the Dynamic GP Service. Use the cache configuration settings found in the WSCachingConfiguration.config file.

   The following code example shows how to import the caching setting from the Dynamics GP Service into the configuration file of sample Leads service. Notice how the cachingConfiguration node specifies the WSCachingConfiguration.config file.

   ```xml
   <configuration>
     <configSections>
       <section name="cachingConfiguration"
         Configuration.CacheManagerSettings,
         Microsoft.Dynamics.EnterpriseLibrary.Caching, Culture=neutral,
         PublicKeyToken=31bf3856ad364e35"/>
     </configSections>
     <cachingConfiguration configSource="WSCachingConfiguration.config"/>
   </configuration>
   ```

3. **Add application settings.**
   To use the same application setting as the Dynamics GP Service, add <appSettings> to your configuration. Import the application setting from the WSServiceAppSettings.config file of the Dynamic GP Service.

   ```xml
   <appSettings file="WSServiceAppSettings.config"/>
   ```

4. **Add bindings.**
   Add a <system.serviceModel> node to your configuration file. Add the <bindings> node as a child of <system.serviceModel>. Import the binding configuration from the WSBindings.config file of the Dynamics GP Service.

   ```xml
   <system.serviceModel>
     <bindings configSource="WSBindings.config"/>
   </system.serviceModel>
   ```
5. **Add services.**  
After `<bindings>`, add `<services>` as a child node of `<system.serviceModel>`. Use `<services>` to add configuration information for each endpoint.

```xml
<system.serviceModel>
  <bindings configSource="WSBindings.config"/>
  <services>
    
  </services>
</system.serviceModel>
```

6. **Add the native service**  
If your service uses the native endpoint of the Dynamics GP Service framework, add `<service>` as child of `<services>`. Use the service name attribute to specify the native endpoint namespace and class name. Set `behaviorConfiguration` to `StandardBehavior`.

```xml
<service name="SampleLeadService.Contract.Leads"
  behaviorConfiguration="StandardBehavior">

Add the default endpoint to `<service>`. The endpoint uses the following settings.

```xml
  <endpoint address="mex" name="http"
    binding="customBinding" bindingConfiguration="CustomBinding"
    contract="IMetadataExchange"/>
```

Add a custom endpoint to `<service>`. Use `address` to specify the name of your contract class. Use `name` to specify a name for your service. Set `binding` to `wsHttpBinding`. Set `bindingConfiguration` to `WSHttpBindingTarget`. Use `contract` to specify the namespace and name of your native endpoint interface.

```xml
  <endpoint address="Leads" name="SampleLeadService"
    binding="wsHttpBinding"
    bindingConfiguration="WSHttpBindingTarget"
    contract="SampleLeadService.Contract.ILeads" />
```

Add host information to service. Use `baseAddress` to specify the URL for your native endpoint. The URL must use the same server name and port as the Dynamics GP Service. The following code example shows the base address used for the native endpoint of the sample Leads service.

```xml
<host>
  <baseAddresses>
    <add baseAddress = "http://localhost:48620/Sample/Leads" />
  </baseAddresses>
</host>
```

7. **Add the legacy service**  
If your service uses the legacy endpoint of the Dynamics GP Service framework, add `<service>` as a child of `<services>`. Use the service name attribute to specify your legacy endpoint namespace and class name. Set `behaviorConfiguration` to `StandardBehavior`.

```xml
<service name="SampleLeadService.Contract.Leads"
  behaviorConfiguration="StandardBehavior">
<service name="SampleLeadService.LegacyContract.Leads"
       behaviorConfiguration="StandardBehavior">

Add the default endpoint to <service>. The endpoint uses the following settings.

<endpoint address="Mex" name="http" binding="customBinding"
          bindingConfiguration="CustomBinding"
          contract="IMetadataExchange"/>

Add a custom endpoint to <service>. Set address to be empty. Use name to specify a name for your service. Set binding to basicHttpBinding. Set bindingConfiguration to BasicHttpBindingTarget. Use contract to specify the namespace and name of your legacy endpoint interface.

<endpoint address="" name="SampleLeadServiceLegacy"
          binding="basicHttpBinding"
          bindingConfiguration="BasicHttpBindingTarget"
          contract="SampleLeadService.LegacyContract.ILeadsLegacy"/>

Add host information to service. Use baseAddress to specify the URL for your legacy endpoint. The URL must use the same server name and port as the Dynamics GP Service. Add DynamicsGPWebServices to the URL and then specify the name of an .asmx file. The following code example shows the base address used for the legacy endpoint of the sample Leads service.

<host>
  <baseAddresses>
    <add baseAddress="http://localhost:48620/
         DynamicsGPWebServices/SampleLeadWebService.asmx"/>
  </baseAddresses>
</host>

8. Add behavior settings.
   After <services> in the <system.serviceModel> node, add <behaviors>. Use the configSource to import the behavior setting in the WSBehaviors.config file of the Dynamics GP Service.

<behaviors configSource="WSBehaviors.config"/>

9. Save the configuration file
   When you have completed work on the configuration file, click File, and then click Save.

Building the service

In the Visual Studio Build menu, choose Build. Visual Studio builds your service.
Chapter 31: Creating XSLT Files

The interface between Web Services for Microsoft Dynamics GP and eConnect requires the exchange of XML documents. The Dynamics GP Service framework uses XSLT transforms to reconcile differences between the document schemas. The following sections describe how to create an XSLT transform for each service operation.

- **Overview of service transforms**
- **Creating XSLT for the GetByKey operation**
- **Creating XSLT for the GetList operation**
- **Creating XSLT for the Create operation**
- **Creating XSLT for the Update operation**
- **Creating XSLT for the Delete operation**

**Overview of service transforms**

The Dynamics GP Service framework uses eConnect to perform operations on the Dynamics GP database. To perform an operation, the service framework and eConnect exchange XML documents. Since the schema of the service framework XML document differs from the schema of the eConnect XML document, the framework uses XSLT to transform XML to and from the eConnect schema. The service framework uses transforms for the following operations:

- To retrieve data, the service framework uses the eConnect Transaction Requester to get a document or a list of summary documents. The framework uses XSLT to transform the Transaction Requester XML into an XML document that the service framework uses to populate the properties of a document object.

- To perform a create, update, or delete operation, the service framework uses an XSLT transform to convert a framework XML document into an eConnect XML document. The transform produces an eConnect XML document that works with a specified eConnect SQL stored procedure to perform the database operation.

When adding a new type of document to the service, you need to supply XSLT transforms that enable the service framework to complete operations using your document. Typically, your XSLT maps property names and values from a service framework XML document to the corresponding property and value in an eConnect XML document.
You must create an XSLT transform for each service operation your document supports. Store each transform in a separate XSLT file.

The Dynamics GP Service framework requires XSLT files to follow an established naming convention. The framework uses the name to match the document type and operation to a transform. If your XSLT file does not include the expected name, you will receive an error when you try to perform the service operation.

To create an XSLT file for the Dynamics GP Service framework, specify a filename that contains the document type name and the operation. In addition, your XSLT filename cannot duplicate the name of an existing XSLT file. While you can precede your XSLT filename with namespace descriptors, your filename must include the following information:

<document type><operation>.xslt

The operation specifies the method. All XSLT filenames use one of the following operation descriptors:

- GetByKey
- GetList
- Create
- Update
- Delete
- Void

For example, the LeadGetByKey.xslt part of the filename for the file with the name Microsoft.Dynamics.GP.Samples.Leads.LeadGetByKey.xslt specifies that the file contains the XSLT transform to use with a GetByKey method that retrieves a Lead document.

The XSLT is also where you implement the create, update, and delete policy for your document. When the service framework serializes a document into XML, it adds XML for the policy object associated with the document. Use XSLT with the policy XML to identify the document data members that are affected by the policy and its behaviors. Also, use the transform process to implement any data changes that the policy and behaviors require.

One challenging part of creating the XSLT is testing the output of your transform. Visual Studio includes a useful XSLT debugger, but requires you to supply a file that contains the XML to transform. To generate the XML file, use the Serialize method of the SerializationManager to create an XML version of your document. SerializationManager is in the Microsoft.Dynamics.Common namespace and the Microsoft.Dynamics.Common.dll assembly. The sample Lead Service in the SDK includes a project named XsltTest that shows how to use the SerializationManager. For more information about installing the sample Lead Service files, see Lead Service on page 315.
Creating XSLT for the GetByKey operation

To retrieve a specified document, create XSLT that transforms the eConnect Transaction Requester document into the business document that the service framework uses. To create the XSLT for the GetByKey operation, complete the following steps:

1. **Create the XSLT file.**
   Start Visual Studio. In the File menu, point to New, and then click File. In the New File window, select General from the list of Categories. Select XSLT File from the Templates, and then click Open. Visual Studio creates an XSLT file.

   If the file contains an `<xsl:template />` example node, delete that node from the file. Your XSLT should be similar to the following example.

   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <xsl:stylesheet version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   </xsl:stylesheet>
   ```

2. **Add required namespaces.**
   Add the following namespaces as attributes to the `xsl:stylesheet` node:

   - `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   - `xmlns:ms ="urn:schemas-microsoft-com:xslt"
   - `xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
   - `xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
   - `xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"

   The following code sample shows the `xsl:stylesheet` node from the LeadGetByKey.xslt file.

   ```xml
   <xsl:stylesheet
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:ms ="urn:schemas-microsoft-com:xslt"
       xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
       xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
       xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"
       version="1.0">
   ```
3. **Import required libraries.**
   Add an `xsl:import` node for the `Microsoft.Dynamics.GP.StandardLibrary`. The file is in the service “XSLT” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT

   Notice how the following XSLT sample uses an `xsl:import` node to add the library. Also notice that the import does not include the file path for the .xslt file. The XSLT file being created must be located in the same folder as the `StandardLibrary.xslt` file.

   ```xml
   <xsl:import href="Microsoft.Dynamics.GP.StandardLibrary.xslt"/>
   ```

4. **Specify the templates.**
   Add `xsl:template` nodes to the stylesheet. The first node identifies XML documents produced by the eConnect Transaction Requester. The `xsl:apply-templates` node applies the remaining templates to the nodes in the XML document.

   ```xml
   <xsl:template match = "root/eConnect">
       <xsl:apply-templates />
   </xsl:template>
   ```

   Next, add an `xsl:templates` node that specifies the name of the document type.

   The following sample adds a template for a Lead. Notice how the XSLT template name of “Lead” matches the value that was placed in the ALIAS field of the eConnect_Out_Setup table.

   ```xml
   <xsl:template match = "Lead">
   </xsl:template>
   ```

5. **Define variables.**
   Add variables that can be used to supply default values for nodes in your document.

   The following sample shows how to create variables named “isocode” and “decimaldigits”. The value of each variable is retrieved from the `StandardLibrary`.

   ```xml
   <xsl:variable name="isocode">
       <xsl:value-of select="gputil:LocalCurrency(
           /root/mbs:Context/mbs:OrganizationKey/mbs:Id)"/>
   </xsl:variable>
   <xsl:variable name="decimaldigits" select="gputil:CurrencyDecimalDigits($isocode)"/>
   ```
6. Add the document nodes.
Add XML to the template that reflects the nodes of the business document that is returned from the service.

The following example shows the XML nodes for a lead document. Notice how the XML specifies individual property names for objects like Key, Phone1, Phone2, Fax, and PotentialRevenue.

```
<Lead>
  <Key>
    <Id />
  </Key>
  <Name />
  <SalespersonID />
  <City />
  <State />
  <Zip />
  <Address1>
    <Address2 />
    <Phone1>
      <Value />
    </Phone1>
    <Phone2>
      <Value />
    </Phone2>
    <Fax>
      <Value />
    </Fax>
    <LeadBusinessCategory />
  </Address1>
  <Country />
  <Contact />
  <PotentialRevenue>
    <Currency />
    <Value />
    <DecimalDigits />
  </PotentialRevenue>
  <QualifiedLead />
</QualifiedLead>
```
7. **Add values to the document nodes.**

To complete the XSLT transform, populate each node with a value from the eConnect Transaction Requester XML document.

When the eConnect Transaction Requester retrieves a document, it creates an XML document with nodes that contain values from the Dynamics GP source table. The Transaction Requester uses the column name from the table as the name of the node. Your XSLT needs to copy values from the eConnect nodes into your business document nodes.

The following example shows XSLT to retrieve a single lead document. Notice the use of the following XSLT nodes:

- The `<xsl:value-of />` node to get the value for the node. The select=attribute specifies the name from the Transaction Requester XML document that maps to current property.

- The `<xsl:variable />` nodes. These nodes add the “isocode” and “decimaldigits” variables from the StandardLibrary. These variables supply values for the PotentialRevenue properties. PotentialRevenue is a MoneyAmount that requires you to supply Currency and DecimalDigits values.

- The LeadBusinessCategory node uses `<xsl:choose \>` to convert the enumeration integer value found in the database to the corresponding LeadBusinessCategory enumeration string value.

- The QualifiedLead property maps the integer value retrieved from the database to the corresponding boolean value.

- The `<xsl:if />` that ensures the Modifi edDate is a valid date for Dynamics GP.

```xml
<Lead>
  <Key>
    <Id>
      <xsl:value-of select="LeadID"/>
    </Id>
  </Key>
</Lead>
```
<Name>
  <xsl:value-of select="LeadName"/>
</Name>
<SalespersonID>
  <xsl:value-of select="SLPRSNID"/>
</SalespersonID>
<City>
  <xsl:value-of select="CITY"/>
</City>
<State>
  <xsl:value-of select="STATE"/>
</State>
<Zip>
  <xsl:value-of select="ZIP"/>
</Zip>
<Address1>
  <xsl:value-of select="ADDRESS1"/>
</Address1>
<Address2>
  <xsl:value-of select="ADDRESS2"/>
</Address2>
<Phone1>
  <Value>
    <xsl:value-of select="PHONE1"/>
  </Value>
</Phone1>
<Phone2>
  <Value>
    <xsl:value-of select="PHONE2"/>
  </Value>
</Phone2>
<Fax>
  <Value>
    <xsl:value-of select="FAX"/>
  </Value>
</Fax>
<LeadBusinessCategory>
  <xsl:choose>
    <xsl:when test="LeadBusinessCategory=1">RealEstate</xsl:when>
    <xsl:when test="LeadBusinessCategory=2">Wholesale</xsl:when>
    <xsl:when test="LeadBusinessCategory=3">Retail</xsl:when>
    <xsl:when test="LeadBusinessCategory=4">Contractor</xsl:when>
    <xsl:when test="LeadBusinessCategory=5">Educational</xsl:when>
    <xsl:when test="LeadBusinessCategory=6">Media</xsl:when>
    <xsl:when test="LeadBusinessCategory=7">Software</xsl:when>
    <xsl:when test="LeadBusinessCategory=8">Restaurant</xsl:when>
  </xsl:choose>
</LeadBusinessCategory>
<Country>
  <xsl:value-of select="COUNTRY"/>
</Country>
<Contact>
  <xsl:value-of select="CONTACT"/>
</Contact>
8. **Save the file.**

In the File menu, choose Save As. In the Save File As window enter a filename that complies with Dynamics GP Service XSLT naming requirements. Review the Save in folder, and then click Save.
CHAPTER 31  CREATING XSLT FILES

Creating XSLT for the GetList operation

To retrieve a collection of summary documents, create XSLT that transforms each eConnect Transaction Requester document into the summary document that the service framework uses. To create the XSLT for the GetList operation, complete the following steps:

1. **Create the XSLT file.**
   
   Start Visual Studio. In the File menu, point to New, and then click File. In the New File window, select General from the list of Categories. Select XSLT File from the Templates, and then click Open. Visual Studio creates an XSLT file.

   If the file contains an `<xsl:template />` example node, delete that node from the file.

2. **Add required namespaces.**
   
   Add the following namespaces as attributes to the `xsl:stylesheet` node:

   - `xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance`
   - `xmlns:ms = "urn:schemas-microsoft-com:xslt"`
   - `xmlns:gpxslt="uri://GreatPlainsTransformLibrary"`
   - `xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"`
   - `xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"`

   The following code sample shows the `xsl:stylesheet` node from the LeadGetList.xslt file.

   ```xml
   <xsl:stylesheet
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:ms = "urn:schemas-microsoft-com:xslt"
       xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
       xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
       xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"
   version="1.0">
```

3. **Import required libraries.**
   
   Add an `xsl:import` node for the Microsoft.Dynamics.GP.StandardLibrary. The file is in the service “XSLT” folder, typically found in the following location:

   `C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT`

   Notice how the following XSLT sample uses an `xsl:import` node to add the library. Also notice that the import does not include the file path for the .xslt file. The XSLT file being created must be located in the same folder as the StandardLibrary.xslt file.

   ```xml
   <xsl:import href="Microsoft.Dynamics.GP.StandardLibrary.xslt"/>
   ```
4. Specify the templates.
Add xsl:template nodes to the stylesheet. Use the first template to identify the root node of the XML document. Add XML to the template that identifies this document as a collection of summary documents.

Also add a xsl:apply-templates node to apply the remaining templates to the XML documents. Use the select attribute to specify the type of nodes to transform.

The following example shows how to add an XSLT template for a list of lead summary objects. Notice how the XML uses the name of the list class (ArrayOfLeadSummary) created in the lead document assembly.

