CHAPTER2. WCF의 진보된 예제

ASP.NET 웹 서비스 개발 경험자에게 또 한가지 심각한 불만사항은 클라이언트 개발을 위해 서비스의 코드(서비스 어셈블리 DLL)를 "참조" 해야만 한다는 점과 클라이언트 프록시 코드를 개발자가 직접 코딩 해야 한다는 점이 불편하다.

WCF 는 하드 코딩을 최소한으로 하기 위해 다양한 설정을 configuration 파일에 명시할 수 있도록 준비하고 있다. 또한, 클라이언트가 서비스의 인터페이스를 참조하거나 서비스 구현으로부터 독립할 수 있도록 서비스의 프록시를 생성해 주는 다양한 방법을 제공한다.

01. Configuration 사용

이전 컬럼에서 사용했던 서비스 코드를 다시 한번 살펴보자. [리스트 1]의 코드는 매우 잘 작동한다. 하지만 이 코드가 단순한 예제가 아니라 실제 업무나 제품의 코드라면 심각한 문제를 가질 수도 있다.

리스트 1. Hello World 서비스의 구현 코드 및 호스트 코드

```
//----- Service Implementation Code -----
using System.ServiceModel;

namespace HelloWorldService
{
    // 서비스 Contract 선언
    [ServiceContract]
    public interface IHelloWorld
    {
        [OperationContract]
        string SayHello();
    }

    // 서비스 타입 구현
    public class HelloWorldWCFService : IHelloWorld
    {
        public string SayHello()
        {
            return "Hello WCF World !";
        }
    }
}

//----- Service Host Code -----
namespace HelloWorldHost
{
    class Program
    {
        static void Main(string[] args)
        {
            ServiceHost host = new ServiceHost(typeof(HelloWorldWCFService),
                new Uri("http://localhost/wcf/example/helloworldservice"));

            host.AddServiceEndpoint(
```

```
            typeof(IHelloWorld),    // service contract
            new BasicHttpBinding(),    // service binding
            "");

        host.Open();
        Console.WriteLine("Press Any key to stop the service");
        Console.ReadLine();
        host.Close();
    }
  }
}
```

[리스트 1] 코드의 문제점은 하드 코딩이 많이 들어 있다는 점이다.

가장 문제가 될 만한 것은 서비스의 주소이다. Hello World 서비스가 매우 유명해 져서 이 프로그램이 여러 회사나 개인에게 판매되는 제품이 되었다고 가정해 보자. 고객들은 이 서비스가 리스닝 하는 서비스의 주소가 /wcf/example/helloworldservice 인 것에 불만을 가질 가능성이 대단히 높다. 고객들은 이 **주소를 수정**하고 싶을 것이고 이를 위해서는 **이 서비스의 소스 코드가 있어야 할 것이다**. 물론 소스를 수정한 후에는 **코드를 다시 컴파일 해야 할 것이므로 소스 코드뿐만 아니라 컴파일에 필요한 Visual Studio 프로젝트 파일과 코드에 서명을 했다면 서명에 필요한 키 파일 등도 필요할 지도 모른다**.

위 코드의 또 한가지 문제점으로 지적할 수 있는 것으로 **서비스 종점을 추가하거나 바인딩 속성을 바꾸기 위해서는 "또" 코드를 수정해야 한다는 점이다.** 이전 컬럼의 코드 예제에서 NetTcpBinding 을 사용하는 종점을 추가하는 코드를 본 적이 있다. 이처럼 종점을 추가하기 위해서는 코드를 수정해야 한다면 서비스 개발자가 서비스의 수행 환경 및 배포 시나리오를 미리 예측하여 코드를 작성해야만 한다는 얘기와 같다. 서비스의 개발이 완료된 시점에서는 새로운 배포 시나리오를 적용하기 어렵게 되고 다시 개발자가 코드를 수정해 주어야만 한다. 또한 서비스가 배포되고 설치된 이후에 바인딩 설정을 바꾸고자 하는 경우도 문제다. 서비스가 배포된 직후에는 텍스트 기반의 메시지 인코딩이 아무런 문제를 일으키지 않다가 추후 다량의 데이터를 전송해야 할 일이 자주 생기면서 MTOM 인코딩 을 바꾸어야 할 지도 모른다. 이 **때 또 다시 코드를 수정하고 컴파일 해야 하며 배포 역시 다시 해야 할 것이다. 너무나도 당연하게 서비스의 주소가 바뀌거나 바인딩 속성이 변경되면 이 서비스를 호출하는 클라이언트 역시 수정되어야 할 것이다.**

만약 서비스와 클라이언트의 하드 코드를 줄이고 개발이 완료된 후에도 개발자가 아닌 시스템 관리자 혹은 운영자에 의해 서비스의 주소를 바꾸거나 바인딩을 추가 하거나 바인딩의 속성을 바꾸는 작업이 가능하다면 보다 유연한 웹 서비스 시스템을 구축할 수 있을 것이다**. 특히 서비스의 주소나 바인딩 설정을 개발자가 아닌 시스템 운영자나 관리자가 변경할 수 있다는 것은 개발자의 부담을 줄여줄 수 있으며 보다 유연한 시스템 운영환경을 제공해 줄 수 있다.**

