Module 3

Creating and Consuming ASP.NET Web API Services

Contents:

Module Overview

ASP.NET Web API provides a robust and modern framework for creating HTTP-based services. In this module, you will be introduced to the HTTP-based services. You will learn how HTTP works and become familiar with HTTP messages, HTTP methods, status codes, and headers. You will also be introduced to the REST architectural style and Hypermedia.

You will learn how to create HTTP-based services by using ASP.NET Web API. You will also learn how to host the services in IIS and how to consume them from various clients. At the end of this module, in the lab "Creating the Traveler ASP.NET Web API Service", you will build an ASP.NET Web API service application and host it in Internet Information Services (IIS).

Objectives

After you complete this module, you will be able to:

- Design services by using the HTTP protocol.
- Create services by using ASP.NET Web API.
- Use the **HttpRequestMessage**/**HttpResponseMessage** classes to control HTTP messages.
- Host and consume ASP.NET Web API services.

Lesson 1
HTTP Services

Hypertext Transfer Protocol (HTTP) is a communication protocol that was created by Tim Berners-Lee and his team while working on WorldWideWeb (later renamed to World Wide Web) project. Originally designed to transfer hypertext-based resources across computer networks, HTTP is an application layer protocol that acts as the primary protocol for many applications including the World Wide Web.

Because of its vast adoption and also the common use of web technologies, HTTP is now one of the most popular protocols for building applications and services. In this lesson, you will be introduced to the basic structure of HTTP messages and understand the basic principles of the REST architectural approach.

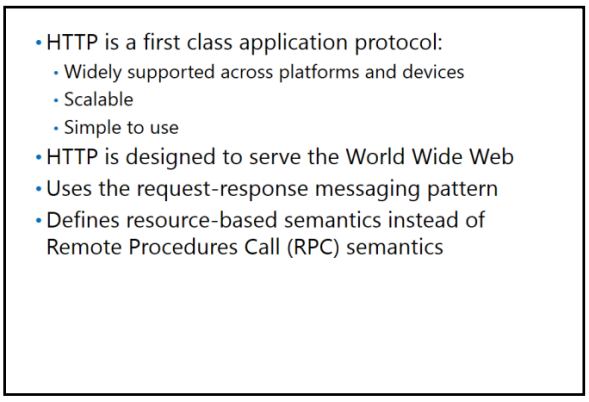
Lesson Objectives

After you complete this lesson, you will be able to:

- Explain the basic structure of HTTP.
- Explain the structure of HTTP messages.
- Describe resources by using URIs.
- Explain the semantics of HTTP verbs.
- Explain how status codes are used.
- Explain the basic concepts of REST.
- Use media types.

Introduction to HTTP

HTTP is a first class application protocol that was built to power the World Wide Web. To support such a challenge, HTTP was built to allow applications to scale, taking into consideration concepts such as caching and stateless architecture. Today, HTTP is supported by many different devices and platforms, reaching most computer systems available today.



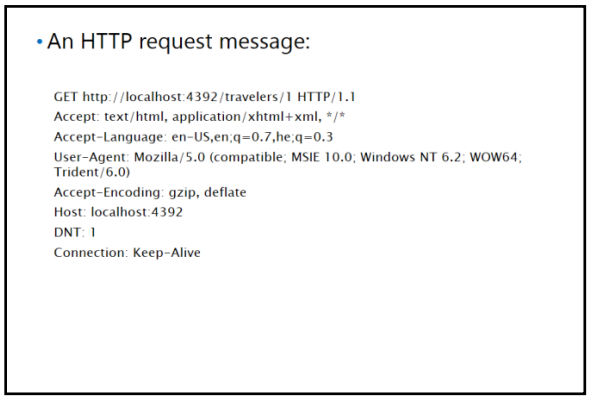
HTTP also offers simplicity, by using text messages and following the request-response messaging pattern. HTTP differs from most application layer protocols because it was not designed as a Remote Procedure Calls (RPC) mechanism or a Remote Method Invocation (RMI) mechanism. Instead, HTTP provides semantics for retrieving and changing resources that can be accessed directly by using an address.

HTTP Messages

HTTP is a simple request-response protocol. All HTTP messages contain the following elements:

- Start-line
- Headers
- An empty line
- Body (optional)

Although requests and responses share the same basic structure, there are some differences between them that you should be aware of.



Request Messages

Request messages are sent by the client to the server. Request messages have a specific structure based on the general structure of HTTP messages.

This example shows a simple HTTP request message.

An HTTP Request

```
GET http://localhost:4392/travelers/1 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:4392
DNT: 1
Connection: Keep-Alive
```

The first and the most distinct difference between request and response messages is the structure of the start-line which are called request-lines.

Request-line

This HTTP request messages start-line has a typical request-line with the following space-delimited parts:

- **HTTP method** – This HTTP request message uses the GET method, which indicates that the client is trying to retrieve a resource. Verbs will be covered in-depth in the topic **Using Verbs** later in this lesson.
- **Request URI** – This part represents the URI to which the message is being sent.
- **HTTP version** – This part indicates that the message uses HTTP version 1.1.

Headers

This request message also has several headers that provide metadata for the request. Although headers exist in both response and request messages, some headers are used exclusively by one of them. For example, the **Accept** header is used in requests to communicate the kinds of responses the clients would prefer to receive. This header is a part of a process known as *content negotiation* that will be discussed later in this module.

Body

The request message has no body. This is typical of requests that use the GET method.

Response Messages

Response messages also have a specific structure based on the general structure of HTTP messages.

This example shows a simple HTTP response message.

The HTTP Response returned by the Server for the above Request

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/11.0.0.0
Date: Tue, 13 Nov 2012 18:05:11 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8
Content-Length: 188
Connection: Close

{"TravelerId":1,"TravelerUserIdentity":"aaabbbccc","FirstName":"FirstName1","LastName":"L
astName1","MobilePhone":"555-555-5555","HomeAddress":"One microsoft
road","Passport":"AB123456789"}
```

Status-Line

HTTP response start-lines are called status-lines. This HTTP response message has a typical status-line with the following space-delimited parts:

- **HTTP version** – This part indicates that the message uses HTTP version 1.1.
- **Status-Code** – Status-codes help define the result of the request. This message returns a status-code of 200, which indicates a successful operation. Status codes will be covered in-depth later in this lesson.
- **Reason-Phrase** – A reason-phrase is a short text that describes the status code, providing a human readable version of the status-code.

Headers

Similar to the request message, the response message also has headers. Some headers are unique for HTTP responses. For example, the **Server** header provides technical information about the server software being used. The **Cache-Control** and **Pragma** headers describe how caching mechanisms should treat the message.

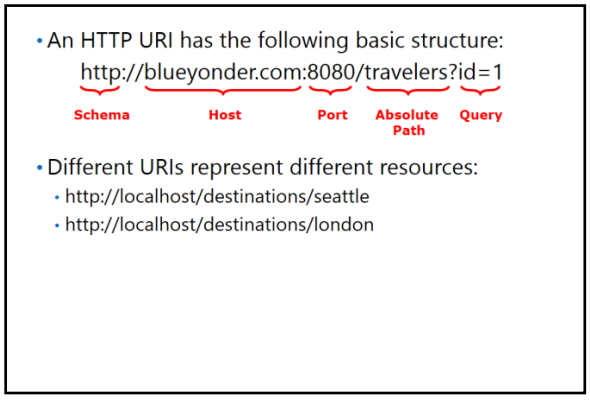
Other headers, such as the **Content-Type** and **Content-Length**, provide metadata for the message body and are used in both requests and responses that have a body.

Body

A response message returns a representation of a resource in *JavaScript Object Notation* (JSON). The JSON, in this case, contains information about a specific traveler in a traveling management system. The format of the representation is communicated by using the **Content-Type** header describing what is known as *media type*. Media types are covered in-depth later in this lesson.

Identifying Resources By Using URI

Uniform Resource Identifier (URI) is an addressing standard that is used by many protocols. HTTP uses URI as part of its resource-based approach to identify resources over the network.



HTTP URIs follow this structure:

"http://" host [":" port] [absolute path ["?" query]]

- **http://**. This prefix is standard to HTTP requests and defines the HTTP URI schema to be used.
- **Host**. The host component of the URI identifies a computer by an IP address or a registered name.
- **Port (optional)**. The port defines a specific port to be addressed. If not present, a default port will be used. Different schemas can define different default ports. The default port for HTTP is 80.
- **Absolute path (optional)**. The path provides additional data that together with the query describes a resource. The path can have a hierarchical structure similar to a directory structure, separated by the slash sign (/).
- **Query (optional)**. The query provides additional nonhierarchical data that together with the path describes a resource.

Different URIs can be used to describe different resources. For example, the following URIs describe different destinations in an airline booking system:

- http://localhost/destinations/seattle
- http://localhost/destinations/london

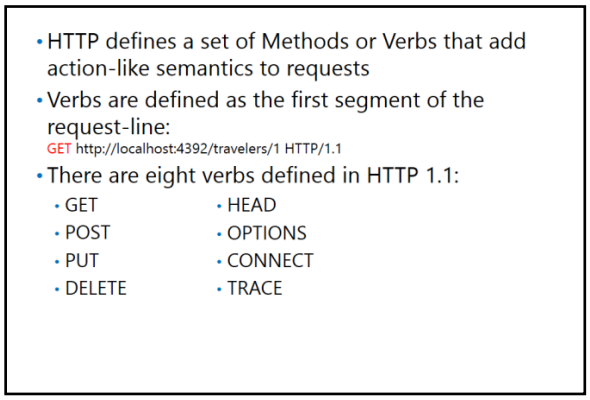
When accessing each URI, a different set of data, also known as a representation, will be retrieved.

The URI Request For Comments (RFC 3986)

http://go.microsoft.com/fwlink/?LinkID=298757&clcid=0x409

Using Verbs

HTTP defines a set of methods or verbs that add an action like semantics to requests. HTTP 1.1 defines an extensible set of eight methods, each with different behavior. For example, the following request uses the GET method to retrieve information about a specific traveler in an airline traveler system.



This example shows an HTTP GET request message.

An HTTP GET request retrieving data about a specific traveler

```
GET http://localhost:4392/travelers/1 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
```

```
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:4392
DNT: 1
Connection: Keep-Alive
```

In the above example, a method is defined in the first segment of the request-line and communicates what the request is intended to perform. For example, the GET method used in the request above communicates that the request is intending to retrieve data about an entity and not trying to modify it. This behavior makes GET compatible with both of the properties an HTTP method might have: it is both *safe* and *idempotent*.

- **Safe verbs**. These are verbs that are intended to have any side effects on the resource state by the server other than retrieving data.
- **Idempotent verbs**. These are verbs that are intended to have the same effect on the resource state when the same request is sent to the server multiple times. For example, sending a single DELETE request to delete a resource should have the same effect as sending the same DELETE request multiple times.

Verbs are a central mechanism in HTTP and one of the mechanisms that make HTTP the powerful protocol it is. Understanding what each verb does is very important for developing HTTP-based services. The following verbs are defined in HTTP 1.1:

Method	Description	Properties	Usage
GET	Requests intended to retrieve data based on the request URI.	Safe, Idempotent	Used to retrieve a representation of a resource.
HEAD	Requests intended to have the identical result of GET requests but without returning a message body.	Safe, Idempotent	Used to check request validity and retrieving headers information without having the message body.
OPTIONS	Requests intended to return information about the communication options and capabilities of the server.	Safe, Idempotent	Used to retrieve a comma-delimited list of the HTTP verbs supported by a resource or a server in the **Allow** header.
POST	Requests intended to send an entity to the server. The actual operation that is performed by the request is determined by the server. The server should return information about the outcome of the operation in result.		Used to create, update, and by some protocols, retrieve entities from the server. POST is the least structured HTTP method.
PUT	Requests intended to store the entity sent in the request URI, completely overriding any existing entity in that URI.	Idempotent	Used to create and update resources.
DELETE	Requests intended to delete the entity identified by the request URI.	Idempotent	Used to delete resources.
TRACE	Requests intended to indicate to clients	Safe,	Rarely implemented, used

Method	Description	Properties	Usage
	what is received at the server end.	Idempotent	to identify proxies the message passes on the way to the server.
CONNECT	Requests intended to dynamically change the communication protocol.	Safe, Idempotent	Used to start SSL tunneling.

For more information about HTTP methods, see the *HTTP 1.1 Request For Comments (RFC 2616)*.

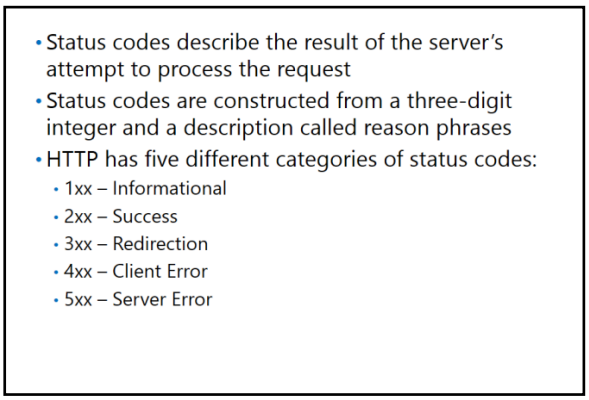
Methods definition in the HTTP 1.1 Request For Comments (RFC 2616)

http://go.microsoft.com/fwlink/?LinkID=298758&clcid=0x409

Status-Codes and Reason-Phrases

Status-codes are 3-digit integers returned as a part of response messages status-lines. Status codes describe the result of the effort of the server to satisfy the request. The next section of the status line after the status code is the reason-phrase, a human-readable textual description of the status-code.

Status codes are divided into five classes or categories. The first digit of the status code indicate the class of the status:



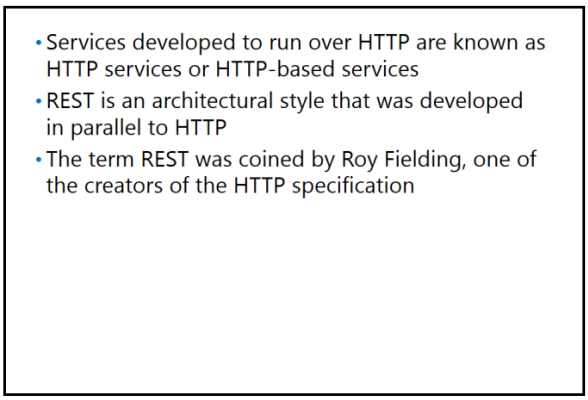
Class	Usage	Examples
1xx – Informational	Codes that return informational response about the state of the connection.	• 101 Switching Protocols
2xx – Successful	Codes that indicate the request was successfully received and accepted by the server.	• 200 OK • 201 Created
3xx – Redirection	Codes that indicate that additional action should be taken by the client (usually in respect to a different network addresses) in order to achieve the result that you want.	• 301 Moved Permanently • 302 Found • 303 See Other
4xx – Client Error	Codes that indicate an error that is caused by the client's request. This might be caused by a wrong address, bad message format, or any kind of invalid data passed in the client's request.	• 400 Bad Request • 401 Unauthorized • 404 Not Found
5xx – Server Error	Codes that indicate an error that was caused by the server while it tried to process a seemly valid request.	• 500 Internal Server • 505 HTTP Version Not Supported

For more information about HTTP status codes, see the *HTTP 1.1 Request For Comments (RFC 2616)*.

HTTP Status-Codes definition in the HTTP 1.1 Request For Comments (RFC 2616)

http://go.microsoft.com/fwlink/?LinkID=298759&clcid=0x409

Introduction to REST



Until now in this module, you have learned how HTTP acts as an application layer protocol. HTTP is used to develop both websites and services. Services developed by using HTTP are generally known as HTTP-based services.

The term Representational State Transfer (REST) describes an architectural style that takes advantage of the resource-based nature of HTTP. It was first used in 2000 by Roy Fielding, one of the authors of the HTTP, URI, and HTML specifications. Fielding described in his doctoral dissertation an architectural style that uses some elements of HTTP and the World Wide Web for creating scalable and extendable applications.

Today, REST is used to add important capabilities to service. These capabilities include the following:

- Service discoverability
- State management

In this lesson, you will learn about these capabilities. For more information about REST, see Roy Fielding's dissertation, **Architectural Styles and the Design of Network-based Software Architectures**.

Architectural Styles and the Design of Network-based Software Architectures by Roy Fielding

http://go.microsoft.com/fwlink/?LinkID=298760&clcid=0x409

Services that use the REST architectural style are also known as RESTful services. A simple way to understand what makes a service RESTful is using a taxonomy called the Richardson Maturity Model, first suggested by Leonard Richardson during his talk during the QCon San Francisco Conference in 2008.

The Richardson Maturity Model

The Richardson Maturity Model describes four levels of maturity for services, starting with the least RESTful level advancing toward fully RESTful services:

- **Level zero services**. Use HTTP as a transport protocol by ignoring the capabilities of HTTP as an application layer protocol. Level zero services use a single address, also known as endpoint and a single HTTP method, which is usually POST. SOAP services and other RPC-based protocols are examples of level zero services.
- **Level one services**. Identify resources by using URIs. Each resource in the system has its own URI by which the resource can be accessed.
- **Level two services**. Uses the different HTTP verbs to allow the user to manipulate the resources and create a full API based on resources.
- **Level three services**. Although the first two services only emphasize the suitable use of HTTP semantics, level three services introduce Hypermedia, an extension of the term Hypertext as a means for a resource to describe their own state in addition to their relation to other resources.

For more information about the Richardson Maturity Model, see Leonard Richardson's presentation and notes.

Leonard Richardson's QCon 2008 presentation and notes

http://go.microsoft.com/fwlink/?LinkID=298761&clcid=0x409

Hypermedia

When the World Wide Web started, it strongly affected the way humans consume data. Alongside abilities, such as remote access to data and the ability to search a global knowledge base, the World Wide Web also introduced Hypertext. Hypertext is a nonlinear format that enables readers to access data related to a specific part of the text by using Hyperlinks. The term Hypermedia describes a logical extension to the same concept. Hypermedia-based systems use Hypermedia elements, known as hypermedia controls, such as links and HTML forms, to enable resources to describe their current state and other resources that are related to them.

Hypermedia and Discoverability

A simple example for resource discoverability can be found in the Atom Syndication Format. At first, the Atom Syndication Format was developed as an alternative to RSS for publishing web feeds. Atom feeds are resources with their own URIs that contain items. Feed items are resources themselves with their own URIs published as links in the feed representation, which makes them discoverable for clients.

This feed describes different instances of a flight in the BlueYonder Companion app. The Hypermedia control entry is used here to refer clients to different instances of a specific flight.

A simple Atom Feed

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/atom+xml
Content-Length: 746
Connection: Close

<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Blue Yonder flights</title>
  <id>uuid:460f9be6-3503-43c5-8168-5cb86127b572;id=1</id>
  <updated>2012-11-16T21:50:17Z</updated>
  <entry xml:base="http://localhost:4392/Flights/BY002/1117">
    <id>BY002</id>
    <title type="text">Flight BY002 November 17, 2012</title>
    <updated>2012-11-16T21:50:17Z</updated>
  </entry>
  <entry xml:base="http://localhost:4392/Flights/BY002/1201">
    <id>BY002</id>
    <title type="text">Flight BY002 December 01, 2012</title>
    <updated>2012-11-16T21:50:17Z</updated>
  </entry>
  <entry xml:base="http://localhost:4392/Flights/BY002/1202">
    <id>BY002</id>
    <title type="text">Flight BY002 December 02, 2012</title>
    <updated>2012-11-16T21:50:17Z</updated>
  </entry>
</feed>
```

Hypermedia and State Transfer

Another pattern supported by hypermedia is state transfer. To manage the state of resources, RESTful services use hypermedia to describe what can be done with the resource when it returns its representation. For example, if a resource representing a flight enables the user to book tickets, a Hypermedia control describing how you can do this should be present. As soon as the flight does not let

the user additionally book because of any number of reasons (it is fully booked, canceled, and so on), the Hypermedia control should not be returned in the resources representation.

This response represents a flight that enables booking in its current state.

A Response with Hypermedia Control for booking flights

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-
8?B?QzpcU2VsYVxNTONcMjA0ODdBXFNvdXJjZVxBbGxmaWxlc1xNb2QwM1xMYWJmaWxlc1xCbHVlWW9uZGVyLkNvb
XBhbmlvblxCbHVlWW9uZGVyLkNvbXBhbmlvbi5Ib3N0XGZsaWdodHM=?=
X-Powered-By: ASP.NET
Date: Wed, 05 Dec 2012 11:12:19 GMT
Content-Length: 312

{
      "Source":{"Country":"Italy","City":"Rome"},
      "Destination":{"Country":"France","City":"Paris"},
      "Departure":"2014-02-01T08:30:00",
      "Duration":"02:30:00",
      "Price":387.0},
      FlightNumber":"BY001",
      "links":[
                        {
                          "rel": "booking",
                          "Link": "http://localhost/flights/by001/booking"
                        }
              ]
}
```

Hypermedia is what differentiates REST from HTTP-based services. It is a simple but powerful concept that enables a range of capabilities and patterns including service versioning, aspect management, and more which are beyond the scope of this course. Today, more and more formats and APIs are created by using Hypermedia.

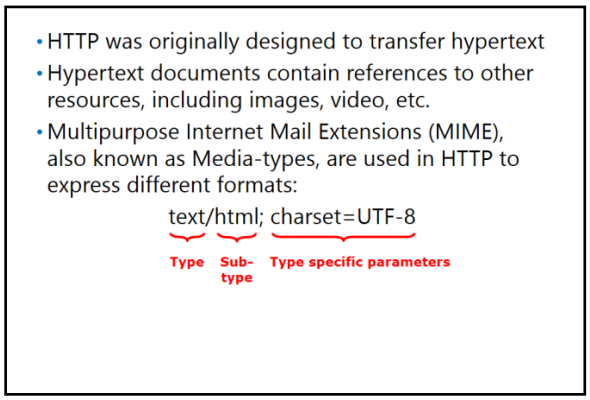
One of the media types supporting Hypermedia is the Hypertext Application Language (HAL). The HAL media type offers link-based Hypermedia. For more information about HAL, see the HAL format specifications.

Hypertext Application Language (HAL)

http://go.microsoft.com/fwlink/?LinkID=298762&clcid=0x409

Media Types

HTTP was originally designed to transfer Hypertext. Hypertext is a nonlinear format that contains references to other resources, some of which are other Hypertext resources. However, some resources contain other formats such as Image files and videos, which required HTTP to support the transfer of different types of message formats. To support different formats, HTTP uses Multipurpose Internet Mail Extensions (MIME) types, also known as media types. MIME types were originally designed for use in defining the content of email messages sent over SMTP.



Media types are made out of two parts, a type and a subtype, optionally followed by type-specific parameters. For example, the type **text** indicates a human-readable text and can be followed by subtypes such as HTML, which indicates HTML content and **plain** indicates a plain text payload.

Common Text Media Types

```
text/html
text/plain
```

In addition, the text type gives a charset parameter, so that the following declaration is also valid.

The Charset Parameter used in Text Media Types

```
text/html; charset=UTF-8
```

In HTTP, media types are declared by using headers as part of a process that is known as content negotiation. Content negotiation is not restricted for media type and includes support for language negotiation, encoding, and more. The following section shows how content negotiation is used for handling media types.

The Accept Header

When a client sends a request, it can send a list of requested media types, and in order of preference, it can accept in the response.

This request message uses the **Accept** header in order to communicate to the server what media types it can accept.

An HTTP request message starting content negotiation

```
GET http://localhost:4392/travelers/1 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:4392
DNT: 1
Connection: Keep-Alive
```

Although the server should try to fulfill the request for content, this is not always possible. Be aware that in the previous request, the type */* indicates that if text/html and application/xhtml+xml are not available, the server should return whatever type it can.

The Content-Type Header

In HTTP, any message that contains an entity-body should declare the media type of the body by using the Content-Type header.

This request message uses the Content-Type header in order to declare what media types it uses for the entity-body.

An HTTP response message returning application/json representation of a traveler entity

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/11.0.0.0
Date: Sat, 17 Nov 2012 13:27:20 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8
Content-Length: 188
Connection: Close

{"TravelerId":1,"TravelerUserIdentity":"aaabbbccc","FirstName":"FirstName1","LastName":"L
astName1","MobilePhone":"555-555-5555","HomeAddress":"One microsoft
road","Passport":"AB123456789"}
```

Media types give the structuring of the HTTP messages. Content negotiation enables servers and clients to set the expectation for what content they should expect during their HTTP transaction. Content negotiation is not limited to media types. For example, content negotiation is used to negotiate content compression by using the **Accept-Encoding** header, localization by using the **Accept-Language** header, and more.

Content negotiation in the HTTP 1.1 Request For Comments (RFC 2616)

http://go.microsoft.com/fwlink/?LinkID=298763&clcid=0x409

Question: Why do you need different Http Verbs?

Lesson 2
Creating an ASP.NET Web API Service

ASP.NET Web API is the first full-featured framework for developing HTTP-based services in the .NET Framework. Using ASP.NET Web API gives developers reliable methods for creating, testing, and deploying HTTP-based services. In this lesson, you will learn how to create ASP.NET Web API services and how they are mapped to the different parts of HTTP. You will also learn how to interact directly with HTTP messages and how to host ASP.NET Web API services.

Lesson Objectives

After you complete this lesson, you will be able to:

- Describe ASP.NET Web API and how it is used for creating HTTP-based services.
- Create routing rules.
- Create ASP.NET Web API controllers.
- Define action methods.
- Create and run an HTTP-based service by using ASP.NET Web API.

Introduction to ASP.NET Web API

HTTP has been around ever since the World Wide Web was created in the early 1990s, but adoption of it as an application protocol for developing services took time. In the early years of the web, SOAP was considered the application protocol of choice by most developers. SOAP provided a robust platform for developing RPC-style services.

With the appearance of Internet scale applications and the growing popularity of web 2.0, it became clearthat SOAP was not fit for such challenges and HTTP received more and more attention.

- ASP.NET Web API is the framework for developing HTTP services
- Provides an all-in-one solution for developing, hosting, consuming, and testing HTTP services

HTTP in the .NET Framework

For the better part of the first decade of its existence, the .NET Framework did not have a first-class framework for building HTTP services. At first, ASP.NET provided a platform for creating HTML-based web-pages and ASP.NET web services, and later-on Windows Communication Foundation (WCF) provided SOAP-based platforms. For these reasons, HTTP never received the attention it deserved.

When WCF first came out in .NET 3.0, it was a SOAP-only framework. As the world started to use HTTP as the an application-layer protocol for developing services, Microsoft started to make investments in extending WCF for supporting simple HTTP scenarios. By the next release of the .NET Framework (.NET 3.5), WCF had new capabilities. These included a new kind of binding called **WebHttpBinding** and the new attributes for mapping methods to HTTP requests.

In 2009, Microsoft released the WCF REST Starter Kit. This added the new **WebServiceHost** class for hosting HTTP-based services, and also new capabilities like help pages and Atom support. When the .NET Framework version 4 was released most of the capabilities of the WCF REST Starter Kit were already rolled into WCF. This includes support for IIS hosting in addition to new Visual Studio templates available

through the Visual Studio Extensions Manager. But even then, WCF was still missing support a lot HTTP scenarios.

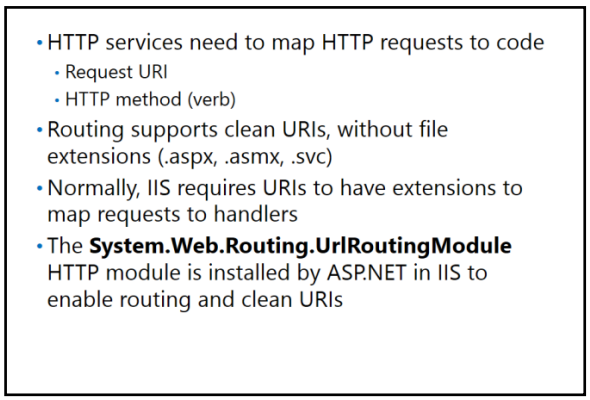
The need for a comprehensive solution for developing HTTP services in the .NET Framework justified creating a new framework. Therefore, in October 2010, Microsoft announced the WCF Web API, which introduces a new model and additional capabilities for developing HTTP-based services. These capabilities included:

- Better support for content negotiation and media types.
- APIs to control every aspect of the HTTP messages.
- Testability.
- Integration with other relevant frameworks like Entity Framework and Unity.

The WCF Web API team released 6 preview versions until in February 2012, the ware united with the ASP.NET team, forming the ASP.NET Web API.

Creating Routing Rules

One of the first challenges when developing HTTP-based services is mapping HTTP requests to the code being executed by the server based on the request-URI and HTTP-method. The process of mapping the request URI to a class or a method is called *routing*.



Routing Tables

ASP.NET uses the **System.Web.Routing.RouteTable** class to hold a data structure that contains different routes that were configured before the initialization of the host. A route contains a URI template and default values for the template. ASP.NET uses routes to map HTTP requests based on their request-URI and HTTP method to the correlating code in the server.

Defining Routes

ASP.NET Web API routes are defined by using the **MapHttpRoute** extension method as is shown in the following code.

This example shows the configuration of a simple route based on the name of the controller.