```xml
<xsl:template match="/">
  <ArrayOfLeadSummary xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsl:apply-templates select="root//eConnect//Lead"/>
  </ArrayOfLeadSummary>
</xsl:template>
```

Next, add an xsl:templates node that specifies the type of document.

The following example shows the template used for a Lead summary document.

```xml
<xsl:template match="Lead"/>
</xsl:template>
```

5. Define variables.
Add variables to the template that supply default values for nodes in the document.

The following sample shows how to create variables named “isocode” and “decimaldigits. The value of each variable is retrieved from the StandardLibrary.

```xml
<xsl:variable name="isocode">
  <xsl:value-of select="gputil:LocalCurrency(
    /root/mbs:Context/mbs:OrganizationKey/mbs:Id)"/>
</xsl:variable>
<xsl:variable name="decimaldigits"
  select="gputil:CurrencyDecimalDigits($isocode)" />
```
6. **Add the document nodes.**
Add XML to the template that represent the nodes of the summary document returned from the service.

The following example shows the XML nodes for a lead summary. Notice how the XML specifies the Id property of the Key object.

```xml
<LeadSummary>
  <Key>
    <Id/>
  </Key>
  <Name/>
  <SalespersonID/>
  <LeadBusinessCategory/>
  <QualifiedLead/>
  <LeadSource/>
  <ModifiedDate/>
</LeadSummary>
```

7. **Add values to the document nodes.**
To complete the XSLT transform, populate each node with a value from the eConnect Transaction Requester XML document.

When the eConnect Transaction Requester retrieves a document, it creates an XML document with nodes that contain values from the Dynamics GP source table. The Transaction Requester uses the column name from the table as the name of the node. Your XSLT needs to copy values from the eConnect nodes into your business document nodes.

The following example shows XSLT to retrieve a lead summary. Notice the use of the following XSLT nodes:

- The `<xsl:value-of />` node to copy the value of the node. The `select=` attribute specifies the name from the Transaction Requester XML document that maps to the current property.
- The `LeadBusinessCategory` node uses `<xsl:choose \>` to convert the enumeration integer value found in the database to the corresponding `LeadBusinessCategory` enumeration string value.
- The `QualifiedLead` property maps the integer value retrieved from the database to the corresponding boolean value.
- The `<xsl:if />` that ensures the `ModifiedDate` is a valid date for Dynamics GP.
8. Save the file.

In the File menu, choose Save As. In the Save File As window, enter a filename that complies with Dynamics GP Service XSLT naming requirements. Review the Save in folder, and then click Save.
Creating XSLT for the Create operation

To add a new document to the Dynamics GP database, use XSLT to transform the service framework document into an eConnect XML input document. To create the XSLT, complete the following steps:

1. **Create the XSLT file.**
   Start Visual Studio. In the File menu, point to New, and then click File. In the New File window, select XSLT File and click Open. Visual Studio creates the file.

   If the file contains an `<xsl:template />` example node, delete that node from the file.

2. **Add required namespaces.**
   Add the following namespaces as attributes to the `xsl:stylesheet` node:

   ```
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:ms ="urn:schemas-microsoft-com:xslt"
   xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
   xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
   xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"
   ```

   The following code sample shows the `xsl:stylesheet` node from the LeadCreate.xslt file.

   ```
   <xsl:stylesheet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                   xmlns:ms ="urn:schemas-microsoft-com:xslt"
                   xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"
                   xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
                   xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
                   version="1.0">
   ```

3. **Import required libraries.**
   Add an `xsl:import` node for the `Microsoft.Dynamics.GP.StandardLibrary`. The file is in the service “XSLT” folder, typically found in the following location:

   ```
   C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT
   ```

   Notice how the following XSLT sample uses an `xsl:import` node to add the library. Also notice that the import does not include the file path for the .xslt file. The XSLT file being created must be located in the same folder as the StandardLibrary.xslt file.

   ```
   <xsl:import href="Microsoft.Dynamics.GP.StandardLibrary.xslt"/>
   ```
4. **Specify the templates.**
Add an `xsl:template` node to the stylesheet. Use the match attribute of the template node to specify the nodes in the document to transform.

The following code sample shows the template used with a lead. Notice how the match attribute specifies to apply the template beginning at the document root. Also, the name attribute allows this template to be used in other transforms.

```xml
<xsl:template name="Lead" match="/">
</xsl:template>
```

5. **Add the document nodes.**
Add XML to the template that reflects the nodes of the eConnect XML document.

The following example shows the XML that eConnect requires to create a lead. Notice the following:

- The XML creates an eConnect document.
- The document contains a `SampleCreateUpdateLead` node.
- The `sampleLeadCreateUpdate` node specifies the name of the eConnect SQL stored procedure.
- The child nodes of `sampleLeadCreateUpdate` specify the names of the input parameters of the SQL stored procedure.

```xml
<eConnect xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SampleCreateUpdateLead>
    <sampleLeadCreateUpdate>
      <LeadID/>
      <LeadName/>
      <SLPRSNID/>
      <SLPRSNID/>
      <CITY/>
      <CITY/>
      <STATE/>
      <STATE/>
      <ZIP/>
      <ZIP/>
      <ADDRESS1/>
      <ADDRESS1/>
      <ADDRESS2/>
    </sampleLeadCreateUpdate>
  </SampleCreateUpdateLead>
</eConnect>
```
6. Add values to the document nodes.

To complete the XSLT transform, populate each node with a value from the service document.

The Dynamics GP service framework produces a serialized version of your document. Your XSLT needs to copy the values from the nodes in your service document to the nodes of the eConnect XML document for the business object that creates records.

The following example shows XSLT to create a lead. Notice the use of the following XSLT nodes:

- The `<xsl:value-of />` node to get the value of the node. The `select` attribute specifies the name of the document property that maps to each eConnect XML document node.

- The `LeadBusinessCategory` node uses `<xsl:choose >` to convert the `LeadBusinessCategory` enumeration string value to the enumeration integer value.

- The `QualifiedLead` property maps the boolean value to the value stored in the database.

- The `<xsl:if />` that ensures the `QualificationDate` is a valid date for Dynamics GP.

```xml
<eConnect xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SampleCreateUpdateLead>
    <sampleLeadCreateUpdate>
      <LeadID>
        <xsl:value-of select="Lead/Key/Id"/>
      </LeadID>
    </SampleCreateUpdateLead>
  </SampleCreateUpdateLead>
</eConnect>
```
7. Implement policy for specified nodes.

Use the policy information that the framework adds to the document to identify behaviors that apply to a node or set of nodes. Add XSLT code that implements each behavior option for each behavior.

The following example shows XSLT for the Lead Create Policy. The configuration of the Lead Create Policy specified by the service administrator indicates whether to use default Qualified and LeadSource values for leads created using the service.

The transform uses the following XSLT when the “Qualified” behavior option is selected. Note the use of XSLT in the following areas:

- The use of an xsl:if statement to identify whether the policy configuration specifies the “Qualified” behavior option. The XSLT uses the GUID of the behavior key to identify the behavior. The value of the SelectedOption/Key/Id property specifies the configured behavior option.
- When configured to use the “Qualified” behavior option, the value of the QualifiedLead property defaults to “true”.
- The use of the Value property of the behavior parameter for the LeadSource.
- The use of the current date to specify the QualificationDate.
The transform uses the following XSLT when the "Not Qualified" behavior option is selected. When configured to use the "Not Qualified" behavior option, the QualifiedLead, LeadSource, and QualificationDate values are taken from the lead document.

```xml
<xsl:if test='Lead/Policy/Behaviors/Behavior[
    Key/Id = '10099a58-64bf-4fa3-b35f-ed43d1b6ff9c']
/SelectedOption/Key/Id=2'>
    <QualifiedLead>
        <xsl:if test='Lead/QualifiedLead != '''>
            <xsl:choose>
                <xsl:when test='Lead/QualifiedLead='true'>2</xsl:when>
                <xsl:otherwise>1</xsl:otherwise>
            </xsl:choose>
        </xsl:if>
    </QualifiedLead>
    <LeadSource>
        <xsl:value-of select='Lead/LeadSource'/>
    </LeadSource>
    <QualificationDate>
        <xsl:choose>
            <xsl:when test='Lead/QualificationDate=
                '0001-01-01T00:00:00'">1/1/1900</xsl:when>
            <xsl:otherwise>
                <xsl:value-of select='gputil:GetDate(Lead/QualificationDate)'/>
            </xsl:otherwise>
        </xsl:choose>
    </QualificationDate>
</xsl:if>
```

8. **Save the file.**

In the File menu, choose Save As. In the Save File As window, enter a filename that complies with Dynamics GP Service XSLT naming requirements. Review the Save in folder, and then click Save.
Creating XSLT for the Update operation

To modify an existing document, use XSLT to transform the document into an eConnect XML input document. Typically, the update operation uses the same eConnect XML document and business object as the create operation. This allows you to re-use the XSLT in your create file. To create the XSLT for an Update operation, complete the following steps:

1. **Create an XSLT file.**
   Start Visual Studio. In the File menu, point to New, and then click File. In the New File window, select XSLT File and click Open. Visual Studio creates a file.

   Remove the existing xsl:template node.

2. **Add required namespaces.**
   Add the following namespaces as attributes to the xsl:stylesheet node:

   - xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   - xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   - xmlns:ms ="urn:schemas-microsoft-com:xslt"
   - xmlns:gpxsilst="uri://GreatPlainsTransformLibrary"
   - xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
   - xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"

   The following code sample shows the xsl:stylesheet node from the LeadUpdate.xslt file.

   ```xml
   <xsl:stylesheet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                   xmlns:ms ="urn:schemas-microsoft-com:xslt"
                   xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"
                   xmlns:gpxsilst="uri://GreatPlainsTransformLibrary"
                   xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
                   version="1.0">
   ...
   </xsl:stylesheet>
   ``

3. **Import required libraries.**
   Add xsl:import nodes that specify files that contain additional templates. The files are in the service “XSLT” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT

   Notice how the following XSLT sample requires you to import two XSLT files.

   - The Microsoft.Dynamics.GPSamples.Leads.LeadCreate.xslt file contains the template you created for the Create operation of the Lead Service. This allows you to use the same eConnect XML document and business object for the update operation.
   - The Microsoft.Dynamics.GP.StandardLibrary.xslt file contains supporting templates and definitions used across all Dynamics GP Service operations.

   Also notice that the import does not include the file path for the .xslt files. The XSLT file being created must be located in the same folder as the LeadCreate.xslt and StandardLibrary.xslt files.
<xsl:import href="Microsoft.Dynamics.GP.Samples.Leads.LeadCreate.xslt"/>
<xsl:import href="Microsoft.Dynamics.GP.StandardLibrary.xslt"/>

4. **Specify the templates.**
Add an `xsl:template` node to the stylesheet. Use the match attribute of the
template node to specify the nodes in the document to transform. Also, specify
the template to use to transform the document type.

The following code sample shows the templates used to update a lead. Notice
how the `xsl:call-template` specifies the Lead template. The Lead template is in
the `Microsoft.Dynamics.GP.Samples.Leads.LeadCreate.xslt` file that was
imported in the previous step.

```xml
<xsl:template match=""/>
  <xsl:call-template name="Lead"/>
</xsl:call-template>
</xsl:template>
```

5. **Implement policy for specified nodes.**
Use the policy information that the framework adds to the document to identify
behaviors that apply to a node or set of nodes. Add XSLT that implements each
behavior option for each behavior.

The following example adds the Lead Update Policy to the LeadCreate.xslt file.
The Lead Update Policy does not have any behaviors. Note the use of XSLT in
the following areas:

- The use of `xsl:if` to identify the policy. The value of `Policy/Key/Id` node is
  compared to the GUID of the Lead Update Policy. If they match, an update
  operation is being performed.

- When an update operation is performed, the template uses values from the
  lead document to populate the QualifiedLead, LeadSource, and
  QualificationDate properties.

```xml
<xsl:if test="Lead/Policy/Key/Id = '443e9afd-50e0-44e2-a2f1-8fffc2dce596'">
  <QualifiedLead>
    <xsl:if test="Lead/QualifiedLead != ''">
      <xsl:choose>
        <xsl:when test="Lead/QualifiedLead='true'">2</xsl:when>
        <xsl:otherwise>1</xsl:otherwise>
      </xsl:choose>
    </xsl:if>
  </QualifiedLead>
  <LeadSource>
    <xsl:value-of select="Lead/LeadSource"/>
  </LeadSource>
</xsl:if>
```
6. **Save the file**

In the File menu, choose Save As. In the Save File As window, enter a filename that complies with Dynamics GP Service XSLT naming requirements. Review the Save in folder, and then click Save.

### Creating XSLT for the Delete operation

To remove an existing document, use XSLT to transform the document into an eConnect XML input document. To create the XSLT for a Delete operation, complete the following steps:

1. **Create the XSLT file.**

   Start Visual Studio. In the File menu, point to New, and then click File. In the New File window, select XSLT File and click Open. Visual Studio creates a file. Remove the existing xsl:template node.

2. **Add required namespaces.**

   Add the following namespaces as attributes to the xsl:stylesheet node:

   - `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   - `xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   - `xmlns:ms = "urn:schemas-microsoft-com:xslt"
   - `xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
   - `xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
   - `xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"

   The following code sample shows the xsl:stylesheet node from the LeadDelete.xslt file.

   ```xml
   <xsl:stylesheet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                   xmlns:ms = "urn:schemas-microsoft-com:xslt"
                   xmlns:gpxslt="uri://GreatPlainsTransformLibrary"
                   xmlns:gputil="urn:Microsoft.Dynamics.GP.TransformUtilities"
                   xmlns:mbs="http://schemas.microsoft.com/dynamics/2006/01"
                   version="1.0">
   ```
3. **Import required libraries.**  
Add an `xsl:import` node for the `Microsoft.Dynamics.GP.StandardLibrary`. The file is in the service “XSLT” folder, typically found in the following location:

```
C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT
```

Notice how the following XSLT sample uses an `xsl:import` node to add the library. Also notice that the import does not include the file path for the .xslt file. The XSLT file being created must be located in the same folder as the StandardLibrary.xslt file.

```
<xsl:import href="Microsoft.Dynamics.GP.StandardLibrary.xslt"/>
```

4. **Specify the templates.**  
Add an `xsl:template` node to the stylesheet. Use the `match` attribute of the template node to specify the nodes in the document to transform.

The following code sample shows the template used with a lead. Notice how the `match` attribute specifies to apply the template beginning at the document root.

```
<xsl:template match="/">
</xsl:template>
```

5. **Add the document nodes.**  
Add XML to the template that reflects the nodes of the eConnect XML document.

The following example shows the XML that eConnect requires to delete a lead. Notice the following:

- The XML identifies this as an eConnect document.
- The document contains the `SampleDeleteLead` node that the eConnect XML document schema requires.
- The `sampleLeadDelete` node specifies the name of the eConnect SQL stored procedure.
- The `LeadID` node specifies the name of the input parameter of the `sampleLeadDelete` business object.

```
<eConnect xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SampleDeleteLead>
    <sampleLeadDelete>
      <LeadID>
      </LeadID>
    </sampleLeadDelete>
  </SampleDeleteLead>
</eConnect>
```
6. **Add values to the document nodes.**
   To complete the XSLT transform, populate each node with a value from the service framework XML document.

   The Dynamics GP Service framework produces a serialized version of your document. Your XSLT needs to copy the values from the nodes in your document to the nodes of the eConnect XML document.

   The following example shows XSLT to delete a lead. Notice the use of the `<xsl:value-of />` node to specify the value of the LeadID node. The `select=` attribute specifies the location of the Id value in the source document.

   ```xml
   <eConnect xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <SampleDeleteLead>
       <SampleDeleteLead>
         <LeadID>
           <xsl:value-of select="Lead/Key/Id"/>
         </LeadID>
       </SampleDeleteLead>
     </SampleDeleteLead>
   </eConnect>
   ```

7. **Implement policy for specified nodes (if required).**
   If you have implemented a policy for the Delete operation, use the policy information that the framework adds to the document to identify the behaviors that apply to a node or set of nodes. Add XSLT that implements each behavior option of each behavior.

   If you have not implemented a policy for the Delete operation, no additional action is required.

8. **Save the file.**
   In the File menu, choose Save As. In the Save File As window, enter a filename that complies with Dynamics GP Service XSLT naming requirements. Review the Save in folder, and then click Save.
Chapter 32: Deploying the Service

To use a new service, deploy the service components you created on the server or servers where you installed Dynamics GP server components. The following sections contain instructions for deploying your service.