WCF 는 바로 그러한 요구 사항을 최대한 만족시켜줄 수 있도록 어플리케이션 configuration(web.config 혹은 app.config)을 통해 서비스와 클라이언트의 다양한 설정을 수행할 수 있는 능력을 가지고 있다. WCF 와 더불어 어플리케이션 configuration 에 추가된 새로운 섹션은 <system.serviceModel> 섹션이다. 이 섹션의 주요 하위 요소(element)들은 다음과 같다.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <services>
      <...List of Services>
    </services>
    <client>
      <...List of Endpoints>
    </client>
    <bindings>
```

```
        <...List of Bindings>
      </bindings>
    </system.serviceModel>
</configuration>
```

<services> 요소는 서비스 호스트에서 사용하는 것으로써 서비스가 호스팅 하는 서비스들이 어떤 것이 있고 이 서비스의 종점들에 대한 목록을 설정하는데 사용되며 <client> 요소는 클라이언트가 접속하고자 하는 서비스의 종점의 목록을 설정하는데 사용된다. 또한 <bindings> 요소는 여러 바인딩의 속성을 설정하는데 사용된다. 이외에도 서비스나 클라이언트의 작동 방식을 설정하는 등의 요소들이 존재한다.

Configuration 을 이용한 서비스 설정

서비스 측에서 코드가 아닌 configuration 을 통해 서비스 호스트를 어떻게 구성하는지 살펴보도록 하자.

WCF 서비스가 어떤 계약을 사용하며, 주소는 어떤 것을 쓰는지 그리고 바인딩은 어떠한지를 나타내기 위해서는 <system.ServiceModel> 섹션의 <services> 요소에 <service> 요소를 추가하면 된다. HelloWorldHost 프로젝트에 app.config 파일을 추가하고 다음과 같이 설정 내용을 추가해 보자.

리스트 2. HelloWorld 서비스를 위한 간단한 configuration 설정

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
            <service name="HelloWorldService.HelloWorldWCFService">
                <endpoint
                    contract="HelloWorldService.IHelloWorld"
                    address="http://localhost/wcf/example/helloworldservice"
                    binding="basicHttpBinding"
                />
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

[리스트 2]는 HelloWorldService.HelloWorldWCFService 클래스를 서비스 타입으로 하는 서비스를 추가하는 설정으로써 이 서비스의 종점이 HelloWorldService.IHelloWorld 인터페이스를 계약으로서 사용하며 서비스 주소는 http://localhost/wcf/example/helloworldservice 를, 바인딩으로 BasicHttpBinding 을 사용하고 있음을 서비스 런타임에게 알리고 있다.

여기서 서비스의 이름, 즉 **<service> 요소의 name 속성이 나타내는 이름이 매우 중요하다. ServiceHost 클래스의 인스턴스가 생성될 때 매개변수로 주어진 서비스 타입과 configuration 에 명시된 여러 <service> 요소들 중 서비스 타입이 일치하는 설정을 읽어 들여 ServiceHost 객체를 초기화하기 때문이다. 따라서 name 속성은 반드시 설정하고자 하는 서비스 타입의 네임스페이스를 포함하는 전체 이름을 명시해야 한다.**

[리스트 1]에서 구현한 서비스 타입의 네임스페이스와 클래스 이름이 HelloWorldService.HelloWorldWCFService 임을 상기할 필요가 있다.

또한 [리스트 2]는 이 서비스의 종점에 대한 설정도 포함하고 있다. 서비스에 추가되는 서비스 종점이 사용하는 서비스 계약과

주소 그리고 바인딩이 모두 명시되어 있음에 주목하자. 주의할 점은 binding 속성에 사용된 바인딩의 이름이 **BasicHttpBinding 이 아닌 basicHttpBinding(소문자로 시작함에 유의)** 이라는 점이다. WCF 는 바인딩들에 대한 configuration 설정을 기본적으로 가지고 있으며 BasicHttpBinding 에 대한 설정 역시 미리 준비되어 있다. 이렇게 미리 준비된 바인딩의 카테고리의 이름이 basicHttpBinding 이며 소문자로 시작하는 이름을 선호하는 XML 의 특성 때문에 basicHttpBinding 이란 이름을 사용하는 것이다.

[리스트 2]는 코드를 통해 서비스를 설정하는 것과 동등한 작업을 수행한다. 즉, [리스트 2]는 다음과 동등한 설정을 수행한다고 보면 되겠다.