Configuring a route by using the System.Web.Http.HttpConfiguration class

```
GlobalConfiguration.Configuration.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

Routes Definition in Details

To understand routes, you have to understand how ASP.NET Web API services are implemented. ASP.NET Web API services are implemented in classes called controllers, each controller exposing one or more public methods called actions. The hosting environment uses routing to deliver HTTP requests to the actions designed to handle those requests.

The following headings discuss controllers and actions in-depth, because understanding what controllers and actions are is important for understanding routes.

How Controllers Are Mapped

ASP.NET Web API services are implemented in classes called controllers. Controllers are implemented by using two constraints:

- They must derive from the **System.Web.Http.ApiController** class
- By convention, they must be named with the **Controller** suffix

When ASP.NET Web API receives a request that matches the template in the route, it looks for a controller that matches the value that was passed in the controller placeholder of the URI template by name. For example, a URI with the following URI relative path, **"api/flights/by001"**, will be evaluated against the template defined in the earlier example (**"api/{controller}/{id}"**). ASP.NET Web API will look for a controller that is named **FlightsController**.

This controller maps when the **flights** value is passed as the value for the {controller} placeholder.

The flights controller

```
public class FlightsController : ApiController
{
}
```

How Actions Are Mapped

Conventions play a big role in the ASP.NET ecosystem and ASP.NET Web API is not different. When looking at the route template, you can see that there is no placeholder for actions even though action methods must be executed in order to handle incoming requests. This is possible because of a convention that maps methods based on their prefix to HTTP verbs.

This action method is chosen when sending a GET request by using the **flights/by001** path.

An Action definition

```
public class FlightsController : ApiController
{
    public HttpResponseMessage Get(string id)
    {
          // Place code here to return an HttpResponseMessage object
    }
}
```

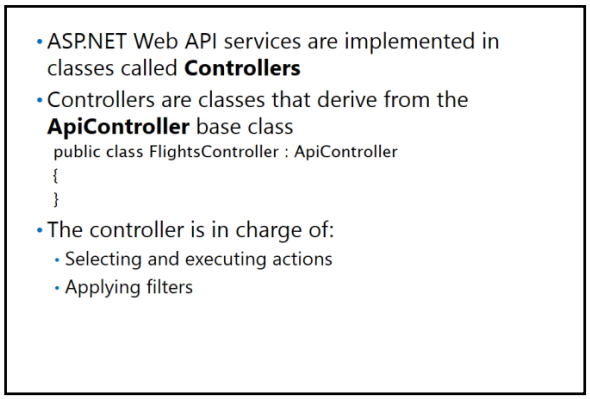
 **Note:** This convention only supports the GET, HEAD, PUT, POST, OPTION, PATCH and DELETE methods. However, actions also support attribute-based routing described later in this lesson.

How Parameters Are Mapped

In this example, a parameter called id was also mapped as a part of the absolute path of the URI. In ASP.NET, parameters are matched as part of a process that is known as Parameter Binding. Parameter Binding default behavior is to bind simple types from the URI and complex types from the entity-body of the request.

For parameter bindings, simple types include all .NET Primitives with the addition of **DateTime**, **Decimal**, **TimeSpan**, **String**, and **Guid**.

The ApiController Class



ASP.NET Web API services are implemented in classes called controllers, which derive from the **System.Web.Http.ApiController** class. As soon as a request is routed to a controller based on a URI, the **ApiController** takes control in finding and executing the appropriate action.

The **ApiController** also provides APIs for handling HTTP request, accessing the **HttpConfiguration**, validating input parameters, and interacting with context of the operation. In fact, a big part of the capabilities of ASP.NET Web API that are described in this module and also in Module 4, "Extending and Securing ASP.NET Web API Services", are exposed and managed by the **ApiController**.

Defining Controllers

To create a controller, you have to do the following:

- Create a class that derives from the **System.Web.Http.ApiController** class.
- Name the class with the *Controller* suffix.

This code example shows how to define a controller.

The flights controller

```
public class FlightsController : ApiController
{
}
```

The Responsibilities of the ApiController

ASP.NET Web API controllers have to derive from the **ApiController** class. The reason for deriving from **ApiController** class is that in addition to defining a logical unit of the service, the **ApiController** class does lots of work. Among the responsibilities of the **ApiController** class, you can find:

- **Action Selection**. The **ApiController** class is responsible for calling the Action Selector that is responsible to execute the action method. Action selection is described in-depth in the next topic, **Action Methods and HTTP Verbs**.
- **Applying Filters**. ASP.NET Web API filters let developers extend the request/response pipeline. Before executing an action method, the **ApiController** class is in charge of applying and executing the filters in the correct order before and after the execution of the action methods.

Additional APIs in the ApiController

The **ApiController** class also exposes additional APIs. Most of them are based on the **System.Web.Http.Controllers.HttpControllerContext** class, which represents the context of the current HTTP request. The request contains information such as the current route that is being used and the request message.

The **ApiController** exposes the **HttpControllerContext** by using the **ControllerContext** property. In addition, the **ApiController** also provides some properties that expose specific data that is a part of the **ControllerContext** property including:

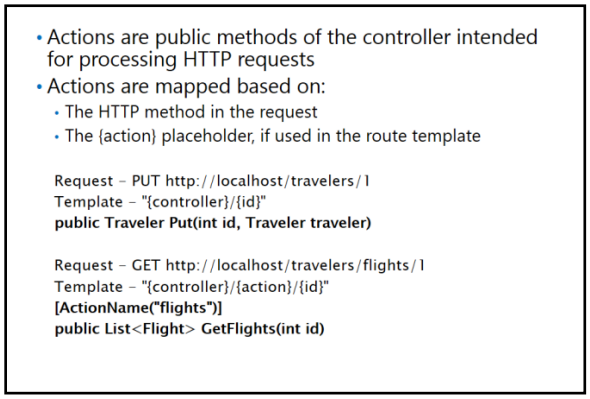
- **The Request property.** This API provides access to the **HttpRequestMessage** representing the HTTP request for the operation. **HttpRequestMessage** class is discussed in-depth in lesson 3, **Handling HTTP Requests and Responses**, of this module.

- **The Configuration property**. The **Configuration** property exposes the configuration that is being used by the host.

Additional Reading: Filters are discussed in-depth throughout Module 4. Action filters are discussed in Module 4, Lesson 1, **The ASP.NET Web API Request Pipeline; Exception** filters are discussed in Module 4, lesson 2, **The ASP.NET Web API Response Pipeline; and Authorization** filters are discussed in Module 4, lesson 4, **Implementing security in ASP.NET Web API Services**.

Action Methods and HTTP Verbs

As soon as ASP.NET Web API chooses a controller, it can start to handle the next step of choosing the method that will handle the request. The selection of the action is performed by the **ApiController** class**.** When choosing an action method, the **ApiController** class gets an instance of a class implementing **System.Web.Http.Controllers.IHttpActionSelector** interface from the **ControllerContext**. The default implementation is using the **System.Web.Http.Controllers.ApiControllerActionSelector** class.



The method selection can be done based on the requests HTTP method has used and on the request-URI. There are several techniques for mapping Actions:

- Mapping to HTTP methods based on convention.
- Mapping to request-URIs based on the {action} placeholder in route templates.

In addition to matching the HTTP method or request-URI to the method name or attribute, ASP.NET Web API takes the parameters that are passed to the method into consideration and makes sure that they match.

Mapping by Action Name

In addition to the controller placeholder in routes, ASP.NET Web API also has a special placeholder for action. When the controller identifies an action in the route, it will first map the action to action methods, which match the name of the action. Matching is performed for methods that fit one of the following criteria:

Criteria	An action matching the name Flights

The name of the method is the same as the action name.	`public HttpResponseMessage Flights()`
	`{`
	`}`

The name of the method matches the action with the prefix of a valid HTTP method name.	`public HttpResponseMessage GetFlights()`
	`{`
	`}`

Criteria	An action matching the name Flights

The method has an action name that is defined by using the **[ActionName]** attribute.	`[ActionName("Flights")]`
	`public HttpResponseMessage AirTrips()`
	`{`
	`}`

Mapping by HTTP Method

The **ApiControllerActionSelector** also maps actions by HTTP methods. This can be done by using one of the following techniques:

Matching by using a prefix or method name.	`public HttpResponseMessage GetFlights()`
	
	`public HttpResponseMessage Get()`

Matching by using the **[AcceptVerbs]** attribute.	`[AcceptVerbs("GET")]`
	`public HttpResponseMessage AirTrips()`
	
	`[AcceptVerbs(AcceptVerbs.Get)]`
	`public HttpResponseMessage AirTrips()`

Matching by using specific implementation of **ActionMethodSelectorAttribute**.	`[HttpGet]`
	`public HttpResponseMessage Flights(int id)`
	
	`[HttpDelete]`
	`public HttpResponseMessage Flights(int id)`

 **Note:** This convention and **HttpVerb** enum support only the GET, HEAD, PUT, POST, OPTION, PATCH, and DELETE methods.

Demonstration: Creating Your First ASP.NET Web API Service

In this demonstration, you will create a new ASP.NET Web application project using the Web API template, view the code generated by Visual Studio 2012, and apply changes to the actions and routing templates.

Demonstration Steps

1. Open **Visual Studio 2012** and create a new **ASP.NET 4 MVC Web Application** project, by using the **Web API** template. Name the new project **MyApp**.
2. Review the content of the **WebApiConfig.cs** file that is under the **App_Start** folder.

3. Review the content of the **ValuesController.cs** file that is under the **Controllers** folder. The parameterless **Get** action method can be invoked by using HTTP (for example, using the */api/values* relative URI).
4. Run the project without debugging, and access the parameterless **Get** action method from the browser.

- In the browser, append the **api/values** to the end of the address, and press Enter.
- Open the **values.json** file in Notepad and observe its content.

5. In the **ValuesController** class, decorate the parameterless **Get** action with the **[ActionName]** attribute, and set the action name to **List**.
6. In the **WebApiConfig** class, add a new route to support MVC-style invocation.

- Add the route by using the **config.Routes.MapHttpRoute** method.
- Set the parameters of the method to the following values.

Parameter	Value
name	ActionApi
routeTemplate	api/{controller}/{action}/{id}
defaults	New anonymous object with an **id** property set to **RouteParameter.Optional**

- Place the call to the method before the existing routing code.

7. Run the project and access the parameter less **Get** action method from the browser, using the MVC-style routing.

- In the browser, append the **api/values/list** to the end of the address, and press Enter.
- Open the **list.json** file in Notepad and observe its content.

  Question: How ASP.NET Web API knows which method to invoke when received a request from the client?

Lesson 3
Handling HTTP Requests and Responses

Creating an instance of a class and finding the method to execute is not always enough. In order to provide a real solution for HTTP-based services, ASP.NET Web API has to provide additional functionality for interacting with HTTP messages. This functionality includes mapping parts of the HTTP request to method parameters in addition to a comprehensive API for processing and controlling HTTP messages. Using that API, you can now easily interact with headers in the requests and response messages, control status codes, and more.

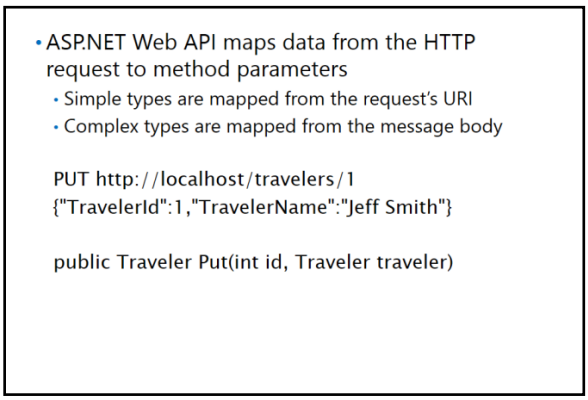
Lesson Objectives

After completing this lesson, you will be able to:

- Describe how parameter binding works in ASP.NET Web API.
- Use **HttpRequestMessage** class to handle incoming requests.
- Use **HttpResponseMessage** class to control the response of an action.
- Throw **HttpResponseExceptions** exception to control HTTP errors.

Binding Parameters to Request Message

After locating the controller and action method, there is still one last task that ASP.NET Web API must handle, which is mapping data from the HTTP request to method parameters. In ASP.NET Web API, this process is known as parameter-binding.



HTTP messages data can be passed in the following ways:

- **The message-URI**. In HTTP, the absolute path and query are used to pass simple values that help identify the resource and influence the representation.
- **The Entity-body**. In some HTTP messages, the message body passes data.

 **Note:** Headers are also used to pass metadata and not as part of the business logic. Headers data is not bound to methods parameters by default and is accessed by using the **HttpRequestMessage** class described later in this lesson.

By default, ASP.NET Web API differentiates simple and complex types. Simple types are mapped from the URI and complex types are mapped from the entity-body of the request. For parameter bindings, simple types include all .NET primitive types (**int**, **char**, **bool**, and so on) with the addition of **DateTime**, **Decimal**, **TimeSpan**, **String**, and **Guid**.

The HttpRequestMessage Class

Invoking a method is an important aspect of HTTP-based services. But HTTP provides a vast functionality that requires analyzing the request message in its entirety. For example, request headers can provide important information, including the version of the entity passed, user credentials, cookie data, and the requested response format. ASP.NET Web API uses the **System.Net.Http.HttpRequestMessage** class to represent incoming HTTP request messages.

- HTTP headers in the request provide information about the client, the resource, and the entity
- Headers provide information such as:
  - Version of the entity passed in the body
  - User credentials
  - Accepted response formats and data
- ASP.NET Web API provides access to the HTTP request using the **HttpRequestMessage** class

The **HttpRequestMessage** class can be accessed by most of the run time components that compose the request pipeline including: message handlers, formatters and filters. **HttpRequestMessage** can also be accessed inside ASP.NET Web API controllers by using the **Request** property.

This code example uses the **AcceptLanguage** property of **HttpRequestMessage** class**,** which lists languages in order to retrieve the value of the **Accept-Language** header returning a localized greeting message.

Retrieve the value of the Accept-Language header by using the Request property

```
public string Get(int id)
{

    var lang = Request.Headers.AcceptLanguage;
    var bestLang = (from l in lang
                    orderby l.Quality descending
                    select l.Value).FirstOrDefault();

    switch (bestLang)
    {
        case "en":
            return "Hello";
        case "da":
            return "Hej";
    }

    return string.Empty;
}
```

The HttpResponseMessage Class

Action methods can return both simple and complex types that are serialized to a format based on the **Accept** header. Although ASP.NET Web API can handle the content negotiation and serialization, it is sometimes required to handle other aspects of the HTTP response message (for example, returning a status code other than 200 or adding headers).

The **System.Net.Http.HttpResponseMessage** class enables programmers to define every aspect of the HTTP response message the action returns.

- HTTP responses use status codes and headers to express the outcome of the request processing
- The **HttpResponseMessage** class provides an API for controlling the content of HTTP responses
- To control the response, first set the return type of the action to **HttpResponseMessage**
  public HttpResponseMessage Get(int id)
- To create an **HttpResponseMessage**, use the **Request.CreateResponse** method
  Request.CreateResponse(HttpStatusCode.NotFound)

In order to control the HTTP response, you must create an action with **HttpResponseMessage** as its return type**.** Inside the action, you have to use the **Request.CreateResponse** or **Request.CreateResponse<T>** methods to create a new **HttpResponseMessage**.

This code example creates a new flight reservation and returns an HTTP message that has two important characteristics: a 201 created status and a **Location** header with the URI of the newly created resource.

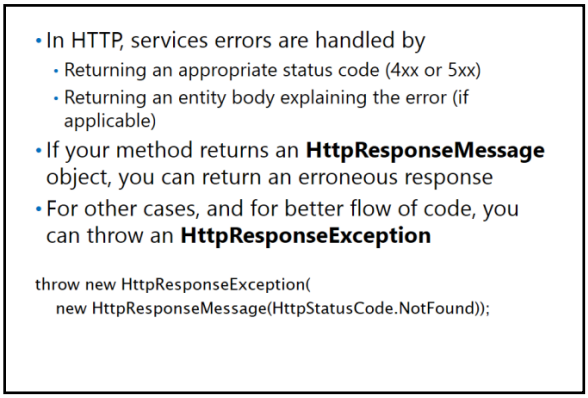
Using the HttpResponseMessage class to control the HTTP response

```
public HttpResponseMessage Post(Reservation reservation)
{
    Reservations.Add(reservation);
    Reservations.Save();

    var response = Request.CreateResponse(HttpStatusCode.Created, reservation);
    response.Headers.Location = new Uri(Request.RequestUri,
reservation.ConfirmationNumber.ToString());
    return response;
}
```

Throwing Exceptions with the HttpResponseException Class

In HTTP, errors are communicated by using two mechanisms:

- **HTTP status-codes –** Provides a numeric representation of the result of the request to the server. Status codes provide an application readable representation of the result of a request.
- **Entity-body –** For most status codes, HTTP accepts an entity body to provide clients with details about the error that occurred.



Although both aspects of HTTP errors can be accessed by using **HttpResponseMessage** class, when you deal with more complex scenarios, returning different results can create a complex code base. Modern programming languages use exceptions to provide simple control flow when an error occurs. ASP.NET Web API provides the **System.Web.Http.HttpResponseException** that enables programmers to control the **HttpResponseMessage** by throwing an exception.

To throw an **HttpResponseException**, you have to create a new response message **HttpResponseMessage** and set the status code, headers, and content that you want the response to have. Next, you throw a new **HttpResponseException** accepting the **HttpResponseMessage** as a constructor parameter**.**

This code example throws **HttpResponseException** to return a 404 Not found response.

Throwing an HttpResponseException

```
if (flight == null)
{
    throw new HttpResponseException(
        new HttpResponseMessage(HttpStatusCode.NotFound);
}
```

Demonstration: Throwing Exceptions

In this demonstration, you will learn how to handle exceptions in ASP.NET Web API. You will learn how to use the **HttpResponseMessage** class to control the status-code of the HTTP response message, and how to use the **HttpResponseException** class to provide better control flow if there is an error.

Demonstration Steps

1. In Visual Studio 2012, open the D:\Allfiles\Mod03\Democode\ThrowHttpResponseException\start\start.sln solution.
2. Open the **DestinationsController.cs** from the **Controllers** folder, and review the contents of the **Get** method.
3. Change the **Get** method so that it returns a **Destination** object and not an **HttpRequestMessage** object.

- Change the return type of the method from **HttpRequestMessage** to **Destination**.
- Remove the if-else statement and return the **destination** variable.

4. Add handling for non-existing destinations by throwing an **HttpResponseException**.

- If the destination was not found, throw a new **HttpResponseException** object.
- Initialize the exception with a new **HttpResponseMessage**, and set the status code of the message to **HttpStatusCode.NotFound**.

5. Run the project without debugging, and verify the **Get** method returns an HTTP 404 for unknown destinations.

- In the browser, append the **api/destinations/1** to the end of the address, and press Enter. Open the file in Notepad and verify you see information for Seattle.
- In the browser, append the **api/destinations/6** to the end of the address, and press Enter. Verify you get an HTTP 404 response.

  Question: In which case should you use HttpResponseException?

Lesson 4
Hosting and Consuming ASP.NET Web API Services

As with any other application, ASP.NET Web API services need a process to give them a run time environment. This run time must accommodate code that potentially serves a large amount of clients. When developing services, hosting environments provide the majority of the capabilities needed to service client requests and maintain a quality of service. After hosting the service, you will learn how to consume the service from various client environments including HTML, JavaScript, and the .NET Framework.

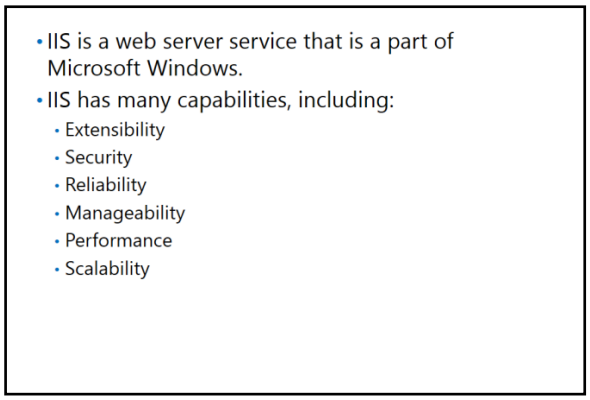
Lesson Objectives

After you complete this lesson, you will be able to:

- Describe the main capabilities of IIS.
- Host ASP.NET Web API in IIS.
- Self-host ASP.NET Web API in .NET applications.
- Consume ASP.NET Web API by using browser-based applications.
- Consume ASP.NET Web API from .NET Framework applications.

Introduction to IIS

Internet Information Services (IIS) is a web server service that is a part of Microsoft Windows. IIS was first released in 1995 as an update to Windows NT 3.51 and has been included in every version of Microsoft server operating systems since. IIS provides a hosting environment for applications and services and also a set of utilities and extensions for managing different aspects of the applications and serves life cycle.



Core capabilities of IIS

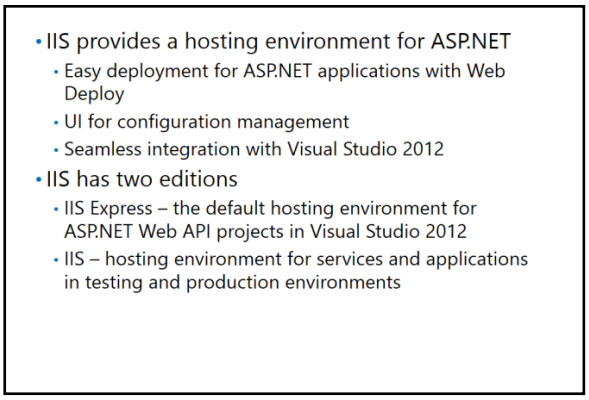
IIS has the following capabilities:

- **Extensibility**. Provides a processing pipeline for messages that is built out of an extensible set of components called modules. Each model is in charge of performing a specific action based on the messages passing through the pipeline.
- **Security.** Provides built-in modules for handling security. This includes capabilities for managing secured conversation with Secure Socket Layer (SSL), handling different kinds of HTTP authentication (Basic, Digest, and so on), and IP restriction.
- **Reliability.** Provides a set of worker processes for application and services. These worker processes are called application pools and provide process management for one or more applications. Application pools are monitored and managed by IIS and provide benefits like isolation between services and applications, resources, and fault management.
- **Manageability.** Provides management tools including an MMC-based UI and a set of Microsoft PowerShell commands that give managing IIS core functionality and extensions. IIS also provides built-in logging and diagnostics capabilities that simplify managing production environments.

- **Performance.** Provides built-in caching and compression modules to improve HTTP applications performance. IIS also uses the HTTP.SYS kernel mode driver to listen to HTTP traffic. HTTP.SYS also provides HTTP caching mechanisms that enable the users to handle cached requests completely in kernel mode providing a high-performance caching mechanism.
- **Scalability.** Provides a set of tools to manage multiple servers. These include centralized configuration, sharing application files between server, and a remote administration module that enables the users to manage servers in a centralized manner. IIS provides infrastructure for load balancing by using extensions, such as Microsoft Application Request Routing (ARR), that provide HTTP-based message routing. It also provides infrastructure for load balancing and also Network Load Balancing (NLB), which provides infrastructure for load balancing at the network layer.

Hosting ASP.NET Web API Services by Using IIS

IIS has been hosting the ASP.NET line of products ever since .NET 1.0. This makes IIS a very appealing host for ASP.NET Web API services. By hosting your ASP.NET Web API projects in IIS, you can gain access to many of the features of IIS. This includes the following:



- Easy deployment to remove computers and web farms with Web Deploy.
- Performance features, such as response compression and caching.
- Reliability support by running your ASP.NET Web API service in a monitored application pool.
- Ability to run multiple ASP.NET Web API projects on the same computer by using IIS virtual directories and isolated web applications.

Note: Deploying web applications by using Web Deploy is covered in Module 8, **Deploying Services**.

By default, when you create an ASP.NET Web API project, Visual Studio 2012 uses IIS Express to host your project, and not the regular IIS. IIS Express is a lightweight, self-contained version of IIS optimized for the development environment. IIS Express can be installed on computers that do not have IIS installed on them, or on computers that cannot be installed with the latest version of IIS. For example, if you are developing on a computer that is running Windows Server 2008, you have IIS 7 and you cannot upgrade it to IIS 8. However, you can install IIS 8.0 Express on that computer.

The following link describes in details the differences between IIS and IIS Express.

IIS Express Overview

http://go.microsoft.com/fwlink/?LinkID=298764&clcid=0x409

The main difference between IIS and IIS Express is in the security context. IIS Express uses the security context of the logged-on user to start the hosting process, whereas IIS use the identity defined in the application pool. This is usually a non-privileged built-in account.

Note: Using a different security context can lead to differences in the behavior of the application. For example, when you host your application in IIS Express, it might be able to access the database because it uses your logged on identity which has administrative permissions in the database. However, when the application is hosted in IIS, it might fail trying to access the database because the identity used by the application pool does not have the required permissions to log on to the database.

Ideally, after you have verified your application is running correctly, use IIS to host your application, instead of IIS Express. To instruct Visual Studio 2012 to use IIS, right-click the ASP.NET Web API project in the **Solution Explorer** window, and then click **Properties**. On the **Web** tab, scroll to the **Servers** group, remove the selection from the **Use IIS Express**, and then click **Create Virtual Directory** to create a directory for your web application in IIS.

This image is a snapshot of the ASP.NET web project properties in Visual Studio 2012, where you set the kind of hosting server (IIS or IIS Express).

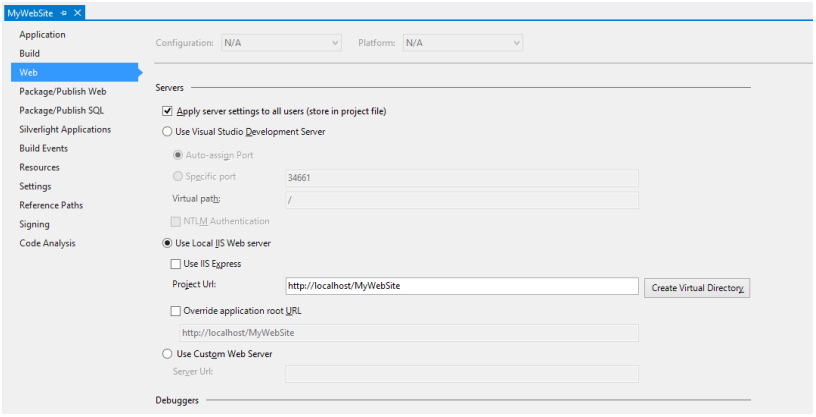
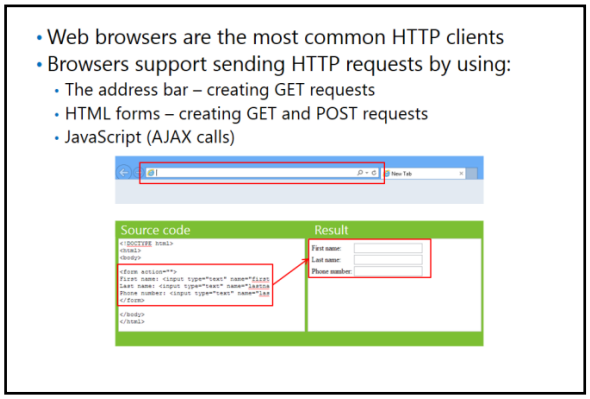


FIGURE 3.1: THE WEB PROPERTIES TAB IN VISUAL STUDIO 2012

Consuming Services from Browsers

When HTTP was built in the early 1990s, it was made for a very specific kind of client: web browsers running HTML. Before the creation of JavaScript in 1995, HTML was using two of the three HTTP methods in HTTP 1.0: GET and POST. GET requests are usually invoked by entering a URI in the address bar or in kinds of hypertext references such as .img and script tags.

For example, entering the http://localhost:2300/api/flights/ URI generates the following GET request.



A GET request invoked by a web browser

```
GET http://localhost:7086/Locations HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:7086
```

```
DNT: 1
Connection: Keep-Alive
```

Another way to start HTTP requests from a browser is using HTML forms. HTML forms are HTML elements that create a form-like UI in the HTML document that lets the user insert and submit data to the server. HTML forms contain sub elements, called input elements, and each represents a piece of data both in the UI and in the resulting HTTP message.

This HTML form lets users submit a new location to the server from a web browser, generating a POST request.

An HTML form for submitting a new flight

```
<form name="newLocation" action="/locations/" method="post">
    <input type="text" name="LocationId" /><br />
    <input type="text" name="Country" /><br />
    <input type="text" name="State" /><br />
    <input type="text" name="City" /><br />
    <input type="submit">
</form>
```

This HTTP message was generated by submitting the **newLocation** HTML form.

An HTML form generated POST request

```
POST http://localhost:7086/locations/ HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Referer: http://localhost:7086/default.html
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Content-Length: 49
DNT: 1
Host: localhost:7086
Pragma: no-cache

LocationId=7&Country=Belgium&State=&City=Brussels
```

The most flexible mechanism to start HTTP from a browser environment is using JavaScript. Using JavaScript provides two main capabilities for that are lacing in other browser-based techniques:

- Complete control over the HTTP requests (including HTTP method, headers, and body).
- Asynchronous JavaScript and XML (AJAX). Using AJAX, you can send requests from the client after the browser completed loading the HTML. Based on the result of the calls, you can use JavaScript to update parts of the HTML page.

Demonstration: Consuming Services Using JQuery

In this demonstration, you will call an ASP.NET Web API service using jQuery AJAX calls.

Demonstration Steps

1. In Visual Studio 2012, open the **D:\Allfiles\Mod03\Democode\ConsumingFromJQuery\Begin\JQueryClient\JQueryClient.sln** solution.

2. In the **JQueryClient** project, expand the **Views** folder, then expand the **Home** folder. Review the content of the **Index.cshtml** file. Locate the script section and observe how the code uses jQuery to retrieve the data from the server.
3. Add the following JavaScript code to the **<script>** element, to override the default behavior for submitting a **DELETE** request to the server.

```
$("#deleteLocation").submit(function (event) {
    event.preventDefault();
    var desId = $(this).find('input[name="LocationId"]').val();
    $.ajax({
        type: 'DELETE',
        url: 'destinations/' + desId
    });
});
```

4. Run the project and use the form to delete a flight.

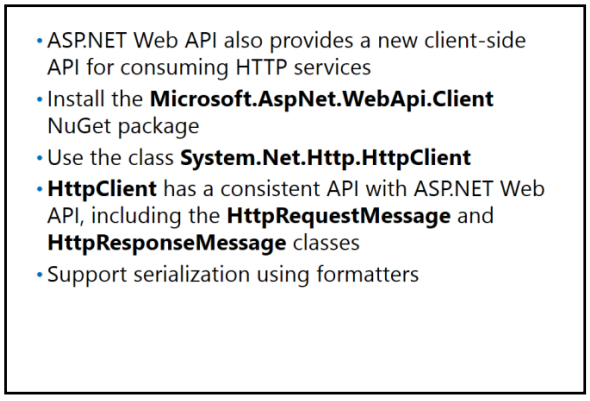
- In the **JQueryClient** project, expand the **Controllers** folder, open the **DestinationsController.cs** file, and place a breakpoint in the **Delete** method.
- Press F5 to debug the application. In the browser, type **1** in the **Location id** box, and then click **Delete**.
- Verify the breakpoint in the **Delete** method is reached.
- Press Shift+F5 to stop the debugger.

Question: Why do you need to use JQuery to consume Web API services?

Consuming Services from .NET Clients with HttpClient

ASP.NET Web API also provides a new client-side API for consuming HTTP in .NET Framework applications. The main class for this API is **System.Net.Http.HttpClient**. This provides basic functionality for sending requests and receiving responses.

HttpClient keeps a consistent API with ASP.NET web API by using **HttpRequestMessage** and **HttpResponseMessage** for handling HTTP messages. The **HttpClient** API is a task-based asynchronous API providing a simple model for consuming HTTP asynchronously.



This code example uses the **HttpClient** to send a GET request receive an **HttpResponseMessage** from the server, and then read its content as a string.

Using the HttpClient GetAsyc method.

```
var client = new HttpClient
{
    BaseAddress = new Uri("http://localhost:12534/")
};

HttpResponseMessage message = await client.GetAsync("destinations");
var res = message.Content.ReadAsStringAsync();
Console.WriteLine(res);
```

Although this code provides a simple asynchronous API, it is not common for the client to require string representation of the data. A more useful approach is to obtain a de-serialized object based on the entity body.

To support serializing and de-serializing objects, **HttpClient** uses a set of extensions defined in the **System.Net.Http.Formatting.dll** that is a part of the **Microsoft ASP.NET Web API Client Libraries** NuGet package. The **System.Net.Http.Formatting.dll** adds the extension methods to the **System.Net.Http** namespace so that no additional using directive is needed.

This code example uses the **ReadAsAsync<T>** extension method to deserialize the content of the HTTP message into a list of Destinations.

Using the ReadAsAsync<T> extension method

```
var client = new HttpClient
{
    BaseAddress = new Uri("http://localhost:12534/")
};