- **Overview of service deployment**
- **Installing the eConnect components**
- **Installing the service components**

### Overview of service deployment

After you create and build the components for your service, you need to deploy your components to the Dynamics GP server or servers. Before you attempt to deploy your service, verify that your components are built and verify the location of the component files. To deploy a service, you need the following components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL stored procedures</td>
<td>The SQL scripts that add the eConnect SQL stored procedures that create, update, or delete Dynamics GP database records.</td>
</tr>
<tr>
<td>Transaction Requester metadata</td>
<td>A SQL script that adds the metadata the eConnect Transaction Requester uses to retrieve fields for a document or summary document.</td>
</tr>
<tr>
<td>eConnect error codes</td>
<td>A SQL script that adds custom error codes and messages to the eConnect error reporting system.</td>
</tr>
<tr>
<td>Document type assembly</td>
<td>A .NET assembly that contains the classes that define a new type of document, summary document, list criteria, and list. The Dynamics GP Service framework uses the classes in this assembly to create, update, delete, and retrieve data in the Dynamics GP database.</td>
</tr>
<tr>
<td>Security metadata</td>
<td>The application and application configuration file you use to add custom security metadata to the Dynamics Security Service.</td>
</tr>
<tr>
<td>Policy metadata</td>
<td>The application and application configuration file you use to add custom policy metadata to Dynamics GP services.</td>
</tr>
<tr>
<td>Policy resource assembly</td>
<td>A .NET assembly that contains the strings and labels the Dynamics Security Console uses when displaying your policy information.</td>
</tr>
<tr>
<td>Service assembly</td>
<td>An assembly and configuration file that define the service interface and its methods. The settings in the configuration file enables the Dynamics GP Service to host your service.</td>
</tr>
<tr>
<td>XSLT files</td>
<td>The XSLT files that allow the Dynamics GP Service framework to transform service XML documents to and from the eConnect XML documents.</td>
</tr>
</tbody>
</table>
Installing the eConnect components

Add the SQL stored procedures that your service uses to create, update, and delete documents. Also, add eConnect Transaction Requester metadata used when retrieving documents and summary documents. To install the eConnect components, complete the following steps:

1. **Copy your SQL files to the Dynamics GP data server.**
   Copy your SQL script files to a folder on your Dynamics GP data server. This is the SQL server that is managing Microsoft Dynamics GP data.

2. **Open Microsoft SQL Server Management Studio on your Dynamics GP data server.**

3. **Open the SQL script files that installs your SQL stored procedures.**
   In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied your SQL script files. Typically, you will have a separate SQL script file for each create/update, delete or void stored procedure that you created. To begin the install, double-click a SQL script file. The SQL script opens in the query window.

4. **Specify the company database.**
   In the list of databases, specify the company database of the company where you plan to install your service.

5. **Run the SQL script.**
   From the Query menu, choose Execute.

   *If you have more than one SQL script file that installs a SQL stored procedure, repeat the last three steps for each file.*

6. **Run the SQL script that adds metadata to the Transaction Requester.**
   In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied your SQL script files. Double-click the file that contains the script you created to add document and summary document metadata to the eConnect Transaction Requester. The script opens in the query window. In the Query menu, choose Execute.

7. **Run the SQL script files for each company.**
   If your service can include data from other company databases, you need to install the SQL stored procedures and Transaction Requester metadata to each company database.

   To install to another company, go back to the step where you specify the company database. Select the company database. Next, complete the steps where you run the SQL scripts that install your SQL stored procedures and Transaction Requester metadata. Repeat these steps for each company that works with your service.
8. **Run the SQL script that adds your custom error codes.**

In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied SQL script files. Double-click the file that contains the script you use to add custom error information to eConnect. The script opens in the query window. Open the Query menu and choose Execute. The script adds the error codes and error messages to the taErrorCode table of the DYNAMICS database.

**Installing the service components**

On the server where you installed Web Services for Microsoft Dynamics GP, add the document type assemblies, metadata helper applications, service assembly, and XSLT files for your service. To install a new service, complete the following steps:

1. **Install the document type assembly.**

   When you build the project, Visual Studio places the assembly in the “\bin\debug” or “\bin\retail” folder of the Visual Studio project folder. Copy the assembly to the Dynamics GP “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

2. **Add your security metadata.**

   When you build the project, Visual Studio places the application and application configuration files in the “\bin\debug” or “\bin\retail” folder of the Visual Studio project folder. Copy your “security helper” application and application configuration files to the Dynamics GP “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

To add your custom security operations, tasks, and role assignments in the Dynamics Security Service, use your “security helper” application. To run the InstallLeadSecurityMetadata application, you must be logged in as a user that has permission to update the Dynamics Security Service.

The permissions required for this login depend upon the security store you selected when you installed Web Services for Microsoft Dynamics GP. You must use a login that has permission to update your specified security store.

<table>
<thead>
<tr>
<th>Security store</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Directory</td>
<td>Log in as user who is an ADAM administrator. If you run the InstallLeadSecurityMetadata application using a login that is not an ADAM administrator, ADAM returns a “System.UnauthorizedAccessException” error. By default, the user who installed Web Services for Microsoft Dynamics GP is an ADAM administrator. To add another user as an ADAM administrator, see the Web Services Installation and Administration Guide.</td>
</tr>
<tr>
<td>Application Mode (ADAM)</td>
<td>Log in as user with permission to open and manage the Active Directory authorization store. Specify the login you use when you open the Authorization Manager console window to view and update application permissions in your Active Directory authorization store.</td>
</tr>
<tr>
<td>Active Directory</td>
<td>Log in as a user with SQL administrator permissions for the database you use to store Dynamics Security Service metadata. The login must have permission to create records in the underlying security tables.</td>
</tr>
</tbody>
</table>
To add your security operations, tasks, and role assignments to the Dynamics Security Service, run your “security helper” application with the “/load” command line parameter.

To add your new security tasks and operations to additional Dynamics Security Service roles, use the Dynamics Security Console.

3. **Add your policy metadata.**
   When you build the project, Visual Studio places the application and application configuration files in the “\bin\debug” or “\bin\retail” folder of the Visual Studio project folder. Copy your “policy helper” application and application configuration files to the Dynamics GP “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

   To add your custom policies and behaviors to the Dynamics GP services, run your “policy helper” application. Run the application with the “/load” and the company Id number as command line parameters.

   To find the company Id of each installed company, query the SY01500 table of the DYNAMICS database.

4. **Install the policy resource assembly.**
   When you build the project, Visual Studio places the assembly in the “\bin\debug” or “\bin\retail” folder of the Visual Studio project folder. Copy the policy resource assembly to the Dynamics GP “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

5. **Check the security permissions for the policy resource assembly.**
   Verify that the policy resource assembly grants Read permission to the user account that is used as the login identity of the Microsoft Dynamics GP Service Host service.

   To check security permissions, open the file properties window and click the Security tab. Verify the groups, users, and permissions specified for the policy resource assembly are the same as those specified for the following assembly:

   Microsoft.Dynamics.GP.DataAccess

6. **Install the service assembly.**
   When Visual Studio created your service project, it placed the assembly file in the “\bin\debug” or “\bin\retail” folder of your Visual Studio project folder. Copy the service your service assembly file to the Dynamics GP “GPWebServices” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

7. **Install the service assembly configuration file.**
   Copy the configuration file for your service to the Dynamics GP “ServiceConfigs” folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs
8. **Add the service to the service host configuration file.**

To enable a service client to connect to your service, your service must be hosted by the Microsoft.Dynamics.GP.ServiceHost.exe service. To add a service to that host, add information about your service to the Microsoft.Dynamics.GP.ServiceHost.exe.config file. The configuration file is typically found in this location:

```
C:\Program Files\Microsoft Dynamics\GPWebServices
```

To ensure you can restore the Dynamics GP Service ot its original functionality, create a copy of the Microsoft.Dynamics.GP.ServiceHost.exe.config file. Save the copy to a safe location.

Open the Microsoft.Dynamics.GP.ServiceHost.config file in a text editor. Add a `<service>` node to `<services>`. If your service uses both the native and legacy endpoints, add a `<service>` entry for each endpoint. The `<service>` node in the configuration file requires the following information about your service:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Specify the name of your service.</td>
</tr>
<tr>
<td>serviceType</td>
<td>Specify information that identifies your service. You must include the following information: The namespace and class name from your service. The name of the assembly for your service. The version number of your service assembly. The culture setting for your service assembly.</td>
</tr>
<tr>
<td>ConfigurationFileLocation</td>
<td>Specify the location and name of the configuration file for your service.</td>
</tr>
</tbody>
</table>

9. **Install the XSLT files.**

Copy your XSLT files to the Dynamics GP service “XSLT” folder, typically found in the following location:

```
C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT
```

10. **Restart the Microsoft Dynamics GP Service Host.**

To ensure the Dynamics GP Service loads and hosts your service, restart the Microsoft Dynamics GP Service Host. To restart the host service, open the Services console, click the service name Microsoft Dynamics GP Service Host and then stop and restart the service.
Part 6: Web Service Samples

This portion of the documentation describes several sample applications that demonstrate how to use the Dynamics GP web service. The following samples are discussed:

- **Chapter 33, “Sales Documents (Legacy),”** describes a sample that uses the legacy endpoint to retrieve lists of sales documents for a specific customer.

- **Chapter 34, “Sales Documents (Native),”** describes a sample that uses the native endpoint to retrieve lists of sales documents for a specific customer.

- **Chapter 35, “Import GL Transactions (Legacy),”** describes a sample that uses the legacy endpoint to import general ledger transactions into Microsoft Dynamics GP.

- **Chapter 36, “Import GL Transactions (Native),”** describes a sample that uses the native endpoint to import general ledger transactions into Microsoft Dynamics GP.

- **Chapter 37, “Update Customers (Legacy),”** describes a sample that uses the legacy endpoint to update customer records in Microsoft Dynamics GP.

- **Chapter 38, “Update Customers (Native),”** describes a sample that uses the native endpoint to update customer records in Microsoft Dynamics GP.

- **Chapter 39, “Lead Service,”** describes the Lead web service that is implemented using the Microsoft Dynamics GP Web Service framework.

*These samples have been created using Visual Studio 2005. To use them with Visual Studio 2008, allow the upgrade wizard to convert each project when you open the solution file.*
Chapter 33: Sales Documents (Legacy)

The Sales Document sample is a C# .NET application that uses the legacy endpoint of the Dynamics GP service to retrieve data. The application displays sales document summary information for a customer. The following topics are discussed:

- Overview of Sales Documents sample
- Running the Sales Documents sample
- How the Sales Documents sample works
- Dynamics GP service use for Sales Documents

Overview of Sales Documents sample

This sample application shows how methods can be used to populate the fields of a user interface. The application displays company, customer, and sales document data, and updates the displayed data to reflect choices made by the user.

The sample application requires the computer to have Visual Studio 2005 or later and the 2.0 .NET framework installed.

Running the Sales Documents sample

To run this sample application, perform the following steps:

1. Verify the Dynamics GP service.
   To ensure Dynamics GP service is installed and ready, enter the service URL into the address bar of the web browser. The default URL for the legacy endpoint is:

   http://machine_name:<port number>/DynamicsGPWebServices

   Typically, the port number is 48620. If the browser displays a message that states you have created a service, the service is ready to use with the sample application.
2. **Start Visual Studio and open the solution file for the sample application.**
   The solution file for this sample is named SalesDocumentSample.sln. The solution file is in the SalesDocument folder inside the Samples folder.

3. **Verify the .NET framework (Visual Studio 2008 and later).**
   The SalesDocumentSample project requires the .NET Framework 2.0. To verify the target framework, open the Visual Studio Solution Explorer and right-click the SalesDocumentSample project name. From the menu, click Properties. In the Application tab of the Properties window, verify that the Target Framework shows .NET Framework 2.0.

   If Target Framework shows another .NET framework, use the drop down list to specify .NET Framework 2.0.

4. **Update the web reference.**
   Open Visual Studio's Solution Explorer and select the Web References folder. Select the DynamicsGPService web reference to view its properties. Change the Web Reference URL property to use the URL for the legacy endpoint of the Dynamics GP service.

   If the URL for the legacy endpoint does not work, it may be due to incompatibility with the static WSDL document. Refer to **WSDL generation** on page 17 for more information about the WSDL document. Try the following URL, which is the direct URL for the static WSDL:

   
   http://machine_name:< port number>/Metadata/Legacy/Full/DynamicsGP.wsd1

5. **Choose Start Debugging from the Debug menu.**
   To build the solution, choose “Start Debugging” in the Debug menu. The sales document sample application loads the web service proxy which may take several seconds to complete. The application will not be visible until loading the web service proxy has completed.

6. **Select a company from the Company drop-down list.**
   Once a company is selected, the Customer drop-down list contains all the customers for that company.

7. **Select a customer from the Customer drop-down list.**
   After a customer is selected, the data grid displays all the sales documents for that customer.

8. **Mark one or more of the check boxes in the Sales Documents list.**
   Use the check boxes to restrict the types of sales documents displayed in the data grid.

9. **Click the Refresh button.**
   The Refresh button forces the data grid to update and display only the specified document types for the customer.
How the Sales Documents sample works

The application consists of two classes: The CustomerSalesForm user interface class manages displaying data and selecting records by the user. The DataManagement class retrieves data using methods from the Dynamics GP service and makes the data available for display by the user interface class. The DataManagement class also shows how to place the summary list data returned by a method into a dataset. The user interface list and grid controls use the datasets as the source of the data they display.

The application uses basic error handling techniques to detect exceptions and displays all error information in message boxes.

Dynamics GP service use for Sales Documents

The sample application uses several Dynamics GP service objects and methods. The DataManagement class loads a proxy class named wsDynamicsGP that allows service objects and methods to be used. The application loads the wsDynamicsGP proxy class using information provided by the service.

Methods

The DataManagement class constructor creates the service proxy and creates a context object. The DataManagement constructor also sets the credentials for the service proxy to use the default credentials.

The GetCompanyList method retrieves the list of all available companies. The method uses a CompanyCriteria object along with a BetweenRestrictionOfNullableOfInt32 restriction object to retrieve a list of company summary objects.

The GetCustomerList method retrieves a list of customers for a specified company. The company is designated by setting the OrganizationKey property of the Context object. The method uses a CustomerCriteria object and a LikeRestrictionOfString object to retrieve the list of customer summary objects.

The GetSalesDocumentListAsync method retrieves a list of sales documents for the specified customer. The method uses a SalesDocumentCriteria object with a ListRestrictionOfNullableOfSalesDocumentType restriction object and a LikeRestrictionOfString object to retrieve the requested sales document summary objects. The asynchronous method ensures the user interface remains responsive even when retrieving the sales document list becomes a lengthy operation.

A GetSalesDocumentListCompletedEventHandler object specifies the method that receives the response to the asynchronous GetSalesDocumentListAsync method. The method wsDynamicsGP_GetSalesDocumentListCompleted places the data supplied by the Dynamics GP service into a dataset and updates the user interface data grid.
Chapter 34: Sales Documents (Native)

The Sales Document sample is a C# .NET application that uses the native endpoint to retrieve data using Web Services for Microsoft Dynamics GP. The application displays sales document summary information for a customer. The following topics are discussed:

- Overview of Sales Documents sample
- Running the Sales Documents sample
- How the Sales Documents sample works
- Dynamics GP service use for Sales Documents

Overview of Sales Documents sample

This sample application shows how methods can be used to populate the fields of a user interface. The application displays company, customer, and sales document data, and updates the displayed data to reflect choices made by the user.

Running the Sales Documents sample

To run this sample application, perform the following steps:

1. **Verify the Dynamics GP service.**
   To ensure Dynamics GP service is installed and ready, enter the service URL into the address bar of the web browser. The default URL for the native endpoint is:

   http://<machine_name>:<port number>/Dynamics/GPService

   Typically, the port number is 48620. If the browser displays a message that states you have created a service, your service is ready to use with the sample application.
2. Start Visual Studio and open the solution file for the sample application.
The solution file for this sample is named SalesDocumentSample.sln. The solution file is in the SalesDocument(native) folder inside the Samples folder.

3. Verify the .NET framework.
Open the Visual Studio Solution Explorer and right-click the SalesDocumentSample project name. From the menu, click Properties. In the Application tab of the Properties window, verify that the Target Framework shows .NET Framework 3.5.

If Target Framework shows another .NET framework, use the drop down list to specify .NET Framework 3.5.

4. Delete the existing service reference.
In the Solution Explorer, expand the Service References folder. Right-click the DynamicsGPService, and then click Delete. If a message box opens and asks whether to proceed, click OK.

5. Update the service reference.
Right-click the Service Reference folder in the Solution Explorer, and click Add Service Reference. In the Add Service Reference window, enter the URL of the native endpoint for the Dynamics GP service in the Address field, and then click Go.

In the list of Services, click Dynamics GP. Enter DynamicsGPService into the Namespace field and then click OK. The service reference is added to your project.

In the Solution Explorer, right-click DynamicsGPService, and then click Configure Service Reference. In the Client section of the Service Reference Settings window, mark Generate asynchronous operations, and then click OK.

7. Generate configuration settings.
Use svcutil.exe to generate configuration settings for your service. To run svcutil.exe open a Visual Studio Command Prompt window. Click the Start menu, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt. To use svcutil.exe you need to provide the URL of the native endpoint for the Dynamics GP service and the namespace of the service reference you just created (DynamicsGPService). Enter the following on a single line and then press Enter.

svcutil.exe http://<machine>:<port>/Dynamics/GPService/n:*,DynamicsGPService

The svcutil.exe utility creates a file named output.config in a specified folder. Note the location of the output.config file.

8. Update the app.config file
In the Visual Studio Solution Explorer, double-click app.config under SalesDocumentSample.

In a text editor, open the output.config file created in the previous step. Copy the <system.serviceModel> node from the output.config file and add it to the app.config file.
To ensure the document sample application works with the messages received from the service, you need to change the default value for attributes of several nodes in the `<wsHttpBinding>` node of the app.config file. The following XML sample shows how to increase values for the attributes of the `<binding>` node.

```xml
maxBufferPoolSize="2147483647"
maxReceivedMessageSize="2147483647"
```

The following XML sample increases the value for an attribute in the `<readerQuotas>` node

```xml
maxNameTableCharCount="2147483647"
```

Save the changes to the app.config file.