```
ServiceHost host = new ServiceHost(typeof(HelloWorldWCFService));

host.AddServiceEndpoint(
    typeof(IHelloWorld),
    new BasicHttpBinding(),
    "http://localhost/wcf/example/helloworldservice");
```

그렇다면 [리스트 2]와 같은 설정이 언제 WCF 에 의해 읽혀 들여지고 효과를 나타내게 되는 것일까? 앞서 언급한바 대로 **ServiceHost 클래스의 인스턴스가 생성되면 이 클래스는 configuration 파일의 <services> 요소의 하위 <service> 요소들 중에서 ServiceHost 클래스의 생성자에 주어진 서비스 타입을 찾는다**. 만약 ServieHost 객체가 이 타입을 <services> 내에서 찾게 되면 발견한 <service> 요소의 설정들을 이용하여 ServiceHost 를 초기화를 수행한다. 즉, 서비스에 대한 설정이 읽혀 들여지고 사용되는 시점은 ServiceHost 객체의 생성자가 호출되는 때인 것이다.

따라서 방금 보인 위 코드를 [리스트 2]와 함께 사용하면 InvalidOperationException 예외가 발생하게 된다. 이유는 간단하다. Configuration 에서 이미 종점이 선언되었지만 코드에 의해 동일한 서비스 주소를 사용하는 종점을 추가하려고 시도했기 때문이다. 이러한 오류를 피하기 위해서는 AddServiceEndpoint 메소드 호출을 코드 상에서 제거해야 한다.

이제 configuration 을 설정하는 방법과 적용하는 방법을 간단히 살펴보았으므로 이전 컬럼에서 2 개의 서비스 종점을 사용하는 예제와 동등한 설정을 configuration 과 코드를 통해 작성하는 예제를 살펴보도록 하자. 다음의 [리스트 3]은 이전 컬럼에서 예제로 보였던 서비스 호스트 코드의 일부이다.

리스트 3. 2 개의 서비스 종점을 사용하는 호스트 코드

```
static void Main(string[] args)
{
    ServiceHost host = new ServiceHost(typeof(HelloWorldWCFService),
        new Uri("http://localhost/wcf/example/helloworldservice"),
        new Uri("net.tcp://localhost/wcf/example/helloworldservice"));

    host.AddServiceEndpoint(
        typeof(IHelloWorld),    // service contract
        new BasicHttpBinding(),    // service binding
        "");            // relative address
    host.AddServiceEndpoint(
        typeof(IHelloWorld),    // service contract
        new NetTcpBinding(),    // service binding
        "");            // relative address

    host.Open();
    Console.WriteLine("Press Any key to stop the service");
```

```
    Console.ReadKey();
    host.Close();
}
```

[리스트 3]에서는 서비스 호스트에 베이스 주소를 사용했으며 BasicHttpBinding 과 NetTcpBinding 을 각각 사용하는 2 개의 서비스 종점을 사용하고 있다. 이러한 코드를 configuration 을 통해 설정하면 [리스트 4]와 같다.

리스트 4. 두 개의 서비스 종점을 사용하는 서비스의 configuration

```
<configuration>
    <system.serviceModel>
        <services>
            <service name="HelloWorldService.HelloWorldWCFService">
                <host>
                    <baseAddresses>
                        <add baseAddress="http://localhost/wcf/example/helloworldservice"/>
                        <add baseAddress="net.tcp://localhost/wcf/example/helloworldservice"/>
                    </baseAddresses>
                </host>
                <endpoint contract="HelloWorldService.IHelloWorld"
                    address=""
                    binding="basicHttpBinding"
                />
                <endpoint contract="HelloWorldService.IHelloWorld"
                    address=""
                    binding="netTcpBinding"
                />
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

[리스트 4]에서 관심을 가질 부분은 두 부분이다. 첫째로 <service> 요소에 새로이 등장한 요소로서 호스트의 베이스 주소를 명시하는 <baseAddresses> 요소와 서비스를 Open 하거나 Close 할 때의 타임아웃을 설정하는 <timeouts> 요소를 명시할 수 있다.

[리스트 3]에서 서비스의 베이스 주소를 2 개 명시했기 때문에 [리스트 4]에서도 2 개의 베이스 주소를 설정으로써 명시했다. 두 번째로 관심을 가질 부분은 <endpoint> 요소를 2 개 사용하여 서비스가 두 개의 서비스 종점을 사용하도록 설정한 부분이다. 코드를 사용할 때는 2 회의 AddServiceEndpoint 메쏘드 호출을 수행한 반면 configuration 을 사용할 경우에는 2 개의 <endpoint> 요소를 사용하면 된다.

이제 [리스트 4]의 configuration 을 사용하는 서비스 호스트 코드인 [리스트 5]를 살펴보도록 하자.

리스트 5. Configuration 을 사용하는 서비스 호스트 코드

```
ServiceHost host = new ServiceHost(typeof(HelloWorldWCFService));

host.Open();
Console.WriteLine("Press Any key to stop the service");
```

```
Console.ReadKey();
host.Close();
```

[리스트 4]의 설정을 사용하는 코드는 당황스러울 정도로 간단하다. 서비스들의 베이스 주소를 설정하거나 AddServiceEndpoint 메소드를 호출하여 서비스 종점을 추가하는 코드는 사라지고 달랑 ServieHost 클래스의 인스턴스를 생성하고 호스트에 대해 Open 메쏘드와 Close 메쏘드를 호출하는 것이 전부인 코드가 되어 버렸다.