HttpResponseMessage message = await client.GetAsync("destinations");
var destinations = await message.Content.ReadAsAsync<List<Destination>>();
Console.WriteLine(destinations.Count);
```

Demonstration: Consuming Services Using HttpClient

In this demonstration, you will learn how to consume HTTP services from .NET Framework applications by using the **HttpClient** class. You will also learn how the **HttpClient** class can serialize and deserialize the body of HTTP messages into objects by using extensions defined in the **System.Net.Http.Formatting.dll**.

Demonstration Steps

1. Open Visual Studio 2012 as an administrator and open the D:\Allfiles\Mod03\Democode\ConsumingFromHttpClient\begin\HttpClientApplication\HttpClientApplication.sln solution.
2. Add the Microsoft ASP.NET Web API Client Libraries NuGet package to the HttpClientApplication.Client project.
3. Add code to perform a GET request for the destinations resource inside the **CallServer** method and print the responses content as string to the console window.

- Create a new **HttpClient** object and store it in a variable named **client**.
- Set the client.BaseAddress property to the URI http://localhost:12534/
- Call the **client.GetAsync** method with the relative URI **api/Destinations**, and use the **await** keyword to call the method asynchronously.
- Store the return value of the **GetAsync** method in a variable of type **HttpResponseMessage**. Name the new variable **message**.
- Read the content of the response message's body by calling the **message.Content.ReadAsStringAsync** method. Use the **await** keyword to call the method asynchronously.
- Write the returned response string to the console output.

4. Add code to deserialize the response into a **List<Destinations>**.

- After the code you added in the previous step, call the **message.Content.ReadAsAsync** method with the generic type **List<Destination>** to deserialize the response message to a list of Destination objects. Use the **await** keyword to call the method asynchronously.
- Write the size of the returned destinations list to the console.

5. Run the server application, and then the client application. Show how the client code retrieves data from the server.

   Question: What are the benefits of HttpClient that makes it more useful than HttpWebRequest and WebClient?

Lab: Creating the Travel Reservation ASP.NET Web API Service

Scenario

Now that the data layer has been created, services that provide travel destination information, flight schedules, and booking capabilities can be developed. Blue Yonder Airlines intends to support many device types. The back-end service must have an HTTP-based service. Therefore, the service is to be implemented by using ASP.NET Web API. In this lab, you will create a Web API service that supports basic CRUD actions over Blue Yonder Airlines' database. In addition, you will update the Travel Companion Windows Store app to consume the newly created service.

Objectives

After you complete this lab, you will be able to:

- Create an ASP.NET Web API service.
- Implement **CRUD** actions in the service.
- Consume an ASP.NET Web API service with the **System.Net.HttpClient** library.

Lab Setup

Estimated Time: 30 Minutes

Virtual Machine: **20487B-SEA-DEV-A, 20487B-SEA-DEV-C**

User name: **Administrator, Admin**

Password: **Pa$$w0rd, Pa$$w0rd**

For this lab, you will use the available virtual machine environment. Before you begin this lab, you must complete the following steps:

1. On the host computer, click **Start**, point to **Administrative Tools**, and then click **Hyper-V Manager**.
2. In Hyper-V Manager, click **MSL-TMG1**, and in the Action pane, click **Start**.
3. In Hyper-V Manager, click **20487B-SEA-DEV-A**, and in the Action pane, click **Start**.
4. In the Action pane, click **Connect**. Wait until the virtual machine starts.
5. Sign in using the following credentials:

- User name: Administrator
- Password: **Pa$$w0rd**

6. Return to Hyper-V Manager, click **20487B-SEA-DEV-C**, and in the Action pane, click **Start**.
7. In the Action pane, click **Connect**. Wait until the virtual machine starts.
8. Sign in using the following credentials:

- User name: **Admin**
- Password: **Pa$$w0rd**

9. Verify that you received credentials to log in to the Azure portal from you training provider, these credentials and the Azure account will be used throughout the labs of this course.

Exercise 1: Creating an ASP.NET Web API Service

Scenario

Implement the travelers' service by using ASP.NET Web API. Start by creating a new ASP.NET Web API controller, and implement CRUD functionality using the POST, GET, PUT and DELETE HTTP methods.

The main tasks for this exercise are as follows:

1. Create a new API Controller for the Traveler Entity

▶ Task 1: Create a new API Controller for the Traveler Entity

1. In the 20487B-SEA-DEV-A virtual machine, open the D:\AllFiles\Mod03\LabFiles\begin\BlueYonder.Companion\BlueYonder.Companion.sln solution file, and add a new class called TravelersController to the BlueYonder.Companion.Controllers project.
2. Change the access modifier of the class to **public**, and derive it from the **ApiController** class.
3. Create a private property named **Travelers** of type **ITravelerRepository** and initialize it in the constructor.

- Create a new property named **Travelers** of type **ITravelerRepository**.
- Create a default constructor for the **TravelersController** class.
- Initialize the **Travelers** property with a new instance of the **TravelerRepository** class.

4. Create an action method named **Get** to handle GET requests.

- The method receives a **string** parameter named **id** and returns an **HttpResponseMessage** object.
- Call the **FindBy** method of the **ITravelerRepository** interface to search for a traveler using the **id** parameter. The ID of the traveler is stored in the traveler's **TravelerUserIdentity** property.
- If the traveler was found, use the **Request.CreateResponse** to return an HTTP response message with the traveler. Set the status code of the response to **OK**.
- If a traveler was not found, use the **Request.CreateResponse** to return an empty message. Set the status code to **NotFound** (HTTP 404).

5. Insert a breakpoint at the beginning of the **Get** method.
6. Create an action method to handle **POST** requests.

- The method receives a **Traveler** parameter called **traveler** and returns an **HttpResponseMessage** object.
- Implement the method by calling the **Add** and then the **Save** methods of the **Travelers** repository.
- Create an **HttpResponseMessage** returning the **HttpStatusCode.Created** status and the newly created traveler object.
- Set the **Location** header of the response to the URI where you can access the newly created traveler. The new URI should be a concatenation of the request URI and the new traveler's ID.

 **Note:** You can refer to the implementation of the **Post** method in the **ReservationsController** class for example of how to set the **Location** header.

- Return the HTTP response message.

7. Insert a breakpoint at the beginning of the **Post** method.
8. Create an action method to handle **PUT** requests.

- The method receives a **string** parameter called **id** and a **Traveler** parameter called **traveler**. The method returns an **HttpResponseMessage** object.
- If the traveler does not exists in the database, use the **Request.CreateResponse** method to return an HTTP response message with the **HttpStatusCode.NotFound** status.

Note: To check if the traveler exists in the database, use the **FindBy** method as you did in the **Get** method.

- If the traveler exists, call the **Edit** and then the **Save** methods of the **Travelers** repository to update the traveler, and then use the **Request.CreateResponse** method, to return an HTTP response message with the **HttpStatusCode.OK** status.

Note: The **HTTP PUT** method can also be used to create resources. Checking if the resources exist is performed here for simplicity.

9. Insert a breakpoint at the beginning of the **Put** method.
10. Create an action method to handle **DELETE** requests.

- The method receives a **string** parameter called **id**.
- If the traveler does not exists in the database, use the **Request.CreateResponse** method to return an HTTP response message with the **HttpStatusCode.NotFound** status.

Note: To check if the traveler exists in the database, use the **FindBy** method as you did in the **Get** method.

- If the traveler exists, call the **Delete** and then the **Save** methods of the **Travelers** repository, and then use the **Request.CreateResponse** method, to return an HTTP response message with the **HttpStatusCode.OK** status.

Results: After you complete this exercise, you will be able to run the project from Visual Studio 2012 and access the travelers' service.

Exercise 2: Consuming an ASP.NET Web API Service

Scenario

Consume the travelers' service from the client application. Start by implementing the **GetTravelerAsync** method by invoking a **GET** request to retrieve a specific traveler from the server. Continue by implementing the **CreateTravelerAsync** method by invoking a POST request to create a new traveler. And complete the exercise by implementing the **UpdateTravelerAsync** method by invoking a **PUT** request to update an existing traveler.

The main tasks for this exercise are as follows:

1. Consume the API Controller from a Client Application

2. Debug the Client App

▶ Task 1: Consume the API Controller from a Client Application

1. In the 20487B-SEA-DEV-C virtual machine, open the D:\AllFiles\Mod03\LabFiles\begin\BlueYonder.Companion.Client\BlueYonder.Companion.Client.sln solution.
2. In the **BlueYonder.Companion.Client** project, open the **DataManager** class from the **Helpers** folder and implement the **GetTravelerAsync** method.

- Remove the **return null** line of code.
- Build the relative URI using the string format **"{0}travelers/{1}"**. Replace the {0} placeholder with the **BaseUri** property and the {1} placeholder with the **hardwareId** variable.
- Call the **client.GetAsync** method with the relative address you constructed. Use the **await** keyword to call the method asynchronously. Store the response in a variable called **response**.
- Check the value of the **response.IsSuccessStatusCode** property. If the value is **false**, return **null**.
- If the value of the **response.IsSuccessStatusCode** property is **true**, read the response into a string by using the **response.Content.ReadAsStringAsync** method. Use the **await** keyword to call the method asynchronously.
- Use the **JsonConvert.DeserializeObjectAsync** static method to convert the JSON string to a **Traveler** object. Call the method using the **await** keyword and return the deserialized traveler object.

3. Insert a breakpoint at the beginning of the **GetTravelerAsync** method.
4. Review the **CreateTravelerAsync** method. The method sets the **ContentType** header to request a JSON response. The method then uses the **PostAsync** method to send a POST request to the server.
5. Insert a breakpoint at the beginning of the **CreateTravelerAsync** method.
6. Review the **UpdateTravelerAsync** method. The method uses the **client.PutAsync** method to send a PUT request to the server.
7. Insert a breakpoint at the beginning of the **UpdateTravelerAsync** method.

▶ Task 2: Debug the Client App

1. Go back to the virtual machine **20487B-SEA-DEV-A** and start debugging the **BlueYonder.Companion.Host** project.
2. Go back to the virtual machine **20487B-SEA-DEV-C** and start debugging the **BlueYonder.Companion.Client** project.
3. Debug the client app and verify that you break before sending a GET request to the server. Press F5 to continue running the code.
4. Go back to the virtual machine **20487B-SEA-DEV-A** and debug the service code.

- The breakpoint you have set in the **Get** method of the **TravelersController** class should be highlighted.
- Inspect the value of the **id** parameter.
- Press F5 to continue running the code.

5. Go back to the virtual machine **20487B-SEA-DEV-C** and debug the client app.

- The code execution breaks inside the **CreateTravelerAsync** method.
- Press F5 to continue running the code.

6. Go back to the virtual machine **20487B-SEA-DEV-A** and debug the service code.

- The breakpoint you have set in the **Post** method solution should be highlighted.
- Inspect the contents of the **traveler** parameter.
- Press F5 to continue running the code.

7. Go back to the virtual machine **20487B-SEA-DEV-C** and use the client app to purchase a flight from *Seattle* to *New York*.

- Display the app bar search for the word **New**. Purchase the trip from *Seattle* to *New York*.
- Fill in the traveler information according to the following table and then click **Purchase**.

Field	Value
First Name	Your first name
Last Name	Your last name
Passport	Aa1234567
Mobile Phone	555-5555555
Home Address	423 Main St.
Email Address	Your email address

- The code execution breaks inside the **UpdateTravelerAsync** method.
- Press F5 to continue running the code.

8. Go back to the virtual machine **20487B-SEA-DEV-A** and debug the service code.

- The breakpoint you have set in the **Put** method solution should be highlighted.
- Inspect the contents of the **traveler** parameter.
- Press F5 to continue running the code.

9. Go back to the virtual machine **20487B-SEA-DEV-C**, close the confirmation message, and then close the client app.
10. Go back to the virtual machine **20487B-SEA-DEV-A** and stop the debugging in Visual Studio 2012.

Results: After you complete this exercise, you will be able to run the BlueYonder Companion client application and create a traveler when purchasing a trip. You will also be able to retrieve an existing traveler and update its details.

Question: Why did you need to return an **HttpRequestMessage** from the **Post** action method?

Module Review and Takeaways

In this module, you learned how HTTP can be used for creating services and how to use ASP.NET Web API to create HTTP-based services. You have also learned how to host ASP.NET Web API services in IIS and consume them from Windows Store apps by using the **HttpClient** class. You also learned how to apply best practices when you develop HTTP services by using ASP.NET Web API.

Best Practices:

- Model your services to describe resources and not functions.
- Use the **HttpResponseMessage** to return a valid HTTP response message.
- When handling errors, throw an **HttpResponseException** exception to avoid complex code.

Review Question(s)

Question: What are ASP.NET Web API controllers used for?

Tools

- IIS

Module 5

Creating WCF Services

Contents:

Module Overview

The previous two modules are about ASP.NET Web Application Programming Interface (API), which is the .NET technology to create Hypertext Transfer Protocol (HTTP)-based services. In the first module, we also explained about another type of service, Simple Object Access Protocol (SOAP)-based services. In the first lesson of this module, you will learn about the differences between HTTP-based and SOAP-based services. The rest of this module will be about implementing SOAP-based services with the Windows Communication Foundation (WCF) framework.

When you develop an application that has client/server architecture, the technologies that you are likely to use is the Windows Communication Foundation (WCF) framework. WCF is the most up-to-date communication infrastructure made by Microsoft, and is designed for building distributed applications that use service-oriented architecture (SOA).

WCF is a very flexible and extensible framework. You can customize and configure WCF to match different application scenarios. You can control almost every aspect of client/server communication, either through configuration or by implementing various extensions.

You can control the following aspects of WCF:

- Communication protocol, such as Transmission Control Protocol (TCP), HTTP, and User Datagram Protocol (UDP).
- Security aspects, such as authentication and authorization.
- Performance tuning, such as throttling, concurrency, and load balancing.
- Hosting environment, such as Internet Information Services (IIS) and Windows Services.

This module describes how to create a simple WCF service, host it using self-hosting, and consume the service from a client application. After you get familiar with WCF and its various configurations and extensions, you will be able to build robust, flexible, and scalable services.

Objectives

After you complete this module, you will be able to:

- Describe why and when to use WCF to create services.
- Define a service contract and implement it.
- Host and configure a WCF service.
- Consume a WCF service from a client application.

Lesson 1
Advantages of Creating Services with WCF

SOAP is a protocol specification for exchange of structured information between peers in a decentralized, distributed environment. SOAP uses Extensible Markup Language (XML) for its message formatting, and usually relies on HTTP for message negotiation and transmission, although there are implementations of SOAP over other transports, such as SOAP-over-UDP, which is used for Universal Plug and Play (UPnP).

SOAP-based services technology has existed for more than a decade. It was created by Microsoft and was later adopted by W3C as a standard. SOAP is used as the underlying layer for ASP.NET Web Services (ASMX) and for WCF. However, unlike Web Services, which are limited to SOAP over HTTP, WCF can use SOAP over TCP and even Named pipes. WCF is also not limited to SOAP - it can be configured to use standard XML (known as plain XML) and JavaScript Object Notation (JSON) also.

Note: SOAP over TCP is a WCF proprietary implementation and is not part of the World Wide Web Consortium (W3C) standards. Therefore, it can be used only between a WCF client and a WCF service.

This lesson briefly reviews the history of SOAP-based services, starting with Web Services and ending with WCF. It discusses the advantages WCF offers as an infrastructure service, and reviews some of these features of WCF and its characteristics, including:

- Multiple transport support, such as TCP and HTTP.
- Various messaging patterns, such as request-response and one-way.
- Complex application scenarios, such as transactions, reliable messaging, and service discovery.

Finally, this lesson explains the features of WCF that are not supported by the ASP.NET Web API.

Lesson Objectives

After you complete this lesson, you will be able to:

- Explain the benefits of using SOAP-based services.
- List the features of WCF that are not supported by ASP.NET Web API.

The Benefits of Using SOAP-Based Services

SOAP, a standard remote procedure call (RPC) protocol, is now maintained by the World Wide Web Consortium (W3C). It was designed as a protocol specification for invoking methods on servers, services, and objects. SOAP was developed as a multi-environment and language-independent way of transferring structured data between services.

SOAP is an XML-based protocol that is mainly (but not exclusively) transmitted over HTTP. The main characteristics of SOAP are as follows:

- Lightweight protocol
- Used for communication between applications

- SOAP characteristics
  - Lightweight
  - HTTP-friendly
  - Cross-platform
  - XML-Based
  - Simple and extensible
- Provides a remote procedure call (RPC)-like protocol for invoking remote service operations
- Windows Communication Foundation (WCF) is the framework for creating SOAP-based services in .NET

- Designed to be transmitted over HTTP
- Not designed for any specific programming language
- XML-based
- Simple and extensible

SOAP is not the only RPC protocol. There are many others, such as Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM) RPC protocol. There are several benefits to consider over other protocols:

- **Versatility**. DCOM and CORBA have not been adopted by many platforms. DCOM is supported only by Windows and CORBA is mainly used in Java.
- **Security**. CORBA, DCOM, and similar protocols are usually not firewall/proxy friendly and might be blocked.

For more information on SOAP, see:

Simply SOAP

http://go.microsoft.com/fwlink/?LinkID=298771&clcid=0x409

The following code presents a sample SOAP request message sent to a service, followed by a sample SOAP response message returned by the service. The service multiplies any number sent in the request message by two.

Request and Response SOAP Messages

Request

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Body>
      <MultiplyByTwo xmlns="http://tempuri.org/">
         <value>123</value>
      </MultiplyByTwo>
   </s:Body>
</s:Envelope>
```

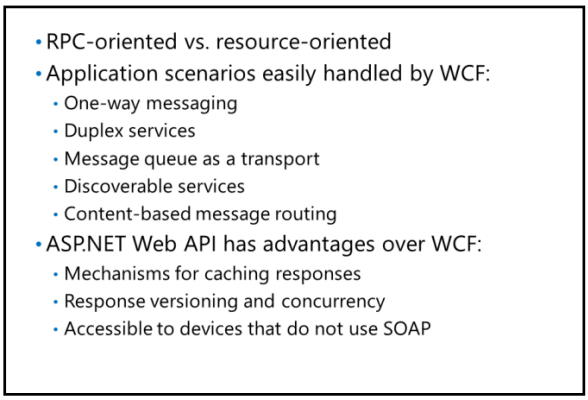
Response

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Body>
      <MultiplyByTwoResponse xmlns="http://tempuri.org/">
         <MultiplyByTwoResult>246</MultiplyByTwoResult>
      </MultiplyByTwoResponse>
</s:Body>
</s:Envelope>
```

SOAP is used as an underlying protocol in ASP.NET Web Services and in WCF services. Both technologies are used to build RPC services.

WCF Features That Are Not Supported by ASP.NET Web API

In the previous topic, we discussed about SOAP, an RPC protocol. Nowadays, HTTP-based services have become popular and are replacing RPC services.



ASP.NET Web API is a simple yet powerful infrastructure for building HTTP-based services. It may seem that WCF is obsolete and you can replace it with ASP.NET Web API. However, that is not always the case. There are several scenarios in which ASP.NET Web API does not provide a solution, or in which WCF is still a preferable infrastructure:

- One way messaging
- Multicast messaging with UDP
- In-order delivery of messages
- Duplex services (for example, publisher-subscriber)
- Using Message Queues (for example, MSMQ) as a transport
- Run time discovery of services
- Content-based message routing
- Distributed transactions support

WCF handles these scenarios by using SOAP and Web Service specifications (commonly referred to as WS-*). The WS-* standards complement SOAP by controlling the needs of distributed applications, such as message security, message reliability, and transaction support. HTTP-based services rely mostly on HTTP headers, which are not designed for these scenarios.

But this is not to say that ASP.NET Web API is less useful than WCF. Services that you create with ASP.NET Web API can take advantage of HTTP and create services that have cacheable responses in clients, built-in concurrency mechanisms, and other features that are part of the HTTP application protocol. In addition, ASP.NET Web API is supported by browsers, which do not support SOAP.

Question: When would you prefer using ASP.NET Web API over WCF?

Lesson 2
Creating and Implementing a Contract

When you create a service, you must answer the following questions:

- What can the service do?
- Where can I find the service?
- How do I send messages to the service?

This lesson explains the service contract, which provides the answer to the first question, "What can the service do?"

The service contract is one of the fundamentals of WCF. It is a definition of the operations supported by the service. Additionally, the service contract defines other aspects of the service and its operations, such as error handling.

This lesson describes how to create and implement a service contract. It also explains how to control the service behavior by using the **[ServiceContract]** attribute, the **[OperationContract]** attribute, and the **[FaultContract]** attribute.

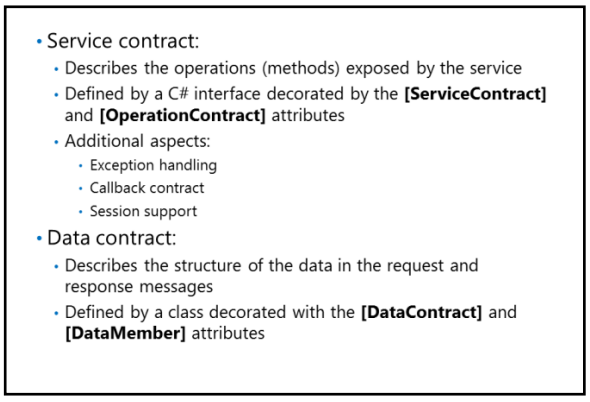
Lesson Objectives

After you complete this lesson, you will be able to:

- Define the WCF service contract.
- Create and implement a service contract.
- Add exception handling to the service.

Creating Service and Data Contracts

A WCF service contract is a standard interface, but a **[ServiceContract]** attribute is added to indicate that the interface defines a WCF service contract. The interface methods that are exposed as service operations will each include an **[OperationContract]** attribute, indicating that the method defines an operation that is part of a service contract.



Best Practice: You can apply the **[ServiceContract]** attribute to an interface or to a class. It is better to use an interface because then you can replace the implementation. It also creates a clear separation between the service contract and its implementation.

These attributes do more than merely mark the interface methods as service operations. They also control various aspects of the service behavior and characteristics, such as session support, exception handling, and callback contracts on duplex services.

Another important aspect of the service contract is the data contract. Data contracts define the data that is exchanged between the service and the client. Data contracts define data that is either returned by a service operation or received by a service operation as a parameter.

Similar to the service contract, the data contract is defined by using special attributes:

- The **[DataContract]** attribute. Applied to the class to mark it as a data contract.
- The **[DataMember]** attribute. Applied to those class properties that will be included in the data contract.

 **Note:** The **ServiceContract** and **OperationContract** attributes are part of the **System.ServiceModel** namespace in the **System.ServiceModel** assembly. The **DataContract** and **DataMember** attributes are part of the **System.Runtime.Serialization** namespace of the **System.Runtime.Serialization** assembly.

The following code example depicts a service contract declaration alongside a data contract (**Reservation**). The contract exposes basic hotel reservation operations.

Service Contract and Data Contract

```
[ServiceContract]
public interface IHotelBookingService
{
   [OperationContract]
     BookingResponse BookHotel(Reservation request);
   [OperationContract]
     Booking GetExistingReservation(string bookingReference);
   [OperationContract]
     string CancelReservation(string bookingReference);
}
 [DataContract]
public class Reservation
{
  [DataMember]
    public string HotelName {get; set; }

   [DataMember]
    public DateTime CheckinDate {get; set; }

   [DataMember]
    public int NumberOfDays {get; set; }

   [DataMember]
    public string GuestFirstName {get; set; }

   [DataMember]
    public string GuestLastName {get; set; }
}
```

The **[DataContract]** and **[DataMember]** attributes are optional. If a class is not decorated with the **[DataContract]** attribute, WCF will automatically serialize every public property and field that it encounters in your class.

You can therefore choose whether to use an inclusive approach or an exclusive approach to mark properties and fields to be serialized:

- **Inclusive approach**. Use **[DataContract]**, and apply **[DataMember]** to each member that needs to be serialized.
- **Exclusive approach**. Do not use **[DataContract]**, and apply the **[IgnoreDataMember]** attribute to each public property and field that you do not want to be serialized.

Choosing the approach to use depends on the characteristics of your class: how large it is, how many serialized and non-serialized members it contains, and whether you can change how it is declared. It is preferable to choose one technique and apply it to all of your classes to prevent confusion.

For more information on inclusive and exclusive data contracts, see:

Using Data Contracts

http://go.microsoft.com/fwlink/?LinkID=298772&clcid=0x409

Implementing a Service Contract

After you define a service contract, you must provide the actual implementation. This involves creating a class that implements the service contract interface.

The previous topic gives an example of a simple service contract, **IHotelBookingService**. The following code example demonstrates implementation of this contract.

- Create a service class that implements the service contract
- Additional aspects of service implementation can be controlled with the **[ServiceBehavior]** attribute

```
[ServiceBehavior(
   InstanceContextMode=InstanceContextMode.PerCall)]
public class HotelBookingService : IHotelBookingService
{
   public BookingResponse BookHotel(Reservation request)
   {
      BookingResponse response = ProcessReservation(request);
      return response;
   }
}
```

Implementing the IHotelBookingService Service Contract

```
public class HotelBookingService :
IHotelBookingService
{
    public BookingResponse BookHotel(Reservation request)
    {
        BookingResponse response = BusinessLogic.ProcessReservation(request);

        return response;
    }

    public Booking GetExistingBooking(string bookingReference)
    {
        Booking booking = BusinessLogic.GetBookingByReference(bookingReference);

        return booking;
    }

    public string CancelReservation(string bookingReference)
    {
        string cancelationReference =
BusinessLogic.CancelBookingByReference(bookingReference);
    }
}
```

As you can see, implementation of a service requires you to implement the service contract interface and provide your business logic. Apart from providing concrete implementation of the service contract, you can control other aspects of the service behavior, such as the instantiation and concurrency model of the service:

- **Instantiation**. When you send a request to a service, the request is executed in an instance of the service class. The service instantiation controls when new instances of your service class are created.
- **Concurrency**. Each request in WCF runs in its own thread. But when several requests running in different threads are executing in parallel, they might attempt to use the same service instance

depending on the instantiation mode. The concurrency setting controls how many requests can use the same service instance concurrently.

The following code example demonstrates how to set the instantiation and concurrency of a service.

Setting the Instantiation and Concurrency of a Service

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall,
ConcurrencyMode=ConcurrencyMode.Single)]
public class HotelBookingService : IHotelBookingService
{
…
}
```

The instancing and concurrency modes are controlled by the **[ServiceBehavior]** attribute. You can control the instancing mode by adding the **InstanceContextMode** parameter, and the concurrency mode by adding the **ConcurrencyMode** parameter.

The instancing options supported in WCF are:

- **Per Call**. A new instance of your service class is created upon each request and destroyed after the request is complete.
- **Single**. A single instance is created for all requests and is destroyed when the service closes.
- **Per Session**. A new instance is created per client connection (session) and is destroyed when the client disconnects or is idle for too long (the default idle time is 10 minutes). This is the default setting.

When you use the **Per Session** or **Single** instancing modes, you can use one of the following concurrency modes to control the number of requests the same service instance can use at the same time:

- **Single**. Only a single request can execute in an instance at a time. This is the default setting.
- **Multiple**. Multiple requests can execute in an instance at a time.
- **Reentrant**. As with **Single**, only a single request can execute in an instance at a time. However, if your instance method calls another service, then instead of the instance being blocked while waiting for the call to return, the instance is released, and a waiting request can start using it.

 **Note:** When you use the **Per Call** instancing mode, each request executes in its own service class instance eliminating the need to manage concurrency issues.

WCF Sessions are not covered in depth in this course. For more information about instances, concurrency, and sessions in WCF, see:

 **Sessions, Instancing, and Concurrency**

http://go.microsoft.com/fwlink/?LinkID=298773&clcid=0x409

Handling Exceptions

- WCF does not return exception objects, but rather **SOAP Faults**
- A general fault is returned if you do not explicitly throw a **FaultException**
- Use **FaultException** with a simple text reason, or **FaultException<T>** with a detailed fault contract
- Fault contracts are defined using a standard data contract, and the **[FaultContract]** attribute should decorate the operation in the service contract

Applications must handle errors. These can include, for example, failed validation of user input or exceptions that may be thrown when the application tries to access a resource such as a file or a database.

Usually, when you develop a stand-alone application that does not involve a server, you can catch exceptions by using **try/catch** blocks. You can then have the application display a friendly error message that explains what happened and what the user can or should do next. If the application does not handle the exception, the .NET Framework will handle it instead, and display a default error dialog box.

However, handling errors in WCF is difficult. A WCF service cannot throw the exception back to the client for several reasons:

- The client may not understand the exception. The client may be running on a different platform, in which .NET Framework exceptions have no meaning.
- A .NET Framework exception exposes implementation details (stack trace, error codes). This is a bad practice because the client should not be familiar with the service implementation. For example, hackers can take advantage of information included in exceptions, such as database name, and file paths on your server.

If an exception cannot be thrown back to the client, what therefore is the solution? The answer is SOAP fault messages. Fault messages are special messages, in XML format, that contain data about an error that occurred on the service and where it originated. By default, an exception that is thrown back to the client from WCF, whether unhandled or thrown deliberately by the service developer, will result in the service returning a message with a SOAP fault message, containing a general error message.

Fault messages are part of the SOAP standard. They can be processed and handled by all compliant platforms, making them interoperable.

The following message shows the default fault message that is returned for unknown errors.

The Default SOAP Fault Error Message of WCF

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header />
  <s:Body>
    <s:Fault>
      <faultcode
xmlns:a="http://schemas.microsoft.com/net/2005/12/windowscommunicationfoundation/dispatch
er">a:InternalServiceFault</faultcode>
      <faultstring>The server was unable to process the request due to an internal error.
For more information about the error, either turn on IncludeExceptionDetailInFaults
(either from ServiceBehaviorAttribute or from the &lt;serviceDebug&gt; configuration
behavior) on the server in order to send the exception information back to the client, or
turn on tracing as per the Microsoft .NET Framework SDK documentation and inspect the
server trace logs.</faultstring>
    </s:Fault>
  </s:Body>
</s:Envelope>
```

If you read the message in the **faultstring** element, you will notice that WCF provides an option to include the actual exception information in the message, if you set the **IncludeExceptionDetailsInFaults**

parameter to **true** in the **[ServiceBehavior]** attribute. You also have the option of configuring this behavior in the service configuration file. You will learn about service configurations in the next lesson, and see how to use the **serviceDebug** configuration behavior in Lesson 3, "Configuring and Hosting WCF Services" in Course 20487.

 **Note:** Including the content of an exception in a SOAP fault message is recommended for debugging. As mentioned, exception messages can expose sensitive information about your service implementation, and non-.NET platforms will not know how to handle this type of content.

Instead of turning on the **IncludeExceptionDetailInFaults** flag, you can throw a **FaultException** that describes the reason for the problem.

The following code shows how to throw a **FaultException**.

Throwing a Fault Exception

```
public BookingResponse BookHotel(Reservation request)
{
    if (!IsHotelNameValid(request))
    {
     throw new FaultException("Unable to find a match for the requested hotel");
    }

    BookingResponse response = BusinessLogic.ProcessBookingRequest(request);

    return response;
}
```

When you throw a **FaultException** instead of a standard exception, the service includes the exception text in the fault section of the SOAP envelope without disclosing sensitive exception information such as the stack trace of the exception. This will make the exception text available to your client, no matter which platform it is written in.

However, sometimes just writing a fault string is not enough, and you will want the fault to contain more information, such as the problematic data that caused the exception or a unique error ID number that your customers can use to contact the service administrator for further investigation. In such a case, you can use the **FaultException<T>** generic exception class. The generic argument **T** specifies the data contract that holds extra information about the error.

The following code returns a fault with extra details contained in a **ReservationFault** object.

Throwing a Detailed Fault

```
public BookingResponse BookHotel(Reservation request)
{
    if (!IsHotelNameValid(request))
    {
     throw new FaultException<ReservationFault>(
       new ReservationFault
       {
         HotelName = request.HotelName,
         ErrorCode = "InvalidHotelName"
       }, " Unable to find a match for the requested hotel");
    }

    BookingResponse response = BusinessLogic.ProcessBookingRequest(request);

    return response;
}
```

The **ReservationFault** class used in the above example is a simple data contract, and does not derive from any **Exception** class.

The following code depicts the declaration of the **ReservationFault** class.

The ReservationFault Class

```
 [DataContract]
public class ReservationFault
{
    [DataMember]
    public string HotelName { get; set; }

    [DataMember]
    public string ErrorCode { get; set; }
}
```

Because clients only recognize data contracts if their classes are used in the method signatures of operations, they will not be aware of data contract used in a **throw** statement within the code. Therefore, you must add the data contract of the fault to the service contract manually. You can include information about fault contracts (the data contracts used in faults), by adding the **[FaultContract]** attribute to the operations that can throw such faults.

The following code shows how to include a fault contract in a service contract.

Service Contract with a Fault Contract

```
 [ServiceContract]
public interface IHotelBookingService
{
    [OperationContract]
  [FaultContract(typeof(ReservationFault))]
    BookingResponse BookHotel(Reservation request);

    [OperationContract]
    Booking GetExistingReservation(string bookingReference);

    [OperationContract]
    string CancelReservation(string bookingReference);
}
```

For an example of how to call the **BookHotel** service operation from a client application, and how to handle the **ReservationFault** fault exception, refer to Lesson 4 in this module, "Consuming WCF Services," and look at the code example shown in the first topic, "Generating Service Proxies with Visual Studio 2012."

Best Practice: It is advised that all service operations use the **[FaultContract]** attribute to inform the client of every kind of fault it can expect to receive.

For more information about handling faults in WCF, see:

Specifying and Handling Faults in Contracts and Services

http://go.microsoft.com/fwlink/?LinkID=298774&clcid=0x409

Question: Why should you avoid returning the entire **Exception** object from a service?

Demonstration: Creating a WCF Service

This demonstration shows how to create a service contract and a data contract, how to implement the service, and how to test the service with the WCF Test Client tool.

Demonstration Steps

1. In Visual Studio 2012, open **D:\Allfiles\Mod05\DemoFiles\CreatingWCFService\begin\CreatingWCFService.sln**. Explore the code in the files of the **Service** project, and notice the use of service contract and data contract attributes.
2. Add a reference to the **System.ServiceModel** and **System.Runtime.Serialization** assemblies.
3. In the **IHotelBookingService** interface, decorate the interface with the **[ServiceContract]** attribute, and decorate the **BookHotel** method with the **[OperationContract]** attribute.
4. Decorate the **BookingResponse** and **Reservations classes** with the **[DataContract]** attribute. Decorate each of their properties with the **[DataMember]** attributes.
5. Run the service in debug and test it by using the built-in WCF Test Client. Run the **BookHotel** operation with the **HotelName** set to **HotelA**, and verify that the response shows the booking reference **AR3254**.

   Question: What are the steps for creating a service contract?

Lesson 3
Configuring and Hosting WCF Services

In the previous lesson, we discussed the logical aspects of WCF services: SOAP messaging, service contracts, data contracts, and service implementation. However, these only explain what the service can do. There are many other questions to be answered:

- What is the address of the service?
- What communication protocols does it support?
- How are the service implementation instances created?
- What are the security characteristics of the service (such as authentication and encryption)?

These are some of the questions answered in this lesson. The features of WCF that control these aspects are the *service host*, *service configuration*, and *endpoint configuration*.

This lesson explains how to host a WCF service, how to configure and control the behavior of a service host, and how to define and configure the service endpoints.

Lesson Objectives

After you complete this lesson, you will be able to:

- Host a WCF service.
- Explain the basics of WCF service endpoints.
- Set the address of a service endpoint.
- Set and configure the binding of a service endpoint.
- Define the contract used by the service endpoint.
- Expose the service metadata with metadata exchange endpoints.

Hosting WCF Services

In the previous lesson, we discussed how to use service contracts and data contracts to describe what a service can do, and how to provide the actual implementation of the service. Implementing a service is not enough to make it run, you also need a host to listen for incoming requests and direct them to your service. The service host prepares the service implementation class that will be addressed by clients, allocating the resources the service requires and managing the execution state of the service.

The service host is responsible for opening ports and listening to requests according to the configuration. The host manages the incoming requests of service, allocates resources such as memory and threads, creates the service instance context, and then passes the instance context through the WCF runtime.

There are two ways to host a WCF service:

- **Self-hosting**. Create your own application, such as a Windows Presentation Foundation (WPF) application or a Windows Service. The host will start after the application starts, and shutdown when the application shuts down.
- **Web hosting**. Hosts the service in IIS. The host will start after IIS receives the first request to the service, and shut down when the web application shuts down.

This lesson focuses on the basics of self-hosting with a simple Console application. Module 6, "Hosting Services," will explain in depth how you can self-host WCF services in Windows Services, and how you can host WCF services with IIS.

The base class that manages WCF hosts is the **ServiceHostBase** type, but this class is an abstract class. The concrete class that you will use to host your services is the **ServiceHost** type, which is declared in the **System.ServiceModel** assembly.

 **Note:** There are other technology-specific service host classes that derive from the **ServiceHostBase** class, such as **WorkflowServiceHost**, which is used to host services that execute Windows Workflow (WF) activities.

The following code demonstrates how to host a WCF service with the **ServiceHost** class.

Hosting a WCF Service with ServiceHost

```
ServiceHost hotelServiceHost = new ServiceHost(typeof(Services.HotelBookingService));

hotelServiceHost.Open();

Console.WriteLine("Service has been hosted. Press Enter to stop");
Console.ReadLine();

hostelServiceHost.Close();
```

A **ServiceHost** instance can manage a single service type, but it can open many listeners for that service type, each with a different configuration. For example, a single **ServiceHost** can listen to both HTTP and UDP communication, and invoke a service method when a request is received on either of these transports.

 **Note:** If you have more than one service, you will need a different instance of **ServiceHost** for each service.

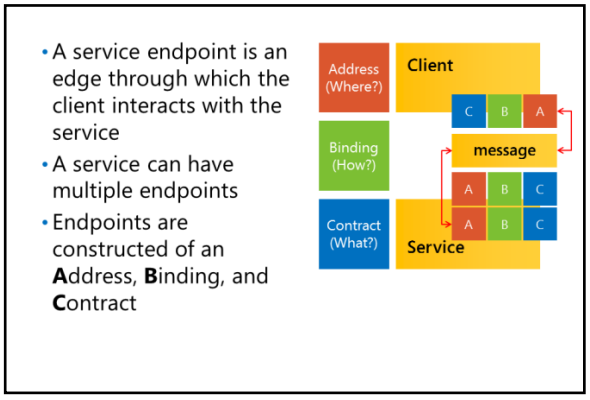
Before the service is opened by using the **Open** method, the host requires configuration that instructs it which communication transports it needs to listen to and at which addresses. You can set this configuration either in the code itself, before calling the **Open** method, or in the application configuration file (app.config). You will learn to create this configuration, both in code and in the configuration file, later in this lesson.

After the service host has opened, you can close it at any time by calling the **Close** method. The **Close** method will stop the host from listening to any communication, making the service unavailable for clients.

 **Best Practice:** The **Open** and **Close** methods can throw an exception if the service host has difficulties listening to ports, such as if a port is already opened by another application. Although the code sample shown above does not demonstrate this, it is advised to wrap the **Open** method and **Close** method calls with a **try/catch** block.

Service Endpoints Overview

As mentioned in previous topics, the service host must listen to one or more communication channels so that it can receive requests from clients. In WCF, these listeners are called endpoints, and the configuration of each listener is referred to as an endpoint configuration.



An endpoint is the end of the client-host communication channel, and therefore is the entry point to the service. An endpoint receives messages from a communications channel and transfers that message to the service implementation for processing. A service can have numerous endpoints, each one listening to a different type of communication, such as HTTP, UDP, or TCP. Each endpoint has a different address to distinguish it from other endpoints of that service and from those of other services running on the same machine.

A service endpoint defines how the service is exposed to the clients. A service endpoint answers the following three questions:

- **A**ddress. Specifies where the service resides. The address is a Uniform Resource Locator (URL) that is used by the client applications to locate the service.
- **B**inding. Specifies how clients should communicate with the service. The binding specifies the message encoding, transport type, security modes, session support, and other protocols.
- **C**ontract. Specifies the operations supported by the endpoint. The contract needs to match one of the contract interfaces implemented by your service class.

These endpoint settings are called the ABCs of an endpoint.

The following code demonstrates how to add an endpoint through code.

Configuring an Endpoint in Code

```
ServiceHost hotelServiceHost = new ServiceHost(typeof(Services.HotelBookingService));

hotelServiceHost.AddServiceEndpoint(
   typeof(Contracts.IHotelBookingService),
   new BasicHttpBinding(),
   "http://localhost:8080/booking/");

hotelServiceHost.Open();
```

The above code adds a single endpoint to the service with the following configuration:

- **Contract**. The **IHotelBookingService** service contract. If a service has more than one contract, you must create several endpoints, one for each contract.
- **Binding**. The endpoint is configured to use **BasicHttpBinding**. This binding listens to HTTP communication, and expects XML messages with SOAP envelopes. You can have multiple endpoints with different bindings. Bindings will be explained in detail in the next topic.
- **Address**. The address **http://localhost:8080/booking** is the listening address of the endpoint. When a client sends a message to the service, it will send the message to **http://*ServerName*:8080/booking/**, where **ServerName** is the DNS or IP address of the server hosting the service.

The following XML configuration demonstrates how to add an endpoint in the application configuration file.

Configuring an Endpoint in a Configuration File

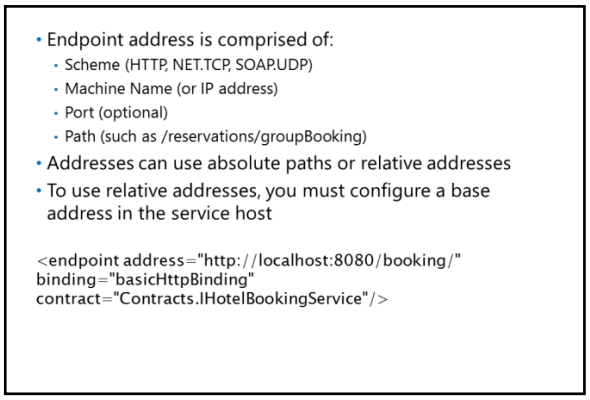
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
          <service name="Services.HotelBookingService">
            <endpoint
                address="http://localhost:8080/booking/"
                binding="basicHttpBinding"
                contract="Contracts.IHotelBookingService">
            </endpoint>
          </service>
        </services>
    </system.serviceModel>
</configuration>
```

The WCF configuration shown in the above example is contained in the **<system.serviceModel>** configuration section group. The **<services>** section contains the list of services, each in its own **<service>** element. The **name** attribute in the **<service>** element is set to the fully qualified name of the service implementation class. Each **<service>** element can contain **<endpoint>** elements, and each such element contains its *ABC* settings (address, binding, and contract).