9. Choose Start Debugging from the Debug menu.
To build the solution, choose “Start Debugging” in the Debug menu. The sales document sample application loads the service proxy which may take several seconds to complete. The application will not be visible until loading the service proxy has completed.

10. Select a company from the Company drop-down list.
Once a company is selected, the Customer drop-down list contains all the customers for that company.

11. Select a customer from the Customer drop-down list.
After a customer is selected, the data grid displays all the sales documents for that customer.

12. Mark one or more of the check boxes in the Sales Documents list.
Use the check boxes to restrict the types of sales documents displayed in the data grid.

13. Click the Refresh button.
The Refresh button forces the data grid to update and display only the specified document types for the customer.

How the Sales Documents sample works

The application consists of two classes: The CustomerSalesForm user interface class manages displaying data and selecting records by the user. The DataManagement class retrieves data using service methods and makes the data available for display by the user interface class. The DataManagement class also shows how to place the summary list data returned by a method into a dataset. The user interface list and grid controls use the datasets as the source of the data they display.

The application uses basic error handling techniques to detect exceptions and displays all error information in message boxes.

Dynamics GP service use for Sales Documents

The sample application uses several Dynamics GP service objects and methods. The DataManagement class loads a proxy class named wsDynamicsGP that allows service objects and methods to be used. The application loads the wsDynamicsGP proxy class using information provided by the service.
**Web Methods**

The DataManagement class constructor creates the service proxy and creates a context object.

The GetCompanyList method retrieves the list of all available companies. The method uses a CompanyCriteria object along with a BetweenRestrictionOfNullableOfInt restriction object to retrieve a list of company summary objects.

The GetCustomerList method retrieves a list of customers for a specified company. The company is designated by setting the OrganizationKey property of the Context object. The method uses a CustomerCriteria object and a LikeRestrictionOfString object to retrieve the list of customer summary objects.

The GetSalesDocumentListAsync method retrieves a list of sales documents for the specified customer. The method uses a SalesDocumentCriteria object with a ListRestrictionOfNullableOfSalesDocumentType restriction object and a LikeRestrictionOfString object to retrieve the requested sales document summary objects. The asynchronous method ensures the user interface remains responsive even when retrieving the sales document list becomes a lengthy operation.

A GetSalesDocumentListCompletedEventHandler object specifies the method that receives the service response to the asynchronous GetSalesDocumentListAsync method. The method wsDynamics GP_GetSalesDocumentListCompleted places the data supplied by the service into a dataset and updates the user interface data grid.
Chapter 35: Import GL Transactions (Legacy)

The Import GL Transaction sample is a C# .NET application that creates new GL transactions from an XML source file. The application uses a Windows Forms-based interface to identify the XML source file and to report progress. The following topics are discussed:

- Overview of Import GL Transactions sample
- Running the Import GL Transactions sample
- How the Import GL Transactions sample works
- Dynamics GP service use for Import GL Transactions

Overview of Import GL Transactions sample

This sample application shows how to use Dynamics GP service methods from the legacy endpoint to create data in Microsoft Dynamics GP. The sample application includes a source file that contains an XML representation of three GL transactions. The application reads the XML file, creates an XML document, and uses the XML document to create new GL transactions.

The sample XML file contains only data required by this solution. This allows the sample to focus on how to use web services in a batch handling solution, and to omit steps to validate the XML and identify missing data. The sample application requires the computer to have Visual Studio 2005 or later and the 2.0 .NET framework installed.

Running the Import GL Transactions sample

To run the application, perform the following steps.

1. Verify the Dynamics GP service.
   To ensure the Dynamics GP service is installed and ready, enter the service URL into the address bar of the web browser. The default URL for the legacy endpoint is:

   http://machine_name:<port number>/DynamicsGPWebServices

   Typically, the port number is 48620. If the browser displays a message that states you have created a service, the service is ready to use with the sample application.
2. **Start Visual Studio and open the solution file for the sample application.**
   The solution file for this sample is named BatchApplication.sln. The solution file is in the ImportGLTransactions folder inside the Samples folder.

3. **Verify the .NET framework (Visual Studio 2008 and later).**
   The BatchApplication project requires the .NET Framework 2.0. To verify the target framework, open the Visual Studio Solution Explorer and right-click the BatchApplication project name. From the menu, click Properties. In the Application tab of the Properties window, verify that the Target Framework shows .NET Framework 2.0.

   If Target Framework shows another .NET framework, use the drop down list to specify .NET Framework 2.0.

4. **Update the web reference.**
   Open Visual Studio's Solution Explorer and select the Web References folder. Select the DynamicsGPService web reference to view its properties. Change the Web Reference URL property to use the URL for the legacy endpoint of the Dynamics GP service.

   If the URL for the legacy endpoint does not work, it may be due to incompatibility with the static WSDL document. Refer to [WSDL generation](#) on page 17 for more information about the WSDL document. Try the following URL, which is the direct URL for the static WSDL:

   \[ \text{http://machine_name:< port number>}/Metadata/Legacy/Full/DynamicsGP.wsdl \]

5. **Choose Start Debugging from the Debug menu.**
   To build the solution, choose “Start Debugging” in the Debug menu.

6. **Click the Open file button.**
   An open file dialog box appears. Navigate to the file named TransactionBatchFile.xml located in the BatchApplication folder found inside the ImportGLTransactions folder.

7. **Select the XML source file.**
   Select the TransactionBatchFile.xml file. The textbox on the user interface displays the contents of the XML file.

8. **Click the Begin button.**
   This button starts the batch processing of the GL transaction from the XML file.

### How the Import GL Transactions sample works

The application consists of three classes: The TransactionProcessor user interface class manages the display of data, and reports the progress of the batch. The XmlManager class opens the XML file and creates an XML document object. The DataManager class uses XML and the web services CreateGLTransaction method to create a GL transaction in Microsoft Dynamics GP.

The TransactionProcessor class handles all errors and logs error messages to a file. TransactionProcessor places the log file in the same directory as the XML source file, and names the log file TransactionProcessorErrorLog.txt.
Dynamics GP service use for Import GL Transactions

The sample application uses several Dynamics GP service objects and methods. The DataManager class loads a proxy class named wsDynamicsGP that allows service objects and methods to be used. The application loads the wsDynamicsGP proxy class using information provided by the Dynamics GP service.

Methods

The DataManager class constructor loads the service proxy and creates a context object. DataManager constructor also sets the credentials for the service proxy to use the default credentials.

The XML for each transaction specifies the company ID for the transaction. DataManager re-sets the context object OrganizationKey property during each transaction to ensure the transaction is applied to the specified company.

The CreateGLTransaction method creates the transaction in Microsoft Dynamics GP. The method uses a GL transaction object to represent the details of the transaction. The GLTransaction object requires a GLTransactionKey object to identify the new transaction.

The GL transaction object uses a collection of GLTransactionLine objects to specify the accounts for the transaction. GLTransactionLines require GLTransactionLineKey, GLAccountNumberKey, and MoneyAmount objects to specify the debit and credit accounts and the amounts for a transaction.
Chapter 36: Import GL Transactions (Native)

The Import GL Transaction sample is a C# .NET application that uses the native endpoint to create new GL transactions from an XML source file. The application uses a Windows Forms-based interface to identify the XML source file and to report progress. The following topics are discussed:

- Overview of Import GL Transactions sample
- Running the Import GL Transactions sample
- How the Import GL Transactions sample works
- Dynamics GP service use for Import GL Transactions

Overview of Import GL Transactions sample

This sample application shows how to use Dynamics GP service methods to create data in Microsoft Dynamics GP. The sample application includes a source file that contains an XML representation of three GL transactions. The application reads the XML file, creates an XML document, and uses the XML document to create new GL transactions.

The sample XML file contains only data required by this solution. This allows the sample to focus on how to use the services in a batch handling solution, and to omit steps to validate the XML and identify missing data. The sample application requires the computer to have Visual Studio 2008 or later and the 3.5 .NET framework installed.

Running the Import GL Transactions sample

To run the application, perform the following steps.

1. **Verify the Dynamics GP service.**
   To ensure the Dynamics GP service is installed and ready, enter the service URL into the address bar of the web browser. The default URL for the native endpoint is:

   \[http://<machine_name>:<port number>/Dynamics/GPService\]

   Typically, the port number is 48620. If the browser displays a message that states you have created a service, your service is ready to use with the sample application.
2. **Start Visual Studio and open the solution file for the sample application.**
   The solution file for this sample is named BatchApplication.sln. The solution file is in the ImportGLTransactions(native) folder inside the Samples folder.

3. **Verify the .NET framework.**
   Open the Visual Studio Solution Explorer and right-click the BatchApplication project name. From the menu, click Properties. In the Application tab of the Properties window, verify that the Target Framework shows .NET Framework 3.5.

   If Target Framework shows another .NET framework, use the drop down list to specify .NET Framework 3.5.

4. **Delete the existing service reference.**
   In the Solution Explorer, expand the Service References folder. Right-click the DynamicsGPService, and then click Delete. If a message box opens and asks whether to proceed, click OK.

5. **Update the service reference.**
   Right-click the Service Reference folder in the Solution Explorer, and click Add Service Reference. In the Add Service Reference window, enter the URL of the native endpoint for the Dynamics GP service in the Address field, and then click Go.

   In the list of Services, click Dynamics GP. Enter DynamicsGPService into the Namespace field and then click OK. The service reference is added to your project.

6. **Generate configuration settings.**
   Use svcutil.exe to generate configuration settings for your service. To run svcutil.exe open a Visual Studio Command Prompt window. Click the Start menu, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt. To use svcutil.exe you need to provide the URL of the native endpoint for the Dynamics GP service and the namespace of the service reference you just created (DynamicsGPService). Enter the following on a single line and then press Enter.

   ```
svcutil.exe http://<machine>:<port>/Dynamics/GPService
/n:*,DynamicsGPService
```

   The svcutil.exe utility creates a file named output.config in a specified folder. Note the location of the output.config file.

7. **Update the app.config file**
   In the Visual Studio Solution Explorer, double-click app.config under BatchApplication.

   In a text editor, open the output.config file created in the earlier step. Copy the `<system.serviceModel>` node from the output.config file and add it to the app.config file.

   To ensure the document sample application works with the messages received from the service, you need to change the default value for attributes of several nodes in the `<wsHttpBinding>` node of the app.config file. The following XML sample shows how to increase values for the attributes of the `<binding>` node.
maxBufferPoolSize="2147483647"
maxReceivedMessageSize="2147483647"

The following XML sample increases the value for an attribute in the <readerQuotas> node

maxNameTableCharCount="2147483647"

Save the change to the app.config file.

8. Choose Start Debugging from the Debug menu.
To build the solution, choose “Start Debugging” in the Debug menu.

9. Click the Open file button.
An open file dialog box appears. Navigate to the file named TransactionBatchFile.xml located in the BatchApplication folder found inside the ImportGLTransactions folder.

10. Select the XML source file.
Select the TransactionBatchFile.xml file. The textbox on the user interface displays the contents of the XML file.

11. Click the Begin button.
This button starts the batch processing of the GL transaction from the XML file.

How the Import GL Transactions sample works
The application consists of three classes: The TransactionProcessor user interface class manages the display of data, and reports the progress of the batch. The XmlManager class opens the XML file and creates an XML document object. The DataManager class uses XML and the services CreateGLTransaction method to create a GL transaction in Microsoft Dynamics GP.

The TransactionProcessor class handles all errors and logs error messages to a file. TransactionProcessor places the log file in the same directory as the XML source file, and names the log file TransactionProcessorErrorLog.txt.

Dynamics GP service use for Import GL Transactions
The sample application uses several Dynamics GP service objects and methods. The DataManager class loads a proxy class named wsDynamicsGP that allows service objects and methods to be used. The application loads the wsDynamicsGP proxy class using information provided by the service.

Methods
The DataManager class constructor loads the services proxy and creates a context object.

The XML for each transaction specifies the company ID for the transaction. DataManager resets the context object OrganizationKey property during each transaction to ensure the transaction is applied to the specified company.

The CreateGLTransaction method creates the transaction in Microsoft Dynamics GP. The method uses a GL transaction object to represent the details of the transaction. The GLTransaction object requires a GLTransactionKey object to identify the new transaction.
The GL transaction object uses a collection of GLTransactionLine objects to specify the accounts for the transaction. GLTransactionLines require GLTransactionLineKey, GLAccountNumberKey, and MoneyAmount objects to specify the debit and credit accounts and the amounts for a transaction.
Chapter 37: Update Customers (Legacy)

The Update Customers sample is a C# .NET application that updates existing customer information. The application uses a Windows Forms-based interface to add or update comments about a customer.

- Overview of Update Customers sample
- Running the Update Customers sample
- How the Update Customers sample works
- Dynamics GP service use for Update Customers

Overview of Update Customers sample

This sample application shows how to use web service methods from the legacy endpoint to update data in Microsoft Dynamics GP. The application provides the ability to edit the comment associated with a customer.

The comment editing process begins with the selection of a company and one of its customers. The Comment textbox displays the existing comment and allows the comment text to be edited. The Update button saves the new comment in Microsoft Dynamics GP. The sample application requires the computer to have Visual Studio 2005 or later and the 2.0 .NET framework installed.

Running the Update Customers sample

To run the application, perform the following steps.

1. **Verify the Dynamics GP service.**
   To ensure Dynamics GP service is installed and ready, enter the service URL into the address bar of the web browser. The default URL for the legacy endpoint is:

   http://machine_name:<port number>/DynamicsGPWebServices

   Typically, the port number is 48620. If the browser displays a message that states you have created a service, the service is ready to use with the sample application.

2. **Start Visual Studio and open the solution file for the sample application.**
   The solution file for this sample is named CustomerUpdate.sln. The solution file is in the UpdateCustomers folder inside the Samples folder.
3. **Verify the .NET framework (Visual Studio 2008 and later).**
   The CustomerUpdate project requires the .NET Framework 2.0. To verify the target framework, open the Visual Studio Solution Explorer and right-click the CustomerUpdate project name. From the menu, click Properties. In the Application tab of the Properties window, verify that the Target Framework shows .NET Framework 2.0.

   If Target Framework shows another .NET framework, use the drop down list to specify .NET Framework 2.0.

4. **Update the web reference.**
   Open Visual Studio’s Solution Explorer and select the Web References folder. Select the DynamicsGPService web reference to view its properties. Change the Web Reference URL property to use the URL for the legacy endpoint of the Dynamics GP service.

   If the URL for the legacy endpoint does not work, it may be due to incompatibility with the static WSDL document. Refer to [WSDL generation](#) on page 17 for more information about the WSDL document. Try the following URL, which is the direct URL for the static WSDL:

   http://machine_name:<port number>/Metadata/Legacy/Full/DynamicsGP.wsdl

5. **Choose Start Debugging from the Debug menu.**
   To build the solution, choose “Start Debugging” in the Debug menu.

6. **Click the Connect button.**
   The Connect button loads the web service proxy which may take several seconds to complete. The Company drop-down list displays the list of available companies.

7. **Select a company from the Company drop-down list.**
   Once a company is selected, the Customer drop-down list contains all the customers for that company.

8. **Select a customer from the Customer drop-down list.**
   After a customer is selected, the Comment textbox displays any existing comment information.

9. **Add or edit the comment information.**
   Edit the existing comment or add a new comment.

10. **Click the Update button.**
    The Update button saves the new comment to Microsoft Dynamics GP.

**How the Update Customers sample works**

The application consists of two classes. The CustomerUpdate user interface class manages the display of data and the selection or editing of customer data. The WebServiceManager class uses web service methods to retrieve data and to save the comment information. The WebServiceManager class also shows how to place the summary list data returned by a web method into a dataset. The user interface's drop-down list controls use the datasets as the source of the data they display.

The application uses basic error handling techniques to detect exceptions and displays all error information in message boxes.
Dynamics GP service use for Update Customers

The sample application uses several Dynamics GP service objects and methods. The WebServiceManager class loads a proxy class named wsDynamicsGP that allows service objects and methods to be used. The application loads the wsDynamicsGP proxy class using information provided by the Dynamics GP service.

Methods

The WebServiceManager class constructor loads the Web Services proxy and creates a context object. The WebServiceManager constructor also sets the credentials for the service proxy to use the default credentials.

The **GetCompanyList method** retrieves the list of all available companies. The method uses a CompanyCriteria object along with a BetweenRestrictionOfNullableOfInt32 restriction object to retrieve a list of company summary objects.

The **GetCustomerList method** retrieves a list of customers for a specified company. The company is designated by setting the OrganizationKey property of the Context object. The method uses a CustomerCriteria object and a LikeRestrictionOfString object to retrieve the list of customer summary objects.

The **GetCustomerByKey method** retrieves a single customer object. The Id property of the CustomerKey object is used to specify the customer to retrieve.

A customer object and customer key object identify the customer to be updated. The contents of the user interface’s Comment textbox populates the customer object’s Comment1 property. The **UpdateCustomer method** saves the updated customer object to the database.
Chapter 38: Update Customers (Native)

The Update Customers sample is a C# .NET application that uses the native endpoint to update existing customer information. The application uses a Windows Forms-based interface to add or update comments about a customer.