[리스트 5]에서 **ServiceHost 객체가 생성되는 시점에서 WCF 런타임은 configuration 파일을 참조하게 되고 configuration 에서 서비스 타입인 HelloWorldWCFService 클래스의 이름을 찾게 된다(**네임스페이스를 포함하는 전체 이름을 찾는다). 물론 [리스트 4]에서 해당 서비스의 설정이 존재하므로 이 설정을 읽어 들여 필요한 베이스 주소 설정과 종점 추가 작업이 내부적으로 일어나게 된다. 이 때문에 추가적인 코드를 사용하여 종점을 설정할 필요가 없어진 것이다.

한 발만 더 나아가 보자. 앞서 바인딩을 설명할 때 바인딩의 속성을 설정할 수 있다고 했었다. 그리고 구체적인 예제로서 BasicHttpBinding 의 MessageEncoding 속성을 MTOM 으로 설정한 예제코드를 보였었다. 다음 코드처럼 말이다.

```
ServiceHost host = new ServiceHost(typeof(HelloWorldWCFService));
BasicHttpBinding binding = new BasicHttpBinding();
binding.MessageEncoding = WSMessageEncoding.Mtom;

host.AddServiceEndpoint(
    typeof(IHelloWorld),    // service contract
    binding,                // service binding
    "http://localhost/wcf/example/helloworldservice");
......
```

이와 동등하게 configuration 을 통해서도 바인딩의 속성을 설정할 수는 없을까? 왜 없겠는가? [리스트 4]에서 사용한 <endpoint>의 binding 속성은 사용할 바인딩의 종류만을 표시한 것일 뿐이다. 다시 말해 [리스트 4]에서 사용한 <endpoint> 요소를 코드로 이야기 하자면 바인딩 객체(이 경우 BasicHttpBinding 객체)를 생성하고 아무런 설정 없이 그대로 사용한 다음 코드와 같다는 말이다.

```
host.AddServiceEndpoint(
    typeof(IHelloWorld),
    new BasicHttpBinding(),
    "http://localhost/wcf/example/helloworldservice");
```

따라서 바인딩 속성은 바인딩 객체가 제공하는 기본값만을 사용하는 것이 된다.

Configuration 상에서 바인딩 객체의 속성을 설정하고자 하면 <bindings> 요소를 사용해야 한다. 이 요소는 각 바인딩 종류별로 바인딩에 대한 속성 설정을 지정하고 나열할 수 있으며 이렇게 나열된 바인딩 설정 중 하나를 <endpoint> 에서 참조하도록 할 수 있다. 다음은 구체적인 <bindings> 요소의 예제를 보여준다.

```
<system.serviceModel>
    <bindings>
        <basicHttpBinding>
            <binding name="MtomSetting" messageEncoding="Mtom" />
            <binding name="Others" ..... />
        </basicHttpBinding>
    </bindings>
    <services>
```

```
      ......
   </services>
</system.serviceModel>
```

<bindings> 요소 내에 <basicHttpBinding> 요소를 사용하여 BasicHttpBinding 에 대해 적용되는 바인딩 설정들을 별도로 모아두고 있음을 알 수 있을 것이며 <basicHttpBinding> 하위에 <binding> 요소를 여러 개 두고 각 바인딩 설정의 구분은 name 속성을 이용하고 있음에도 주목하기 바란다.

<bindings> 요소에 나열된 다양한 바인딩 속성 설정들은 바인딩 속성을 설정하는 템플릿들에 지나지 않는다. 이렇게 나열된 템플릿들 중에서 하나를 실제 바인딩 객체가 사용하도록 지시할 필요가 있다. 이 때 사용되는 것이 <endpoint> 요소의 bindingConfiguration 속성이다. bindingConfiguration 속성에 <bindings> 내에 나열된 바인딩 설정의 이름을 명시하면 해당 바인딩 설정에 나열된 다양한 속성(이 경우, messageEncoding 설정밖에 없지만)이 바인딩 객체에 적용된다. 적용되는 시점은 종점이 서비스 호스트에 추가되는 시점이며 이 시점은 ServiceHost 클래스의 인스턴스가 만들어지는 시점과 동일하다.

```
<endpoint contract="HelloWorldService.IHelloWorld"
   address=""
   binding="basicHttpBinding"
   bindingConfiguration="MtomSetting" />
```

완전한 configuration 파일의 모습은 [리스트 6]과 같다. 이 설정을 사용하면 Hello World 서비스는 MTOM 메시지 인코딩을 사용하는 BasicHttpBinding 기반의 종점과 NetTcpBinding 을 사용하는 종점을 갖는 서비스가 될 것이다.