Defining a Service Endpoint Address

The endpoint address identifies an endpoint uniquely and informs potential clients of its location. The endpoint address has the following parts:

- Scheme (HTTP, TCP)
- Machine name (or IP address)
- Port (optional)
- Path (for example /reservations/groupBooking/)



There are two ways to specify an address for an endpoint in WCF:

- Specify an absolute address for each endpoint.
- Specify a base address for the service host and then specify a relative address for each endpoint.

In each case, you can specify the address both in code and in configuration.

The following example demonstrates how to use relative endpoint addresses in the configuration file.

Using a Relative Endpoint Address in Configuration Files

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
          <service name="Services.HotelBookingService">
            <host>
```

```
            <baseAddresses>
               <add baseAddress="http://localhost:8080/reservations/" />
            </baseAddresses>
         </host>
         <endpoint
             address=""
             binding="basicHttpBinding"
             contract="Contracts.IHotelBookingService ">
         </endpoint>
         <endpoint
             address="secured"
             binding="wsHttpBinding"
             contract="Contracts.IHotelBookingService ">
         </endpoint>
         <endpoint
             address="net.tcp://localhost:8081/reservations/"
             binding="netTcpBinding"
             contract="Contracts.IHotelBookingService ">
         </endpoint>
      </service>
   </services>
</system.serviceModel>
</configuration>
```

In the above example, the service configuration has three endpoints for the **IHotelBookingService** contract. The first two endpoints use the **http://localhost:8080/reservations/** base address specified in the **<host>** element. Of these, the first endpoint uses an empty relative address, so the actual address of the endpoint is the same as the base address. The second uses the **secured** relative address, so its actual address is **http://localhost:8080/reservations/secured**. The third endpoint uses the **net.tcp://localhost:8081/reservations/** address. The third address cannot use the HTTP base address, since it uses a different binding (TCP rather than HTTP). You will learn about bindings in the next topic.

 **Note:** The service host matches the address and the base address according to the binding of the endpoint and the scheme of the base addresses. For example, both **basicHttpBinding** and **wsHttpBinding** use HTTP communication, and therefore the service host will search for a base address with the **HTTP** scheme. Therefore, you can have only one base address per URI scheme.

The **http://** URI scheme in endpoint addresses is part of the HTTP URI structure, but most other addresses do not have URI schemes and are WCF-proprietary. This is why some address schemes have the "net." prefix, such as **net.tcp://** and **net.pipe://**. The **soap.udp://** scheme and **ws://** scheme are different, as these schemes do have URI structures: **soap.udp** is used in UDP URIs, and **ws** is used in WebSocket URIs.

You can also use relative endpoint addresses when you declare endpoints in code. You can specify the base address of the service host in the **ServiceHost** constructor method.

The following code demonstrates how to use relative endpoint addresses in code.

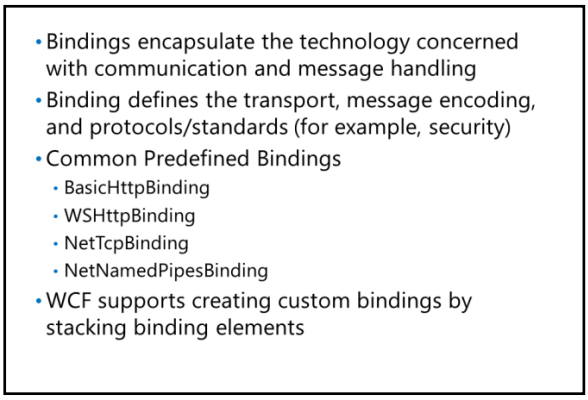
Using a Relative Endpoint Address in Code

```
ServiceHost hotelServiceHost = new ServiceHost(
   typeof(HotelBookingService), new Uri("http://localhost:8080/reservations/"));

hotelServiceHost.AddServiceEndpoint(
    typeof(IHotelBookingService), new BasicHttpBinding(), "");

hotelServiceHost.AddServiceEndpoint(
    typeof(IHotelBookingService), new WSHttpBinding(), "secured");

hotelServiceHost.Open();
```

Defining Service Endpoint Bindings

When a client sends a message to a service, it handles many technology related issues, such as the kind of transfer protocol to use, the message be streamed or buffered, the XML be sent as text or as binary, and the message needs to contain a security header of some sort.

To make those choices easily, WCF offers the binding mechanism. Binding encapsulates all the technology decisions required to pass a message from point A to point B:



- The binding defines which transport to use to send the messages.
- The binding defines how to encode the messages onto the wire.
- The binding defines which protocols, such as security and sessions, are required.

A binding in WCF is a combination of these three elements: transport, encoding, and protocols. In WCF, you can create all the definitions and configurations of the binding in a single place - either in code or in configuration files - which simplifies the amount of work required.

Note: The binding also defines the properties of the communications channel and messages, such as timeouts and maximum message size.

Predefined Bindings

Instead of setting the three elements of the binding each time you define an endpoint, WCF provides a collection of predefined bindings for the most common combinations of binding elements. You can use these bindings with their default values or fine-tune the bindings to your needs. The following are some typically used predefined bindings:

Binding name	Transport	Encoding	Usage
BasicHttpBinding	HTTP	Text	Interoperability with older service technologies. Uses SOAP 1.1 with no WS-* protocols.
WSHttpBinding	HTTP	Text	Interoperability with new service technologies. Uses SOAP 1.2, and supports all WS-* protocols.
NetTcpBinding	TCP	Binary	Non-interoperable binding. Optimized for intranet communication scenarios. Supports all WS-* protocols.
NetNamedPipeBinding	Named pipes	Binary	Non-interoperable binding. Optimized for inter-process communication on the same machine. Lacks some of the WS-* options, such as message security.
UdpBinding	UDP	Text	Interoperable with service

Binding name	Transport	Encoding	Usage
			technologies that implement SOAP-over-UDP. Lacks some of the WS-* options, such as reliable messaging.
NetHttpBinding	HTTP/WebSockets	Binary	Non-interoperable binding. Optimized for Internet communication and duplex communication with WebSockets. Supports all WS-* protocols.

For more information on predefined bindings available in WCF, see:

Configuring System-Provided Bindings

http://go.microsoft.com/fwlink/?LinkID=298775&clcid=0x409

Although most bindings work even in scenarios for which they are not designed, it is a good practice to choose the correct binding for a given endpoint. There are many considerations to take into account when deciding which binding to use. Covering all those is beyond the scope of this topic. Here however are some examples:

- Bindings that support end-to-end security:
  - **NetTcpBinding**
  - **WSHttpBinding**
- Interoperable bindings you can use with other service platforms:
  - **BasicHttpBinding**
  - **WsHttpBinding**
  - **UdpBinding** (most service platforms do not currently support SOAP-over-UDP)
- Bindings that cross firewalls easily:
  - **BasicHttpBinding**
  - **WSHttpBinding**
- Bindings that support duplex (two-way) communication:
  - **NetTcpBinding**
  - **WSDualHttpBinding**
  - **NetHttpBinding**

Configuring Bindings

Each binding has a set of configurable properties that you can change to modify the binding to your needs. You can change the settings either through code or by using the configuration file.

The following example shows how you can configure the basic HTTP binding in the configuration file.

Configuring a Binding in a Configuration File

```
<system.serviceModel>
     <bindings>
          <basicHttpBinding>
               <binding name="increasedSettings"
                        sendTimeout="00:10:00"
                        receiveTimeout="00:30:00"
```

```
                        maxReceivedMessageSize="5000000"/>
            </basicHttpBinding>
        </bindings>
</system.serviceModel>
```

- The **<bindings>** element needs to be placed inside the **<system.serviceModel>** element in the configuration file.
- Inside the **<bindings>** element, place an element by the name of the binding type that you wish to change. Note that the name of the binding is in camel case (the first letter of the first word is in lowercase, and each subsequent word is capitalized).
- Inside this element, place a **<binding>** element and give it a name. The **name** attribute will then be applied to any endpoints that make use of that binding.
- Inside the **<binding>** tag, add the attributes and elements that you wish to set.

In the previous example, three attributes were changed. The **sendTimeout** attribute, which sets the maximum time a service waits for a message to be sent, was changed to 10 minutes. The **receiveTimeout** attribute, which sets the maximum time a service waits until a message is fully received, was changed to 30 minutes. The **maxReceivedMessageSize** attribute, which sets the maximum allowable size of a message sent to the service, was set to 5000000 bytes.

To apply this binding to an endpoint, you will need to set the binding configuration of the endpoint accordingly.

The following example shows how to configure an endpoint with the new binding configuration.

Configuring an Endpoint with a Modified Binding Configuration

```
<services>
   <service name="Services.HotelBookingService">
      <endpoint
          address="http://localhost:8080/booking/"
          binding="basicHttpBinding"
      bindingConfiguration="increasedSettings"
          contract="Contracts.IHotelBookingService">
      </endpoint>
   </service>
</services>
```

In addition to modifying the binding in configuration, you can also modify the binding in code. All the predefined bindings are exposed as .NET classes. You can configure a binding by setting the properties of the binding instance object.

The following code sets the same binding configuration as the previous code example, this time in code.

Configuring a Binding in Code

```
ServiceHost hotelServiceHost = new ServiceHost(typeof(Services.HotelBookingService));

BasicHttpBinding basicHttpWithIncreasedSettings = new BasicHttpBinding
  {
    ReceiveTimeout = TimeSpan.FromMinutes(30),
    SendTimeout = TimeSpan.FromMinutes(10),
    MaxReceivedMessageSize = 5000000
  };

hotelServiceHost.AddServiceEndpoint(
    typeof(Contracts.IHotelBookingService),
    basicHttpWithIncreasedSettings,
    "http://localhost:8080/booking/");
```

```
hotelServiceHost.Open();
```

Custom Binding

In addition to predefined bindings, you can create your own custom binding. When using a custom binding, you can select the binding elements that compose the binding. You can define the transport element, the message encoding, and other elements such as security and transaction support. You can define custom bindings by adding a **<customBinding>** element to the **<bindings>** section in the configuration file, or by creating a new instance from the **CustomBinding** class in code.

 **Note:** To use the **CustomBinding** class in your code, add a using directive for the **System.ServiceModel.Channels** namespace.

The following example demonstrates how to create a custom binding that uses HTTP and binary XML encoding.

Creating Custom Bindings

```
<system.serviceModel>
    <bindings>
   <customBinding>
    <binding name="compressedBinaryHttpBinding">
        <reliableSession/>
        <binaryMessageEncoding compressionFormat="GZip"/>
        <httpTransport/>
    </binding>
   </customBinding>
    </bindings>
    <services>
      <service name="HotelBooking.HotelBookingService">
         <endpoint
             address="http://localhost:8080/booking/"
       binding="customBinding"
       bindingConfiguration="compressedBinaryHttpBinding"
             contract="Contracts.IHotelBookingService">
         </endpoint>
      </service>
    </services>
  </system.serviceModel>
```

The custom binding created in the above example uses HTTP transport. Instead of encoding the message as plain text it encodes the message by using a WCF-proprietary binary encoding and compresses the content with GZIP so that it can be sent faster on slow networks. In addition, the binding uses the **WS-ReliableMessaging** protocol, which helps to cope with network failures while sending messages from end to end.

This course does not cover the reliable messaging support of WCF. For more information on reliable messaging in WCF, see:

 **Introduction to Reliable Messaging with the Windows Communication Foundation**

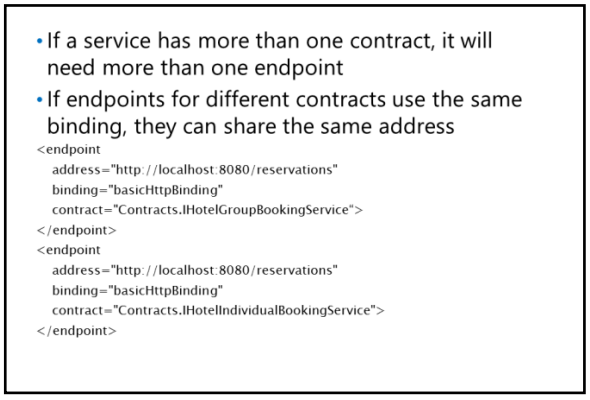
http://go.microsoft.com/fwlink/?LinkID=298776&clcid=0x409

Question: What are the advantages of using the built-in bindings rather than creating custom bindings?

Defining Service Endpoint Contracts

The final setting of the endpoint is the service contract. The endpoint must indicate the exposed service contract and, in so doing, indicates the operations supported by the endpoint.

As you saw before, a service can have multiple endpoints if it wants to expose a contact through different bindings. Another case in which you will create multiple endpoints for a service is when the service has multiple service contracts (that is, implements multiple interfaces).



If your service class implements more than one contract, you will need to create multiple endpoints to expose these contracts to clients. It is common to create these multiple endpoints such that they use the same binding and binding configuration, but have different addresses.

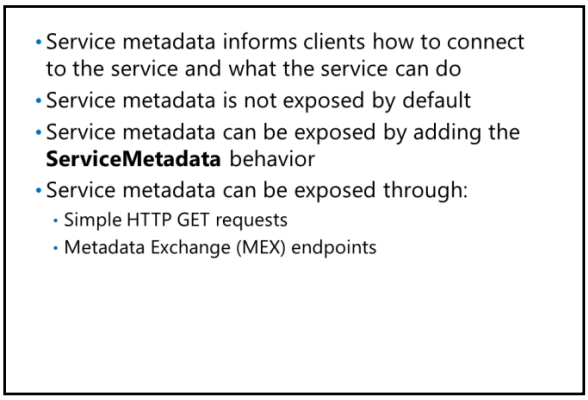
The following example shows two endpoints, each for a different contract.

Endpoint Per Service Contract

```
<system.serviceModel>
     <services>
       <service name="Services.HotelBookingService">
         <endpoint
             address="http://localhost:8080/reservations/GroupBooking"
             binding="basicHttpBinding"
             contract="Contracts.IHotelGroupBookingService">
         </endpoint>
         <endpoint
             address="http://localhost:8080/reservations/IndividualBooking"
             binding="basicHttpBinding"
             contract="Contracts.IHotelIndividualBookingService">
         </endpoint>
       </service>
     </services>
</system.serviceModel>
```

If you create different endpoints for different contracts but use the same binding and binding configuration for all the endpoints, you can use the same address for all the endpoints. In the above example, both addresses could be replaced with the same address, **http://localhost:8080/reservations**. In such a case, WCF will automatically identify which endpoint was addressed by checking the message headers. Every message sent to a service has an **Action** header that holds the name of the contract and the name of the requested operation.

Exposing Service Metadata



The last step before clients can start consuming your service is to expose the service contract and endpoint information. This information is known as the service metadata. The service metadata includes all the information that is required to interact correctly with the service.

WCF exposes service metadata in the form of Web Services Definition Language (WSDL) documents. WSDL is an XML-based format that is used to describe the functionality offered by a service. A WSDL description of a service provides a list of service operations, the parameters and data structures each operation expects, and any security or messaging policies the client has to obey.

In WCF, the WSDL of the service describes the following:

- Service contracts
- Data contracts
- Fault contracts
- Service endpoint addresses
- Binding-related policies, such as security and transactions

WCF services expose metadata by using two techniques:

- **HTTP GET**. You can get the WSDL document by sending a simple GET request over HTTP to the service. This technique is useful if you want other developers to download your WSDL through a browser. This technique is also useful to test if your service has loaded correctly (by browsing to the WSDL address and verifying that you do not get any exceptions).
- **Metadata Exchange (MEX) endpoints**. You can get the WSDL document by calling a special service endpoint that uses SOAP messages with the WS-MetaDataExchange protocol. If you expose a service metadata as an endpoint rather than over HTTP, you have more control over the type of transport you use, and you can control other binding-related configurations. For example, you can decide if you want to expose the service metadata over TCP and prevent unauthorized clients from accessing the metadata.

By default, for security reasons, WCF services do not expose their metadata. If you want to expose your service metadata, you will need to change the behavior of your service. You will recall from earlier in this module that you can control service behavior through the **[ServiceBehavior]** attribute. However, exposing metadata is one of several service behaviors that cannot be controlled through the service code with the **[ServiceBehavior]** attribute. Instead, you must configure it either through the **ServiceHost** class or in the service configuration file.

 **Note:** Some behaviors, such as concurrency and instancing, are more development-oriented. Others, such as service metadata, are more deployment-and-hosting-oriented. Development-related behaviors are set in the service implementation by using attributes, while hosting-related behaviors are set in the host project (either in the configuration file or in the service host code, in the **ServiceHost** instance).

To add a service behavior configuration to your configuration file, open your application configuration file and perform the following steps:

1. Add a **<behaviors>** section to the **<system.serviceModel>** section group, and in it create a **<serviceBehaviors>** element.

 **Note:** In addition to service behaviors, you can also use the **<behaviors>** section to configure endpoint behaviors. Endpoint behaviors are beyond the scope of this course.

2. Add a new **<behavior>** element to the **<serviceBehaviors>** element, and set its **name** attribute to a name describing the use of the behavior.
3. Add service behavior elements to the **<behavior>** element to configure various behaviors of your service and your hosting environment.
4. After you create the service behavior configuration, set your service to use that configuration by adding the **behaviorConfiguration** attribute to your **<service>** element and setting the value of the attribute to the name of the service behavior. As long as you created the service behavior in the configuration file first, Visual Studio 2012 will open a drop-down list showing the names of the existing service behaviors when you add the **behaviorConfiguration** attribute.

You can change the behavior of your service so that it exposes metadata by creating a service behavior element and adding the **<serviceMetadata>** element to it.

The following code demonstrates how to configure a service to expose metadata.

Exposing Metadata of a Service

```
<system.serviceModel>
   <behaviors>
    <serviceBehaviors name="metadata">
     <behavior>
       <serviceMetadata httpGetEnabled="True"/>
     </behavior>
    </serviceBehaviors>
   </behaviors>
     <services>
       <service name="Services.HotelBookingService" behaviorConfiguration="metadata">
         <endpoint
             address="http://localhost:8080/reservations/GroupBooking"
             binding="basicHttpBinding"
             contract="Contracts.IHotelGroupBookingService">
         </endpoint>
       </service>
     </services>
</system.serviceModel>
```

The above example adds the **<serviceMetadata>** element, which changes the default behavior of the service so that it exposes its metadata. You can also control how the service exposes metadata by adding the **httpGetEnabled** attribute and setting it to **true**. This attribute configures the service to expose the metadata with simple HTTP GET requests.

 **Note:** You can set other attributes of the **<serviceMetadata>** element to control how the metadata is exposed. For example, you can set the **httpsGetEnabled** attribute to **true** to expose the metadata over HTTPS.

In addition to the **<serviceMetadata>** behavior, many more behaviors are available, such as the **<serviceDebug>** behavior that was mentioned in Lesson 2, "Creating and Implementing a Contract." You can also define the service behaviors in code before opening the service host.

 **Note:** If you are going to host multiple services in the same hosting project, and you want several services to use the same behavior configuration, you can omit the **behaviorConfiguration** attribute from the **<service>** element and the **name** attribute from the **<behavior>** element. Behaviors without a name will automatically apply to every service that does not have a specific service behavior configuration.

You can also configure service behavior in code before opening your service host.

The following code demonstrates how to add service behaviors to the service host.

Using Service Behaviors in Code

```
ServiceHost hotelServiceHost = new ServiceHost(typeof(Services.HotelBookingService));

hotelServiceHost.Description.Behaviors.Add(new ServiceMetadataBehavior { HttpGetEnabled =
true });
hotelServiceHost.Description.Behaviors.Add(new ServiceDebugBehavior {
IncludeExceptionDetailInFaults = true });

hotelServiceHost.Open();
```

The above example adds two behaviors, **ServiceMetadataBehavior**, and **ServiceDebugBehavior**. You can find these two behaviors in the **System.ServiceModel.Description** namespace. You can mix adding behaviors in code and in configuration, but make sure you do not add the same behavior twice.

If you prefer exposing your service metadata with a Metadata Exchange (MEX) endpoint, you can do so by adding such an endpoint to your service endpoints list.

The following configuration demonstrates how to add a MEX endpoint.

Adding a MEX Endpoint

```
<system.serviceModel>
     <behaviors>
       <serviceBehaviors>
         <behavior>
            <serviceMetadata/>
         </behavior>
       </serviceBehaviors>
     </behaviors>
     <services>
       <service name="Services.HotelBookingService">
         <endpoint
             address="http://localhost:8080/reservations/GroupBooking"
             binding="basicHttpBinding"
             contract="Contracts.IHotelGroupBookingService"/>
         <endpoint
             address="http://localhost:8080/reservations/MEX"
             kind="mexEndpoint"/>
       </service>
     </services>
</system.serviceModel>
```

As you can see, the new MEX endpoint has no binding and no contract, but instead has the **kind** attribute. The **kind** attribute is used when creating standard endpoints, such as MEX endpoints and service discovery endpoints. When you use the **mexEndpoint** kind, the endpoint is configured to use HTTP

binding and the **IMetadataExchange** contract automatically. The **IMetadataExchange** contract is a special built-in contract that handles WS-MetaDataExchange messages.

Note: Instead of using the **kind** attribute, you can set the endpoint to use the **mexHttpBinding** binding and the **IMetadataExchange** contract. This has the same result as using the **kind** attribute.
You can change the binding attribute if you want to use non-HTTP bindings for the MEX endpoint, such as **mexHttpsBinding** and **mexTcpBinding**, and you can provide additional binding configuration if required.

In a single project, if you find yourself hosting more and more services, with more and more contracts, you will have a very big configuration file to manage. Instead of writing the configuration file yourself, you can use the **Microsoft Service Configuration Editor** tool (**SvcConfigEditor.exe**). **SvcConfigEditor.exe** is a graphical utility that you can use to add new services and service endpoints to your configuration file, and to edit WCF settings such as the binding configuration and service behaviors. You can open the **Service Configuration Editor** in Visual Studio 2012 from the **Tools** menu (on the **Tools** menu, click **WCF Service Configuration Editor**), or in Solution Explorer, by right-clicking **App.config** and clicking **Edit WCF Configuration**.

The following screenshot shows the Service Configuration Editor tool.

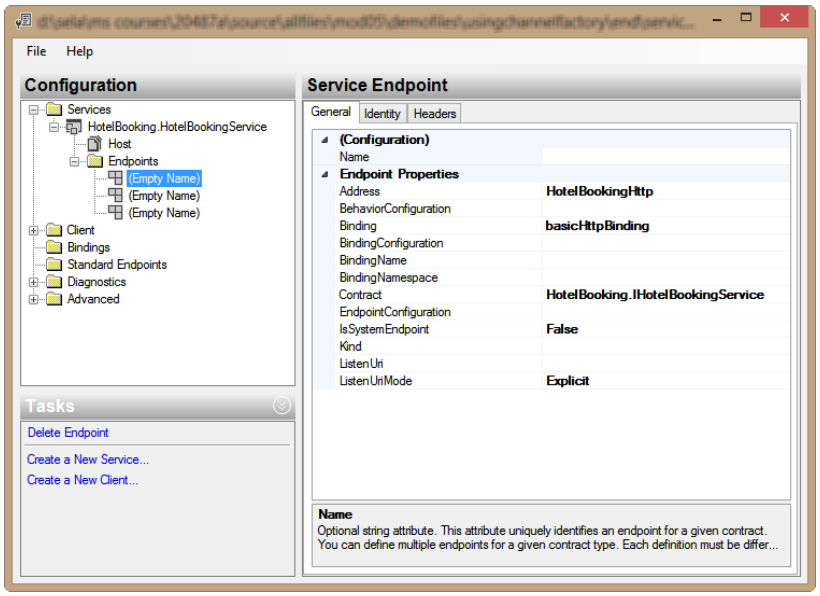


FIGURE 5.1: THE SERVICE CONFIGURATION EDITOR TOOL

For more information on how to use the WCF Service Configuration tool, see:

Configuration Editor Tool (SvcConfigEditor.exe)

http://go.microsoft.com/fwlink/?LinkID=298777&clcid=0x409

Demonstration: Configuring Endpoints in Code and in Configuration

This demonstration shows how to add endpoints to a service in configuration and in code.

Demonstration Steps

1. Open **D:\Allfiles\Mod05\DemoFiles\DefineServiceEndpoints\begin\DefineServiceEndpoints.sln**.

2. In the **ServiceHost project**, open the **App.config** file, locate the **<serviceBehaviors>** section, and then add a **<behavior>** element with the **serviceMetadata** behavior.

Note: Refer to Topic 6, "Exposing Service Metadata" in this lesson for a code sample of how to add the **serviceMetadata** behavior.

3. In the **App.config**, add a base address to the **<service>** element by using the address **http://localhost:8733/**.

Note: Refer to Topic 3, "Defining a Service Endpoint Address" in this lesson for a code sample of how to add a base address in configuration.

4. Save the changes you made to the **App.config** file, open the file with the WCF Configuration Editor, and then add a new endpoint to the service.

- To open the **App.config** file with the WCF Configuration Editor tool, right-click **the App.config file in Solution Explorer,** and then click **Edit WCF Configuration.**
- Add a new service endpoint with the following settings:

Property	Value
Address	**booking**
Binding	**basicHttpBinding**
Contract	**HotelBooking.IHotelBookingService**

- Save the changes you have made to the confirmation, and then close the Service Configuration Editor window.

5. In the **ServiceHost** project, open the **Program.cs**, and add an endpoint that uses **NetTcpBinding**.
   - Add the endpoint before calling the **host.Open** method.
   - Call the **host.AddServiceEndpoint** method by using the following code.