- Overview of Update Customers sample
- Running the Update Customers sample
- How the Update Customers sample works
- Dynamics GP service use for Update Customers

Overview of Update Customers sample

This sample application shows how service methods can update data in Microsoft Dynamics GP. The application provides the ability to edit the comment associated with a customer.

The comment editing process begins with the selection of a company and one of its customers. The Comment textbox displays the existing comment and allows the comment text to be edited. The Update button saves the new comment in Microsoft Dynamics GP. The sample application requires the computer to have Visual Studio 2008 or later and the 3.5 .NET framework installed.

Running the Update Customers sample

To run the application, perform the following steps.

1. Verify the Dynamics GP service.
   To ensure Dynamics GP service is installed and ready, enter the service URL into the address bar of the web browser. The default URL for the native endpoint is:

   http://<machine_name>:< port number>/Dynamics/GPService

   Typically, the port number is 48620. If the browser displays a message that states you have created a service, your service is ready to use with the sample application.

2. Start Visual Studio and open the solution file for the sample application.
   The solution file for this sample is named CustomerUpdate.sln. The solution file is in the UpdateCustomers(native) folder inside the Samples folder.
3. **Verify the .NET framework.**
Open the Visual Studio Solution Explorer and right-click the CustomerUpdate project name. From the menu, click Properties. In the Application tab of the Properties window, verify that the Target Framework shows .NET Framework 3.5.

If Target Framework shows another .NET framework, use the drop down list to specify .NET Framework 3.5.

4. **Delete the existing service reference.**
In the Solution Explorer, expand the Service References folder. Right-click the DynamicsGPService, and then click Delete. If a message box opens and asks whether to proceed, click OK.

5. **Update the service reference.**
Right-click the Service Reference folder in the Solution Explorer, and click Add Service Reference. In the Add Service Reference window, enter the URL of the native endpoint for the Dynamics GP service in the Address field, and then click Go.

In the list of Services, click Dynamics GP. Enter DynamicsGPService into the Namespace field and then click OK. The service reference is added to your project.

6. **Generate configuration settings.**
Use svcutil.exe to generate configuration settings for your service. To run svcutil.exe open a Visual Studio Command Prompt window. Click the Start menu, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt. To use svcutil.exe you need to provide the URL of the native endpoint for the Dynamics GP service and the namespace of the service reference you just created (DynamicsGPService). Enter the following on a single line and then press Enter.

```
svcutil.exe http://<machine>:<port>/Dynamics/GPService /n:*,DynamicsGPService
```

The svcutil.exe utility creates a file named output.config in a specified folder. Note the location of the output.config file.

7. **Update the app.config file**
In the Visual Studio Solution Explorer, double-click app.config under CustomerUpdate.

In a text editor, open the output.config file created in the earlier step. Copy the `<system.serviceModel>` node from the output.config file and add it to the app.config file.

To ensure the document sample application works with the messages received from the service, you need to change the default value for attributes of several nodes in the `<wsHttpBinding>` node of the app.config file. The following XML sample shows how to increase values for the attributes of the `<binding>` node.

```
<binding name="DynamicsGPServiceBinding" maxBufferPoolSize="2147483647"
maxReceivedMessageSize="2147483647">
```
The following XML sample increases the value for an attribute in the <readerQuotas> node

```
maxNameTableCharCount="2147483647"
```

Save the change to the app.config file.

8. Choose Start Debugging from the Debug menu.
   To build the solution, choose “Start Debugging” in the Debug menu.

9. Click the Connect button.
   The Connect button loads the web service proxy which may take several seconds to complete. The Company drop-down list displays the list of available companies.

10. Select a company from the Company drop-down list.
    Once a company is selected, the Customer drop-down list contains all the customers for that company.

11. Select a customer from the Customer drop-down list.
    After a customer is selected, the Comment textbox displays any existing comment information.

12. Add or edit the comment information.
    Edit the existing comment or add a new comment.

13. Click the Update button.
    The Update button saves the new comment to Microsoft Dynamics GP.

How the Update Customers sample works

The application consists of two classes. The CustomerUpdate user interface class manages the display of data and the selection or editing of customer data. The WebServiceManager class uses service methods to retrieve data and to save the comment information. The WebServiceManager class also shows how to place the summary list data returned by a method into a dataset. The user interface drop-down list controls use the datasets as the source of the data they display.

The application uses basic error handling techniques to detect exceptions and displays all error information in message boxes.

Dynamics GP service use for Update Customers

The sample application uses several Dynamics GP service objects and methods. The WebServiceManager class loads a proxy class named wsDynamicsGP that allows service objects and methods to be used. The application loads the wsDynamicsGP proxy class using information provided by the service.

Methods

The WebServiceManager class constructor loads the services proxy and creates a context object.

The GetCompanyList method retrieves the list of all available companies. The method uses a CompanyCriteria object along with a BetweenRestrictionOfNullableOfInt restriction object to retrieve a list of company summary objects.
The GetCustomerList method retrieves a list of customers for a specified company. The company is designated by setting the OrganizationKey property of the Context object. The method uses a CustomerCriteria object and a LikeRestrictionOfString object to retrieve the list of customer summary objects.

The GetCustomerByKey method retrieves a single customer object. The Id property of the CustomerKey object is used to specify the customer to retrieve.

A customer object and customer key object identify the customer to be updated. The contents of the user interface Comment textbox populates the customer object Comment1 property. The UpdateCustomer method saves the updated customer object to the database.
Chapter 39: Lead Service

The Lead Service sample shows how to use the Dynamics GP Service framework to integrate a new type of document with Web Services for Microsoft Dynamics GP.

Information about the Lead Service is divided into the following sections:

- Overview of the Lead Service
- Lead Service sample files
- Installing the sample application
- Installing the Lead Service
- Testing the native service
- Testing the legacy service
- Uninstalling the Lead Service

Overview of the Lead Service

The Lead Service is a new Windows Communication Foundation (WCF) service that creates, updates, deletes, and retrieves lead documents for the Dexterity-based sample integration application for Microsoft Dynamics GP. The sample integrating application is included with Dexterity. To perform these operations, the Lead Service adds the lead document type to the Dynamics GP Service framework and eConnect.

The Lead Service also includes the following applications:

- An application that adds security metadata for the new service to the Microsoft Dynamics Security Service.

- An application that adds policy metadata for creating, updating, and deleting leads.

Before you install the Lead Service, verify the following components are installed:

- The Fabrikam sample company and data. The procedures that follow add the Lead Service to the sample company database.

- Microsoft Dynamics GP Web Services. For information about how to install web services, see the Web Services Installation and Administration Guide.

- Visual Studio. The Lead Service sample includes several Visual Studio projects that you need to open, update, and build.
Lead Service sample files

The Web Service for Microsoft Dynamics GP SDK includes several files and Visual Studio projects you use to build and install the Lead Service. There are also applications for adding security and policy metadata for the Lead Service. The sample also includes several test projects that create applications you use to test and debug specific Lead Service components. The Lead Service sample includes the following files and folders.

<table>
<thead>
<tr>
<th>Category</th>
<th>File or folder name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata install</td>
<td>InstallLeadPolicyMetadata</td>
<td>A Visual Studio C# project that creates the application and configuration file that add policy metadata for leads. A second project named Sample.PolicyResources creates an assembly that contains the string resources used by the Dynamics Security Service Console to display policy information.</td>
</tr>
<tr>
<td></td>
<td>InstallLeadSecurityMetadata</td>
<td>A Visual Studio C# project that creates the application and configuration file that add security metadata for the Lead Service.</td>
</tr>
<tr>
<td>eConnect components</td>
<td>LeadBusinessObjects</td>
<td>Includes the SQL stored procedures and scripts that enable the WCF service to retrieve, create, update, and delete leads. A second Visual Studio C# project named TestLeadBusObjects creates an application you can use to test the SQL stored procedures for leads and the Transaction Requester update for leads.</td>
</tr>
<tr>
<td>WCF service components</td>
<td>Microsoft.Dynamics.GPSamples.Leads</td>
<td>A Visual Studio C# project that produces the document type assembly for leads. The document type assembly defines the lead document type, the lead summary, and the lead criteria for the Lead Service.</td>
</tr>
<tr>
<td></td>
<td>SampleLeadService.Contract</td>
<td>A Visual Studio C# WCFService Library project. The project produces an assembly that creates a WCF service interface and the methods that provide access to leads.</td>
</tr>
<tr>
<td></td>
<td>SampleLeadXSLT</td>
<td>A collection of XSLT files that the Microsoft Dynamics GP Service framework uses to transform XML documents. The service framework uses the XSLT to transform XML data documents for eConnect. A second project named XsltTest creates an application that produces an Xml file that contains a serialized lead object. The Xml file can be used with the Visual Studio XSLT debugger to debug the XSLT files.</td>
</tr>
<tr>
<td>WCF service testing</td>
<td>SampleLeadServiceTest(legacy)</td>
<td>A collection of Visual Studio C# projects that test each of the Lead Service methods.</td>
</tr>
<tr>
<td></td>
<td>SampleLeadServiceTest(native)</td>
<td></td>
</tr>
</tbody>
</table>
Installing the sample application

To install the Lead Service, you must first install the Dexterity-based back-office sample application. This sample application is included with Dexterity. For more information about installing the back-office sample application, see the Microsoft Dynamics GP Integration Guide that is included with Dexterity. To install the back-office sample application, complete the following procedure:

1. **Install the sample application files.**
   Copy the Application.SampleIntegratingApp.dll, and Develop.cnk files to the Microsoft Dynamics GP client install folder. The client install is typically in the location:

   C:\Program Files\Microsoft Dynamics\GP2010

2. **Start Microsoft Dynamics GP.**
   A message will be displayed asking whether you want to include new code. Click Yes. When the Login window appears, log in as the System Administrator (sa) or DYNSA.

   You must log in as the System Administrator (sa) orDYNSA the first time after installing the sample, so that the SQL tables for the application are properly created.

3. **Log into the sample company.**
   Choose to log into the sample company, Fabrikam, Inc.

4. **Verify the installation.**
   If the sample application was installed properly, the IG sample toolbar will be displayed in Microsoft Dynamics GP.

Installing the Lead Service

The Lead Service consists of several components that enable the Microsoft Dynamics GP Services framework to work with lead documents. To install the Lead Service, complete the steps in each of the following sections.

**Install the eConnect components**
To create, update, delete, and retrieve lead documents, the Lead Service uses custom eConnect SQL stored procedures and Transaction Requester components. To add leads to eConnect, complete the following procedure:

1. **Copy the SQL files to the Dynamics GP data server.**
   Open the \LeadBusinessObjects\sampleLeadBusObjects\ folder of the Leads Web Service sample. Copy the following SQL script files from the to your Dynamics GP data server:

   - sampleErrorCode.sql
   - sampleLeadCreateUpdate.sql
   - sampleLeadDelete.sql
   - sampleLeadRequester.sql

2. **Open Microsoft SQL Server Management Studio on your Dynamics GP data server.**
3. **Run the sampleLeadCreateUpdate SQL script.**

   In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied the SQL script files. Double-click sampleLeadCreateUpdate.sql. The script opens in the query window. In the list of databases, click TWO. To install the stored procedure, choose Execute from the Query menu.

4. **Run the sampleLeadDelete SQL script.**

   In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied the SQL script files. Double-click sampleLeadDelete.sql. The script opens in the query window. In the list of databases, click TWO. To install the stored procedure, choose Execute from the Query menu.

5. **Run the sampleLeadRequester SQL script.**

   In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied the SQL script files. Double-click sampleLeadRequesters.sql. The script opens in the query window. In the list of databases, click TWO. To add the Transaction Requester metadata, choose Execute from the Query menu. The script adds metadata for Leads to the eConnect_Out_Setup table of the TWO database.

6. **Run the sampleErrorCodes SQL script.**

   In the File menu, point to Open, and then choose File. From the Open File window, navigate to the folder where you copied the SQL script files. Double-click sampleErrorCodes.sql. The script opens in the query window. To add the error codes, choose Execute from the Query menu. The script adds the error codes and error messages to the taErrorCode table of the DYNAMICS database.

---

**Install the lead document type assembly**

The Lead document type assembly defines the lead, lead summary, and lead criteria objects. To install the assembly, complete the following procedure:

1. **Open the solution file with Visual Studio.**

   Using Visual Studio, open the file Microsoft.Dynamics.GP.Samples.Leads.sln.

2. **Verify the Reference for the project.**

   Use the Solution Explorer to verify the References. Reload any references that are not able to find the specified .dll file. To reload a reference, browse to the Dynamics GP web services folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

3. **Build the assembly.**

   In the Build menu, choose Build Solution. Visual Studio builds the assembly in the “\bin\debug” or “\bin\retail” folder of the project.

4. **Copy the assembly to the web services server.**

   Copy the Microsoft.Dynamics.GP.Samples.Leads.dll file to the server where you installed Web Services for Microsoft Dynamics GP. Place the assembly file in the Dynamics GP web services folder, typically found in this location:

   C:\Program Files\Microsoft Dynamics\GPWebServices
Add security metadata
To prevent the Lead Service from providing unauthorized access to lead data, add metadata to the Microsoft Dynamics Security Service. To update the Dynamics Security Service, build and run the InstallLeadSecurityMetadata.exe application.

InstallLeadSecurityMetadata adds metadata to the Dynamics Security Service that define five security operations and two tasks you use to manage access to lead operations. In addition, the application adds the operations to the Dynamics Security Service Superuser role.

To build and run the application, complete the following procedure:

1. **Open the solution file with Visual Studio.**
   Using Visual Studio, open the file named InstallLeadSecurityMetadata.sln.

2. **Verify references for the project.**
   Use Solution Explorer to verify the References. Reload any references that are not able to find the specified .dll file. To reload a reference, browse to the Dynamics GP web services folder, typically found in the following location:
   C:\Program Files\Microsoft Dynamics\GPWebServices

3. **Build the application.**
   In the Build menu, choose Build Solution. Visual Studio builds the application file and the application configuration file in the “\bin\debug” or “\bin\retail” folder of the project.

4. **Install the application and application configuration files.**
   Find the following application and application configuration files:
   - InstallLeadSecurityMetadata.exe
   - InstallLeadSecurityMetadata.exe.config

   Copy both files to the Dynamics GP web service folder, typically found in this location:
   C:\Program Files\Microsoft Dynamics\GPWebServices

5. **Run the application.**
   To run the InstallLeadSecurityMetadata application, you must be logged in as a user that has permission to update the Dynamics Security Service. The permissions required for this login depend upon the security store you selected when you installed Web Services for Microsoft Dynamics GP. You must use a login that has permission to update your specified security store.
The following table describes the permissions required by each security store.

<table>
<thead>
<tr>
<th>Security store</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Directory</td>
<td>Log in as user who is an ADAM administrator. If you run the InstallLeadSecurityMetadata application using a login that is not an ADAM administrator, ADAM returns a “System.UnauthorizedAccessException” error. By default, the user who installed Web Services for Microsoft Dynamics GP is an ADAM administrator. To add another user as an ADAM administrator, see the Web Services Installation and Administration Guide.</td>
</tr>
<tr>
<td>Active Directory Application Mode (ADAM)</td>
<td>Log in as a user with permission to open and manage the Active Directory authorization store. Specify the login you use when you open the Authorization Manager console window to view and update application permissions in your Active Directory authorization store.</td>
</tr>
<tr>
<td>SQL</td>
<td>Log in as a user with SQL administrator permissions for the database you use to store Dynamics Security Service metadata. The login must have permission to create records in the underlying security tables.</td>
</tr>
</tbody>
</table>

Open a command prompt and change the working location to the folder where you installed InstallLeadSecurityMetadata.exe.

Type the following command and press Enter:

```
InstallLeadSecurityMetadata.exe /load
```

6. **View the security metadata changes (optional).**

To verify that the InstallLeadSecurityMetadata application succeeded, open the Dynamics Security Console, expand the Microsoft Dynamics GP Web Services node, click Roles, click the Superuser role, and then click Properties. Verify the list of operations includes the Create Leads, Delete Leads, Modify Leads, Query Leads, and View Leads operations.

**Add policy metadata**

The Lead Service includes methods that create, update, and delete leads. The Dynamics GP WCF service requires a policy object for each of these operations. To add policy metadata, build and run the InstallLeadPolicyMetadata.exe application.

InstallLeadPolicyMetadata adds policies for creating, updating, and deleting a lead. The application also adds a behavior to the create policy that allows you to specify a lead source for a qualified lead. The sample includes a resource assembly that contains the labels displayed for the policy items in the Dynamics Security Console.

To build and run the application, complete the following procedure:

1. **Open the solution file with Visual Studio.**
   Using Visual Studio, open the file named InstallLeadPolicyMetadata.sln.

2. **Verify references for the project.**
   Use Solution Explorer to verify the References. Reload any references that are not able to find the specified .dll file. To reload a reference, browse to the Dynamics GP web services folder, typically found in the following location:

   `C:\Program Files\Microsoft Dynamics\GPWebServices`
3. **Build the application.**
In the Build menu, choose Build Solution. Visual Studio builds the application file and the application configuration file in the “\bin\debug” or “\bin\retail” folder of the project.