리스트 6. Binding 설정을 수행하는 Configuration

```
<configuration>
   <system.serviceModel>
      <bindings>
         <basicHttpBinding>
            <binding name="MtomSetting" messageEncoding="Mtom" />
         </basicHttpBinding>
      </bindings>
      <services>
         <service name="HelloWorldService.HelloWorldWCFService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost/wcf/example/helloworldservice"/>
                  <add baseAddress="net.tcp://localhost/wcf/example/helloworldservice"/>
               </baseAddresses>
            </host>
            <endpoint contract="HelloWorldService.IHelloWorld"
               address=""
               binding="basicHttpBinding"
               bindingConfiguration="MtomSetting"
            />
            <endpoint contract="HelloWorldService.IHelloWorld"
               address=""
               binding="netTcpBinding"
            />
         </service>
```

```
        </services>
    </system.serviceModel>
</configuration>
```

이제 마음을 가라 앉히고 좀 정리를 해보자. [리스트 4]과 [리스트 5]와 같은 코드를 사용하면 앞서 언급했던 하드 코드 문제와 관리적인 편의를 올릴 수 있을까? 그렇다. 컴파일 된 코드 안에는 하드 코딩 된 부분이 없으며 configuration 파일 내에서 서비스의 베이스 주소를 수정한다던가 종점에서 상대 주소(relative address)를 수정하여 서비스 주소를 수정할 수 있을 것이다. 그 뿐인가? 필요에 따라 종점을 추가하는 것 역시 코드를 전혀 수정하지 않고 <endpoint> 요소만 추가하면 된다. 또한 바인딩에 적용할 다양한 속성이 바뀌었을 때 코드를 수정하기 보다는 configuration 에서 바인딩 설정을 변경하는 방법으로 코드에 대한 종속성을 없앨 수 있었다. 어떤가? 이젠 보다 유연한 설정이 가능한 Hello World 웹 서비스가 탄생하지 않았나?

02. WSDL과 서비스 메타 정보

지금까지 살펴본 Hello World 서비스의 구현은 적은 코드를 가지고 다양한 기능을 제공하는 웹 서비스와 클라이언트를 작성하게 해주는 것이었다.

하지만 여전히 서비스와 클라이언트의 구현에서 불만 사항이 남아 있다. 특히, 클라이언트 측의 구현 사항에 대해 심각한 불만 사항이 존재하는데 그것은 다름 아닌 서비스 코드를 클라이언트가 직접 "참조" 해야만 한다는 점이다. 즉, ChannelFactory 와 프록시를 생성하기 위해서는 반드시 IHelloWorld 인터페이스를 클라이언트 코드가 참조해야 하지만 이 인터페이스의 선언은 서비스 구현 코드, 즉 HelloWorldService 프로젝트 내에 존재한다는 것이다. 그리고 이 프로젝트에는 IHelloWorld 인터페이스뿐만 아니라 서비스의 실제 구현을 맡고 있는 HelloWorldWCFService 클래스의 구현까지도 포함되어 있지 않은가?

클라이언트를 구현하기 위해 서비스의 계약 인터페이스에 대한 정보가 필요하고 이를 위해서 서비스의 구현을 담는 어셈블리를 클라이언트에 배포해야 한다면 내부 구현을 클라이언트에게 알리지 않는다는 서비스 지향 아키텍처(SOA; Service Oriented Architecture)에 위반될 뿐 더러, 자칫 잘못하면 서비스 구현 기술에 대한 중요한 기밀이 외부에 새어 나갈 수도 있다는 말이 되겠다. 간단한(?) 해결책으로 HelloWorldService 프로젝트를 다시 두 개의 프로젝트로 나누고 한 프로젝트에는 IHelloWorld 인터페이스에 대한 선언만을, 또 한 프로젝트에는 IHelloWorld 인터페이스의 구현인 HelloWorldWCFService 클래스를 넣어두는 것을 생각해 볼 수 있겠다. 물론 클라이언트에게는 인터페이스 만이 포함된 어셈블리를 배포해야 할 것이다.

비록 인터페이스만이 포함된 어셈블리만을 배포하는 방식으로 구현을 숨기는 방법을 사용한다 하더라도 문제는 여전히 남아 있다. WCF 클라이언트에게는 그런 방식으로 클라이언트 구현이 가능하도록 한다 할지라도 WCF 가 아닌 일반 닷넷 클라이언트나 닷넷이 아닌 ASP, JAVA, C/C++ 클라이언트에게도 닷넷 어셈블리를 배포할 수 있을까? 그들에게 닷넷 어셈블리는 아무짝에도 쓸모 없는 바이너리 코드 덩어리일 뿐이다. 따라서 서비스는 자신의 인터페이스 계약을 클라이언트에게 알려줄 다른 방법이 필요하며, 이 방법은 서비스의 플랫폼이나 구현 기술, 프로그래밍 언어에 무관해야 할 것이고 절대 다수의 클라이언트들이 이 방법을 이해하고 서비스의 인터페이스 계약이 어떠한 것인지 알아 들을 수 있는 표준적인 방법이어야만 한다.

WSDL(Web Service Description Language)이 바로 서비스의 계약을 표준적인 방법으로 클라이언트에게 알리기 위한 웹 서비스 표준이다. WSDL 은 웹 서비스가 어떤 메소드를 가지고 있고 이 메소드의 매개변수와 반환 값이 무엇인가를 XML 을 통해 알려줄 뿐만 아니라 웹 서비스에 접근하는 방법으로 HTTP GET, HTTP POST, SOAP 등 중에서 어느 것이 지원되는가도 알려준다. 클라이언트는 서비스의 WSDL 로 부터 서비스가 어떤 메소드를 제공하는지 매개변수의 타입은 어떠한지 알아낼 수 있으므로 서비스 호출에 필요한 모든 정보들을 알아낼 수 있는 것이다. [리스트 12]는 Hello World 서비스에 대한 WSDL 의 일부를 보여주고 있다.