```
host.AddServiceEndpoint(typeof(IHotelBookingService), new NetTcpBinding(),
"booking");
```

6. Run the service host in debug and test it by using the built-in WCF Test Client. Run the **BookHotel** operation by using the TCP endpoint. Set the **HotelName** to **HotelA**, and then verify that the response shows the booking reference **AR3254**.
7. Browse to the base address of the service, and then view the WSDL file with the service metadata.
8. Close the browser and the WCF Test Client. Return to Visual Studio, stop the debugger, and then close Visual Studio 2012.

   Question: Give several examples for creating service configuration from code instead of using configuration files.

Lesson 4
Consuming WCF Services

The last step in developing a service is consuming a WCF service, which is performed once to make the service run. There are several ways to consume a WCF service. The most productive way is to use WCF on the client side. However, you can also consume a service from non-.NET clients.

WCF and Visual Studio 2012 provide different tools that can help you consume services easily. In this lesson, you will learn how to use Visual Studio 2012 to generate a service proxy class in design-time, and how to create a service proxy at run time with the **ChannelFactory<T>** generic class. You will also learn how to use the proxy created with these two techniques to call your service.

Lesson Objectives

After you complete this lesson, you will be able to:

- Generate a client proxy with the **Add Service Reference** dialog box of Visual Studio 2012.
- Create a client proxy with the **ChannelFactory<T>** generic class.

Generating Service Proxies with Visual Studio 2012

For your client application to consume a service, it needs to communicate with the service, send it the correct messages, and translate the returned message. To enable a client to communicate with a service without having to manage all the implementation that is required to transfer the message, WCF uses the proxy pattern.

- A proxy reflects an entity over a technology boundary
- The proxy translates method calls to messages exchanged over a transport
- Use the **Add Service Reference** dialog box in Visual Studio 2012 to generate the proxy class
- All the service contracts and data contracts are reflected in the client project
- The WCF client configuration is appended to the client's configuration file

A proxy is an adapter that reflects the capabilities of a service over the networking and messaging technological boundaries that are used by the service. A proxy on one side reflects the service contract by implementing the same contract (interface) that is exposed by the service. On the other side, each operation the proxy implements internally performs all the necessary operations that are required to communicate with the service. The following steps, implemented by the proxy, are required when communicating with a service:

1. Serialize the request to a message object.
2. Open a channel to the service and send the message to the service, according to the required transport and other binding settings.
3. Wait for the service to respond to the request and send its response.
4. Deserialize the message and return the value to the caller.

If you want to use the proxy pattern to consume a service, you need to build a class that implements the service contract and that is responsible for all the transformations and communication with the service. WCF can build those proxy classes by using the **Add Service Reference** dialog box of Visual Studio 2012. This tool reads WSDL documents, extracts the service contract, and creates proxy classes that match the service contracts. In addition, this tool also creates data classes according to the data contracts specified in the WSDL file. To use the **Add Service Reference** dialog box:

1. In **Solution Explorer**, right-click your project, and then click **Add Service Reference**.

2. In the **Add Service Reference** dialog box, enter the WSDL file address of the service, and then click **Go**. WCF will try to connect with the service and request the service's WSDL file.

Note: If you are trying to consume a WCF service that you have developed, make sure the service has exposed its WSDL document. To expose the document, the service must use the **ServiceMetadata** service behavior.

3. After Visual Studio finds the WSDL, the list of service contracts will display. Enter the name of the namespace with which you want to create proxies, and then click **OK**. WCF will create the proxy classes that are required for every service contract and data contract classes exposed in the WSDL. Visual Studio will also place all service endpoint configurations in the configuration file of the client. This allows you to use the proxy easily with any of the service's endpoints.

The following screenshot shows the **Add Service Reference** dialog box in Visual Studio 2012.

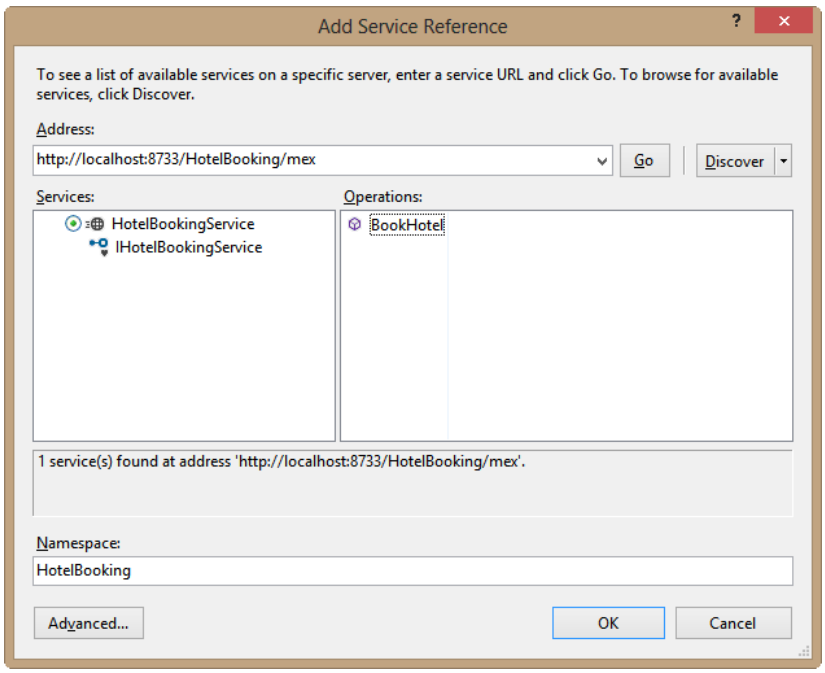


FIGURE 5.2: THE ADD SERVICE REFERENCE DIALOG BOX IN VISUAL STUDIO 2012

The **Advanced** button in the **Add Service Reference** dialog box opens a window in which you can configure code generation settings. You can configure settings such as the accessibility level of proxy classes (public or internal), whether to generate asynchronous client calls in addition to synchronous calls, and which type to use when creating collection properties (for example, generating them as an array or as a generic **List<T>**).

For more information about the **Add Service Reference** dialog box, see:

Add Service Reference dialog box

http://go.microsoft.com/fwlink/?LinkID=313733

Each of the generated proxies is named according to the name of the contact, without the "I" prefix, and will be appended with the suffix "Client". For example, if the name of the service contract is **IHotelBookingService**, then the generated proxy class will be named **HotelBookingServiceClient**.

To use the generated proxy, create an instance of it in your code, and use it to call the service.

The following example demonstrates how to use a generated proxy to call a service.

Using a Generated Proxy to Call a Service

```
var proxy = new HotelBooking.HotelBookingServiceClient();

HotelBooking.Reservation reservation = new HotelBooking.Reservation()
{
   FirstName = "James",
   LastName = "Alvord",
   HotelName = "Contoso Motel",
   NumberOfNights = 3
};

try
{
   proxy.BookHotel(reservation);
}
catch (FaultException<ReservationFault> reservationEx)
{
   Console.WriteLine(reservationEx.Message + Environment.NewLine +
reservationEx.Detail.ErrorCode);
}
catch (FaultException faultEx)
{
   Console.WriteLine(faultEx.Message);
}
catch (Exception ex)
{
   Console.WriteLine("An unknown error has occurred: " + ex.Message);
}
```

The above example creates a proxy object from the generated **HotelBookingServiceClient** class, a reservation object from the generated **Reservation** class, and then calls the **BookHotel** method of the proxy. Calling this method will invoke the **BookHotel** operation in the service.

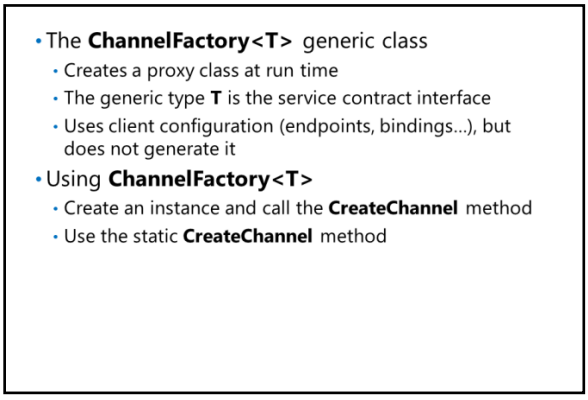
 **Note:** One of the risks in working with proxies is that developers can forget that they are using an object that crosses the boundary of the application, and possibly even the device. Be aware that, though it might seem that you are working with a local object, there is an underlying mechanism that adds overhead on each method call. Communication latency, serialization, and many other factors can cause a latency penalty.

The above code example also has a **try/catch** block to handle possible exceptions and service faults. The code handles the fault exception that returns a **ReservationFault** object, a more general fault exception for any other unknown service faults, and general exception handling in case of a more catastrophic exception, such as a communication exception. You can see the server-side implementation of the **ReservationFault** class in Lesson 2, "Creating and Implementing a Contract," in the "Handling Exceptions" topic.

If the service contract changes over time and you want your proxies to reflect the new state of the contract and/or data contract, you do not have to delete the service reference and start all over again. The **Add Service Reference** dialog box also supports an update option. In **Solution Explorer**, expand the **Service References** folder under your project, select the service reference that you want to update, right-click it, and then click **Update Service Reference**.

If you want to create a cleaner configuration file, you can use the **Svcutil.exe** tool, which is similar to the Add Service Reference dialog box. However, you can use the **Svcutil.exe** tool from a command prompt, and it offers more options than the **Add Service Reference** dialog box, including metadata export and service validation.

Creating a Service Proxy with ChannelFactory<T>



Another way to create a service proxy is to use the **System.ServiceModel.ChannelFactory<T>** generic class. There are several differences between creating a proxy through the **Add Service Reference** dialog box and through **ChannelFactory<T>**. Here are some of the major differences:

1. Adding a service reference creates the proxy at design time, while **ChannelFactory<T>** creates a proxy class at run time (by using the .NET Remoting transparent proxy).
2. Adding a service reference creates a proxy class together with all the data contracts and required configuration, while **ChannelFactory<T>** only generates a proxy. You, as the client developer, need to provide the service contract interface and the data contract classes.
3. Adding a service reference requires the service metadata to generate a proxy, while **ChannelFactory<T>** requires the service contract interface as a generic type parameter.

To use the **ChannelFactory<T>** generic class, your client needs to have access to the service contract interface and the data contracts. You can achieve this through either a shared assembly or a shared, linked C# file that contains the service interface.

There are two ways to use the **ChannelFactory<T>** generic class:

- Use the static **CreateChannel** method without creating an instance of **ChannelFactory<T>**.
- Create an instance of **ChannelFactory<T>** by using the service contract interface as a generic type parameter, and then use the **CreateChannel** method of the instance.

The following code example demonstrates the different ways to use **ChannelFactory<T>**.

Creating a Service Proxy with ChannelFactory<T>

```
// Create a service proxy with the static CreateChannel method
IHotelBookingService proxyA = ChannelFactory<IHotelBookingService>.CreateChannel(
   new BasicHttpBinding(),
   new EndpointAddress(@"http://localhost:8733/booking"));

// Create a service proxy with an instance of ChannelFactory<T>
ChannelFactory<IHotelBookingService> factory = new ChannelFactory<IHotelBookingService>(
   "HotelBooking_http");

IHotelBookingService proxyB = factory.CreateChannel();

(proxyA as ICommunicationObject).Open();
(proxyB as ICommunicationObject).Open();

var reservation = new Reservation()
{
   FirstName = "James",
   LastName = "Alvord",
   HotelName = "Contoso Motel",
   NumberOfNights = 3
};

proxyA.BookHotel(reservation);
proxyB.BookHotel(reservation);
```

```
(proxyA as ICommunicationObject).Close();
(proxyB as ICommunicationObject).Close();
```

Note: When you use the **Add Service Reference** dialog box, the generated proxy class derives from the **ClientBase<T>** abstract class. Under the hood, this abstract class uses **ChannelFactory<T>** to generate the inner proxy that is called by the generated proxy class.

The first **CreateChannel** method call receives two parameters: a binding object and the service endpoint address. The third part of the service endpoint's ABC, the contract, is passed as the generic type of the **ChannelFactory<T>** class. If you choose to use the second technique, by creating an instance on **ChannelFactory<T>**, you can either call the constructor with a binding instance and a service address, as you do with the static method, or pass a string representing the name of an already configured endpoint, as shown in the example.

After creating the proxy objects, their channels are opened by calling the **ICommunicationObject.Open** method. When you call the **CreateChannel** method, it dynamically creates a proxy object that implements both the **IHotelBookingService** and the **ICommunicationObject** interfaces. You can use the **ICommunicationObject** interface to open and close the communication channel manually, as well as registering to channel-related events, such as **Opening**, **Closing**, and **Faulted**. If you do not open the channel manually by calling the **Open** method, it will be opened when you send the first request to the service.

Note: Opening the channel can take several seconds if there is a lengthy negotiation process between the client and the service, for example when you call a secured service that requires the client to authenticate. In such cases, opening the communication channel ahead of time, when the application starts or the form is loaded, can save some time when calling the service for the first time.

For the above example to work, you need to have a **<client>** endpoint configuration element in the application configuration file of your client, and you need to set the **name** attribute of the element to the name you used in the constructor.

The following configuration demonstrates how to create client endpoints in configuration.

Creating Client Endpoints in Configuration

```
<system.serviceModel>
    <client>
        <endpoint address="http://localhost:8733/booking"
            binding="basicHttpBinding"
            contract="HotelBooking.IHotelBookingService"
            name="HotelBooking_http" />
        <endpoint address="net.tcp://localhost:8734/booking"
            binding="netTcpBinding"
            contract="HotelBooking.IHotelBookingService"
            name="HotelBooking_tcp">
        </endpoint>
    </client>
</system.serviceModel>
```

The advantage of using channel factories is in how they handle breaking changes in contracts. Breaking changes are changes that force you to fix your proxy code, such as when another parameter is added to an operation, or when the name of a data contract is changed. If you use channel factories, a breaking change will stop your code from compiling, but if you use a generated proxy, you might end up having exceptions at run time.

Demonstration: Adding a Service Reference

This demonstration shows how to add a service reference in Visual Studio 2012, and how to use the generated proxy to call the service.

Demonstration Steps

1. Open **D:\Allfiles\Mod05\DemoFiles\AddingServiceReference\begin\AddServiceReference.sln**. In the **ServiceHost** project, open the **App.config** file, and then view the service configuration, including the endpoint configuration and service behaviors.
2. Run the **ServiceHost project** without debugging.
3. In the **ServiceClient** project, use the **Add Service Reference** dialog box to add a service reference to the **HotelBookingService** service.
   - Use the address **http://localhost:8733/HotelBooking to locate the service metadata**
   - Use the **HotelBooking** namespace for the generated proxy.
4. Open the **Program.cs** file in the **ServiceClient** project, and then in the **Main** method, uncomment the code. Observe the use of the generated proxy and data contracts.
5. Run the **ServiceClient** project and verify that the client is able to connect to the service.

   Question: What are the advantages and disadvantages of using the **Add Service Reference** dialog box of Visual Studio 2012?

Demonstration: Using Channel Factories

This demonstration shows how to use the **ChannelFactory<T>** generic class to create a proxy and call a service.

Demonstration Steps

1. Open **D:\Allfiles\Mod05\DemoFiles\UsingChannelFactory\begin\UsingChannelFactory.sln**, and then view the service configuration, including the endpoint configuration and service behaviors.
2. In the **ServiceClient** project, add a reference to the **Common** project and the **System.ServiceModel** assembly.
3. In the **ServiceClient** project, open the **Program.cs** file, and then add the following code before the commented code.

```
ChannelFactory<IHotelBookingService> serviceFactory =
    new ChannelFactory<IHotelBookingService>
        (new BasicHttpBinding(),
        "http://localhost:8733/HotelBooking/HotelBookingHttp");
IHotelBookingService proxy = serviceFactory.CreateChannel();
```

4. In the **Main** method, uncomment the code, run the service and the client, and then test whether the client can connect to the service.

- The console application should displays the message *Booking response: Approved, booking reference: AR3254*

Question: What are the requirements for using the **ChannelFactory<T>** generic class?

Lab: Creating and Consuming the WCF Booking Service

Scenario

Until now, most of Blue Yonder Airlines' booking systems were internal systems, connected to the booking database of the company. Since there are plans to move the newly created ASP.NET Web API service to a location outside internal network of the company (probably to the Windows Azure cloud), there is a need to create a new service that can receive booking requests from both internal and external applications. Since the requirements from the new service include features such as support for TCP and MSMQ communication, it was decided that the new service will be a WCF service. In this lab, you will create a WCF service for the booking subsystem. In addition, you will update the ASP.NET Web API booking service to use the new WCF booking service.

Objectives

After completing this lab, you will be able to:

- Create service and data contract, and implement the service contract.
- Configure a WCF service for TCP and host it in a console application.
- Consume a WCF service from a client application.

Lab Setup

Estimated Time: 40 minutes

Virtual Machine: **20487B-SEA-DEV-A**, **20487B-SEA-DEV-C**

User name: **Administrator**, **Admin**

Password: **Pa$$w0rd**, **Pa$$w0rd**

For this lab, you will use the available virtual machine environment. Before you begin this lab, you must complete the following steps:

1. On the host computer, click **Start**, point to **Administrative Tools**, and then click **Hyper-V Manager**.
2. In Hyper-V Manager, click **MSL-TMG1**, and in the Action pane, click **Start**.
3. If you executed a later lab before this one, follow these instructions:

- In Hyper-V Manager, click the **20487B-SEA-DEV-A** virtual machine.
- In the **Snapshots** pane, right-click the **StartingImage** snapshot and then click **Apply**.
- In the **Apply Snapshot** dialog box, click **Apply**.

4. In Hyper-V Manager, click **20487B-SEA-DEV-A**, and in the Action pane, click **Start**.
5. In the Action pane, click **Connect**. Wait until the virtual machine starts.
6. Sign in using the following credentials:

- User name: **Administrator**
- Password: **Pa$$w0rd**

7. Return to Hyper-V Manager, click **20487B-SEA-DEV-C**, and in the Action pane, click **Start**.
8. In the Action pane, click **Connect**. Wait until the virtual machine starts.
9. Sign in using the following credentials:

- User name: **Admin**
- Password: **Pa$$w0rd**

10. Verify that you received credentials to log in to the Azure portal from you training provider, these credentials and the Azure account will be used throughout the labs of this course.

Exercise 1: Creating the WCF Booking Service

Scenario

The first step in creating a WCF service is to define the service contract and data contracts. Only afterwards can you begin implementing the service contract. In this exercise, you will define a service contract interface for the booking service along with the required data contracts, and then you will implement the service contract.

The main tasks for this exercise are as follows:

1. Create a data contract for the booking request

2. Create a service contract for the booking service

3. Implement the service contract

▶ Task 1: Create a data contract for the booking request

1. In the **20487B-SEA-DEV-A** virtual machine, open the **BlueYonder.Server.sln** solution file from the **D:\AllFiles\Mod05\LabFiles\begin\BlueYonder.Server folder**.

2. In the **BlueYonder.BookingService.Contracts** project, add the **TripDto** data contract class.

- Set the access modifier of the class to **public** and decorate it with the **[DataContract]** attribute.
- Add the following properties to the class.

Name	Type
FlightScheduleId	int
Status	**FlightStatus**
Class	**SeatClass**

- Decorate each of the new properties with the **[DataMember]** attribute.

 **Note:** In order to use data contract object, the **System.ServiceModel** and **System.Runtime.Serialization assemblies need to be added to the project references. The begin solution already contains those assemblies.**

3. Add the **ReservationDto** data contract class.

- Set the access modifier of the class to **public** and decorate it with the **[DataContract]** attribute.
- Add the following properties to the class.

Name	Type
TravelerId	int
ReservationDate	**DateTime**
DepartureFlight	**TripDto**

Name	Type
ReturnFlight	**TripDto**

- Decorate each of the new properties with the **[DataMember]** attribute.

Note: Review the **ReservationCreationFault** class in the **Faults** folder. The class will be used later, as a data contract object to mark a fault reservation.

▶ Task 2: Create a service contract for the booking service

1. In the **BlueYonder.BookingService.Contracts** project, add a new interface named **IBookingService.** **S**et the access modifier of the interface to **public**.
2. Decorate the interface with the **[ServiceContract]** attribute and set the **Namespace** parameter of the attribute to **http://blueyonder.server.interfaces/**.
3. Add the **CreateReservation** method to the interface, and define it as an operation contract.

- The method should receive a parameter named **request** of type **ReservationDto**, and return a string.
- Decorate the method with the **[OperationContract]** attribute.
- Decorate the method with the **[FaultContract]** attribute, and set the attribute's parameter to the type of the **ReservationCreationFault** class.

▶ Task 3: Implement the service contract

1. In the **BlueYonder.BookingService.Implementation** project, implement the IBookingService in the **BookingService** class.

- Change the class declaration, so it will implement the **IBookingService** interface.
- Decorate the class with the **[ServiceBehavior]** attribute and set the attribute's **InstanceContextMode** parameter to **InstanceContextMode.PerCall**.

2. In the service implementation class, implement the service contract.

- Create the **CreateReservation** method, but do not fill it with code yet.

Note: At this point, the class will not compile because no value is returned from the method. Ignore this for now, as you will soon write the missing code.

3. Start implementing the **CreateReservation** method by verifying whether the request contains information for the departure flight. If the departure flight information is missing, throw a fault exception.

- If the request.DepartureFlight property is null, throw a FaultException of type ReservationCreationFault.
- In the **FaultException** constructor, create a new instance of **ReservationCreationFault** with the following property values.

Property	Value
Description	Reservation must include a departure flight

Property	Value
ReservationDate	request.ReservationDate

- In the **FaultException** constructor, set the second constructor parameter to the reason string **"Invalid flight info"**.

4. Continue implementing the **CreateReservation** method by creating a **Reservation** object.

- Initialize the **Reservation** object according to the following table.

Property	Value	
TravelerId	request.TravelerId	
ReservationDate	request.ReservationDate	
DepartureFlight	A new Trip object with the following values.	
	Property	**Value**
	Class	request.DepartureFlight.Class
	Status	request.DepartureFlight.Status
	FlightScheduleID	request.DepartureFlight.FlightScheduleID

5. Continue implementing the **CreateReservation** method by checking whether the return flight is not **null**. If the request.**ReturnFlight** is not null, add a trip to the reservation object you created.

- Initialize the **reservation.ReturnFlight** property with a new **Trip** object, and set its properties according to the following table.

Property	Value
Class	request.ReturnFlight.Class
Status	request.ReturnFlight. Status
FlightScheduleID	request.ReturnFlight.FlightScheduleID

6. Continue implementing the **CreateReservation** method by adding the new reservation object to the database:

- Create a new **ReservationRepository** object and initialize it with the **ConnectionName** field.
- Use the **ReservationUtils.GenerateConfirmationCode** static method to generate a confirmation code and assign in to the **reservation.ConfirmationCode** property before saving the new reservation.
- Call the **Add** and then the **Save** methods of the repository to save the newly created **reservation**.
- Return the generated confirmation code to the client.

Note: To make sure the context and the database connection are disposed properly at the end of the service operation, you should place the repository-related code in a **using** block.

7. Insert a breakpoint at the beginning of the **CreateReservation** method.

Results: You will be able to test your results only at the end of the second exercise.

Exercise 2: Configuring and Hosting the WCF Service

Scenario

The second step in creating the WCF service is to create a project for hosting the service. In this exercise, you will create a service host, configure it with a TCP endpoint and use it to make the service available for clients.

The main tasks for this exercise are as follows:

1. Configure the console project to host the WCF service with TCP endpoint

2. Create the service hosting code

▶ Task 1: Configure the console project to host the WCF service with TCP endpoint

1. In the **BlueYonder.BookingService.Host** project, add a reference to the **System.ServiceModel** assembly.

Note: The begin solution already contains all the project references that are needed for the project. This includes the **BlueYonder.BookingService.Contracts**, **BlueYonder.BookingService.Implementation BlueYonder.DataAccess**, and **BlueYonder.Entities** projects, as well as the Entity Framework 5.0 package assembly.

2. Review the **FlightScheduleDatabaseInitializer.cs** file in the **BlueYonder.BookingService.Host** project. Observer how the **Seed** method initializes the database with predefined locations and flights.
3. In the **BlueYonder.BookingService.Host** project, open the **App.config, and** add a service configuration section for the Booking WCF service.

- Add the **<system.serviceModel>** element to the configuration, and in it add the **<services>** element.
- In the <services> element, add a <service> element, and then set its name attribute to BlueYonder.BookingService.Implementation.BookingService.

4. Add an endpoint configuration to the service configuration you added in the previous step.

- In the **<service>** element, add an **<endpoint>** element with the following attributes.

Attribute	Value
name	BookingTcp
address	net.tcp://localhost:900/BlueYonder/Booking/
binding	netTcpBinding

Attribute	Value
contract	BlueYonder.BookingService.Contracts.IBookingService

5. In the **App.config**, add a connection string to the local SQL Express.

```
<connectionStrings>
  <add name="BlueYonderServer" connectionString="Data
Source=.\SQLEXPRESS;Database=BlueYonder.Server.Lab5;Integrated Security=SSPI"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Note: You can copy the connection string from the ASP.NET Web API services configuration file in **D:\Allfiles\Mod05\Labfiles\begin\BlueYonder.Server\BlueYonder.Companion.Host\Web.config**. Make sure you change the database parameter to **BlueYonder.Server.Lab5**.

▶ Task 2: Create the service hosting code

1. In the **BlueYonder.BookingService.Host** project, open the **Program.cs** file, and then add two static event handler methods to handle the **ServiceOpening** and **ServiceOpened** events of the service host.

- Each of the methods receives two parameters: **sender**, of type **object**, and **args**, of type **EventArgs**.
- In each method, write a short status message to the console window.

2. In the Main method, add **the following code to initialize the database.**

```
var dbInitializer = new FlightScheduleDatabaseInitializer();
dbInitializer.InitializeDatabase(new
TravelCompanionContext(Implementation.BookingService.ConnectionName));
```

3. Remaining in the Main method, add code to host the **BookingService** service class.

- Create a new instance of the **ServiceHost** class for the **BookingService** service class.
- Register to the service host's Opening and Opened events with the **ServiceOpening and ServiceOpened methods, respectively.**
- **Open the service host, wait for user input, and the close the service host.**

Note: Refer to Lesson 3, "Configuring and Hosting WCF Services", Topic 1, "Hosting WCF Services", for an example on how to open the service host, wait for user input, and the closing the service host.

4. Run the **BlueYonder.BookingService.Host** project in debug mode and verify it opens without throwing exceptions. Keep the console window open, as you will need to use it later in the lab.

Results: You will be able to start the console application and open the service host.

Exercise 3: Consuming the WCF Service from the ASP.NET Web API Booking Service

Scenario

After you create the WCF service, you can consume it from the ASP.NET Web API web application. In this exercise, you will configure the client endpoint in the ASP.NET Web API web application, and use the **ChannelFactory<T>** generic class to create a client proxy. You will then use the new proxy to call the WCF service, create the reservation on the backend system, and get the reservation confirmation code in return.

The main tasks for this exercise are as follows:

1. Add a reference to the service contract project in the ASP.NET Web API projects
2. Add client configuration to Web.Config
3. Call the Booking service by using ChannelFactory<T>
4. Debug the WCF service with the client app

▶ Task 1: Add a reference to the service contract project in the ASP.NET Web API projects

1. Open the **D:\AllFiles\Mod05\LabFiles\begin\BlueYonder.Server\BlueYonder.Companion.sln** solution file in a new Visual Studio 2012 instance, and add the **BlueYonder.BookingService.Contracts** project from the **D:\Allfiles\Mod05\Labfiles\begin\BlueYonder.Server\BlueYonder.BookingService.Contracts folder** to the solution.
2. In the BlueYonder.Companion.Controllers project, add a reference to the BlueYonder.BookingService.Contracts project.
3. In the **BlueYonder.Companion.Host** project, add a reference to the **BlueYonder.BookingService.Contracts** project.

▶ Task 2: Add client configuration to Web.Config

1. In the **BlueYonder.Companion.Host** project, open the **Web.config file**, and add a client endpoint configuration to call the Booking WCF service.

- Add the **<system.serviceModel>** element to the configuration, and in it add the **<client>** element.
- In the **<client>** element add an **<endpoint>** element with the following attributes.

Configuration Parameter	Value
address	net.tcp://localhost:900/BlueYonder/Booking
binding	netTcpBinding
contract	BlueYonder.BookingService.Contracts.IBookingService
name	BookingTcp

 **Note:** Make sure you set the **name** attribute of the endpoint to **BookingTcp**, as you will use this endpoint name in code to locate the endpoint configuration.

▶ Task 3: Call the Booking service by using ChannelFactory<T>

1. In the **BlueYonder.Companion.Controllers** project, open the **ReservationsController.cs** file. In the **ReservationsController** class, create a channel factory object for the **IBookingService** service contract, and store it in a field named **factory**.

- In the channel factory constructor, use the **BookingTcp** endpoint configuration name.

2. In the **CreateReservationOnBackendSystem** method, uncomment the code that creates the **TripDto and ReservationDto** objects.
3. In the **CreateReservationOnBackendSystem** method, create a new channel by using the channel factory you have created.

- Before the **try** block, create the channel by calling the **factory.CreateChannel** method.
- Store the newly created channel in a variable of type **IBookingService**.

4. In the **try** block, call the **CreateReservation** operation of the Booking service.

- The **CreateReservation** operation returns the confirmation code string. Store the returned string in a local variable.
- After calling the service, close the channel by casting the channel object to the **ICommunicationObject** interface, and then call its **Close** method.
- Return the confirmation code string and remove the **return** statement that is currently at the end of the method.

 **Note:** Refer to Lesson 4, "Consuming WCF Services", Topic 2, "Creating a Service Proxy with ChannelFactory<T>", for an example on how to close a channel.

5. Change the first catch block from

```
catch (HttpException fault)
```

to

```
catch(FaultException<ReservationCreationFault> fault)
```

- Inside the **catch** block, throw an **HttpResponseException** with an **HttpResponseMessage** object.
- Create the **HttpResponseMessage** by using the **Request.CreateResponse** method. Set the status code to **BadRequest** (HTTP 400), and the content of the message to the description of the fault.
- Abort the connection in case of Exception, by calling the **Abort** method on the proxy object.

6. In the second **catch** block, abort the connection before calling the **throw** statement.

- Abort the connection by casting the channel object to the **ICommunicationObject** interface, and then calling its **Abort** method.

7. In the **Post** method, before adding the new reservation to the repository, call the Booking WCF service and set the reservation's confirmation code.

- Call the CreateReservationOnBackendSystem method with the newReservation object.
- Store the returned confirmation code of the reservation in the **newReservation.ConfirmationCode** property.

Note: The reservation should be saved to the database with the confirmation code you got from the WCF service, so make sure you set the confirmation code property before adding the reservation to the repository.

8. Build the solution.

Note: The **BlueYonder.Companion.Host** project is already configured for Web hosting with IIS. Building the solution will make the Web application ready for use.

▶ Task 4: Debug the WCF service with the client app

1. Place a breakpoint on the line of code that calls the **CreateReservationOnBackendSystem** method, and start debugging the Web application.
2. In the **20487B-SEA-DEV-C** virtual machine, open the **BlueYonder.Companion.Client** solution from the **D:\AllFiles\Mod05\LabFiles\begin** folder, and run it without debugging.
3. Search for **New** and purchase a new trip from *Seattle* to *New York*.
4. Go back to the **20487B-SEA-DEV-A** virtual machine, and debug the **BlueYonder.Companion** and **BlueYonder.Server** solutions. Verify the ASP.NET Web API service is able to call the WCF service. Continue running both solutions and verify the client is showing the new reservation.

Results: After you complete this exercise, you will be able to run the Blue Yonder Companion client application and purchase a trip.

Question: Why should you use the **NetTcpBinding** in your endpoints instead of **BasicHttpBinding** or **WSHttpBinding**?

Question: Why did you create the service contract and service implementation in separate C# projects?

Module Review and Takeaways

At the beginning of this module you learned about SOAP-based services and their benefits compared to other technologies. You also learned about the differences between ASP.NET Web API and WCF and the factors that should be considered when choosing between the two.

You then learned the basics of WCF services: defining a service contract interface and data contracts, implementing a service, and the proper way to handle exceptions. Next, you learned about hosting WCF services; the responsibilities of the host, the options that are available in hosting, and the details of how to self-host. You also learned how to configure service endpoints in code and in the configuration file of your host. You then learned how to expose your service metadata, including information on the various contracts and endpoints, by using service behaviors and MEX endpoints.

Finally, you learned how to create a proxy that can call a service by using the **Add Service Reference** dialog box of Visual Studio 2012, or by using the generic **ChannelFactory<T>** class.

This module covered the fundamentals of creating WCF services. If you wish to go in depth into other features of WCF, such as messaging patterns, WCF extensibility, and how to secure WCF services, refer to Appendix 1, "Designing and Extending WCF Services", and Appendix 2, "Implementing Security in WCF Services" in Course 20487.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
When running the service host from Visual Studio, an exception of type **AddressAccessDeniedException** is thrown, saying "HTTP could not register URL…"	
When running the service host from Visual Studio, an exception of type **AddressAlreadyInUseException** is thrown saying "HTTP could not register URL…"	

Review Question(s)

Question: When should you favor WCF over of ASP.NET Web API?

Question: When should you use the **Add Service Reference** dialog box, and when should you use the **ChannelFactory<T>** generic class?

Tools

WCF Test Client, Microsoft Service Configuration Editor, Svcutil.exe

Appendix A

Designing and Extending WCF Services

Contents:

Module Overview

Windows Communication Foundation (WCF) is the framework for developing Simple Object Access Protocol (SOAP)-based services. In the You learned how to create a WCF service by creating a service contract, implementing it, and hosting it.

When you create services in WCF, there are additional design principles and techniques that you can apply to your code to enhance the reliability and performance of your service.. For example, you can create asynchronous service operations to better utilize how WCF uses the managed Thread Pool. You can also create services that can be a part of a distributed transaction that spans over several services and databases. Or you can create your own custom error handlers that log any unhandled exception thrown in your service code.

This module describes how to design service contracts with various patterns, such as one-way operations and asynchronous operations, and then explains how to implement those contracts in your service implementation. You will also learn about distributed transactions, what they are and how you can create a WCF service that supports distributed transactions. The last lesson in this module is about the WCF pipeline, where you will learn how you can extend the message handling pipeline by creating your own custom runtime components and extensible objects, and then apply them to services and operations.

Objectives

After completing this module, you will be able to:

- Design and create services and clients to use different kinds of message patterns.
- Configure a service to support distributed transactions.
- Extend the WCF pipeline with runtime components, custom behaviors, and extensible objects.

Lesson 1
Applying Design Principles to Service Contracts

Message patterns describe how clients and services exchange data between each other. The most common message pattern in WCF is the Request-Response pattern, where the client sends a request and waits until a response returns from the service. However, WCF supports several other kinds of message patterns that you can use in your service contract. For example, you can define a service contract with one-way operations, which clients can call without waiting for any response (even if it is a fault response).

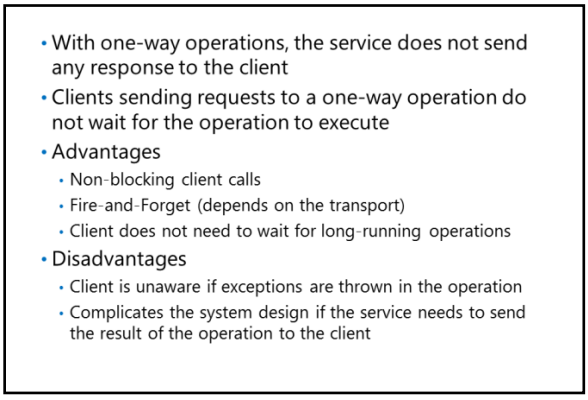
In this lesson, you will learn about the different message patterns that you can choose from when you design your service contract.

Lesson Objectives

After completing this lesson, you will be able to:

- Create service contracts with one-way operations.
- Produce services that use streams for requests and responses.
- Build duplex services.
- Make asynchronous operations in both client and service.

One-Way Operations



Defining an operation as one-way means that the operation is not expected to return a value. Therefore, clients can send a request to the service and continue to execute without waiting for a response. This pattern is also known as *fire-and-forget*. Because one-way operations require that no value is returned, operations that are marked as one-way must have **void** as their return type. This nonblocking behavior of the client differs from the behavior when the client calls an operation that uses the request-response pattern. This is because in request-response patterns, the client blocks until the service responds, even if the operation returns **void**.