4. **Install the application and application configuration files.**
Find the following application and application configuration files:

- InstallLeadPolicyMetadata.exe
- InstallLeadPolicyMetadata.exe.config

Copy both files to the Dynamics GP web service folder, typically found in this location:

C:\Program Files\Microsoft Dynamics\GPWebServices

5. **Install the lead policy resource assembly.**
The solution includes a second project named Sample.PolicyResources. Copy the Sample.PolicyResources.dll assembly in the “\bin\debug” or “\bin\retail” folder of the project to the Dynamics GP web services folder, typically found in the following location:

C:\Program Files\Microsoft Dynamics\GPWebServices

6. **Check the security permissions of the lead policy resource assembly.**
Verify that the Sample.PolicyResources assembly grants Read permission to the user account that is being used as the log on identity for the Microsoft Dynamics GP Service Host service.

   To check security permissions for the resource assembly, open the file properties window and click the Security tab. Verify that the groups, users, and permissions specified for the Sample.PolicyResources assembly are the same as those specified for the Microsoft.Dynamics.GP .DataAccess assembly.

7. **Run the application.**
Open a command prompt and change the working location to the folder where you installed the InstallLeadPolicyMetadata application. Use command line parameters to install the policy metadata. The second command line parameter specifies the Id of the sample company. Type the following command and press Enter:

   InstallLeadPolicyMetadata.exe /load -1

8. **View the policy metadata changes (optional).**
To verify that the InstallLeadPolicyMetadata application succeeded, open the Dynamics Security Console, expand the Microsoft Dynamics GP Web Services node, expand the Policy node, and then click Leads. The console lists policies named Create Lead Policy, Delete Lead Policy, and Update Lead Policy.
Install the service components
The Leads service includes an assembly that defines the interface for the service. To install these components, complete the following procedure:

1. **Open the solution file with Visual Studio.**

2. **Verify references for the project.**
   Use Solution Explorer to verify the References. Reload any references that are not able to find the specified .dll file. To reload a reference, browse to the Dynamics GP web services folder, typically found in the following location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

3. **Verify the base address in the service configuration file**
   The SampleLeadService.Contract project includes a file named SampleLeadService.config. The file includes configuration information for the endpoints of the native and legacy implementations of the Leads service.

   Verify that the machine name and port number of the baseAddress attributes in the configuration file identify the server and port for the native and legacy endpoints of the Leads service. The following example shows the format of the baseAddress for the native endpoint.

   ```xml
   <add baseAddress="http://<machine_name>:<port_number>/
   SampleLeadService.Contract/LeadService" />
   ```

4. **Build the application.**
   In the Build menu, choose Build SampleLeadService.Contract. Visual Studio builds the service SampleLeadService.Contract.dll assembly in the “\bin\debug” or “\bin\retail” folder of the project. Visual Studio also places the file SampleLeadService.config in the “\bin\debug” or “\bin\retail” folder of the project

5. **Install the service assembly.**
   Copy the SampleLeadService.Contract.dll assembly to the server where you installed Web Services for Microsoft Dynamics GP. Place the SampleLeadService.Contract.dll assembly in the Dynamics GP web service folder, typically found in this location:

   C:\Program Files\Microsoft Dynamics\GPWebServices

6. **Install the service configuration file.**
   Copy the SampleLeadService.config file to the server where you installed Web Services for Microsoft Dynamics GP. Place the SampleLeadService.config file in the Dynamics GP web service “ServiceConfigs” folder, typically found in this location:

   C:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs
7. **Edit the service host configuration file**

To enable a web service client to connect to the Leads service, the service must be hosted by the Microsoft.Dynamics.GP.ServiceHost.exe service. To add a service to this host, add information about the Leads service to the Microsoft.Dynamics.GP.ServiceHost.exe.config file. The configuration file is typically found in this location:

```
C:\Program Files\Microsoft Dynamics\GPWebServices
```

To ensure you can restore the web service to its original functionality, create a copy of the `Microsoft.Dynamics.GP.ServiceHost.exe.config` file. Save the copy to a safe location.

Open the `Microsoft.Dynamics.GP.ServiceHost.exe.config` file in a text editor. Add the following `<service>` elements to the `<services>` node:

```xml
<service
    name="SampleLeadService"
    serviceType="SampleLeadService.Contract.Leads,
                SampleLeadService.Contract,
                Version=1.0.0.0,
                Culture=neutral"
    ConfigurationFileLocation="ServiceConfigs\SampleLeadService.config"
/>
<service
    name="SampleLeadServiceLegacy"
    serviceType="SampleLeadService.LegacyContract.Leads,
                SampleLeadService.Contract,
                Version=1.0.0.0,
                Culture=neutral"
    ConfigurationFileLocation="ServiceConfigs\SampleLeadService.config"
/>
```

8. **Install the XSLT files.**

The Leads service sample includes a Visual Studio solution named `SampleLeadXSLT` that contains the XSLT files for the Leads service. Find the following files:

- `Microsoft.Dynamics.GP.Samples.Lead.LeadCreate.xslt`
- `Microsoft.Dynamics.GP.Samples.Lead.LeadDelete.xslt`
- `Microsoft.Dynamics.GP.Samples.Lead.LeadGetByKey.xslt`
- `Microsoft.Dynamics.GP.Samples.Lead.LeadGetList.xslt`
- `Microsoft.Dynamics.GP.Samples.Lead.LeadUpdate.xslt`

Copy the files to the Dynamics GP web service “XSLT” folder, typically found in this location:

```
C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT
```

9. **Restart the Microsoft Dynamics GP Service Host.**

To add the Leads service, you must restart the Microsoft Dynamics GP Service Host. Open the Services console, click the service named Microsoft Dynamics GP Service Host. and then stop and restart the service.
Testing the native service

To test the native endpoint of the Lead Service, the SDK includes a collection of test applications you use to verify each method. To use these test applications, you must first log in as a user that was assigned the Superuser role in the Dynamics Security Console. To verify the Leads service, complete the following procedure:

1. **Open the solution file with Visual Studio.**
   Using Visual Studio, open the file named SampleLeadServiceTest.sln in the SampleLeadServiceTest(native) folder.

2. **Verify the References for each project.**
   Use Solution Explorer to verify the References for each test project. Reload any references that are not able to find the specified assembly.

3. **Add a Service Reference to each project.**
   Use the Solution Explorer to add a Service Reference to the Leads service for each test project. Right click the project name and click Add Service Reference. In the Address box of the Add Service Reference specify the URL of the Leads service:

   http://machine_name:port_number/Sample/Leads

   In the Namespace box, enter SampleLeadService. Click OK.

4. **Build the test applications.**
   In the Build menu, choose Build Solution. Visual Studio builds a separate console application for each project. The application files are in the “\bin\debug” or “\bin\retail” folder of each project.

5. **Run each test application.**
   To test each method, click the corresponding project folder in Solution Explorer. From the Debug menu, choose Start Debugging to run the application. To ensure the required data exists for each test application, run the test applications in the following order.

<table>
<thead>
<tr>
<th>Test application</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateLeadTest</td>
<td>Creates a new lead. Use the Dynamics GP client to verify that the new lead was created.</td>
</tr>
<tr>
<td>UpdateLeadTest</td>
<td>Updates the contact information for the lead created by CreateLeadTest. Use the Dynamics GP client to verify the change to the contact information of the lead.</td>
</tr>
<tr>
<td>DeleteLeadTest</td>
<td>Removes the lead that was created by CreateLeadTest. Use the Dynamics GP client to verify the lead was deleted.</td>
</tr>
<tr>
<td>GetLeadTest</td>
<td>Retrieves a single lead record. The application displays the name of a lead in the console window. GetLeadTest retrieves a single lead using the sample data provided for the Lead Maintenance sample application.</td>
</tr>
<tr>
<td>GetListTest</td>
<td>Retrieves a list of leads. The application displays the number of leads in the list. GetListTest retrieves a collection of leads from the sample data provided for the Lead Maintenance sample application.</td>
</tr>
</tbody>
</table>
Testing the legacy service

To test the legacy endpoint of the Lead Service, the SDK includes a collection of test applications you use to verify each method. To use these test applications, you must first log in as a user that was assigned the Superuser role in the Dynamics Security Console. To verify the Leads service, complete the following procedure:

1. **Open the solution file with Visual Studio.**
   Using Visual Studio, open the file named SampleLeadServiceTest.sln in the SampleLeadServiceTest(legacy) folder.

2. **Verify the References for each project.**
   Use Solution Explorer to verify the References for each test project. Reload any references that are not able to find the specified assembly.

3. **Add a Web Reference to each project.**
   Use the Solution Explorer to add a Web Reference to the Leads service for each test project. Right click the project name and click Add Web Reference. In the Add Web Reference window, use the URL box to specify the Leads service:

   http://machine_name:port_number/DynamicsGPWebServices/
   SampleLeadWebService.asmx

   In the Web reference name box, enter SampleLeadService. Click Add Reference.

4. **Build the test applications.**
   In the Build menu, choose Build Solution. Visual Studio builds a separate console application for each project. The application files are in the “\bin\debug” or “\bin\retail” folder of each project.

5. **Run each test application.**
   To test each service method, click the corresponding project folder in Solution Explorer. From the Debug menu, choose Start Debugging to run the application. To ensure the required data exists for each test application, run the test applications in the following order.

<table>
<thead>
<tr>
<th>Test application</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateLeadTest</td>
<td>Creates a new lead. Use the Dynamics GP client to verify that the new lead was created.</td>
</tr>
<tr>
<td>UpdateLeadTest</td>
<td>Updates the contact information for the lead created by CreateLeadTest. Use the Dynamics GP client to verify the change to the contact information of the lead.</td>
</tr>
<tr>
<td>DeleteLeadTest</td>
<td>Removes the lead that was created by CreateLeadTest. Use the Dynamics GP client to verify the lead was deleted.</td>
</tr>
<tr>
<td>GetLeadTest</td>
<td>Retrieves a single lead record. The application displays the name of a lead in the console window. GetLeadTest retrieves a single lead using the sample data provided for the Lead Maintenance sample application.</td>
</tr>
<tr>
<td>GetListTest</td>
<td>Retrieves a list of leads. The application displays the number of leads in the list. GetListTest retrieves a collection of leads from the sample data provided for the Lead Maintenance sample application.</td>
</tr>
</tbody>
</table>
Uninstalling the Lead Service

Removing the Lead Service requires you to delete the files that were added during the install process and to remove the policy and security metadata. To remove the Lead Service, complete the following procedure:

1. **Remove the policy metadata.**
   Open a command prompt and change the working location to the folder where you installed the InstallLeadPolicyMetadata application. Type the following command and press Enter. Notice how the second command line parameter specifies the Id of the sample company:

   ```
   InstallLeadPolicyMetadata.exe /remove -1
   ```

2. **Delete the InstallLeadPolicyMetadata.exe and configuration files.**
   Go to the folder where you installed the InstallLeadPolicyMetadata application, typically found in this location:

   `C:\Program Files\Microsoft Dynamics\GPWebServices`

   Delete the following files.
   
   - InstallLeadPolicyMetadata.exe
   - InstallLeadPolicyMetadata.exe.config

3. **Remove the security metadata.**
   From the command prompt, change the working location to the folder where you installed the InstallLeadSecurityMetadata application. Type the following command and press Enter:

   ```
   InstallLeadSecurityMetadata.exe /remove
   ```

4. **Delete the InstallLeadSecurityMetadata.exe application and configuration files.**
   Go to the folder where you installed the InstallLeadSecurityMetadata application, typically found in this location:

   `C:\Program Files\Microsoft Dynamics\GPWebServices`

   Delete the following files:
   
   - InstallLeadSecurityMetadata.exe
   - InstallLeadSecurityMetadata.exe.config

5. **Delete the lead service assemblies.**
   Go to the Dynamics GP web services folder typically found in this location:

   `C:\Program Files\Microsoft Dynamics\GPWebServices`

   Delete the following files:
   
   - Microsoft.Dynamics.GP.Samples.Leads.dll
   - Sample.PolicyResources.dll
   - SampleLeadService.Contract.dll
6. **Delete the lead service configuration file.**
   Go to the Dynamics GP web services configuration folder typically found in this location:
   
   C:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs
   
   Delete the following file:
   
   - SampleLeadService.Contract.config

7. **Restore the Service Host configuration file**
   Remove the Lead Service configuration information from the Microsoft.Dynamics.GP.ServiceHost.exe.config file. The host configuration file is typically found in this location:
   
   C:\Program Files\Microsoft Dynamics\GPWebServices
   
   *If you saved a copy of the Microsoft.Dynamics.GP.ServiceHost.exe.config file, you can replace the existing file with the original version.*
   
   Open the configuration file with a text editor, remove the <service> elements with the names SampleLeadService and SampleLeadServiceLegacy. Save the changes and close the file.

8. **Delete the lead XSLT files.**
   Go to the Dynamics GP web services “XSLT” folder typically found in this location:
   
   C:\Program Files\Microsoft Dynamics\GPWebServices\XSLT
   
   Delete the following files:
   
   - Microsoft.Dynamics.GP.Samples.Lead.LeadCreate.xslt
   - Microsoft.Dynamics.GP.Samples.Lead.LeadDelete.xslt
   - Microsoft.Dynamics.GP.Samples.Lead.LeadGetByKey.xslt
   - Microsoft.Dynamics.GP.Samples.Lead.LeadGetList.xslt
   - Microsoft.Dynamics.GP.Samples.Lead.LeadUpdate.xslt

9. **Restart the Microsoft Dynamics GP Service Host.**
    Restart the Microsoft Dynamics GP Service Host. Open the Services console, click the service named Microsoft Dynamics GP Service Host, and then stop and restart the service.
Appendix

The following appendix is included for this manual:

- **Appendix A, “Culture Codes.”** lists the culture codes used for the Context object in the Dynamics GP web service.

- **Appendix B, “ISO Currency Codes.”** lists the standard 3-letter ISO currency codes used to identify currency types in the Dynamics GP web service.

- **Appendix C, “Troubleshooting.”** provides information about resolving issues you may encounter with Web Services for Microsoft Dynamics GP.
Appendix A:   Culture Codes

The following chart lists the culture codes commonly used for the Context object in the Dynamics GP web service.

<table>
<thead>
<tr>
<th>Culture</th>
<th>Language-Country/Region</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>invariant culture</td>
</tr>
<tr>
<td>de-DE</td>
<td>German - Germany</td>
</tr>
<tr>
<td>da-DK</td>
<td>Danish - Denmark</td>
</tr>
<tr>
<td>nl-BE</td>
<td>Dutch - Belgium</td>
</tr>
<tr>
<td>nl-NL</td>
<td>Dutch - The Netherlands</td>
</tr>
<tr>
<td>en-AU</td>
<td>English - Australia</td>
</tr>
<tr>
<td>en-CA</td>
<td>English - Canada</td>
</tr>
<tr>
<td>en-NZ</td>
<td>English - New Zealand</td>
</tr>
<tr>
<td>en-GB</td>
<td>English - United Kingdom</td>
</tr>
<tr>
<td>en-US</td>
<td>English - United States</td>
</tr>
<tr>
<td>fr-CA</td>
<td>French - Canada</td>
</tr>
<tr>
<td>fr-FR</td>
<td>French - France</td>
</tr>
<tr>
<td>pl-PL</td>
<td>Polish - Poland</td>
</tr>
<tr>
<td>pt-BR</td>
<td>Portuguese - Brazil</td>
</tr>
<tr>
<td>es-CO</td>
<td>Spanish - Colombia</td>
</tr>
<tr>
<td>es-MX</td>
<td>Spanish - Mexico</td>
</tr>
<tr>
<td>es-ES</td>
<td>Spanish - Spain</td>
</tr>
</tbody>
</table>
## Appendix B: ISO Currency Codes

The following chart lists the standard 3-letter ISO currency codes used to identify currency types in the Dynamics GP web service.

<table>
<thead>
<tr>
<th>ISO Code</th>
<th>Country/Region</th>
<th>Currency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUD</td>
<td>Australia</td>
<td>Dollars</td>
</tr>
<tr>
<td>CAD</td>
<td>Canada</td>
<td>Dollars</td>
</tr>
<tr>
<td>EUR</td>
<td>European Union</td>
<td>Euros</td>
</tr>
<tr>
<td>JPY</td>
<td>Japan</td>
<td>Yen</td>
</tr>
<tr>
<td>MXN</td>
<td>Mexico</td>
<td>Pesos</td>
</tr>
<tr>
<td>NZD</td>
<td>New Zealand</td>
<td>Dollars</td>
</tr>
<tr>
<td>PLN</td>
<td>Poland</td>
<td>Zlotych</td>
</tr>
<tr>
<td>SGD</td>
<td>Singapore</td>
<td>Dollars</td>
</tr>
<tr>
<td>ZAR</td>
<td>South Africa</td>
<td>Rand</td>
</tr>
<tr>
<td>GBP</td>
<td>United Kingdom</td>
<td>Pounds</td>
</tr>
<tr>
<td>USD</td>
<td>United States</td>
<td>Dollars</td>
</tr>
</tbody>
</table>
Appendix C: Troubleshooting

You may encounter problems that prevent the Dynamics GP service from working properly. The following is a list of common issues you may encounter, and possible solutions.