리스트 12. Hello World 서비스에 대한 WSDL 의 일부

```
<?xml version="1.0" encoding="utf-8" ?>
<wsdl:definitions targetNamespace="http://www.simpleisbest.net/wcf/helloworld" ...>
    <wsdl:types>
        <xsd:schema targetNamespace="...">
        ......
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="IHelloWorld_SayHello_InputMessage">
        <wsdl:part name="parameters" element="tns:SayHello" />
    </wsdl:message>
    <wsdl:message name="IHelloWorld_SayHello_OutputMessage">
        <wsdl:part name="parameters" element="tns:SayHelloResponse" />
    </wsdl:message>
    <wsdl:portType name="IHelloWorld">
        <wsdl:operation name="SayHello">
            <wsdl:input wsaw:Action="..."
```

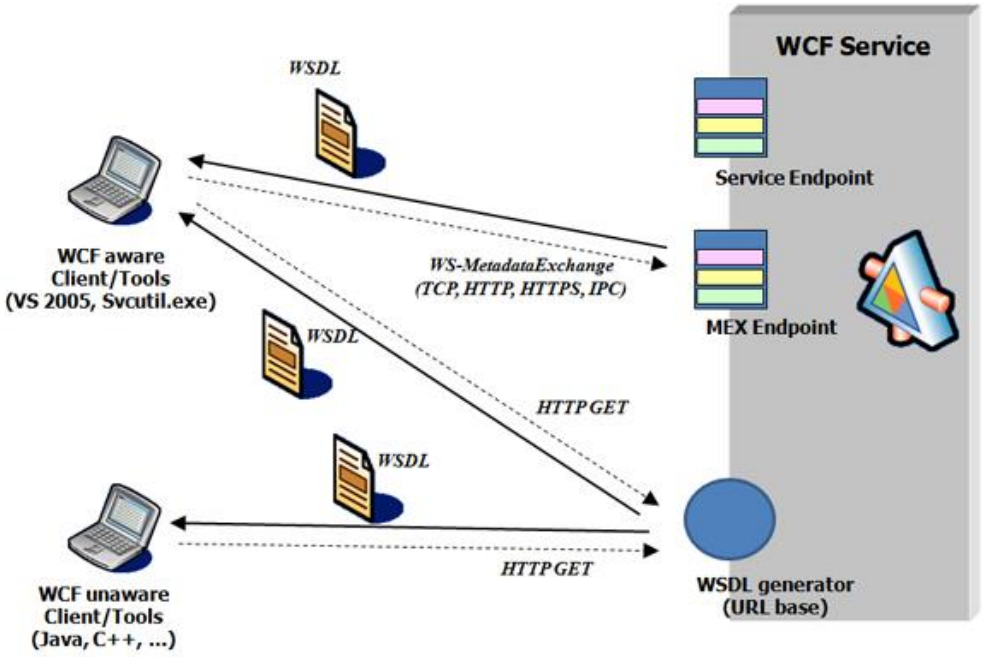
```
                message="tns:IHelloWorld_SayHello_InputMessage" />
        <wsdl:output wsaw:Action="..."
                message="tns:IHelloWorld_SayHello_OutputMessage" />
      </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

[리스트 12]는 Hello World 서비스의 계약인 IHelloWorld 인터페이스가 SayHello 메소드를 가지고 있고 이 메소드를 호출하기 위한 SOAP 메시지의 스키마와 SayHello 메소드가 반환하는 SOAP 메시지의 스키마에 대한 정보를 포함한다. 이들 스키마 정보들로부터 SayHello 메소드의 매개변수가 무엇인지 그리고 반환 값의 타입이 무엇인지 알아낼 수 있으며 이로부터 다시 서비스의 메소드인 SayHello 에 대한 닷넷 메소드 프로토타입(prototype; 함수의 이름과 매개변수 그리고 반환 타입을 함수 프로토타입이라 한다)을 생성해 낼 수 있는 것이다.

웹 서비스를 처음 접하는 개발자라면 [리스트 12]의 XML 을 이해하려 들지 않는 것이 좋다. 비록 XML 이란 것이 사람이 읽기 쉬운 텍스트 기반의 마크업(Markup) 언어라고 할지라도 WSDL 정도 되면 복잡한 참조와 XML 네임스페이스 때문에 가독성(readability)이 크게 떨어질뿐더러 사람이 이해하기란 매우 어려워진다. 또한 WSDL 자체의 스키마 정의도 잘 알아야만 [리스트 12]를 이해할 수 있으므로 이제 막 WCF 를 배우기 시작하는 독자들에겐 당연히 이해가 안될 것이다. 사실 WSDL 을 잘 이해하고 있으면 분명 도움이 되지만 필수 조건은 아니다. 개발자는 WSDL 이 무엇이며 어떤 역할을 하는지를 잘 이해하고 WSDL 을 통해 무엇을 할 수 있는지 정도만 잘 이해 하도록 하자.