You can use one-way operations for any of the following scenarios:

- When the client does not require the operation to return a value, either successful or failed.
- When the operation is long-running and you do not want to block your client's execution.
- When the service has a different way of notifying the client of the operation's result, such as responding with an email message.

For example, a client tracking system can send one-way messages to a service with the Global Positioning System (GPS) location of the client every second. The service operation only logs the coordinates and therefore does not send any response back to the client. The business logic of the application handles the possibility that some calls fail, because if a call fails, a new location is sent shortly afterward. In this case, it is also expected that the business logic handles the scenario where the client loses connectivity and does not send updates for long periods of time.

Note: The client and service implementation of one-way operations differs between transports. When using a bidirectional transport such as Hypertext Transfer Protocol (HTTP) or Transmission Control Protocol (TCP), the client sends a request and waits for an acknowledgement from the service. When the service receives the request, it immediately responds with an acknowledgement and only then handles the request. For example, when using HTTP transports with one-way messaging, the service immediately responds with an HTTP 202 (Accepted) response, and only then handles the request. If the service is unavailable to receive the request, the client fails and throws an exception. On the other hand, when using a unidirectional transport such as User Datagram Protocol (UDP), the client sends a request and does not wait for any service acknowledgement. Therefore, the client cannot verify whether the service is online and available.

To mark an operation as one-way, add the **IsOneWay = true** initialization to the **[OperationContract]** attribute decorating the operation in the service contract.

The following code example demonstrates how to create a one-way operation in a service contract.

Service Contract With One-Way Operations

```
 [ServiceContract]
public interface ILogging
{
   [OperationContract(IsOneWay = true)]
   void Log(string header, string message);

   [OperationContract]
   List<LogLine> GetLog();
}
```

Note: If the **IsOneWay** parameter is set to **false** or omitted from the attribute, the default messaging pattern becomes request-response. If the operation does not return **void**, an exception is thrown in runtime when opening the service host. You can have both one-way and request-response operations in the same service contract.

When you use one-way messages, the client is unaware of what has occurred during the execution of the service operation. Therefore, the client does not know whether the service operation completed execution or failed because of an exception. If the client requires information on the operation's result, the service has to send the information to the client. The service can send this information by using either an out-of-band communication, such as an email or a text message (SMS), or by sending a message to the client application For example, by hosting a WCF service in the client application and sending it a message that contains the result.

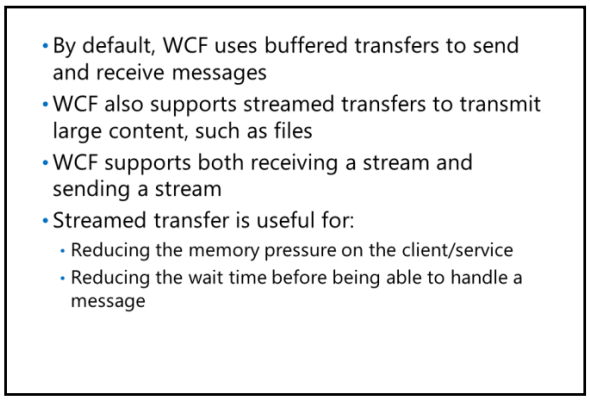
For additional information on the one-way messaging pattern, see:

One-Way Services

http://go.microsoft.com/fwlink/?LinkID=298778&clcid=0x409

Streamed Requests and Responses

The default behavior of WCF for sending or receiving a message is to use buffered transfer. In buffered transfers, all the data the client or service has to send is loaded into the memory (into a buffer), and is stored there until the whole message is sent to the other end. This results in several memory allocations that have to be freed. If you send a very large message, you might even get an **OutOfMemoryException** exception if your object is too big to fit in memory. The same buffering behavior applies when receiving a message.



Note: To prevent services from failing due to large memory allocation attempts, the default configuration of WCF limits the maximum size of a received message to 64 kilobytes (KB). An additional benefit of this restriction is that it helps prevent denial of service attacks. You can change this limit by setting the **maxReceivedMessageSize** and **maxBufferSize** attributes in the binding configuration. This configuration also exists on the client-side, and affects the size of responses that clients can receive from services.

In addition, buffered transfers require the receiving side to wait until the whole message arrives before it can be read and used. When you send large messages on slow networks, it can cause your service and client to stop responding for a long time, and even time out if the message takes too long to be received.

If you have to send large amounts of data from the client to the service or from the service to the client such as when uploading or downloading a file, you might consider sending the data using a streamed transfer instead of using the standard buffered transfer.

In streamed transfers, both the receiving and sending sides work with streams. When you use streams, you can send small amounts of data, which are known as chunks, from one end to the other without having to allocate memory for the whole message in advance. This is possible because the allocated memory is only as big as a single chunk. In addition, the receiving end can handle each chunk, while it is being received, without having to wait for the whole message to arrive. WCF can use streamed transfers in most of its bindings, including HTTP, TCP, and Named pipes.

Streaming is supported on both ends of a WCF service call: an operation can receive a stream, return a stream, or do both. The first part in creating an operation that supports streaming is to define the operation contract to use stream, either as an input, output, or both.

The following code example demonstrates how to define operations that work with stream content.

Contract with Operations that Receive and Return Streams

```
 [ServiceContract]
public interface IVideoService
{
   [OperationContract]
   Stream DownloadVideo(string videoName, string format);

   [OperationContract]
   Guid UploadVideo(Stream video);

   [OperationContract]
   Stream DecodeVideo(Stream videoInput);
```

```
}
```

Note: If your operation receives a stream, it cannot receive any other parameter other than that stream. Instead of using the **Stream** class for the input/output, you can also use the **Message** class or any other type that implements the **IXmlSerializable** interface. If you plan on passing very large data structures, consider working with the **Message** class or implementing the **IXmlSerializable** interface, because these types provide streamed access to Extensible Markup Language (XML) content by using the **XmlReader** abstract classes for reading XML and **XmlWriter** abstract classes for writing XML.

For a demonstration on how to use the **Message** class for streamed transfers, see:

Using the Message Class

http://go.microsoft.com/fwlink/?LinkID=298779&clcid=0x409

Setting an operation to accept or return a **Stream** class is not enough. You also have to change the binding configuration of your endpoint to use streamed transfers. Each part of the communication can be streamed: the request (**StreamedRequest**), the response (**StreamedResponse**), or both (**Streamed**).

The following code example demonstrates setting the binding configuration to use streaming on both request and response.

Configuring a Binding for Streamed Transport

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Services.VideoStreamingService">
        <endpoint
          address="streamService"
          binding="basicHttpBinding"
          bindingConfiguration="HttpStreaming"
          contract="Contracts.IVideoService" />
      </service>
    </services>

    <bindings>
      <basicHttpBinding>
        <binding
            name="HttpStreaming"
            maxReceivedMessageSize="2147483647"
            transferMode="Streamed"/>
      </basicHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

In this example, the **maxReceivedMessageSize** is changed from the default size of **65KB** to **int32.MaxInt**, to enable the service to receive large files.

When you change the **transferMode** of the binding to any of the streamed options, it affects all the operations in the contract that you defined for the endpoint. This includes operations that do not use streams for either input or output. In this case, the streamed content is buffered before serializing/deserializing it. If your contract has a mix of operations that are more suitable for buffering and operations that you want to stream, consider splitting the contract to two contracts. You can then expose each contract through a different endpoint with a different binding configuration: one that uses buffering and the other that uses streaming.

After you configure your service contract for streamed transfers, you also have to make sure that your client-side configuration is configured for streamed transfers. For TCP and Named pipes, WCF includes policy settings in the service metadata so clients can create matching binding configuration. If you use the **Add Service Reference** dialog box of Visual Studio 2012, your client-side configuration can be configured for streamed transfer. However, for the HTTP transport, WCF does not include policy settings in the service metadata, and after you use the **Add Service Reference** dialog box, you have to manually edit the client configuration file and change the binding configuration to set which part is streamed: the request (**StreamedRequest**), the response (**StreamedResponse**), or both (**Streamed**).

For additional information about how to stream messages in WCF, see:

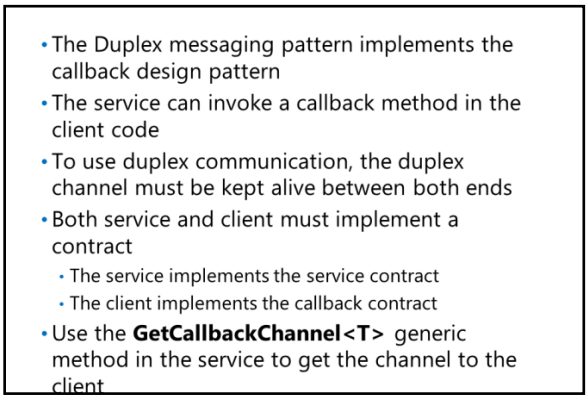
Streaming Message Transfer

http://go.microsoft.com/fwlink/?LinkID=298780&clcid=0x409

Duplex Services

The WCF duplex messaging pattern implements the callback design pattern. In this pattern, a callback function is invoked after a certain function finishes its work. The callback pattern is used widely in the .NET Framework, for example:

- The .NET Asynchronous Programming Model uses callbacks (BeginInvoke and EndInvoke).
- The ASP.NET Cache uses callbacks to indicate when items are removed from the cache.
- Windows Presentation Foundation (WPF) uses callbacks in its dependency property mechanism.



You can view the duplex channel in WCF as the implementation of the callback pattern in the distributed world.

For a service to use a callback that resides on the client-side, the service has to open a connection to the client. To make this possible, the client must perform two actions:

1. Create a service, and host it inside the client application. The remote service can open a channel to the client-side service and send a message to it.
2. Send information about the client's address to the service so that the service knows how to call the client.

When you use a duplex channel, such as a TCP channel, WCF automatically performs the above two actions.

To call a client's callback from the service-side, the client must perform the following steps:

1. When the client calls the service, save the callback channel until it is needed.
2. When the service is ready to send the message, open the saved channel, and then call the callback operation.

To create a duplex service, start by creating the client contract to implement in the client. Your service uses the client contract when calling the client.

The following code example creates the **IStockCallback** client contract.

The Client Contract to be Implemented by the Client

```
 [ServiceContract]
public interface IStockCallback
{
    [OperationContract(IsOneWay=true)]
    void StockUpdated(string stockId, float value);
}
```

After you create the client contract, create the service contract, which is implemented on the service-side.

The following code example creates the **IStock** service contract.

The Service Contract to be Used in the Service

```
 [ServiceContract(CallbackContract = typeof(IStockCallback))]
public interface IStock
{
     [OperationContract(IsOneWay=true)]
     void RegisterForQuote(string stockId);

     [OperationContract(IsOneWay=true)]
     void UpdateStockQuote(string stockId, float newValue);
}
```

You can correlate the service and client contracts in the service contract by adding the **CallbackContract** parameter to the **[ServiceContract]** attribute, and setting it to the kind of the client contract interface. There are two contracts that have to be implemented: the service contract and the client contract (also known as the *callback* contract). Both contracts are designed when the service is built, but only the service contract is implemented by the service itself.

 **Note:** The callback and service contracts use only one-way operations, but you can also declare operations that use the request-response messaging pattern in these contracts.

When you add a reference to the service on the client-side, you have access to both the service and callback contracts. You use the service contract when you create a proxy for calling the service and you implement the callback contract on the client-side.

The following code example demonstrates how the service can call the client by using the callback when it has information for it.

The Implementation of the Service Contract

```
public class StockService : IStock
{
   var _callbacks = new Dictionary<string, IStockCallback>();

   public void RegisterForQuote(string stockId)
   {
      _callbacks[stockId] =
OperationContext.Current.GetCallbackChannel<IStockCallback>();
   }

   public void UpdateStockQuote(string stockId, float newValue)
   {
      try
      {
         _callbacks[stockId].StockUpdated(stockId, newValue);
      }
      catch (CommunicationException e)
      {
         // Log exception
```

```
        }
    }
}
```

When the client sends a request to the **RegisterForQuote** operation method, the method retrieves the callback channel from the WCF context by calling the **GetCallbackChannel<T>** generic method, and stores it in a generic dictionary. When the **UpdateStockQuote** method is called by a different client (or even a different service), the callback channel is retrieved from the dictionary, and a message is sent to the selected client.

 **Note:** The preceding code example registers a single callback for every stock ID to keep the code short. In real-world scenarios, you should support multiple registrations per stock, as shown in the demonstration that follows this topic.

To handle the message sent from the service, the client must implement the callback contract.

The following code example demonstrates how to implement the **IStockCallback** callback contract.

Client-Side Implementation of the IStockCallback Callback Contract

```
public class StockCallback : IStockCallback
{
    public void StockUpdated(string stockId, float value)
    {
        MessageBox.Show(string.Format("{0} = {1}", stockId, value), "Stock Updated");
    }
}
```

Implementing the callback contract is not enough. You also have to host this implementation in the client before calling the service for the first time. However, you do not have to manually create a service host and host the implementation, because WCF can automatically handle the hosting for you if you create the client proxy with duplex communication support.

The following code example demonstrates how to create a duplex channel for the **IStock** service contract with the **StockCallback** client callback class.

Creating a Duplex Chanel with the DuplexChannelFactory<T> Generic Class.

```
InstanceContext context = new InstanceContext(new StockCallback());
DuplexChannelFactory<IStock> factory = new DuplexChannelFactory<IStock>(context);
IStock proxy = factory.CreateChannel();
proxy.RegisterForQuote("MSFT");
```

When you have to call a duplex service, use the **DuplexChannelFactory<T>** generic class instead of the standard **ChannelFactory<T>** generic class. The **DuplexChannelFactory<T>** has a constructor that expects an **InstanceContext** parameter. The **InstanceContext** class manages the local service hosting context. You call the **InstanceContext** constructor with an instance of the class that you want to host, which in this case is the **StockCallback** class.

If you use the **Add Service Reference** dialog box of Visual Studio 2012 to add reference to a duplex service, the generated proxy is generated with a constructor method that expects an **InstanceContext** object.

The only part that remains is to verify that you are using a binding that supports duplex communication. TCP and Named pipes support duplex communication, but HTTP and UDP transports cannot handle duplex communication. If you want to use duplex communication with HTTP, you have to use either the **WsDualHttpBinding** or the **NetHttpBinding**.

Note: The specification of HTTP, which is defined by the World Wide Web Consortium (W3C), prevents the usage of HTTP as a duplex channel. Therefore, you cannot use standard HTTP channels, such as those used with **BasicHttpBinding** and **WsHttpBinding**, for duplex communication. You can use the **WsDualHttpBinding** with duplex communication, because this binding creates two request-response channels: one for calling the service, and another for receiving calls from the service. You can also use the **NetHttpBinding** for duplex communication, because this binding can upgrade its HTTP channel to use **WebSockets**, which is a new protocol that supports bidirectional duplex communication.

For additional information on duplex services, see:

Duplex Services

http://go.microsoft.com/fwlink/?LinkID=298781&clcid=0x409

For additional information on **NetHttBinding** and its **WebSockets** support, see:

Using the NetHttpBinding

http://go.microsoft.com/fwlink/?LinkID=298782&clcid=0x409

Demonstration: Duplex Services

This demonstration shows how to create a service contract and a callback contract, and how to implement the service contract on the service-side and the callback contract on the client-side. This demonstration also shows how to create a proxy by using the **DuplexChannelFactory<T>** generic class.

Demonstration Steps

1. Open the D:\Allfiles\Apx01\DemoFiles\DuplexServices\DuplexServices.sln solution file.
2. View the callback contract and service contract. Observe how the correlation is made between the two contracts.

- The contracts are located under the **Contracts** project, in the **IStockCallback.cs** and **IStock.cs** files.

3. View the service implementation. You set the instancing mode of the service to **Single** because the service host directly calls the **UpdateStockQuote** method, and the reason that you set the concurrency mode to **Multiple** is to support multiple concurrent registration calls.

- The service implementation is located in the **StockService.cs** file, under the **Service** project.

4. Observe the use of the **GetCallbackChannel<T>** generic method in the **RegisterForQuote** method.

- The **GetCallbackChannel<T>** generic method retrieves the callback channel to the client and stores the channel in the dictionary.
- The lock around the dictionary access is there to prevent multiple requests from writing the same key to the dictionary.

5. View the code in the **UpdateStockQuote** method, and observe how the callback channel is casted to the **ICommunicationObject** interface.
6. View the service host code, and observe that it uses TCP-based endpoints because TCP supports duplex communication.

- The service host code is located in the **Program.cs** file under the **Service** project.

7. View the implementation of the callback contract in the **Client** project, in the **StockCallback.cs** file.

8. View the client proxy creation. Note the use of the **DuplexChannelFactory<T>** generic classes instead of the **ChannelFactory<T>** generic class.
9. Run both projects without debugging and wait until both console windows open.

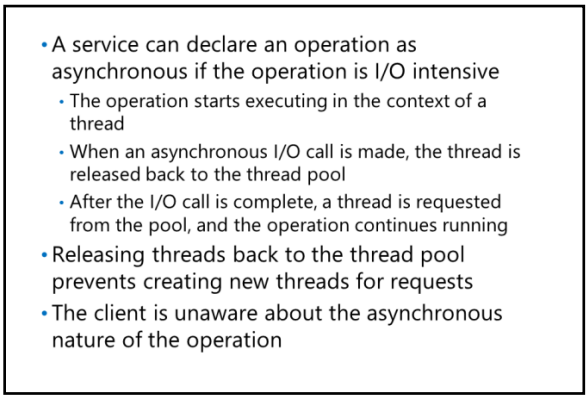
- Wait until the message "*Enter stock name followed by new price. Enter Q to quit*." appears in the **Service Host** console window, and until the message "*Press Enter when the service is ready*" appears in the **Client** console window.
- Move to **Client** console window and press Enter. Wait until the following message appears in the window: "*Waiting for stock updates. Press Enter to stop*".

10. Enter stock changes in the **Service Host** console window, and view how the updates appear in the **Client** console window.

- In the **Service Host** console window, type **MSFT 1200**, and then press Enter. Verify the message "*MSFT = 1200*" appears in the **Client** console window.
- In the **Service Host** console window, type **MSFT 1400**, and then press Enter. Verify the message "*MSFT = 1400*" appears in the **Client** console window.
- Close both console windows when you are finished.

Question: What will happen if the callback contract uses a request-response instead of a one-way operation?

Asynchronous Service Operations



When the client sends requests to a WCF service, each request is executed in the service in a dedicated thread from the managed *Thread Pool*. As long as an operation is executing, the attached thread cannot be used for other incoming calls. If your operation is doing CPU related tasks, such as data manipulation or some computations, the handling thread is used. But if your operation is doing I/O related tasks, such as reading a file from the hard-drive, or waiting for a database call to return, the handling thread is blocked. And if the thread is blocked, it is not doing any work. When you have several concurrent requests in WCF that are waiting for I/O operations to complete, you end up having many blocked threads, and this forces the thread pool to create more threads for incoming requests. Creating more threads when other threads are blocked and not being used is a waste of resources because there is an overhead to creating and managing many threads, and each new thread increases your service's memory consumption.

If you have an operation that does blocking I/O work, you can switch it to an asynchronous service operation. Asynchronous service operations act as follows:

1. The operation starts to execute in the context of a thread from the thread pool.
2. When the operation requires some I/O-bound work, it executes an asynchronous I/O call instead of a synchronous I/O call.
3. As soon as the I/O call starts, the thread is released back to the thread pool during the asynchronous I/O call. As soon as the thread is back in the thread pool, the thread can be assigned to other incoming requests.

4. When the I/O call completes and the control is returned to the operation, a thread is requested from the thread pool, assigned to the operation, and the operation continues its execution.

When you use the asynchronous operation call pattern with I/O-bound operations, you can better use the thread pool and decrease the memory consumption of your service.

You can change your service operation from being synchronous to asynchronous by changing the way the operation method is declared in the service contract and by the way that you implement it in your code.

The following code example demonstrates how to declare a service contract with an asynchronous service operation.

Service Contract with an Asynchronous Service Operation

```
 [ServiceContract]
public interface IFileHandler
{
    [OperationContract]
    Task<int> ProcessFile(string fileName);
}
```

In this example, the return kind of the operation method is the **Task<T>** generic class. When the service host sees that the operation uses a **Task**, it treats this operation as asynchronous.

Note: Although the operation contract's signature uses the **Task<T>** type, the contract exposed by the service, by using the WSDL file, shows this operation as synchronous. This is because the asynchronicity of the operation is internal to the service implementation and is transparent to clients.

The following code example demonstrates how to implement the **IFileHandler** service contract.

Implementing the Asynchronous Service Contract

```
public class FileHandlerService : IFileHandler
{
    public async Task<int> ProcessFile(string fileName)
    {
        using (var file = File.OpenRead(fileName))
        {
            byte[] buffer = new byte[4096];
            int numRead;
            int result = 0;
            while ((numRead = await file.ReadAsync(buffer, 0, buffer.Length)) > 0)
            {
                result += ProcessData(buffer, numRead);
            }

            return result;
        }
    }

    private int ProcessData(byte[] buffer, int numRead)
    {
        // Do some processing and return the partial result
        return partialResult;
    }
}
```

To implement the asynchronous service operation, you use the **async** keyword in the signature of the method, and in the implementation, you signify any asynchronous operation with the **await** keyword.

When the **await** keyword is encountered during execution, the thread is released and returned to the thread pool, until the asynchronous operation is complete. In this sample, the **FileStream.ReadAsync** method starts the asynchronous I/O call and returns an instance of the **Task<T>** class.

Note: Before WCF 4.5, asynchronous service operations were implemented by creating two methods: **BeginXXX** and **EndXXX**, marking the **BeginXXX** method with the **[OperationContract]** attribute, and setting the **AsyncPattern** parameter of the attribute to **true**. This declaration technique, although not obsolete, is more difficult to implement.

For additional information about the different asynchronous programming patterns that you can use in the .NET Framework, see:

Asynchronous Programming Patterns

http://go.microsoft.com/fwlink/?LinkID=298783&clcid=0x409

All the built-in **Stream** types in the .NET Framework, such as **FileStream** and **NetworkStream**, expose asynchronous methods for both reading and writing that return a **Task** object. ADO.NET also provides asynchronous operations with the **DbCommand** class. For example, although the **DbCommand.ExecuteNonQuery** method is a synchronous blocking call, the **DbCommand.ExecuteNonQueryAsync** method provides a nonblocking asynchronous call that you can use to run lengthy SQL statements in the database.

If your service operation has to call another WCF service, and you do not want to block the thread while you wait for the other service to respond, you can use asynchronous client calls. You can generate a client proxy with asynchronous methods with the **Add Service Reference** dialog box of Visual Studio 2012. When you add a service reference in the **Add Service Reference** dialog box, click **Advanced**, and then select the **Allow generation of asynchronous operations** option button.

The following code example demonstrates how to call a WCF service by using an asynchronous WCF client call.

Invoking a Service Asynchronously

```
using (var proxy = new RemoteServices.ProcessingServiceClient())
{
    int result += await proxy.ProcessDataAsync(buffer, numRead);
}
```

In this example, the asynchronous **ProcessDataAsync** method was generated in addition to the synchronous **ProcessData** method. You can use the same **async/await** pattern here as you would use it when working with streams, because asynchronous WCF client calls also return a **Task<T>** instance.

For additional information about the **async** and **await** keywords, see:

Asynchronous Programming with Async and Await

http://go.microsoft.com/fwlink/?LinkID=298784&clcid=0x409

Note: Before Visual Studio 2012, the **Add Service Reference** feature generated client codes with asynchronous methods that used the event-based asynchronous pattern or the **IAsyncResult** asynchronous pattern (also known as the Asynchronous Programming Model, or APM). You can still generate those methods by clicking **Advanced** in the **Add Service Reference** dialog box, and then selecting the **Generate asynchronous operations** option.

For additional information about how to create asynchronous service and client operations, see:

Synchronous and Asynchronous Operations

http://go.microsoft.com/fwlink/?LinkID=298785&clcid=0x409

If you choose to use channel factories instead of generating proxy classes, and the service contract interface that you have only contains synchronous operations, you have to create a new service contract interface that returns **Task<T>,** and replace the **T** generic type with the current return kind of the method. If the method returns a **void**, substitute it with the **Task** return type. The **IFileHandler** service contract shown at the beginning of this topic is the result of replacing a synchronous method signature returning an **int** value with an asynchronous method signature returning **Task<int>**.

Note: When you use client-side asynchronous operations, the operations on the service-side will remain synchronous, unless they were also replaced with asynchronous operations (which requires changing to implementation of the service as well).

Lesson 2
Handling Distributed Transactions

The most used type of transactions is a database transaction. When you use a database, you can start a transaction, execute several insert/update/delete commands, and then choose whether to commit the transaction or rollback the transaction, canceling the changes that you made. Although you might think of transactions as only being relevant to databases, there are other transactional resources that you can manage, such as the file system of your computer and registry settings.

A single transaction only spans one resource (for example, a specific database), a distributed transaction can span multiple resources, such as two databases or a database and a file system of machine.

In distributed transactions, the scope of the transaction is not limited to a single or a chain of resources. Instead, the distributed transaction spans multiple systems, collaborating the work that is performed on the client-side and service-side, to create a transaction that can even span multiple databases in different networks. For example, a client can start a distributed transaction and call two WCF services, each working in an internal transaction. If the second service call fails, the distributed transaction will rollback, causing the transaction in the first service to rollback.

In this lesson, you will learn how to configure your service to support distributed transactions, and how to use transactions in your service implementation. You will also learn how to call a service from the client while in a distributed transaction.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe what a distributed transaction is, and how the two-phase commit protocol works.
- Configure your service contract and binding to support distributed transactions.
- Use distributed transactions in your service implementation and in client-side code.

Explaining Transactions Terms: Transaction Manager, Transaction Coordinator, and Two Phase Commit

To set up a distributed transaction, each computer involved in the transaction manages its own local transaction by using a local transaction manager, and interacts with the other computer by using a Distributed Transaction Coordinator (DTC). The DTC is responsible for handling the completion and abortion of the distributed transaction.

- Distributed transactions coordinate activities in a distributed system
- WCF supports two transaction protocols:
  - OLE Transaction (OleTx) – **Non-Interoperable**
  - WS-AtomicTransaction (WS-AT) – **Interoperable**
- The Distributed Transaction Coordinator (DTC) service must be running and properly configured
- Transactions have performance and coupling costs

Note: The **System.Transactions.TransactionScope** class is responsible for locally managing the transaction and communicating with the DTC. You have already used this class to create local transactions with Entity Framework in Module 2, "Querying and Manipulating Data Using Entity Framework", Lesson 4, "Manipulating Data", in Course 20487. This class can also handle distributed transactions.

When the computer that started the distributed transaction finishes its work, its coordinator starts the distributed transaction commit process by using a two-phase commit protocol.

The following diagram describes the two-phase commit protocol.

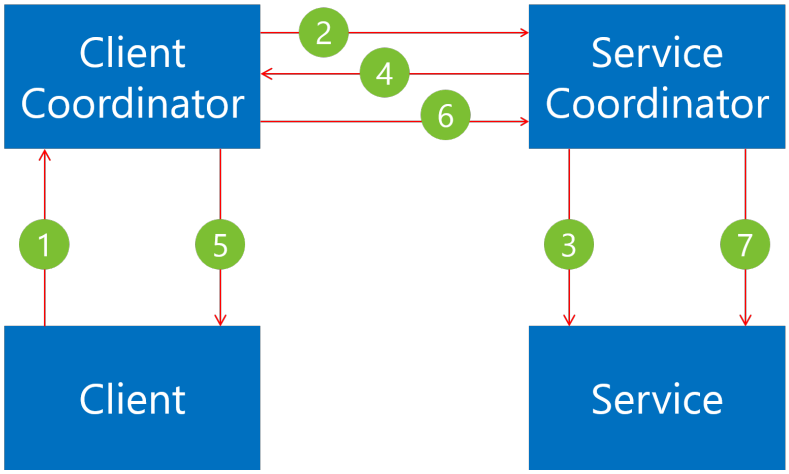


FIGURE 13.1: THE TWO-PHASE COMMIT PROTOCOL

The two-phase commit protocol stages are:

1. When the client finishes its work, it asks its coordinator to commit the distributed transaction.
2. The coordinator of the client sends a commit preparation request to the coordinator of the service.
3. The coordinator of the service instructs the service to prepare itself to commit.
4. If the service can commit the transaction, an approval message is sent to the coordinator of the client.
5. After the coordinator of the client makes sure both parties are prepared to commit, it sends the client application a request to commit the transaction.
6. In addition to sending the client a request, the coordinator also sends a request to commit the transaction to the coordinator of the service.
7. The coordinator of the service passes the request to commit the transaction to the service implementation.

For a distributed transaction to work, both computers must have a DTC installed. In Windows operating systems, the Microsoft DTC (MSDTC) service processes distributed transactions. Therefore, you have to make sure that the DTC service is installed and is in the **Started** state. To verify this, open the services list by clicking the **Administrative Tools** tile on the Start screen, and then opening **Services**.

WCF supports distributed transactions in the following modes:

- **OLE Transactions (OleTx)**. The OleTx protocol is a Microsoft transaction protocol that is supported by clients and services developed in the .NET Framework, and by MSDTC. OleTx uses Remote Procedure Calls (RPC) to coordinate the transaction between the client and the service and therefore is not interoperable. OleTx is the default transaction mode of WCF when using non-interoperable bindings such as the **NetTcpBinding** and **NetNamedPipesBinding**.
- **WS-AtomicTransaction (WS-AT)**. WS-AT is a transaction protocol created by the OASIS (Organization for the Advancement of Structured Information Standards) consortium, and is part of the WS-* sets of standards. With WS-AT, the transaction coordination is handled by passing information in the SOAP headers of requests, and therefore is interoperable. WS-AT is also supported by MSDTC to communicate with remote DTCs. WCF uses WS-AT with its interoperable bindings, such as **WSHttpBinding**, but you can also configure non-interoperable bindings to use WS-AT.

Configuring non-interoperable bindings to use WS-AT is useful (for example, when OleTx RPC calls are blocked because of firewall rules).

If you plan on using WS-AT with MSDTC, you have to configure MSDTC. For the MSDTC configuration steps required for using WS-AT, see:

Configuring WS-Atomic Transaction Support

http://go.microsoft.com/fwlink/?LinkID=298786&clcid=0x409

WCF prefers to use OleTx when possible, because it provides better performance than WS-AT and requires fewer configurations in MSDTC. Even if your binding is configured for WS-AT, WCF tries to upgrade the transaction flow to OleTx, if possible.

You can change the way WCF coordinates transactions by changing the binding configuration of your service endpoints. This setting will be discussed later in this lesson. Before you decide to use distributed transactions in your services, remember that transactions have coupling and performance costs caused by resource locks and additional message traffic. One way to avoid this coupling is to use mechanisms other than transactions for undoing changes, such as compensations, as explained in the following link.

Compensation vs. Transactions

http://go.microsoft.com/fwlink/?LinkID=298787&clcid=0x409

Configuring Contracts and Bindings for Transactions

If you want your service to accept distributed transactions, you have to specify this in the service contract by using the **[TransactionFlow]** attribute. By decorating your service operations with the **[TransactionFlow]** attribute, you can state which operations supports flowing transactions and which operations do not.

The following code example demonstrates how to define a service contract that supports flowing transactions.

- Transaction flow can be configured at the contract level as **Mandatory**, **Allowed**, or **Not Allowed**

```
[ServiceContract]
public interface IBankContract
{
  [OperationContract]
  [TransactionFlow(TransactionFlowOption.Mandatory)]
  bool Transfer(Account from, Account to, decimal amount);
}
```

- Transaction flow requires the **TransactionFlow** binding element to be enabled in the binding

```
<netTcpBinding>
  <binding name="TransactionalBinding" transactionFlow="true"/>
</netTcpBinding>
```

Configuring Service Contract for Transaction Flow

```
 [ServiceContract]
public interface IBankContract
{
   [OperationContract]
   [TransactionFlow(TransactionFlowOption.Mandatory)]
   bool Transfer1(Account from, Account to, decimal amount);

   [OperationContract]
   [TransactionFlow(TransactionFlowOption.Allowed)]
   bool Transfer2(Account from, Account to, decimal amount);
}
```

You can use the **TransactionFlowOption** enumeration to determine whether a transaction can flow from the client to the service. The possible values for this enumeration are as follows:

- **Allowed**. The transaction of the client may flow to the service.

- **NotAllowed**. The transactions of the client may not flow to the service. If the client sends a request with transaction information, the request is rejected with a protocol exception. This is the default setting for the **[TransactionFlow]** attribute.
- **Mandatory**. The transactions of the client must flow to the service. If a request is made from a client without transaction information, the request is rejected with a protocol exception.

The **[TransactionFlow]** attribute controls only whether the service can receive flow transactions. It does not state anything about whether and how the service uses the transaction. That depends on the service implementation.

To enable transaction flow through your service, you have to set the **TransactionFlow** setting of your binding to **true**. The **TransactionFlow** setting can be applied to any binding that supports SOAP header. For example, you can set it with **WSHttpBinding** and **NetTcpBinding**, but you cannot set **TransactionFlow** in **BasicHttpBinding**.

The following code example demonstrates how you can configure a **NetTcpBinding** to enable transaction flow.