- **Exceptions**
- **The stored data differs from what was submitted**
- **Performance**
- **Dynamics GP service does not respond**
- **Security**
- **Policy**

**Exceptions**

The following exceptions may occur when working with the Dynamics GP service:

**An unhandled system exception occurs**

A system exception occurs when an unexpected event prevents the normal completion of a method in the Dynamics GP service. A system exception returns the following message:

"The application encountered an unhandled system exception. Contact your system administrator for details."

The Dynamics GP Web Service Exceptions console logs the details surrounding each system exception. Refer to Chapter 14, “Exceptions,” for a description of the Exceptions console. The additional information the console provides may help identify the source of the system exception.

Another source of exception information is the system’s event logs. Use the system event viewer to open and review the system logs. Relevant errors, warnings and informational updates for the Dynamics GP service may be found in the Dynamics group of the Application log.

Additional information surrounding system exceptions or other unexpected results can be obtained by enabling tracing. Tracing logs information to a file while the service performs operations. All tracing occurs on the server that is running the Dynamics GP Service Host.

A setting in the service endpoint configuration file enables tracing. These files can be found in the ServiceConfigs folder of the Web Services for Microsoft Dynamics GP installation. This is typically found in the following location:

C:\Program Files\Microsoft Dynamics\GPWebServices\ServiceConfigs\n
To enable web service tracing, edit the configuration file for the endpoint. You will see the following <system.diagnostics> element with the settings that control tracing:
<system.diagnostics>
  <switches>
    <add name="ApplicationTraceSwitch" value="4" />  
  </switches>
  <trace autoflush="false" indentsize="4">
    <listeners>
      <add name="dynamicsListener" />  
    </listeners>
  </trace>
</system.diagnostics>

These settings will create a log file in the Logs folder of the Web Services for Microsoft Dynamics GP installation. The value assigned to the “ApplicationTraceSwitch” element enables or disables tracing. It also controls the level of detail recorded in the log. The following table contains the possible values for the trace switch:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable logging.</td>
</tr>
<tr>
<td>1</td>
<td>Log only error messages.</td>
</tr>
<tr>
<td>2</td>
<td>Log warning messages and error messages.</td>
</tr>
<tr>
<td>3</td>
<td>Log informational messages, warning messages, and error messages.</td>
</tr>
<tr>
<td>4</td>
<td>Log all messages including informational messages, warning messages, and error messages.</td>
</tr>
</tbody>
</table>

To disable tracing, set the trace switch values to value="0".

Restart the Microsoft Dynamics GP Service Host so that the change to the configuration file that enables tracing will take effect.

*Tracing should be disabled immediately after completing your research. If tracing is allowed to run, service performance will diminish and the log file will quickly become quite large.*

Use a text editor to view the contents of the log. The log contains a series of text messages detailing the activities performed by the web service. Look for actions and operations surrounding the error. The logs help identify the actions of the service leading to the exception and may contain additional exception information that is not available in the exception console.

**The security object does not exist**

When viewing a system exception in the exception console, you may find the following message:


The message indicates the web method call attempted to access a company that does not exist. The company is specified by the OrganizationKey property of the Context object. The "Key = 20" in the example indicates the OrganizationKey was set to a Company ID of 20. This exception also occurs if the OrganizationKey is empty.

To eliminate this exception, ensure the OrganizationKey is always populated with a valid Company ID. The OrganizationKey should be null only when retrieving system-level information like the list of companies in Microsoft Dynamics GP.
**Insufficient authorization to perform this action**

When attempting to use the web service, an exception may return the message:

"Insufficient authorization to perform this action."

This exception indicates the current user does not have sufficient security authorization to perform the requested operation. Logging on as a user with the necessary security authorization should resolve the exception. Another option is to assign the current user to a role that includes the required security authorization.

This error may also occur when using the “working on behalf of another user” option. This option allows the user and role performing the operation to be different from the logged-on user. The exception occurs when the user specified by the WorkingOnBehalfOf property of the Context object does not have authorization to perform the requested operation. Use the Security console to view the role or roles assigned to the user.

The “working on behalf of another user” option may also produce this exception if the current user that is attempting to work on behalf of another user has not been granted access to the “Work on Behalf of Other Users” task. Use the Security console to determine whether the user has been assigned the “Work on Behalf of Other Users” task and to assign the role to the user.

Entity ID filtering is another possible source of this error. If the application is requesting filtered results, users can receive this error if they don’t have access to the restricted operation used for the entity ID filtering. They may also receive this error if the entity ID assignment that maps a Windows User ID to a back office object ID cannot be found.

**Business object not found**

When viewing a system exception in the exception console, you may find the following message:

"Business object not found."

This will occur when a Get or Update operation specifies an object that could not be retrieved. Check the key value used to identify the business object to ensure it specifies an object that exists in Microsoft Dynamics GP.

If the object is known to exist, check the Context used for the request. Ensure the OrganizationKey property of the Context object is using the correct Company ID for the object being retrieved.

**The stored data differs from what was submitted**

The following issues can produce data different from the properties in the submitted business object:

**Policy-related changes**

You find the data values written or updated were not what you expected after a Create or Update operation saves the object to the database. A policy and its behaviors may be controlling some characteristics of the operation. Refer to Chapter 15, “Policy,” for additional information on policies and behaviors.
In some instances, the unexpected data changes occur when a specific policy instance associated with the user’s role does not run. Use the Dynamics Security console to view the user’s role assignments. If the user is assigned to more than one role, the web services will use the system’s default policy instead of a specific policy instance associated with an assigned role.

Review the specific behaviors associated with a policy by opening the Dynamics Security console and navigating to the policy. The middle-pane of the console will list the behaviors for the policy. Highlight a policy and select Properties from the Action menu in the right-pane. The Policy Instance Property window will display the behavior and behavior options. View the behavior options to determine whether the policy is the source of specific data change.

Check the policy for a behavior option with a behavior type of External. An external policy allows the application calling the web method to specify a behavior option. If the application uses an external behavior, the application’s behavior choice produces data that was not submitted by the user.

eConnect tracing may help by showing the actual data sent to Dynamics GP. Refer to the eConnect documentation for details about using eConnect tracing. Review the eConnect trace log to see the data eConnect used to create or update the record in the Dynamic GP database.

**Extension-related changes**

Another source of unexpected data changes includes extensions to the Dynamics GP service. Refer to Part 4, *Extending the Web Service*, for additional information on extensions. Other applications may react to the update or create event and add or change object properties. Review the BusinessObjectFile.config file to determine whether any assemblies are registered for the events associated with a create or update operation.

**Performance**

The following are suggestions for ensuring optimum performance from the Dynamics GP service:

**Ensure optimum performance**

To ensure optimum performance from Get List operations, whenever possible use object or summary object properties that correspond to indexed columns in the Dynamics GP database. The Dynamics GP Web Service Reference online help file specifies the properties that are indexed.

**Minimize the number of service instances**

Applications using Dynamics GP services should create proxy instances only when necessary. Creating a new instance of the service and loading the proxy can be a lengthy operation. The best performing applications avoid repeatedly deleting and re-instantiating the service proxy object.

For most applications, you should be able to instantiate a single proxy object that can be used throughout the lifetime of the application. To work with more than one company, change the OrganizationKey of the Context object or create a separate Context object to specify each company.
Optimized proxy
The Dynamics GP service contains a large number of methods. The proxy generated when you create a web reference or service reference is quite large. Loading this large proxy at runtime can take a significant time. Most applications that integrate with the Dynamics GP service use method only from a specific functional area. In many cases, they can use an optimized proxy to improve performance. Refer to Chapter 19, “Optimized Proxy,” for details about how to do this.

Dynamics GP service does not respond
The following issues can cause the Dynamics GP service to stop responding:

Faulted state
The native endpoint of the Dynamics GP service may enter a faulted state, which means that it will no longer process requests. The most common cause for this is applications that do not close their connection with the native endpoint when they have finished accessing it. Always be sure to use the Close() method for the proxy object when your application no longer needs to use the native endpoint.

Extensions
Dynamics GP service extensions require changes to the BusinessObjectsFile.config to register the extension for a Dynamics GP service event. If the edit creates an error in the contents of the configuration file, the Dynamics GP service may no longer respond.

Always make a backup copy of the BusinessObjectsFile.config prior to editing the file. Store the copy to a safe location. Use the backup copy to restore the Dynamics GP service if problems occur.

If changes to the BusinessObjectsFile.config prevent a method from responding, open the Exception Management Console to identify the source of the error. Edit the BusinessObjectsFile.config to correct the specified error. Restart the Microsoft Dynamics GP Service Host to ensure the changed BusinessObjectsFile.config is used.

Refer to Chapter 23, “Registering Events,” for additional information about the business object configuration file.

Configuration file changes
Changes made to the configuration files for the service endpoints, the binding definitions, and the service host can cause the Dynamics GP service to stop responding. If an error is introduces while editing a configuration file, the Dynamics GP service may become unavailable.

Always make a backup copy of the configuration file prior to editing the file. Store the copy to a safe location. Use the backup to restore the file if problems occur.
Security

The following is a list of issues associated with the Dynamics Security Administration service:

**A security authorization change is not used**
A change occurs to a user’s security authorization to add or restrict access to a service operation. Testing by that user reveals the ability to perform the specified operation remains unchanged.

To optimize the responsiveness of the Dynamics GP service, a memory cache stores the security settings. The security service reloads the cache at 20 minute intervals. Changes to security authorization will not take effect until they are loaded into the security cache.

If testing of a new security authorization change does not immediately show the expected result, re-test the operation after 20 minutes. The delay allows the security service to update its security cache with your change. Restarting the Microsoft Dynamics GP Service Host can force an immediate reload of the security cache. This should be performed only after careful consideration of the impact it will have on current Dynamics GP service users.

**The security service is not working**
The Dynamics Security Administration service uses two system logs to record error and warning messages. The Dynamics and ADAM(DynamicsSecurityService) logs contain error and warning messages associated with the Dynamics Security Administration service. If the security service is not running or is producing error messages, use the system event viewer to find detailed errors or warnings messages that specify the source of the problem.

Policy

The following is an issue that occurs when using policies with the Dynamics GP service:

**The expected policy is not used**
The Create, Update, and Delete operations can use a policy to control the characteristics of an operation. The user role determines the specific policy instance used by the operation.

If an operation does not use the expected policy, view the user’s role assignments in the Dynamics Security console. A user that has more than one role will always use the default policy for the operation. To ensure a specific policy is used with an operation, assign the user to a single role.
Glossary

Behavior
One characteristic for the service operation being performed. A policy object has a collection of behaviors.

Behavior option
A specific configuration option associated with a behavior.

Criteria object
An object that contains the restrictions that define what is to be returned from a GetList method in the Dynamics GP service.

Document type assembly
A Microsoft .NET assembly that contains classes that define a document type for the Dynamics GP Service framework.

Dynamics GP Service framework
The components that enable the Dynamics GP Service and other services to perform operations using data in the Microsoft Dynamics GP database.

eConnect
A set of SQL stored procedures and supporting code used by integrating applications to access data in Microsoft Dynamics GP.

Entity ID assignments
For the Dynamics GP service, the things that assign Windows User IDs to specific objects in Microsoft Dynamics GP that have identity information. See also User-assignable business objects.

Extension assembly
An assembly that contains the processing code for a service extension that is extending one or more business objects for the Dynamics GP service.

External behavior
A behavior that can be specified by the application that is calling the method for the Dynamics GP service and passing the policy object.

Internal behavior
A behavior that can be specified by only the web service administrator.

Legacy endpoint
An endpoint for the Microsoft Dynamics GP Service Host. It uses the BasicHTTPBinding, which has better performance and default security than the legacy endpoint.

Policy
A feature of the Dynamics GP service that allows the web service administrator to control how business objects are created, updated, or deleted.

Policy instance
The set of behaviors for a policy, set a specific way for a particular role.

Proxy
A special set of classes that will act as a wrapper for the operations, objects, and enumerations defined by the service.

Resource assembly
A Microsoft .NET assembly that contains the text message and label resources that the Dynamics Security Console uses to display policy information.

SOAP
Simple Object Access Protocol. The XML-based protocol used to communicate with a web service.

Summary object
An object that contains only the most important details of the main object. For example, the customer summary object contains only the most important details of the customer object. Summary objects are returned by the GetList methods in the Dynamics GP service.

Transaction Requester
An eConnect service that retrieves XML that represents a document.

User-assignable business objects
Those objects in Microsoft Dynamics GP that have identity information and can be associated with a Windows User ID. Examples include customers or salespeople.

WCF
Windows Communication Foundation. This is a framework included in the .NET Framework that can be used to build services that allow applications to exchange data. WCF supports several standard protocols and authentication methods.

Web reference
A URL that points to service that supports ASMX-compatible operations.

Web service
A software system that provides data and services to other applications. Web services use standard Internet transport protocols such as Hypertext Transfer Protocol (HTTP) and standard XML-based document formats such as Simple Object Access Protocol (SOAP) to exchange information.
Index

A
account numbers
determining format of 38
length of 38
using in Dynamics GP service 38
actions, tracking for business objects 111
Adding Policy Metadata, chapter 223-239
Adding Security Metadata, chapter 209-222
app.config file
generating settings with SvcUtil.exe 29
sample 28
ApplicationKey, using with the security helper application 212
asynchronous methods, adding to native endpoint proxy 26
asynchronously calling methods 99

B
base objects, described 31
basic restrictions
described 61
element 61
behavior keys, creating 226
behavior option
adding behavior parameters 231
adding to the policy resource assembly 239
creating 226
defined 341
finding in policy instance 96
parameters for 97
parameters for the behavior option constructor 230
behavior parameters
adding to the policy resource assembly 239
parameters for the behavior parameter constructor 231
behaviors
adding behavior options 230
adding policy metadata 229
adding to the policy resource assembly 238
defined 341
described 12
external behaviors 12, 93
finding behavior option 96
finding in policy instance 96
setting selected behavior option 97
setting with code 96
implementing for new service operations 169
internal behaviors 12, 93
behaviors (continued)
parameters for the LoadBehavior method 230
setting selected behavior option 97
benefits, of web services 7
between restrictions
described 62
example 62
buffer sizes, configuring for native endpoint 27
BuildArrays method, using in a criteria class 206
business document objects
described 31
user-assignable 103
business logic, implementing in stored procedures 179
business object. see SQL stored procedures
BusinessObject
using in a document type 194
using with a document summary 201
BusinessObjectsFile.config
described 127, 145
editing 145
impact of Repair operation 146
modifications by other applications 146
modifying 146
structure of 145
context (continued)
role used 42
WorkOnBehalfOf 42
contract class
legacy endpoint 254
native endpoint 248
conventions, in documentation 3
Create methods
described 35, 45
description of how processed 165
element 45
implementing with SQL stored procedures 167
SQL stored procedure for 174
XSLT 271
Create policy
described 47
element 48
retrieving 47
setting behaviors 48
Created, web service event 128
Creating
sample event handler method 138
web service event 128
Creating a Document Type, chapter 191-208
Creating a New Web Service, part 158-287
Creating Objects, chapter 45-51
Creating the Service, chapter 241-258
Creating XSLT Files, chapter 259-281
credentials, for legacy endpoint service instance 24
criteria objects
creating 55
creating a class 204
defined 341
described 53
for entity action tracking 113
indexed columns 60
inheriting form the Criteria class 204
overriding the BuildArrays method 206
overriding the GetWhereClause method 207
restrictions for 61
culture codes
in Dynamics GP service context 41
list of for Context object 331
currency
see also MoneyAmount values
in Dynamics GP service context 41
ISO codes list 333
data modification action restrictions 114
data types
chapter 37-40
choice types 39
datetime values 37
decimal values 37
enumerations 38
data types (continued)
  integer values 37
  MoneyAmount values 37
  naming differences for endpoints 33
  Percent values 38
  standard types 37
  string values 37

database transactions, events included in 127

date values, empty date values 37
datetime values, described 37
decimal values described 37
empty decimal values 37

DefaultingForCreate, web service event 127
DefaultingForDelete, web service event 128
DefaultingForUpdate, web service event 128
DefaultingForVoid, web service event 128

Defining eConnectOperations, chapter 171-190
Delete methods
described 35, 77
example 77
implementing with SQL stored procedures 167
SQL stored procedure for 185

XSLT 279

Delete policy
described 79
example 79
retrieving 79
setting behaviors 79
Deleted, web service event 128
Deleting
template event handler method 144

web service event 128
Deleting or Voiding Objects, chapter 77-81
Deploying the Web Service, chapter 283-287
design patterns
  choosing which to use 160
  for new service operations 160
Designing a Service, chapter 159-169
document conversions, in Dynamics GP Service framework 168
document type
  adding a criteria class 204
  adding data members 194
  adding enumerations 208
  adding keys 198
  adding properties for each data member 195
  adding the GUID attribute 193
  classes 191
  creating a class 193
  creating a document type 192
  creating a stored procedure 174, 185

document type (continued)
  creating a stored procedure for a new document 174
  creating a summary document class 200
data members 192
defining the summary document list 203
described 191
inheriting from the BusinessObject class 194
installing the assembly 285
naming 172
required classes 191
document type assembly, defined 341
documentation, symbols and conventions 3
dynamic WSDL generation 17