클라이언트가 서비스를 성공적으로 호출하기 위해서는 서비스가 어떤 메소드를 가지고 있는지 그리고 그 메소드의 매개변수, 반환 값은 어떤지에 대한 인터페이스 정보가 있어야 한다. 뿐만 아니라 서비스가 사용하는 트랜스포트, 메시지 인코딩과 같은 바인딩 속성에 대한 정보도 클라이언트에게 알려주어야 한다. 이처럼 서비스 호출을 위한 다양한 정보들을 서비스 메타 데이터(Metadata)라고 하고 **서비스에 대한 메타데이터는 WSDL 을 통해 다양한 플랫폼을 가진 클라이언트에게 알려줄 수 있다.** 또한 WCF 는 서비스의 메타 데이터를 제공하기 위해 **서비스에 미리 정의되어 있는 종점(endpoint)를 추가하여 메타데이터를 제공하는 방법도 제공한다**. 전통적인 HTTP GET 을 통해 WSDL 을 제공하는 방법과 WCF 에서 미리 구현되어 있는 MEX(Metadata Exchange) 종점을 추가하는 방법 중 하나를 선택하거나 두 기능을 모두 제공할 수 있다. MEX 종점에 대해서는 추후에 메타 데이터를 상세히 다루는 6 장에서 다시 살펴보도록 하자.

그림 1. WSDL 과 MEX 를 이용한 서비스 메타 데이터 게시



WCF 에서 WSDL 을 제공하기 위해서는 약간의 작업을 요구한다. 첫 번째 방법은 오프라인으로 WSDL 을 생성하는 방법으로써

Windows SDK 에 포함된 svcutil.exe 이란 유틸리티를 사용하면 된다. 이 유틸리티에 서비스 호스트에 대한 어셈블리 혹은 서비스 구현에 대한 어셈블리를 매개변수로 다음과 같이 명시하면 WSDL 파일(.wsdl)과 관련 스키마(XSD) 파일(.xsd)을 생성해 준다.

svutil.exe HelloWorldService.dll

위와 같이 생성한 .wsdl 파일과 .xsd 파일을 클라이언트에게 배포하거나 코드를 생성하는데 사용할 수 있다. 구체적으로 이들 파일을 사용하여 클라이언트 코드를 생성하는 방법은 조금 후에 살펴보기로 하자.

이렇게 WSDL 을 생성하는데 유틸리티까지 써가면서 생성할 필요가 있을까? ASP.NET 웹 서비스처럼 기본적으로 WCF 런타임이 온라인 상에서 WSDL 을 제공해주는 기능은 없을까? 물론 있다. 하지만 보안상의 이유로 서비스에 대한 정보 공개를 최소화하기 위해 WSDL 자동 생성은 기본적으로 해제(disable) 되어 있다.

온라인 상에서 HTTP GET 을 통해 WSDL 을 읽어 갈 수 있도록 하기 위해서는 다음과 같은 configuration 설정을 해 주어야만 한다.

리스트 13. HTTP GET 을 통해 WSDL 생성을 위한 configuration 설정

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <bindings>
        ......
        </bindings>
        <services>
            <service name="HelloWorldService.HelloWorldWCFService"
                    behaviorConfiguration="HelloWorldService">
                    ......
            </service>
        </services>
        <behaviors>
            <serviceBehaviors>
                <behavior name="HelloWorldService">
                    <serviceMetadata httpGetEnabled="true"/>
                </behavior>
            </serviceBehaviors>
        </behaviors>
    </system.serviceModel>
</configuration>
```