Enabling Transaction Flow with NetTcpBinding

```
<netTcpBinding>
    <binding name="TransactionalBinding" transactionFlow="true">
            . . .
    </binding>
</netTcpBinding>
```

In some supporting bindings, such as **NetTcpBinding**, you can also set the **transactionProtocol** attribute to configure, which protocol WCF uses to coordinate the transaction: **OleTransactions**, **WSAtomicTransactionOctober2004**, or **WSAtomicTransaction11**. The first protocol uses the OleTx transaction protocol, and the latter two protocols use different versions of the WS-AT standard. **WSAtomicTransactionOctober2004** uses version 1.0 from October 2004 and **WSAtomicTransaction11** uses version 1.1 from July 2007.

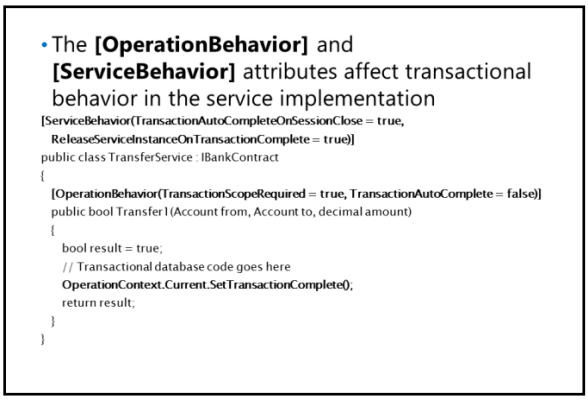
The following code example demonstrates how you can use the **<transactionFlow>** binding element if you build your own custom binding.

Creating a Custom Binding that Supports Transactions

```
<customBinding>
  <binding name="transactionalBinding">
    <reliableSession/>
    <transactionFlow transactionProtocol="WSAtomicTransaction11"/>
    <textMessageEncoding/>
    <httpTransport/>
  </binding>
</customBinding>
```

The **[TransactionFlow]** attribute should also appear in your client-side binding configuration. If you use the **Add Service Reference** dialog box of Visual Studio 2012, this attribute is added automatically to the client configuration. If you use the **ChannelFactory<T>** generic class and manually create the client configuration file, you have to add this attribute yourself.

Implementing Transactions in Services and Clients



When you use transactions in services, you do not have to explicitly commit or roll back the transaction. Instead, your code executes in a transaction scope, where all the code that executes participates in a transaction. When you execute a service operation in the transaction scope, you have to toggle the **Complete** flag of the transaction to state that the transaction can be completed successfully. If the flag is not toggled by the end of the scope, or if the code fails while you are in the scope, the transaction aborts. When you call several service operations while in the transaction scope, only the last operation of each service has to toggle the **Complete** flag and the previous flag settings are ignored.

Note: If you recall the DTC two-phase commit diagram, when the coordinator for the service checks that the service can complete its transaction, it actually checks that the transaction scope was marked as completed.

When you use transactions in your service implementation, you have to specify how scopes are created and used, and when a scope is marked as complete. You can control these settings by using the **[ServiceBehavior]** attribute and the **[OperationBehavior]** attribute.

The following code example demonstrates how you can set transaction-related settings by using the **[ServiceBehavior]** attribute.

Setting Transaction-related Settings with the [ServiceBehavior] Attribute

```
[ServiceBehavior(
   TransactionAutoCompleteOnSessionClose = true,
   ReleaseServiceInstanceOnTransactionComplete = true)]
public class TransferService : IBankContract
{
    . . .
}
```

The **TransactionAutoCompleteOnSessionClose** property controls if the transaction automatically becomes marked as complete when the client's session closes without errors. Setting this property to **true** requires that the service use a session. The default value of this property is **false**.

The **ReleaseServiceInstanceOnTransactionComplete** property controls if the service instance is destroyed as soon as the transaction is marked as complete. When you set this property to **true** and use the **PerSession** instancing mode, every time that a transaction is marked as completed, and the service instance is destroyed.

In addition to setting your service behavior, you can configure each of your operations to define how they handle transactions.

The following code example demonstrates how you can use the **[OperationBehavior]** attribute to configure transactions in service operations.

Setting Transaction-related Settings with the [OperationBehavior] Attribute

```
[OperationBehavior(
   TransactionScopeRequired = true,
```

```
   TransactionAutoComplete = false)]
public bool Transfer1(Account from, Account to, decimal amount)
{
   bool result = true;
   // Transactional database code goes here
   OperationContext.Current.SetTransactionComplete();
   return result;
}
```

Setting the **TransactionScopeRequired** property to **true** indicates that the operation requires a transaction for its work. If a transaction scope does not flow from the client, the service creates its own scope. Creating a scope depends on how you have set the **TransactionFlow** attribute in your contract, and whether the client flows a transaction into the service. The default value of the **TransactionScopeRequired** property is **false**.

The following table describes the different scenarios of configuring the **TransactionScopeRequired** property, and how it is affected by transactions that flow from the client.

TransactionScopeRequired	TransactionFlow	Client passes transaction	Result
True	Allowed / Mandatory	Yes	Operation uses the flowed transaction.
True	Allowed / NotAllowed	No	Operation uses a new transaction.
False	Allowed / Mandatory	Yes	Operation is executed without a transaction.
False	Allowed / NotAllowed	No	Operation is executed without a transaction.

 **Note:** Incorrect settings that may throw exceptions are omitted from the table. For example, a service throws an exception if a transaction scope is required and the transaction flow is mandatory but the client did not pass a transaction to the service.

When an operation uses a transaction scope, you can mark it as complete in one of two ways:

- Set the **OperationBehavior.TransactionAutoComplete** property to **true**. By setting this property to **true**, the transaction automatically sets to complete if the operation finishes without faulting. The default value for this property is **false**.
- Set the transaction completion in code. If the **TransactionAutoComplete** property is set to **false**, you can set the transaction to complete by calling the **SetTransactionComplete** method, as shown in the earlier example. You have to manually complete a transaction when certain execution paths require the transaction to be aborted.

To call a WCF service within a transaction, you have to wrap your code in a transaction scope, and then set the transaction to a completed state before ending it.

The following code example demonstrates an example of how to use a distributed transaction on the client.

Using Transactions in the Client

```
using (TransactionScope scope = new TransactionScope())
{
```

```
    bank1Proxy.Transfer1(accountOne, accountTwo, amount);
    bank2Proxy.Transfer1(accountThree, accountFour, amount);
    bank2Proxy.Transfer2(accountThree, accountFour, amount);
    scope.Complete();
}
```

When you create a transaction scope, you can involve several services in one distributed transaction. Because all participating parties in a distributed transaction have to mark the transaction as completed, the client also must do so by calling the **TransactionScope**.**Complete** method.

 **Note:** To use the **TransactionScope** class, add a reference to the **System.Transactions** assembly, and add a using directive for the **System.Transactions** namespace.

Demonstration: Creating WCF transactional services

This demonstration shows how to create a service contract that requires transactions, how to configure the binding to flow the transaction from client to service, how to implement a service that has transactional behaviors, and how to call the service from the client that has a distributed transaction.

Demonstration Steps

1. Open the D:\Allfiles\Apx01\DemoFiles\Transactions\Transactions.sln solution file and observe the client-side code.

- Open the **Program.cs** file from the **Client** project and examine the contents of the **Main** method.
- Observe how the client application first tests a successful transaction, and then tests an unsuccessful transaction.

2. View the service contract and the use of the **[TransactionFlow]** attribute. Observe the use of the different flow levels (**Mandatory** and **Allowed**).
3. In the **Service** project, open the **App.config** file, and view the binding configuration where the **transactionFlow** attribute is set to **true**.
4. Open the **TransferService.cs** from the **Service** project and inspect the service implementation.

- View the **[OperationBehavior]** attribute decorating the **Transfer** method. Note how the **TransactionScopeRequired** and **TransactionAutoComplete** parameters are set to **true**.

5. Observe how Entity Framework is used in the **Transfer** method. The transaction started by the **SaveChanges** method is elevated to a distributed transaction.
6. In the **Client** project, open the **Program.cs** file. View how the channel factory is created and how the binding is configured to flow the transaction from the client to the service. Note that in the first transaction scope block, the **Complete** method is called, but in the second transaction scope block it is not called.
7. Run both client and service projects, and view the printouts in the **Service** console window. The first distributed transaction is committed, and the second distributed transaction rolls back.

Lesson 3
Extending the WCF Pipeline

When a WCF service receives a message, the message travels a long way through various parts of the WCF infrastructure until it reaches the service operation to which it is addressed. Most of the work performed on the message—such as inspecting it, extracting information from it, and deserializing it—is performed by built-in mechanisms of WCF. Nevertheless, you can add custom message handling implementation to different parts of the infrastructure. As soon as you gain access to the message, WCF offers you an easy application programming interface (API) to examine and change the WCF contents, and to store relevant message information for you to use later when processing the message in the operations of the service.

In this lesson, you will learn how WCF controls the behavior of the service and its components, and how you can customize this behavior to suit your needs. You will also learn how you can inspect messages and save state information by using various techniques.

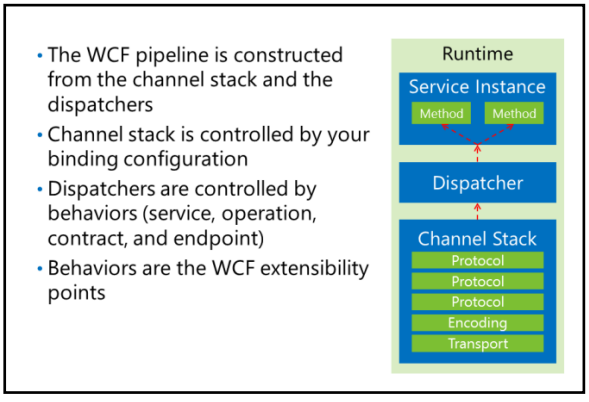
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the architecture of the WCF pipeline.
- Explain the responsibilities of the channel stack.
- Explain the responsibilities of the dispatchers.
- Create custom runtime components.
- Apply custom runtime components to the WCF pipeline by using custom behaviors.
- Attach custom behaviors to services, endpoints, contracts, and operations.
- Create and use extensions objects.

High-level Architecture of the WCF Pipeline

Messages that are received by a WCF service have a long distance to travel, from the moment they arrive until the moment they are sent to your service implementation for execution. After a message arrives at the transport channel, it flows through the channel stack, through the encoding, and through any other channel stack protocols—such as a security protocol, or a reliable messaging protocol.



Note: The channel stack will be thoroughly explained in the next topic. For now, consider the channel stack as the building blocks of your binding. For example, if your endpoint is configured to use the **BasicHttpBinding** binding, your channel stack has an HTTP transport channel, with a text encoder and no other channel stack protocols, because **BasicHttpBinding** does not use any WS-* protocol.

After all the protocols in the channel stack have verified the message, the message has several more steps to pass before reaching the service implementation. For example, some part of the pipeline has to

deserialize the message to the appropriate common language runtime (CLR) types, and inspect it to determine which service method to invoke and which service instance to use.

These decisions and actions are performed by the dispatchers, which are the components responsible for translating the content of the message to a method call. The way the dispatcher is constructed is defined by behaviors—service behaviors, endpoint behaviors, contract behaviors, and operation behaviors. Instancing, serialization, throttling, and authorization are only a few examples of the different behaviors that control dispatcher operations.

The throttling behavior is not covered in this course. However, you should become familiar with it as it can help you control your service's performance.

For more information about this service behavior and how to configure it, see:

Using ServiceThrottlingBehavior to Control WCF Service Performance

http://go.microsoft.com/fwlink/?LinkID=298788&clcid=0x409

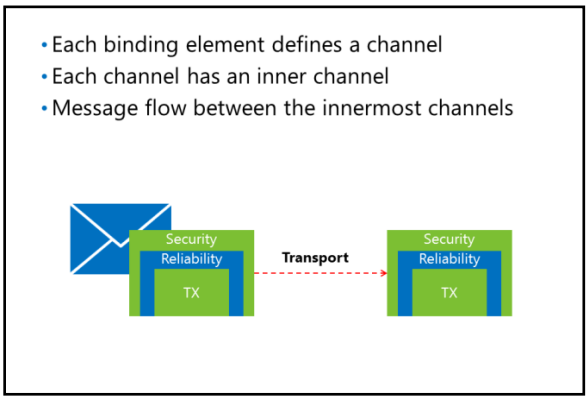
You define and configure behaviors by using code or configuration. For example, adding the **<serviceMetadata>** element under the **<serviceBehavior>** element in the service's configuration file adds the metadata publishing behavior to your service. The **[OperationBehavior]** attribute decorates your service operations and configure how to create transaction scopes. Each of these behaviors changes the operation of the dispatcher.

In addition to using the behaviors that are built-in to WCF, you can write custom behaviors. With custom behaviors, you can add message handling logic, and change the way messages process in the dispatcher. For example, you can change the way errors are handled and logged, or provide custom message validation.

You can apply custom behaviors—just like standard behaviors—to your services, endpoints, contracts, and operations, by using either code or configuration. The ability to extend dispatchers by using custom behaviors is the most important extensibility point in WCF.

Responsibilities of the Channel Stack

The message pipeline is implemented in WCF as a pipeline of channels, or a channel stack. The channel stack is built out of two channel types: the protocol channel and the transport channel. Each protocol channel has an inner channel property that contains the next channel to be used. However, the transport channel does not hold an inner channel, because it is the last channel that is called as the message is transmitted to the client.



There are three kinds of binding elements: the protocol, the encoder, and the transport. Each protocol channel holds one protocol binding element. Therefore, there may be zero or more protocol channels in the channel stack, each one connected to the other, exactly like a pipeline, where each pipe has another pipe attached at its end.

The transport channel, which is responsible for taking the message and transmitting it to the client, contains both the encoder binding element and the transport binding element. For example, if the binding elements are HTTP transport, text encoding, and security and reliability protocols, the channels that are used are as follows: protocol channel (reliability element), protocol channel (security element),

and transport channel (HTTP and text elements). When a message is sent, it passes from the outermost channel to the transport channel. The binding you select for your endpoint is a representation of a composition of channels.

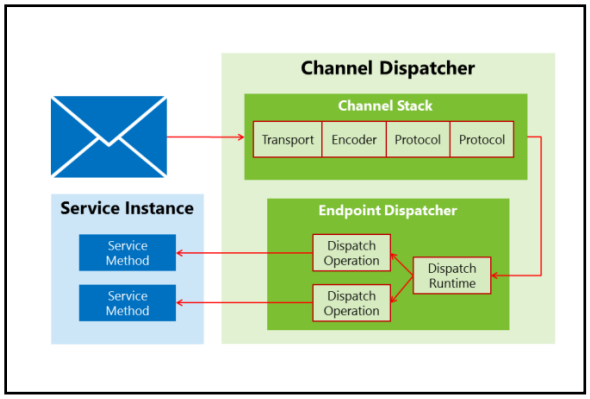
In WCF, you can extend both the Channel Stack and the Dispatchers by creating new channels and runtime components. For example, you can create new transport channels for protocols that are not currently supported by WCF, such as the Pragmatic General Multicast (PGM) protocol.

Creating new channels is an advanced topic and will not be covered in this course. However, the other topics in this lesson will explain in depth how to extend the Dispatchers.

Responsibilities of the Dispatchers

WCF dispatchers are divided into several types of scopes as follows:

- **Channel scope**. Responsible for handling all the messages that are received by a specific binding type (a specific URI and binding configuration).
- **Endpoint scope.** Responsible for handling all the messages directed to a specific endpoint.
- **Operation scope**. Responsible for handling all the messages directed to a specific operation in a specific endpoint.



Channel Dispatcher

The **ChannelDispatcher**—which is created by the service host—listens to messages on a specific channel, and associates messages from the channel with specific endpoints by using the appropriate endpoint dispatcher.

When a service host opens, it creates a **ChannelDispatcher** object for every combination of Uniform Resource Identifier (URI) and binding element—such as transfer, encoding, and protocols—to which it listens. For example, if a host is listening on its base address for service metadata requests and has two more endpoints with **WSHttpBinding** and **NetTcpBinding** bindings, then it has three channel dispatchers: one channel dispatcher for the base address, one for the **WSHttpBinding** binding, and another for the **NetTcpBinding** binding.

A single **ChannelDispatcher** can be created for more than one endpoint. For example, if your service has endpoints for two contracts of the service, and both endpoints use the same binding and same address, then your host creates a single **ChannelDispatcher** for both endpoints. When a message is received by the **ChannelDispatcher**, it checks the message to see which endpoint should receive the message.

Each channel dispatcher contains information about the URI on which it listens, its channel stack, and the endpoints (represented by **EndpointDispatcher** objects) that are used by this channel. When the channel dispatcher receives a message, the dispatcher passes the message across to the endpoint dispatchers of which it contains, to see which of them can handle the message. When the matching endpoint dispatcher is found, the channel dispatcher passes the message to it for processing.

The channel dispatcher is also responsible for various behaviors of the service, such as handling error, throttling, and time-out settings for receiving and sending messages. For example, you can extend the channel dispatcher by creating a new error handler that logs all uncaught exceptions to a log file.

Endpoint Dispatcher

The **EndpointDispatcher** class is responsible for receiving messages that are sent to a specific endpoint, and then passes them to the appropriate service operation by using a **DispatchOperation** object. The endpoint dispatcher receives a message from the channel dispatcher, and then checks if the message was sent to it by checking the **To** and the **Action** elements that are specified in the message header. The check is performed by using the **AddressFilter** and the **ContractFilter** properties. After the endpoint dispatcher receives a message, it passes it to its **DispatchRuntime** object. In turn, it passes the messages to the relevant **DispatchOperation** object, which is responsible for invoking the service instance method.

You can extend the endpoint dispatcher by supplying your own implementation of address and contract filter. For example, you can create a new address filter that ignores the host name in the address, to support services that are hosted behind load balancers.

Dispatch Runtime

Each endpoint dispatcher holds a **DispatchRuntime** object, which is responsible for selecting the most suitable **DispatchOperation** object according to the content of the message. In addition to selecting the **DispatchOperation** object, the dispatch runtime is also responsible for the following tasks:

- Performing message inspection by using custom message inspectors.
- Initializing the service instance context and managing its lifetime.
- Applying the role provider and authorization manager on the service instance.

 **Note:** The role provider and authorization manager will be discussed in Appendix B, "Implementing Security in WCF Services" in Course 20487.

You can extend the dispatch runtime and add your own custom processing by adding message inspectors and instancing providers to the dispatcher. The extensions that you add to the dispatch runtime affect all the messages that relate to the endpoint that contains the dispatch runtime. For example, you can use the dispatch runtime to add a custom message validation that guarantees that messages directed to a specific endpoint contain a custom SOAP header. Or it can add a special service instance provider that supplies a pool of service instances instead of using the default service instance creation techniques the dispatch runtime offers (per-call, per-session, and single).

Dispatch Operation

The **DispatchOperation** object is responsible for invoking the service method that is specified in the message, and passing the parameters contained in the message to it. To do this, the **DispatchOperation** object performs the following tasks:

1. Obtains the service implementation instance from the dispatch runtime.
2. Deserializes the message to the matching Common Language Runtime (CLR) types.
3. Applies parameter inspectors to all the parameters sent to the method.
4. Invokes the method with the appropriate parameters.
5. Serializes the response, and creates the response message object.

You can customize and extend each of those tasks. For example, you can add a custom parameter inspector that validates CLR data after it is deserialized, to check for irregular values such as string length, and integer values range.

 **Note:** You can also use message inspectors to validate data. However, it is more difficult than using the parameter inspector, because message inspectors are called when the message is

still in XML format. The XML format requires more work to parse and validate specific parts of the message.

By extending the dispatch operation, you can add custom processing to the operation level. It affects all the messages sent to a specific operation that is specified in the contract, regardless of the endpoint through which the message was received.

Client-Side Dispatchers

The dispatcher components that were mentioned earlier are used on the service side. On the client side, other components are responsible for creating and handling messages that are sent to the service. The proxies, generated in WCF clients by the **Add Service Reference** dialog box of Visual Studio 2012 and by the **ChannelFactory<T>** generic class, are responsible for converting method calls and parameters to outgoing messages, and for converting response messages back to return values.

When the client calls an operation, such as a proxy method, the **ClientOperation** object—which is a part of the proxy—takes the parameters that are sent to the method, inspects them, and then serializes the parameters to a message. After the client operation finishes building the message, it sends it to the **ClientRuntime** object. The **ClientRuntime** object then inspects the message, creates the outgoing channel, and passes the message to the channel and from there to the service.

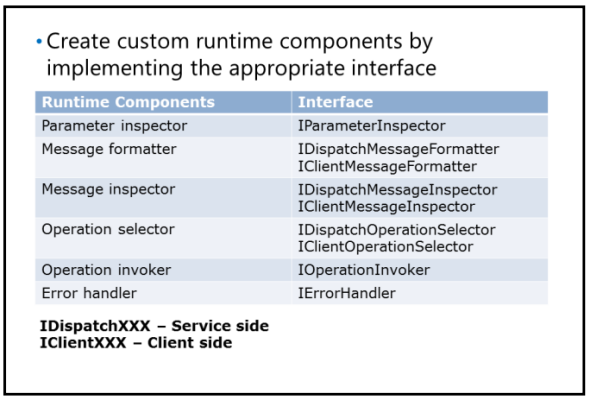
You can use the **ClientRuntime** and the **ClientOperation** objects to customize the behavior of your service. You can customize the **ClientRuntime** to change the behavior of all the operations in the contract that are exposed by the proxy, whereas the **ClientOperation** offers extensibility for a specific operation. For example, you can build a message inspector that adds a custom SOAP header to all outgoing messages, and then put this inspector in the **MessageInspectors** collection of the **ClientRuntime** object.

For additional information about the client runtime, see:

Extending Clients

http://go.microsoft.com/fwlink/?LinkID=298789&clcid=0x409

Creating Custom Runtime Components



You can use the extensibility points in the dispatchers to add custom runtime components that can handle various tasks, such as message and parameter inspection, custom serialization and deserialization of messages, and service instance creation. To build custom runtime components, you have to implement specific interfaces according to the kind of the component that you want to build. For example, to build a custom runtime component that performs message inspection, you have to write a class that implements the **IDispatchMessageInspector** interface and add it to the **DispatchRuntime** object that you want to customize. There are several runtime components that you can create to control the message flow in your service. All the runtime component interfaces are declared in the **System.ServiceModel.Dispatcher** namespace, which is a part of the **System.ServiceModel** assembly.

Some of these interfaces are as follows:

- **Parameter inspectors**. You can use parameter inspectors to check the value of parameters before invoking the operation's method. You can use them to validate constraints on your data types, such as maximum length of strings or the size of arrays. To implement a parameter inspector, you have to create a class that implements the **IParameterInspector** interface. You can add the parameter inspector component by adding it to the **ParameterInspectors** collection of the **ClientOperation** object (on the client-side) or the **DispatchOperation** object (on the service-side).
- **Message formatters**. You can use message formatters to customize how messages deserialize to parameters, how return values are serialized to messages on the service-side, how parameters serialize to messages, and how response messages deserialize to return values on the client-side. If you want to build a custom message formatter, you can implement either the **IDispatchMessageFormatter** or the **IClientMessageFormatter**. **IDispatchMessageFormatter** is required for customizing the service-side, and **IClientMessageFormatter** is required customizing the client-side. You can apply this runtime component by setting the **Formatter** property of your **DispatchOperation** object on the service-side, or of the **ClientOperation** object on the client-side.
- **Message inspectors**. You can use message inspectors to validate and extract information that is contained inside the message sent from a client to the service, and inside the message sent from the service to the client. You can implement the inspectors on either side: client or service. If you want to build an inspector that you use in the service, you have to implement the **IDispatchMessageInspector** interface. If you want to build an inspector for the client-side, you have to implement the **IClientMessageInspector** interface. To use the message inspector runtime component, add it to the **MessageInspectors** collection of your **DispatchRuntime** object on the service-side, or of the **ClientRuntime** object on the client-side. You can have multiple custom message inspectors in your service pipeline, each handling a different aspect of the message. For example, you can have one message inspector to verify the presence of a SOAP header, and another message inspector that adds missing elements in messages sent by older versions of the clients.
- **Operation selectors**. You can create custom operation selectors to change the way the dispatch runtime finds the dispatch operation that can process the incoming message. You can use operation selectors when you have a new version of a service that has changes to operation names, and you want the service to be backward compatible with older clients that still use the old operation names. If you want to build a selector for the service-side, you have to implement the **IDispatchOperationSelector** interface. If you want to build a selector for the client-side, you have to implement the **IClientOperationSelector** interface. You can use the client-side operation selector to map between proxy methods and service operations. You can apply this runtime component by setting the **OperationSelector** property of your **DispatchRuntime** object on the service-side, or of the **ClientRuntime** object on the client-side.
- **Operation invokers**. You can change the way the operation invocations translate to method calls by using the operation invoker runtime component. For example, you can change the orders of parameters sent to the method or log each method call. The operation invoker runtime component can only be applied on the service-side. To build an operation invoker, you have to implement the **IOperationInvoker** interface, and apply it in your service by setting the **Invoker** property of your **DispatchOperation** object.
- **Error handlers**. You can create custom error handlers to extend the way WCF handles exceptions. For example, you might want to catch every exception and log it to the database, or replace any unhandled exception with a general fault message that contains the message "*Please call the help desk at (555) 555-5555 to report this problem*". To create a custom error handler, you have to implement the **IErrorHandler** interface, and apply it in your service by adding it to the **ErrorHandlers** collection of the **ChannelDispatcher** object. Error handlers can only be applied on the service-side, and you can have multiple error handlers in the same channel. For example, you can create one error handler that is specific to Structured Query Language (SQL) Server exceptions, and another error handler to handle other kinds of exceptions.

The following code example demonstrates how to create a custom operation invoker.

Creating a Custom Operation Invoker

```
public class TimingOperationInvoker : IOperationInvoker
{
    IOperationInvoker _previousInvoker = null;

    public TimingOperationInvoker(IOperationInvoker previousInvoker)
    {
        _previousInvoker = previousInvoker;
    }

    public object[] AllocateInputs()
    {
        return _previousInvoker.AllocateInputs();
    }

    public object Invoke(object instance, object[] inputs, out object[] outputs)
    {
        Stopwatch timer = new Stopwatch();
        timer.Start();
        object result = _previousInvoker.Invoke(instance, inputs, out outputs);
        timer.Stop();
        Console.WriteLine("Operation took {0} milliseconds to execute",
timer.ElapsedMilliseconds);
        return result;
    }

    public IAsyncResult InvokeBegin(object instance, object[] inputs, AsyncCallback
callback, object state)
    {
        // Use the default invoker. Don't time async methods
        return _previousInvoker.InvokeBegin(instance, inputs, callback, state);
    }

    public object InvokeEnd(object instance, out object[] outputs, IAsyncResult result)
    {
        // Use the default invoker. Don't time async methods
        return _previousInvoker.InvokeEnd(instance, out outputs, result);
    }

    public bool IsSynchronous
    {
        get { return _previousInvoker.IsSynchronous; }
    }
}
```

As you can see in this example, the operation invoker has a constructor that receives the previous invoker. The previous invoker is the invoker currently used by the **OperationDispatcher**. If this is the first custom operation invoker that is attached to the pipeline, the previous invoker becomes the default operation invoker of WCF. Then, the previous invoker has the required code to invoke the service method, send it the parameters it requires, and return its return value.

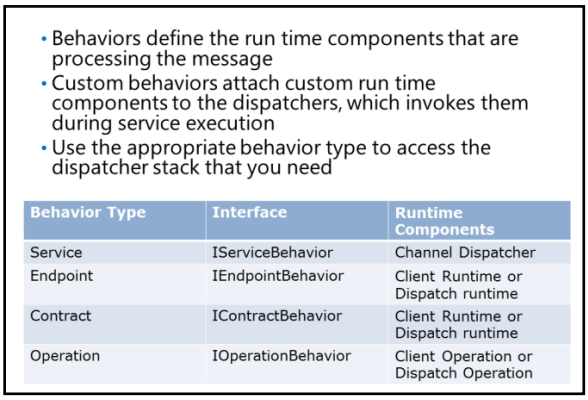
 **Note:** If you create several operation invokers for an operation, you must connect each of them by sending each invoker to the constructor of its successor. The last custom operation invoker receives the default operation invoker to its constructor. For example, you can have an operation invoker that logs the result of each service method, followed by an operation invoker that calculates the time of each method execution, followed by the default operation invoker that performs the actual invocation of the service method.

Through the code, the previous invoker is used to perform the actual invocation of the service method. The custom invoker adds code before and after invoking the service method, to calculate how much time that it took for the service method to execute.

The **IOperationInvoker** declares the following methods and properties:

- **AllocateInputs**. This method is called before invoking the operation to obtain the parameters that are to be sent to the invoked method. This method calls the message formatter, which deserializes the message to an array of objects.
- **IsSynchronous**. This property returns a value that specifies if the operation is to be invoked synchronously or asynchronously.
- **InvokeBegin** and **InvokeEnd**. These methods are called if the operation is to be invoked asynchronously.
- **Invoke**. This method is called if the operation is to be invoked synchronously.

Applying Runtime Components with Custom Behaviors



After you build custom runtime components, you have to decide which dispatcher you want to attach to these components. This must be done at runtime, when your service host opens. To do this, you must have a custom behavior. Custom behaviors are the mechanism that you use to control which runtime components are added to the various dispatchers. Using custom behaviors, you can access the **ChannelDispatcher**, **EndpointDispatcher**, **DispatchRuntime**, and **DispatchOperation** objects, and then add runtime components to them. You can also use custom behaviors to remove existing runtime components from the dispatchers.

When you build a custom behavior, you have to decide to which scope you want to apply the behavior. For example, you can build a custom behavior and attach it to a specific endpoint so that it only applies to the operations of that endpoint. Or you can build a custom behavior and attach it to the whole service so that it applies to all the operations in all the different endpoints exposed by the service.

WCF offers you the following kinds of behaviors:

- **Service behaviors**. By creating a service behavior and applying it to your service, you can change the runtime components of all the operations in all the endpoints of your service. For example, if you want to add a custom error handler runtime component that handles exceptions from all the service operations, you have to add it using a service behavior. Only service behaviors have access to the channel dispatcher where this runtime component is declared. In addition to customizing the runtime components, you can also customize the service host itself. To create a custom service behavior, you have to implement the **IServiceBehavior** interface. You can apply service behaviors to a service by creating the custom behavior as a custom attribute and then adding it on the service implementation. Or you can add the behavior to the service's behavior configuration in the configuration file.
- **Endpoint behaviors**. You can use endpoint behaviors to apply changes to a specific endpoint and its operations. Building an endpoint behavior instead of a service behavior is useful if you only have to customize the runtime components of a specific endpoint. For example, you can take a message inspector runtime component that performs message logging, create an endpoint behavior for it, and only apply the behavior to endpoints that do not require clients to authenticate. To create a custom

endpoint behavior, you have to implement the **IEndpointBehavior** interface. You cannot apply endpoint behaviors by using attributes, because they must be applied directly to an endpoint declaration. Endpoint behaviors can be added to the endpoint's behavior configuration in the configuration file.

- **Contract behaviors**. When using a contract behavior, you can apply changes to all the operations of a specific contract, regardless of the endpoint in which it is declared. To create a custom contract behavior, you have to implement the **IContractBehavior** interface. You can apply contract behaviors to your contract or service implementation only by using custom attributes, because there is no contract configuration in the configuration file.
- **Operation behaviors**. You can use an operation behavior to change the runtime components that are used for a specific operation, regardless of the endpoint dispatcher through which it was invoked. For example, you can create a custom operation invoker that logs the duration of time it takes an operation to execute, and then apply it to several operations while testing the service. To create a custom operation behavior, you have to implement the **IOperationBehavior** interface. Like the contract behavior, operation behaviors can only be attached to an operation by using custom attributes.

The following table lists which behavior type is most suitable to the scope that you want to control with the custom runtime component.

Behavior type	Message scope
Service behavior	All the messages sent to the service
Endpoint behavior	All the messages sent to a specific endpoint
Contract behavior	All the messages sent to a specific contract in the service, regardless of the endpoint that exposes it
Operation behavior	All the messages sent to a specific service operation

Each interface mentioned earlier contains the **ApplyDispatchBehavior** method, which gives you access to the dispatchers so that you can apply custom runtime components to them. If you want to customize the runtime components on the client-side, you can create an endpoint behavior, a contract behavior, or an operation behavior. Each behavior contains the **ApplyClientBehavior** method, which you can use to gain access to either the client runtime or the client operation to apply the necessary runtime component to them.

The following code example demonstrates how to build a custom operation behavior that attaches the custom operation invoker shown in the previous code example, "Creating a Custom Operation Invoker".