Dynamics GP service architecture 9
  capabilities 8
  configurations 10
  context 41
creating a proxy for 19
  creating objects 45
data types 37
  deleting objects 77
documentation for 34
enumerations 38
framework, described 160
learning object model 33
not responding 339
object model 31
overview 7
policy 12
retrieving documentation 34
retrieving individual objects 53
retrieving list of summary objects 55
service instance for legacy endpoint 24
service instance for native endpoint 30
updating objects 65
URL 15
using existing methods in a new service 252
voiding objects 77

WSDL file 17

Dynamics GP Service framework
creating documents, illustration 164
defined 341
deleting documents, illustration 164
document conversions 168
GreatPlainsBusinessService 241
naming requirements for XSLT files 260
operations supported 160
retrieving documents, illustration 162
updating documents, illustration 164
voiding documents, illustration 164

Dynamics GP Web Service Reference, described 34
Dynamics GP Web Services Exceptions console, described 12
Dynamics Security Administration console
  creating entity ID assignments 103
  described 11
  illustration 11
Dynamics Security service
  adding metadata 209
  adding operations 215
  adding roles 218
  adding tasks 216
  configuration file 214
described 11
  removing metadata 220
  role assignments 218
Dynamics Security web service, troubleshooting 340

E
eConnect
  adding error codes 188
described 10
eConnect_Out table 118
locking 182
removing locks 184
services 171
SQL stored procedures 167
Transaction Requester 167, 171
use in creating new services 167
using the error reporting system 188
empty values
  for dates 37
  for decimals 37
  for integers 37
  for MoneyAmounts 37
  for Percents 38
  for strings 37
endpoints
  chapter 15-19
  choosing which to use 15
  configuring interfaces used with 120
differences between legacy and native 33
  viewing with a web browser 15
entity action tracking
  actions to track 111
criteria for actions to retrieve 113
data modification action restrictions 114
described 111
eConnect_Out table 118
enabling 111
example 115
overview 111
retrieving tracking information 113
supported business objects 111

344  W E B  S E R V I C E  P R O G R A M M E R ’ S  G U I D E
entity ID assignments
  defined 341
  described 103
entity ID filtering
  chapter 103-109
  example 106
  GetByKey operations described 104
  GetList operations described 105
  implementing 105
  limitations 106
  operations for 104
  overview 103
  roles for 104
  Scope property 105
  security settings for 104
  setting up 103
  tasks for 104
enumerations
  adding for a new document type 208
  described 38
  example 38
error codes
  adding error codes to eConnect 188
  adding error codes to the database 188
event handler argument types, list of for web service events 132
event handler methods
  event handler argument types 132
  in extension assembly 132
  properties passed with event arguments 133
  writing code for 133
event viewer, events logged for Dynamics GP service 335
EventHandlerType, for web service events 127
exceptions
  (continued)
  list of 335
  native endpoint 89
  overview 83
  system exceptions 85
  types of 83
  unhandled system exceptions 335
  validation exceptions 86
  viewing list of 83
Extending the Web Service, part 124-156
  extending web service objects
  how to extend 125
  overview 125
  parts of the extension 126
  procedure 126
  web service events 127
extension assembly
  chapter 131-144
  creating 131
  defined 341
  deploying 132
  described 126
  event handler methods 132
  namespaces required 131
  public static class for 132
  referenced assemblies 131
Extension objects, working with 149
ExtensionList collection
  for objects 125
  working with 149
external behaviors
  defined 341
  described 12, 93
  finding behavior option 96
  finding in policy instance 96
  setting selected behavior option 97
  setting with code 96
faulted state, for the native endpoint 339
filtering, see entity ID filtering
format, for account numbers 38
framework
  Dynamics GP Service framework
Get List methods
  criteria objects for 55
  described 35, 55
  examples 56, 58
  implementing with TransactionRequester 167
  optimizing performance 60
  using results 60
  with entity ID filtering 105
  XSLT 267
Get methods
  described 35, 53
  examples 53
  implementing with TransactionRequester 167
Get methods (continued)
  with entity ID filtering 104
GetByKey methods, XSLT 261
GetChanged methods 113
GetPolicyByKey, using with a new service 241
Getting Started, part 14-42
GetWhereClause method, using with a criteria class 207
GreatPlainsBusinessService
  described 241
  error messages 242
  operations 241
  parameters for a Create method 250
  parameters for a Delete method 251
  parameters for a GetByKey method 249
  parameters for a GetList method 250
  parameters for an Update method 251
  policy requirements 241
GUIDAttribute
  adding to document type class 193
  using with security operation keys 212
Helper objects, described 32
Import GL Transactions (Legacy), sample application 299
Import GL Transactions (Native), sample application 303
indexed columns, using for criteria objects 60
inheritance
  example 32
  in Dynamics GP service object model 32
insufficient authorization errors, troubleshooting 337
integer values
  described 37
  empty integer values 37
interfaces
  available to use for endpoints 120
  overview of using for optimized proxy 119
  predefined 120
internal behaviors
  defined 341
  described 12, 93
ISO codes
  for MoneyAmount values 37
  list of 333
Kerberos authentication, use with native endpoint 27
keys, adding to a new document type 198
L

Lead Service
- adding policy metadata 320
- adding security metadata 319
- file list 316
- helper applications 315
- installing the document type assembly 318
- installing the files for the Leads service 322
- installing the sample application 317
- installing the service 317
- installing the XSLT files 323
- overview 315
- prerequisites 315
- sample 315
- test applications for the legacy service 325
- test applications for the native service 324
- testing the legacy service 325
- testing the native service 324
- uninstalling the service 326

Leads service, installing service components 322
- learning the Dynamics GP service object model 33

legacy endpoint
- adding a service interface 246
- connecting to 21
- connecting to with Visual Studio 2005 21
- connecting to with Visual Studio 2008 22, 25
- defined 341
- described 9
- differences from native endpoint 33
- documentation convention 4
- handling exceptions for 86
- implementing an interface 254
- ImportGLTransactions sample 299
- namespace for proxy 24
- Sales Documents sample 291
- service instance 24
- UpdateCustomers sample 307
- URL for 15
- when to use 15
- WSDL document for 17
- light bulb symbol 3
- like restrictions
  - described 63
  - example 63
- list restrictions
  - described 61
  - example 62
- LoadBehavior method, parameters 230
- locking rows, concurrency issues for 71
- locks
  - locking rows in stored procedures 182
  - removing locks from rows in stored procedures 184
- login credentials, for legacy endpoint service instance 24

M

maps, see XSLT
- margin notes 3

metadata
- for policy 169
- for security 168
- naming document types 172
- policy 223
- security 209
- updating the Transaction Requester 171

methods
- calling asynchronously 99
- categories 35
- chapter 35-36
- Create methods 35
- Delete methods 35
- Exception methods 36
- framework methods 36
- Get methods 35
- GetList methods 35
- Policy methods 36
- Update methods 35
- Void methods 35
- MoneyAmount values
  - described 37
  - empty MoneyAmount values 37
- ISO codes for 37

N

namespace
- for legacy endpoint proxy 24
- for legacy endpoint WSDL 17
- for native endpoint proxy 29
- for native endpoint WSDL 17
- required for extension assembly 131

native endpoint
- adding an interface to a service 244
- application configuration file 27
- client settings 27
- closing the service instance 30
- configuring buffer sizes 27
- connecting to 25
- defined 341
- described 10
- differences from legacy endpoint 33
- documentation convention 4
- faulted state 339
- handling exceptions for 89
- implementing an interface 248
- ImportGLTransactions sample 303
- Kerberos authentication use 27
- namespace for proxy 29
- Sales Documents sample 295
- security settings for 27
- service instance 30
- UpdateCustomers sample 311
- URL for 16

native endpoint (continued)
- userPrincipalName 27
- when to use 15
- WSDL document for 17

O

Object Browser, in Visual Studio 19
- object model
  - for Dynamics GP service 31
  - inheritance 32
  - learning 33
  - object categories 31
- objects
  - creating 45
  - deleting 77
- ExtensionList collection 125
- filtering 103
- how to extend 125
- naming differences for endpoints 34
- retrieving individual 53
- updating 65
- voiding 77
- which can be extended 125

operations
- for entity ID filtering 104
- supported by the Dynamics GP Service framework 160

optimized proxy
- code changes required 122
- configuring endpoints 120
- creating 119
- overview 119
- predefined interfaces 120
- using in an application 121

optimizing
- Get List methods 60
- service access 119
- web service applications 338

organization, in web service context 41

Other Programming Techniques, chapter 99-102

output type parameters, using with Transaction Requester 172
- overflow, for strings 37

P

parameters, for behavior option 97
- patterns
  - choosing which to use 160
  - for new service operations 160

Percent values
- described 38
- empty Percent values 38

performance, optimizing for web service applications 338

policy
- adding to the policy resource assembly 238
- behaviors for 12, 93
- chapter 93-97
- configuring from code 94
policy (continued)
configuring with Dynamics Security Console 94
defined 341
described 12, 93
for Create operations 47
for Delete operations 79
for Update operations 69
for Void operations 79
impacting data 337
implementing for new service operations 169
listing available policies 94
overview 93
policy instances 93
reference information for 93
roles for 95
troubleshooting 340
using in an application 95
using XSLT to implement 169, 275
Policy Administrator role, described 94
Policy helper application
adding a policy helper resource assembly 237
adding behaviors to a policy 234
configuration information 228
creating a behavior option 226
creating behavior keys 226
creating behaviors 229
creating policies 233
creating policy keys 226
creating the application 225
described 223
implementing command line parameters 227
loading policy metadata 286
removing policy metadata 234
using the ServiceFactory 234
Policy instances
creating with code 95
defined 341
deleting with code 95
expressed 39
retrieving 95
roles for 95
upgrading with code 95
Policy keys, creating 226
Policy metadata
behavior options 230
behavior parameters 231
behaviors 229
component 223
installing 286
policies 233
removing 234
Policy methods, described 36
Policy object, parameters for the constructor 233
Policy resource assembly
defined 341
examples 238
policy resource assembly (continued)
installing 286
naming behavior options 239
naming behavior parameters 239
naming behaviors 238
naming policies 238
security permissions 286
specifying a root object name 238
PolicyBusinessService, using with the policy helper application 234
port, for Dynamics GP service 16
prerequisites, for using web services 3
product support, for Microsoft Dynamics GP web services 4
proxy
defined 341
described 19
for Dynamics GP service 19
optimized proxy 119
Q
queries, see Get List methods
R
Registering Events, chapter 145-148
registration example, for web service events 147
Requester Setup Tool 111
resource assembly, defined 341
restriction objects
basic restrictions 61
between restrictions 62
like restrictions 63
list restrictions 61
types of 61
used with criteria 61
retrieve methods
described 53
description of how processed 162
Retrieved
sample event handler method 134
web service event 127
Retrieving Objects, chapter 53-63
role key
automatically assigned for context 42
in Dynamics GP service context 42
roles
for entity ID filtering 104
for policy instances 95
S
Sales Documents (Legacy), sample application 291
Sales Documents (Native), sample application 295
samples
Import GL Transactions (Legacy) 299
Import GL Transactions (Native) 303
Lead Service 315
Sales Documents (Legacy) 291
Sales Documents (Native) 295
Update Customers (Legacy) 307
samples (continued)
Update Customers (Native) 311
Scope property, for criteria objects 105
security
adding security metadata 168
for stored procedures 184, 187
securing a new service 168
security helper application
adding command line parameters 214
adding security metadata 209
configuration settings 214
creating 210
creating a role key 213
creating operation keys 212
creating task keys 213
removing security metadata 220
specifying the ApplicationKey 212
updating roles 218
using the Dynamics Security Service 215
security metadata
adding new operations 215
adding new roles 218
adding new tasks 216
described 209
installing metadata 285
removing 220
roles 218
uninstalling security metadata 209
security operation keys
identification 212
using the GuidAttribute 212
security operations, listed 212
security role keys
identification 213
using the Superuser role 213
security task keys, identification 213
Security web service, see Dynamics Security web service
segments, for account numbers 38
separator character, for account numbers 39
service
adding a Create method 250
adding a Delete method 251
adding a GetByKey method 249
adding a GetList method 250
adding a legacy endpoint interface 246
adding a native endpoint interface 244
adding an Update method 251
adding Dynamic GP methods 252
adding Dynamics GP service methods 242
adding the GetPolicyByKey method 241
configuration file 256
contract for a legacy endpoint 254
contract for a native endpoint 248
create documents 164

WEB SERVICE PROGRAMMER’S GUIDE 347
service (continued)
creating a new service 242
creating your own 159
design considerations 159
designing your own 159
designs for new operations 160
document conversions 168
eConnect 167, 171
handling error messages 242
operations 241
policy 169
retrieving documents 162
security 168
transforms 259
using policy 241
using the context object 241
using the eConnect error reporting system 188

service document type
data members 166
described 166
naming 166

service reference
adding to Visual Studio 2008 project 25
described 19
in C# project 26

SOAP
defined 341
described 7
exceptions from Dynamics GP service operations 83

SQL stored procedures, see stored procedures

static WSDL generation 17

stored procedures
adding business logic 179
adding local variables 177, 186
adding output parameters 176
creating a record 181
creating for a new document type 174
deleting a new document type 185
deleting a record 187
error codes 188
identifying required parameters 176
installing new eConnect stored procedures 284
locking a record 182
naming input parameters 176
removing a lock from a record 184
specifying security permissions 184, 187
structure of the stored procedure, illustration 174
upgrading an existing record 183
use with services 167
validating input parameters 178, 186
verifying a record exists 178, 187

string values (continued)
truncation issues 37
summary document
creating a class 200
data members 201
getting a list of summary documents 203
inheriting from the BusinessObject class 201
naming 203
specifying the data type 203
summary objects
defined 341
described 53
retrieving list of 55
Superuser role, security role key 213
support, for Microsoft Dynamics GP web services 4
SvcUtil.exe
described 29
generating proxy and configuration settings 29
symbols in documentation 3
system database, specifying in web service context 41
system exceptions
described 83, 85
handling in code 86
viewing details 84

T
tasks, for entity ID filtering 104
technical support, for Microsoft Dynamics GP web services 4
timeout, specifying for web service instance 4
tracing
modes 336
to log Dynamics GP service issues 335

Transaction Requester
adding metadata for a new document type 172
adding metadata for Get operations 171
defined 341
output type parameters 172
SQL stored procedures 172
use with services 167
transforms, see XSLT
troubleshooting, Dynamics GP service 335
truncation, for strings 37

U
unhandled system exceptions 335
Update Customers (Legacy), sample application 307
Update Customers (Native), sample application 311
Update methods
concurrency issues 71
described 35, 65

Update methods (continued)
example 65
implementing with SQL stored procedures 167
SQL stored procedure for 174
XSLT 277

Update policy
described 69
example 69
retrieving 69
setting behaviors 69

Updated, web service event 128
Updating
sample event handler method 142
web service event 128

Updating Objects, chapter 65-76
URL, for Dynamics GP service 15
user IDs, for Dynamics GP objects 103
user-assignable business objects
defined 341
described 103
Using the Web Service, part 44-102
Using Web Service Extensions, chapter 149-156

V
validating, stored procedure input parameters 178
ValidatingForCreate
sample event handler method 135
web service event 127
ValidatingForDelete, web service event 128
ValidatingForUpdate
sample event handler method 140
web service event 128
ValidatingForVoid, web service event 128
validation exceptions
described 83, 86
example 86
handling in code 86
viewing details 85
Visual Studio 2005, connecting to legacy endpoint 21
Visual Studio 2008, connecting to legacy endpoint 22, 25
Void methods, described 35, 77
Void policy
described 79
retrieving 79
Voided, web service event 129
Voiding, web service event 128

W
warning symbol 3
WCF
defined 341
described 9
web methods, see methods 2
web reference
adding to Visual Studio 2005 project 21
defined 341
described 19
in C# project 22
web service
architecture 9
benefits 7
components 283
configurations 10
context 41
defined 341
deploying a new web service 283
described 7
document types 191
installing components 285
installing eConnect components 284
installing the interface files 286
installing the XSLT files 287
object model 31
overview 7
proxy 19
troubleshooting 335
URL 15
Web Service Architecture, chapter 9-12
Web Service Basics, part 6-12
web service document type, see document type
web service events
chapter 127-129
defined 341
how processed 127
list of 127
registering 127, 145
registration example 147
types of 127
web service extensions
creating or updating data 152
retrieving data from 149
using 149
Web Service Object Model, chapter 31-34
Web Service Samples, part 290-327
web services, support 4
Windows Communication Foundation, see WCF
Windows User IDs, associating with Dynamics GP objects 103
Work on Behalf of Other Users task 101
WorkOnBehalfOf property
for Context object 101
impact of entity ID filtering 106
in web service context 42
WSDL
accessing for a service 17
defined 341
dynamically-generated WSDL 17
generation 17
static WSDL 17
using a WSDL document 17

X
XML, defined 341
XML element, used for web service object extension 125
XSLT
Create methods 271
defined 341
Delete methods 279
GetByKey methods 261
GetList methods 267
implementing policy for a Create method 275
implementing policy for a Delete method 281
implementing policy for an Update method 278
implementing policy for service methods 260
installing the files 287
naming requirements for files 260
operation names 260
testing and debugging 260
Update methods 277
uses 259
using for document conversions 168
using to implement policy for new service operations 169