Creating a Custom Operation Behavior

```
public class TimingOperationBehavior : IOperationBehavior
{
    public void AddBindingParameters(
       OperationDescription operationDescription,
       System.ServiceModel.Channels.BindingParameterCollection bindingParameters)
    {
    }

    public void ApplyClientBehavior(
       OperationDescription operationDescription,
       System.ServiceModel.Dispatcher.ClientOperation clientOperation)
    {
    }

    public void ApplyDispatchBehavior(
```

```
        OperationDescription operationDescription,
        System.ServiceModel.Dispatcher.DispatchOperation dispatchOperation)
    {
        dispatchOperation.Invoker = new
TimingOperationInvoker(dispatchOperation.Invoker);
    }

    public void Validate(OperationDescription operationDescription)
    {
    }
}
```

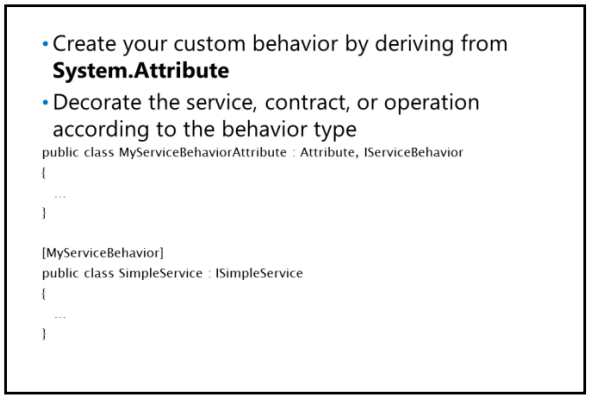
You can create operation behaviors for either the client side or service side. This is the reason why the interface exposes both the **ApplyClientBehavior** and **ApplyDispatchBehavior** methods. In the previous example, the operation invoker can be used in the service, and that is why the **ApplyClientBehavior** method is not implemented. By implementing the **ApplyDispatchBehavior** method, you can customize the runtime components used by the operation dispatcher. In this example, the method is used to plug-in the new operation invoker by wrapping the default invoker supplied by the WCF infrastructure.

In addition to the **ApplyClientBehavior** and **ApplyDispatchBehavior**, the **IOperationBehavior** interface exposes two more methods:

- **AddBindingParameters**. This method controls the binding elements in the channel stack. You can use this method to configure the binding elements. For example, you can use this method to change the time-out properties of the binding.
- **Validate**. Use this method to verify that you can successfully use the behavior by inspecting the operation description, such as verifying that the operations support a required fault contract.

Attaching Custom Behaviors to Services

A common practice when applying custom behaviors is to use custom attributes. If you write a custom service behavior, you can apply it to your service. If you write a contract behavior, you can apply it to your contract. Or if you write an operation behavior, you can apply it to your operation. You can implement all of these options by using custom attributes.



Note: You cannot use custom attributes to apply endpoint behaviors. You can only attach endpoint behaviors to endpoints by using the configuration file.

To build your custom behavior so that you can use it as a custom attribute, you have to derive your custom behavior from the **Attribute** class.

The following code example demonstrates how to create a custom service behavior by using a custom attribute.

Creating a Custom Attribute for the Custom Behavior

```
 [AttributeUsage(AttributeTargets.Method)]
public class TimingOperationBehaviorAttribute : Attribute, IOperationBehavior
{
```

```
…
}
```

You can create any custom behavior—whether service, endpoint, contract, or operation behavior—as a custom attribute by deriving from the **Attribute** class. You can also decorate the custom behavior class by using the **[AttributeUsage]** attribute to specify where this attribute can be used. The rest of the implementation of the class remains the same as it was.

 **Note:** In the previous example, the name of the class is changed from **TimingOperationBehaviorAttribute** to **TimingOperationBehaviorAttribute**, because the naming convention for custom attribute classes is to add the **Attribute** suffix to the name of the class.

After you create the new custom attribute, all that remains is to decorate the service, contract, or operation with the new attribute.

The following code example demonstrates how to apply the custom behavior to a service operation.

Applying Custom Behaviors in Code

```
public class SimpleService : ISimpleService
{
    [TimingOperationBehavior]
    public string PerformLengthyTask()
    {
        // Do something
        …
        return "Done";
    }
}
```

If you do not want to apply the custom behavior by using custom attributes, or if you build a custom endpoint behavior that cannot be applied by using custom attributes, you can apply the behavior by using the configuration. To use custom behaviors in configuration files, you have to create a class that derives from the **BehaviorExtensionElement** class.

The following code example demonstrates a behavior extension element.

Creating a Behavior Extension Element Class

```
public class TimingBehaviorConfigurationElement : BehaviorExtensionElement
{
    public override Type BehaviorType
    {
        get { return typeof(TimingEndpointBehavior); }
    }

    protected override object CreateBehavior()
    {
        return new TimingEndpointBehavior();
    }
}
```

The **BehaviorExtensionElement** declares two abstract members that you must override when you build your custom configuration extension:

1. **BehaviorType**: You must override this property to return a **Type** object that represents the kind of the custom behavior this configuration element creates.
2. **CreateBehavior**: You must override this method to return an instance of your custom behavior.

 **Note:** The preceding code example creates a behavior extension element class for the **TimingEndpointBehavior** endpoint behavior class. The content of this class is not shown here. However, the implementation of this class resembles the operation behavior that was demonstrated before.

To apply this behavior in the configuration, you have to introduce this behavior as an XML element. To do this, you have to add the newly created type to the extension element of the **<system.serviceModel>** element.

The following code example demonstrates how to add the previous configuration extension to the configuration file.

Adding a Custom Endpoint Behavior to the Configuration File

```
<system.serviceModel>
  <extensions>
    <behaviorExtensions>
      <add name="timingBehavior" type="Service.TimingBehaviorConfigurationElement,
Service"/>
    </behaviorExtensions>
  </extensions>
<system.serviceModel>
```

First you add the behavior extension and set its name and type by setting the **name** and the **type** attributes. Then you can use its name to apply the custom behavior in the configuration file of your service or in your endpoint configuration, depending on the kind of custom behavior that you create.

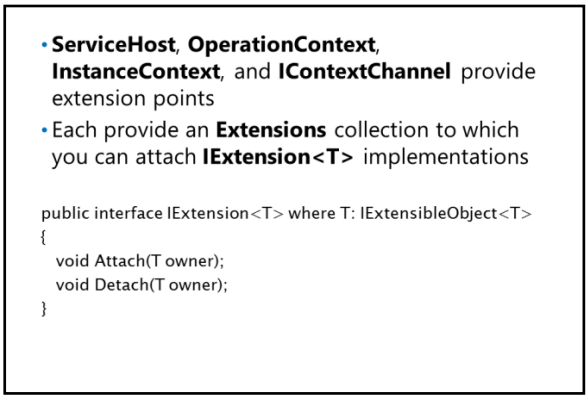
The following code example demonstrates how to apply the custom endpoint behavior in configuration.

Applying a Custom Endpoint Behavior in Configuration

```
<system.serviceModel>
  <extensions>
    <behaviorExtensions>
      <add name="timingBehavior" type="Service.TimingBehaviorConfigurationElement,
Service"/>
    </behaviorExtensions>
  </extensions>
  <behaviors>
    <endpointBehaviors>
      <behavior name="timing">
        <timingBehavior/>
      </behavior>
    </endpointBehaviors>
  </behaviors>
  <services>
    <service name="Service.SimpleService">
      <endpoint
          address=""
          binding="basicHttpBinding"
          contract="Service.ISimpleService"
          behaviorConfiguration="timing"/>
    </service>
  </services>
</system.serviceModel>
```

 **Note:** The **type** attribute must be set to the fully qualified name of the class, including its assembly name. If the class is from a different assembly, you should use the fully qualified name of the assembly.

Adding State and Functionality with Extensible Objects



Another method that you can use to extend various parts of your service is to use the extensible object pattern that WCF implements. By using the extensible object pattern, you can add new functionality to your services, operations, and channels.

The extensibility mechanism in WCF uses the **IExtensibleObject<T>** generic interface, which declares an **Extensions** property that holds a collection of extension objects. There are several types that implement this interface and enable extensions to attach to them. An extension object is any type that implements the **IExtension<T>** interface. By adding and removing extension objects from the collection, you can customize the functionality of the extensible objects.

The **IExtension<T>** interface declares the following methods:

- **Attach**. This method is called when the extension is added to the extensions collection of the extensible object. You can use this method to apply the required functionality changes.
- **Detach**. This method is called when the extension is removed from the extensions collection. Use this method to undo the changes that you have made to the extensible object.

The following WCF types implement the **IExtensibleObject<T>** interface:

- **ServiceHostBase**. This is the base type of the **ServiceHost** class. You can add extensions to this object to extend the behavior of your service host.
- **InstanceContext**. This class provides access to both the service instance that is created at run time, and to its contained service host. By adding extensions to the instance context, you can add functionality that is executed when the instance is created or destroyed.
- **OperationContext**. This class provides access to all the information about the current operation, such as the incoming message and the security identity of the client. By using extensions, you can change the behavior of the endpoint dispatcher of the operation, examine and change message content, and apply other customizations. For example, you can create a parameter inspector that stores the values of the o input parameters of the operation, in an extension, and use the extension object in an error handler to log the parameters. If an operation throws an exception, the parameters, which might have been the cause of the exception, as well as the error handler, can log the exception.
- **IContextChannel**. This interface is implemented by the service and client channels. You can add extensions to the channels to attach custom data to them, and use it at some point by custom behaviors and runtime components. For example, you can create an extension for the service-side that keeps a list of message headers (name, namespace, and value) that are added to every outgoing message. Because the channel can be accessed from anywhere in the service, any service operation and runtime component can add headers to the extension object. When a message is ready to be sent to the client, a message inspector can take all the headers from the extension, and include them in the returned message.

In addition to adding functionality to the runtime classes by customizing the dispatchers and the host, you can also use the extensions to hold state information that you use in your runtime components or in your service implementation, just as the example described for the **OperationContext** and **IContextChannel** extension objects.

For additional information about extensible objects in WCF, see:

Extensible Object

http://go.microsoft.com/fwlink/?LinkID=298790&clcid=0x409

You can use extensions to maintain state for different scopes of your service, to configure the behavior of your service, and to add functionality to various parts of your service. For example, you can create a service host extension that performs special processing when a host opens or closes.

The following code example demonstrates an extension that provides a singleton instance of a log manager that can be accessed from anywhere in the service.

Creating an Extension

```
public class SingletonLoggerExtension : IExtension<ServiceHostBase>
{
     public LogManager Logger { get; private set; }

     public SingletonLoggerExtension(string outputLogFile)
     {
         Logger = new LogManager(outputLogFile);
     }

     public void LogMessage(string message)
     {
       Logger.LogMessage(message);
     }

     public void Attach(ServiceHostBase owner)
     {
        Logger.Initialize();
     }

     public void Detach(ServiceHostBase owner)
     {
     }
}
```

After you attach the extension to the host, a logger instance is created and becomes available to anyone who needs it.

Note: Using extensions for singletons gives you more flexibility than by using the standard singleton design pattern or static classes. That is because with extensions, you can also declare singleton objects for scopes other than the whole service. For example, you can create a singleton extension that you can apply to the operation context. Meaning, all the runtime components and the service implementation code share the same singleton object, whereas other operation contexts have their own singleton object. This behavior can be very difficult to achieve when using the standard singleton design pattern or when using static classes.

To add this extension to the service host, you have to use the service host's **Extensions** collection.

The following code example demonstrates how to add the new extension to the service host.

Adding an Extension Object to Service Host's Extensions Collection

```
ServiceHost host = new ServiceHost(typeof(ISimpleService));
host.Extensions.Add(new SingletonLoggerExtension(@"c:\serviceLogs\log.txt"));
host.Open();
```

After you add the extension to the host, you can use it anywhere you want inside the service.

The following code example demonstrates how to use this extension in a service operation.

Using an Extension

```
public class SimpleService : ISimpleService
{
    public string PerformLengthyTask()
    {
        string result;

        // Do some work
        …

        ServiceHost host = OperationContext.Current.Host;
        SingletonLoggerExtension extension
=host.Extensions.Find<SingletonLoggerExtension>();
        extension.LogMessage("the result was: " + result);

        return result;
    }
}
```

By using the **Find<T>** generic method, you can locate a specific extension inside the extension collection. If you add several instances of the same extension type—for example, if the logger extension is added several times to create different output files—you can use the **FindAll<T>** generic method that returns a collection of extension objects.

Demonstration: Extending the WCF Pipeline

This demonstration shows how to create a custom operation invoker and how to apply it to the service implementation with a custom operation behavior. This demonstration also shows how to create an extension object, and how to access it from the service host and a custom runtime component.

Demonstration Steps

1. Open the **D:\Allfiles\Apx01\DemoFiles\CustomOperationInvoker\CustomOperationInvoker.sln** solution file. This solution shows how to create a custom operation invoker that prints time taken to execute each service operation.
2. Open the operation invoker code, and observe how the custom operation invoker uses the previous operation invoker for the methods and properties.

- In the **Service** project, open the **TimingOperationInvoker.cs** and observe how the class implements the **IOperationInvoker** interface.
- View the code of the constructor method. The constructor receives the previous operation invoker, in this case, the default invoker of WCF.
- Inspect the **AllocateInputs**, **Invoke**, **InvokeBegin**, and **InvokeEnd** methods, and the **IsSynchronous** property. Each method calls the matching method or property from the underlying **_previousInvoker** object.

3. Observe how you retrieve the extension object and log the execution time in the **Invoke** method by calling the **LogMessage** method of the **SingletonLoggerExtension** extension object.
4. View the code of the **SingletonLoggerExtension** class, and how it implements a service host extensible object.
5. Open the **Main** method and observe how the **SingletonLoggerExtension** instance is created and added to the service host's **Extensions** collections.
6. View the implementation of the custom operation behavior (the **TimingOperationBehavior** class), and observe how the new operation invoker replaces the existing one.

7. View the **SimpleService** class and observe how the custom operation behavior is applied to the **PerformLengthyTask** operation method.
8. Run the project without debugging, open the WCF Test Client utility, and then connect to the service by using the address **http://localhost:8080/SimpleService**. Verify the logger prints the execution time in the console window after you invoke the service.

- Open the WCF Test Client utility by double-clicking the **WcfTestClient** shortcut in the **D:\AllFiles** folder.
- After adding the service in the WCF Test Client, call the **PerformLengthyTask** method.
- After you invoke the method, switch to the console window, and verify that you see the message "*Operation took XXXX milliseconds to execute*" (XXXX will be replaced by the time that it took the operation to execute).

Lab: Designing and Extending WCF Services

Scenario

After running several testing cycles on the newly created booking service, the Blue Yonder Airlines software testing department wanted the error logs of the service to contain more information about the input of operations so that it is easier for them to analyze service faults. In this lab, you will add an error handler to the booking service, which logs the operation parameters for failed requests.

In addition, Blue Yonder Airlines wants to enable its Blue Badge members (from the Blue Yonder Airlines' frequent flyer program) to earn miles for checked-in flights. In this lab, you will update the WCF booking service to call the frequent flyer WCF service, and update the two databases—reservations and frequent flyers—using a single distributed transaction.

Objectives

After completing this lab, you will be able to:

- Create an error handling runtime component and apply it to a WCF service.
- Configure a WCF service to support distributed transactions.

Lab Setup

Estimated Time: 60 minutes

Virtual Machine: **20487B-SEA-DEV-A**

User name: **Administrator**

Password: **Pa$$w0rd**

For this lab, you will use the available virtual machine environment. Before you begin this lab, you must complete the following steps:

1. On the host computer, click **Start**, point to **Administrative Tools**, and then click **Hyper-V Manager**.
2. In Hyper-V Manager, click **MSL-TMG1**, and in the Action pane, click **Start**.
3. In Hyper-V Manager, click the **20487B-SEA-DEV-A** virtual machine.
4. In the **Snapshots** pane, right-click the **StartingImage** snapshot and then click **Apply**.
5. In the **Apply Snapshot** dialog box, click **Apply**.
6. In Hyper-V Manager, click **20487B-SEA-DEV-A**, and in the Action pane, click **Start**.
7. In the Action pane, click **Connect**. Wait until the virtual machine starts.
8. Sign in using the following credentials:

- User name: Administrator
- Password: **Pa$$w0rd**

9. Verify that you received credentials to log in to the Azure portal from you training provider, these credentials and the Azure account will be used throughout the labs of this course.

Exercise 1: Create a Custom Error Handler Runtime Component

Scenario

To log unhandled exceptions together with the parameters that were sent to the service operation, you have to create several runtime components and attach them to the WCF message handling pipeline. You can start by creating a parameter inspector which you can use to store the parameters sent to any service operation in an extension object. Then you can continue by creating an error handler that can retrieve the

stored parameters, serialize them to XML, and output the exception and parameter values to a log file. The last step of this exercise is to apply these custom runtime components to your service by using the configuration file of your service host.

The main tasks for this exercise are as follows:

1. Create an Operation Extension to Hold the Parameter Values
2. Create a Parameter Inspector that Stores the Parameter Values in an Operation Extension
3. Create an Error Handler that Traces Parameter Values for Faulty Operations
4. Create a Custom Service Behavior for the Error Handler and Apply it to the Service
5. Configure Tracing for WCF and the Custom Trace

▶ Task 1: Create an Operation Extension to Hold the Parameter Values

1. Open BlueYonder.Server.sln from D:\AllFiles\Apx01\LabFiles\begin\BlueYonder.Server.
2. In the **BlueYonder.BookingService.Implementation** project, create and implement an extensible object class named **ParametersInfo**, under a new folder named **Extensions**.

- Set the access modifier of the class to **public**.
- Implement the IExtension<OperationContext> interface
- Create the **Attach** and **Detach** methods, but leave them empty, as you will not use them.
- In the class, create a constructor which receive an array of objects. Store the array in a publicly available property named **Parameters**.

 **Note:** You do not have to add any code to the **Attach** and **Detach** methods, because you only use the extension for state management, not to add functionality to the operation context.

▶ Task 2: Create a Parameter Inspector that Stores the Parameter Values in an Operation Extension

1. In the **BlueYonder.BookingService.Implementation** project, create and implement a new parameter inspector class named **ParametersInspector**, under the **Extensions** folder.

- The class should implement the **IParameterInspector** interface.
- In the **BeforeCall** method, store the list of parameters sent to the operation in a new extension object of type **ParametersInfo**.
- Add the new extension object to the current operation context's **Extensions** collection.
- The method should return null when it completes.

 **Note:** You have to implement the **BeforeCall** method to save the parameters of the operation before the operation is invoked. You do not have to implement the **AfterCall** method, because it only executes after the operation is complete without exceptions.

▶ Task 3: Create an Error Handler that Traces Parameter Values for Faulty Operations

1. To the BlueYonder.BookingService.Implementation project, add D:\Allfiles\Apx01\Labfiles\Assets\Extensions\ErrorLoggingUtils.cs under the Extensions folder.
2. In the **BlueYonder.BookingService.Implementation** project, create and implement a new error handler class named **LoggingErrorHandler**, under the **Extensions** folder.

- Set the access modifier of the class to **public** and implement the **IErrorHandler** interface.
- Add a new private field of type **TraceSource** to the class and name it **_traceSource**. Initialize the trace source to use the trace source **ErrorHandlerTrace**.
- Implement the **ProvideFault** method and leave it empty.
- Implement the **HandleError** method by retrieving the parameters that you stored in the extension object.
- To retrieve the parameters, use the **Find** method of current operation context's **Extensions** collection.
- If the parameters were found, create a string containing the type and message of the exception, and the values of the parameters.
- Use the **ErrorLoggingUtils.GetObjectAsXml** static method to convert each of the parameters to an XML string.
- Write the string to the log by using the **_traceSource** field.
- Return **true** at the end of the method.

The resulting code of the **HandleError** method should resemble the following code.

```
ParametersInfo parametersInfo =
OperationContext.Current.Extensions.Find<ParametersInfo>();
if (parametersInfo != null)
{
  string message = string.Format
    ("Exception of type {0} occurred: {1}\n operation parameters are:\n{2}\n",
    error.GetType().Name,
    error.Message,
  parametersInfo.Parameters.Select
        (o =>   ErrorLoggingUtils.GetObjectAsXml(o)).Aggregate((prev, next) => prev +
"\n" + next));
  _traceSource.TraceEvent(TraceEventType.Error, 0, message);
}
return true;
```

Note: The **IErrorHandler** interface provides two methods, **ProvideFault** and **HandleError**. You can implement the **ProvideFault** method to provide a fault message to WCF based on the thrown exception. The **HandleError** is called after WCF returns the fault message to the client so that you can log the thrown exception without making the client wait until the logging procedure is complete.

▶ Task 4: Create a Custom Service Behavior for the Error Handler and Apply it to the Service

1. In the **BlueYonder.BookingService.Implementation** project, create and implement a new service behavior class named **ErrorLoggingBehavior**, under the **Extensions** folder.

- Implement the **IServiceBehavior** interface.
- Add the **AddBindingParameters** and **Validate** methods of the interface and leave them empty.
- Implement the **ApplyDispatchBehavior** method by iterating the **serviceHostBase.ChannelDispatchers** collection, and adding a new **LoggingErrorHandler** object to each channel dispatcher's **ErrorHandlers** collection.
- In each channel dispatcher iteration, iterate each of the endpoints, and in each endpoint, iterate the endpoint's **DispatchRuntime.Operations** collection.

- In each dispatch operation iteration, add a new **ParametersInspector** object to the operation's **ParameterInspectors** collection.

2. In the BlueYonder.BookingService.Implementation project, add a reference to the System.Configuration assembly,
3. In the **BlueYonder.BookingService.Implementation** project, create and implement a new behavior extension element class named **ErrorLoggingBehaviorExtensionElement**, under the **Extensions** folder.

- Set the access modifier of the class to **public** and inherit it from the **BehaviorExtensionElement** class.
- Override the **BehaviorType** property by returning type **Type** object for the **ErrorLoggingBehavior** class. Use the **typeof** operator to get the Type object of a class.
- Override the **CreateBehavior** method by returning a new instance of the **ErrorLoggingBehavior** class.

 **Note:** You can use a custom behavior in the configuration file only if you create a class for its configuration element. The configuration element class has to provide two things: the kind of the custom behavior class and an instance of it.

4. In the **BlueYonder.BookingService.Host** project, open the **App.config** file, and add a **<behaviorExtension>** element for the new configuration element that you created.

- In the **<system.serviceModel>** element, create a new **<extensions>** element, and in it create a **<behaviorExtensions>** element.
- In the **<behaviorExtensions>** element, add an **<add>** element with the following values.

Attribute	Value
name	errorLoggingBehavior
type	BlueYonder.BookingService.Implementation.Extensions.ErrorLoggingBehavior ExtensionElement, BlueYonder.BookingService.Implementation

 **Note:** The **type** attribute must be set to the qualified name of the configuration element class, including the name of its containing assembly. You can set the value of the **name** attribute to any name that you think best represents your custom behavior.

5. In the **<serviceBehaviors>** element, add the **<errorLoggingBehavior/>** element to the service's **<behavior>** element.

 **Note:** Visual Studio Intellisense uses built-in schemas to perform validations. Therefore, it will not recognize **errorLoggingBehavior** behavior extension, and will display a warning. Please disregard this warning.

▶ Task 5: Configure Tracing for WCF and the Custom Trace

1. Remaining in the **App.config**, add a **<system.diagnostics>** element and set the trace to automatic flush.

- In the newly added **<system.diagnostics>** element, add a **<trace>** element and set its **autoflush** attribute to **true**.

 **Note:** The **autoflush** attribute controls whether log messages are immediately written to the log, or cached in memory and periodically flushed. The value of the attribute is set to **true** so that you can view the results immediately without waiting for the log to flush its content to the file.

2. Add a shared listener to the **<system.diagnostics>** element.

- In the **<system.diagnostics>** element, add a **<sharedListeners>** element, and in it, add an **<add>** element with the following values.

Attribute	Value
name	ServiceModelTraceListener
type	System.Diagnostics.XmlWriterTraceListener
initializeData	D:\AllFiles\Apx01\LabFiles\WCFTrace.svclog

3. In the **<system.diagnostics>** element, add two sources, one for **System.ServiceModel** and another for **ErrorHandlerTrace**. Set both sources to use the shared listener you created before.

- In the **<system.diagnostics>** element, add a **<sources>** element, and in it, add two **<source>** elements.
- Set the attributes in the elements according to the following table.

Source element	Attribute	Value
First	name	System.ServiceModel
	switchValue	Error,ActivityTracing
Second	name	ErrorHandlerTrace
	switchValue	Error,ActivityTracing

- In each of the **<source>** elements, add a **<listeners>** element with the following configuration.

```
<listeners>
 <add name="ServiceModelTraceListener">
   <filter type="" />
 </add>
</listeners>
```

The resulting configuration should resemble the following configuration.

```
<source name ="System.ServiceModel" switchValue="Error,ActivityTracing">
  <listeners>
    <add name="ServiceModelTraceListener">
      <filter type="" />
    </add>
  </listeners>
</source>
<source name ="ErrorHandlerTrace" switchValue="Error,ActivityTracing">
  <listeners>
    <add name="ServiceModelTraceListener">
```

```
      <filter type="" />
    </add>
  </listeners>
</source>
```

 **Note:** The **System.ServiceModel** source is used for tracing WCF activities, and the **ErrorHandlerTrace** source is used by the **LoggingErrorHandler** class, in the **TraceSource** constructor.
WCF tracing is covered in Lesson 2, **Configuring Service Diagnostics**", of Module 10, **Monitoring and Diagnostics**.

4. Run the **BlueYonder.BookingService.Host** project without debugging, and test the service using the WCF Test Client utility

- After running the project, open the WCF Test Client utility by double-clicking the **WcfTestClient** shortcut from the **D:\AllFiles** folder.
- Add the service by using the address **http://localhost/BlueYonder/Booking**.
- Test the **UpdateTrip** operation by sending a null object. Wait until an error dialog box appears that states "*The confirmation code of the reservation is invalid*".
- Close the WCF Test Client utility.

5. From **D:\AllFiles\Apx01\LabFiles**, open the **WCFTrace.svclog** trace log file and verify that you see the exception with the XML of the **TripUpdateDto** parameter. Close the **Microsoft Service Trace Viewer** utility and the service host console window, and return to Visual Studio 2012.

Results: You can use the WCF Test Client utility to test the service, cause exceptions to be thrown in the code, and check the log files to verify that the exception message is logged together with the parameters that are sent to the service operation.

Exercise 2: Add Support for Distributed Transactions to the WCF Booking Service

Scenario

To support distributed transactions, you have to update both the Frequent Flyer service and the Booking service that acts as the client of the Frequent Flyer service. Start by updating the Frequent Flyer service contract and the service host configuration file to support transaction flow, and then continue to update the service implementation to use the flowed transactions. After updating the Frequent Flyer service, apply the same binding configuration to the client-side configuration in the Booking service, and use a transaction scope to create a distributed transaction that spans the service call and the changes that were made to the local database.

The main tasks for this exercise are as follows:

1. Add Transaction Flow Attributes to the Frequent Flyer Service Contract

2. Configure the Service Endpoint's Binding to Flow Transaction

3. Add the Transaction Scope Attribute to the Service Implementation

4. Add Code to the WCF Booking Service that Calls the Frequent Flyer WCF Service

5. Execute the Service Call and the Reservations Database Updates in a Distributed Transaction

6. Update the WCF Client Configuration with the Frequent Flyer Service Endpoint and the Support for Transaction Flow in the Bindings

▶ Task 1: Add Transaction Flow Attributes to the Frequent Flyer Service Contract

1. In the BlueYonder.FrequentFlyerService.Contracts project, open the IFrequentFlyerService.cs file, and set the AddFrequentFlyerMiles and RevokeFrequentFlyerMiles methods to allow the flow of transactions.

- Decorate the AddFrequentFlyerMiles and RevokeFrequentFlyerMiles methods with the [TransactionFlow] attribute and set the flow option to TransactionFlowOption.Allowed.

▶ Task 2: Configure the Service Endpoint's Binding to Flow Transaction

1. In the **BlueYonder.FrequentFlyerService.Host** project, open the **App.config** file, and add a new binding configuration for **netTcpBinding** with transaction flow.

- In the **<system.serviceModel>** element, add a new <bindings> element with the following configuration.

```
<bindings>
  <netTcpBinding>
    <binding name="TcpTransactionalBind" transactionFlow="true" />
  </netTcpBinding>
</bindings>
```

2. Apply the new binding configuration to the existing service endpoint

- Locate the **<endpoint>** element for the Frequent Flyer service.
- In the element, add the **bindingConfiguration** attribute, and set it to **TcpTransactionalBind**.

▶ Task 3: Add the Transaction Scope Attribute to the Service Implementation

1. In the BlueYonder.FrequentFlyerService.Implementation project, open the FrequentFlyerService.cs file, and add transaction scope setting to the AddFrequentFlyerMiles and RevokeFrequentFlyerMiles methods.

- Decorate the AddFrequentFlyerMiles and RevokeFrequentFlyerMiles methods with the [OperationBehavior] attribute.
- Set the parameters in each **[OperationBehavior]** attribute according to the following table.

Parameters	Value
TransactionAutoComplete	true
TransactionScopeRequired	true

▶ Task 4: Add Code to the WCF Booking Service that Calls the Frequent Flyer WCF Service

1. In the **BlueYonder.BookingService.Implementation** project, open the **BookingService.cs** file, and add a private field named **_frequentFlyerChannnelFactory** to store the channel factory for the **IFrequentFlyerService** service contract. Use the **FrequentFlyerEP** configuration name in the channel factory constructor.
2. In the **UpdateTrip**, check whether the traveler is checking in, and if this is the case, call the Frequent Flyer service to update their miles.

- To check if the traveler is checking in, verify the original and new status are different, and that the new status is **FlightStatus.CheckedIn**.
- Retrieve the earned miles from the originalTrip.FlightInfo.Flight.FrequentFlyerMiles property.

- Use the **_frequentFlyerChannnelFactory.CreateChannel** method to create a new proxy to the Frequent Flyer service.
- Call the Frequent Flyer service's **AddFrequentFlyerMiles** method, and pass it the traveler's ID and the earned miles.
- Call the service before saving the local changes to the database.

▶ Task 5: Execute the Service Call and the Reservations Database Updates in a Distributed Transaction

1. Add a reference to the **System.Transactions** assembly, and surround the service call and the database update with a transaction scope. Make sure that you set the **Complete** flag before leaving the scope. The resulting code segment should resemble the following code.

```
using (TransactionScope scope = new TransactionScope())
{
    // TODO: 2 - Call the Frequent Flyer service to add the miles if the traveler has
checked-in
    if (originalStatus != newStatus && newStatus == FlightStatus.CheckedIn)
    {
        IFrequentFlyerService proxy = _frequentFlyerChannnelFactory.CreateChannel();
        int earnedMiles = originalTrip.FlightInfo.Flight.FrequentFlyerMiles;
        proxy.AddFrequentFlyerMiles(reservation.TravelerId, earnedMiles);
    }
    // TODO: 3 - Wrap the save and the service call in a transaction scope
    reservationRepository.Save();
    scope.Complete();
}
```

▶ Task 6: Update the WCF Client Configuration with the Frequent Flyer Service Endpoint and the Support for Transaction Flow in the Bindings

1. In the **BlueYonder.BookingService.Host** project, open the **App.config** file, add a client endpoint for the Frequent Flyer service, and configure its binding to flow transactions. Name the client endpoint **FrequentFlyerEP** to match the name that you used in the Booking service implementation

 **Note:** You can use the service endpoint configuration from the Frequent Flyer service host to set the address, binding, and contract settings of the client endpoint. You can also copy the binding configuration from the Frequent Flyer service host configuration.

2. Make sure that the MSDTC service is running, and start both service hosts (Booking and Frequent Flyer) without debugging.

- To open the services list, on the Start screen, click the **Administrative Tools** tile, and in the **Administrative Tools** window, double-click **Services**.
- In the **Services** window, look for the **Distributed Transaction Coordinator** service and check its **Status** column. If the status of the service is not **Running**, right-click it, and then click **Start**.
- After you run both service hosts, wait until both console windows show the "service is running" message.

3. Start the WCF Test Client utility, add the two services, and verify the distributed transaction works by calling the **UpdateTrip**.

- Open the WCF Test Client utility by double-clicking the **WcfTestClient** shortcut from the **D:\AllFiles** folder.

- Add the services by using the http://localhost/BlueYonder/Booking and http://localhost/BlueYonder/FrequentFlyer service addresses http://localhost/BlueYonder/Booking.
- Test the **UpdateTrip** method by using the following request values.

<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>FlightDirection</td><td>Departing</td></tr>
<tr><td>ReservationConfirmationCode</td><td>Aa123</td></tr>
<tr><td>TripToUpdate</td><td>
<table>
<tr><th>Property</th><th>Value</th></tr>
<tr><td>Class</td><td>First</td></tr>
<tr><td>FlightScheduleID</td><td>1</td></tr>
<tr><td>Status</td><td>CheckedIn</td></tr>
</table>
</td></tr>
</table>

4. Verify the miles were added to the traveler by invoking the **GetAccumulatedMiles** operation with traveler ID **1**. Verify the returned miles are **5026**. Close the WCF Test Client utility and the two console windows when done.

Results: You can run the WCF Test Client utility, call an operation in the Booking service that starts a distributed transaction, and verify that the Frequent Flyer service indeed committed its transaction.

Question: Why did you log the error in the **HandleError** method of the error handler class and not in the **ProvideFault** method?

Module Review and Takeaways

In this module, you learned how to apply various design principles to your service contract and how to implement complex communication scenarios, such as duplex services. You also learned how to support distributed transactions with WCF services. In the last lesson of this module, you learned the most important parts of WCF: the WCF message handling pipeline. You learned how the pipeline is constructed, how a message flows through the dispatchers, and how you can customize the pipeline by creating custom runtime components. This module is the second of three modules that cover WCF. The next module will discuss the various ways by which you can secure your WCF services.

Review Question(s)

Question: When is it useful to use asynchronous operations on the service-side?

Question: What are dispatchers?

Tools

- Microsoft Service Configuration Editor, Microsoft Service Trace Viewer