여기에서 새로이 등장하는 개념이 서비스 행동(service behavior)이란 것이다. **WCF 에서 Behavior 란 것은 클라이언트와 서비스가 동의한 인터페이스와 무관하게 서비스가 내부적으로 사용하는 작동 방식을 지칭한다**. 예를 들어 어떤 웹 서비스가 여러 클라이언트의 동시 호출을 처리하기 위해 내부적으로 쓰레드 풀을 쓴다고 가정해 보자. 이 때 클라이언트는 서비스가 내부적으로 쓰레드 풀을 쓰는지 단일 쓰레드를 쓰는 지는 관심사 밖이며, 또한 서비스 계약 인터페이스에 이러한 내부 구현 여부가 명시되지도 않는다. 사실 우리가 Google 이나 E-bay 에서 제공하는 웹 서비스를 사용할 때 그 내부 구현이 C++인지 PHP 인지 알게 무엇인가? 우리가 알아야 할 것은 이 웹 서비스가 제공하는 메소드와 매개변수가 무엇이고 그 메소드의 수행결과가 무엇인가, 즉 서비스 인터페이스가 중요한 것이다.

Behavior 는 WCF 의 서비스 혹은 서비스의 메소드의 내부 구현에 관계된 사항을 제어할 때 사용되는 용어이며, 특정 서비스가 내부 작동 방식에 대한 설정이 필요한 경우 configuration 을 사용하거나 소스 코드 내에 ServiceBehavior 와 같은 특성(attribute)를 사용하여 서비스의 행동 방식을 결정할 수 있다. WCF 는 쓰레드 관리, 인스턴스 관리, 디버깅 등 다양한 서비스의 내부 구현에

관련된 다양한 Behavior 들을 제공하고 있으며, 서비스의 각 메소드 별로 트랜잭션 사용 여부 등을 결정하는데 적용되는 OperationBehavior 역시 제공하고 있다. 이들 Behavior 들에 대해서는 이 책의 전반에 걸쳐 지속적으로 등장하게 될 것이며 13 장에서 상세하게 다시 언급될 것이다.

이 장에서 처음 등장한 서비스 Behavior 는 서비스가 자동으로 WSDL 을 제공할 것인가를 결정하는 ServiceMetadataBehavior 클래스이다. 이 Behavior 가 서비스에 지정되면 WCF 런타임은 서비스를 위한 WSDL 을 런타임에 자동을 생성해 줄 수 있다. 이 Behavior 를 사용하는 것만으로 자동으로 생성된 WSDL 을 클라이언트에게 제공할 수는 없다. **이 Behavior 클래스의 HttpGetEnabled 속성을 true 로 설정해야만 HTTP GET 방식으로 WSDL 을 제공할 수 있게 된다**. 다시 이해하기 쉽게 말하자면 웹 브라우저에서 WSDL 의 XML 을 눈으로 확인하고자 하면(HTTP GET 방식), HttpGetEnabled 속성이 true 이어야 한다는 말이 되겠다. 웹 브라우저 외에도 곧 설명할 서비스 프록시 생성기(generator)를 사용하기 위해서도 이 속성의 값은 true 로 설정되어야 한다. 왜냐면 프록시 생성기가 WSDL 을 알아내기 위해 HTTP GET 방식으로 WSDL 을 조회하기 때문이다.

[리스트 13]과 같이 configuration 을 사용하여 HttpGetEnabled 속성을 바꾸고자 한다면 먼저 <behavior> 요소를 추가해 넣어야 한다. 그리고 <serviceMetadata> 요소의 httpGetEnabled 속성의 값을 true 로 설정해 주면 된다. 그리고 서비스가 이 Behavior 의 적용을 받기 위해서는 서비스를 선언하는 <service> 요소에 behaviorConfiguration 속성을 추가하고 적용하고자 하는 Behavior 의 이름을 명시하면 된다. 간단하지 않은가?

WCF 에서는 configuration 으로 수행하는 설정이라면 모두 코드로도 표현이 가능하다. ServiceHost 클래스의 Description 속성으로 제공되는 ServiceDescription 객체는 서비스가 가진 Behavior 들에 대한 컬렉션을 제공한다. 이 컬렉션은 Behaviors 속성으로 제공되며 이 속성은 주어진 Behavior 에 대한 타입을 근거로 객체을 찾아주는 Find 메쏘드와 새로운 Behavior 를 추가할 수 있는 Add 메쏘드를 가지고 있다. 따라서 코드를 통해 적용할 Behavior 객체를 만들고 이를 ServiceHost 에 추가하면 해당 Behavior 를 적용할 수 있는 것이다.

```
public class ServiceHost : ServiceHostBase
{
    public ServiceDescription Description { get; }
    ......
}

public class ServiceDescription
{
    public KeyedByTypeCollection<IServiceBehavior> Behaviors { get; }
    ......
}

public class KeyedByTypeCollection<TItem> : KeyedCollection<Type, TItem>
{
    public T Find<T>();
    ......
}
```

[리스트 14]는 코드를 통해 Behavior 를 추가하거나 기존 설정을 변경하는 전형적인 예제를 보여주고 있다. [리스트 14]에서 관심을 가질 부분은 ServiceMetadataBehavior 객체가 Behaviors 컬렉션에 이미 존재하는가 살펴보는 코드이다. Configuration 의 <serviceMetadata> 요소에 의해 ServiceMetadataBehavior 가 이미 설정되었다면 이 객체는 Behaviors 컬렉션 내에 이미 존재할 것이고 이미 존재하는 Behavior 에 대해 동일한 Behavior 가 추가되면 오류가 발생하기 때문이다. 의도하지는 않았지만 [리스트 14]의 코드는 ServiceBehavior 들에 대한 configuration 의 설정을 오버라이드 하는 예제도 된다. Configuration 에서 HttpGetEnabled 속성을 false 로 지정하더라도 [리스트 14]의 코드에 의해 HttpGetEnabled 속성은 true 로 오버라이드 되게 된다. 앞서 configuration 오버라이드에 대한 설명은 충분히 했으므로 더 이상 반복하진 않겠다.

리스트 14. WSDL 에 대한 HTTP GET 을 enable 시키는 코드

```
// 다음 코드에서 configuration 을 읽어 ServiceHost 를 초기화 한다.
ServiceHost host = new ServiceHost(typeof(HelloWorldWCFService));

// ServiceMetadataBehavior 설정
ServiceMetadataBehavior behavior =
    host.Description.Behaviors.Find<ServiceMetadataBehavior>();
if (behavior == null) {
    behavior = new ServiceMetadataBehavior();
    host.Description.Behaviors.Add(behavior);
}
behavior.HttpGetEnabled = true;

.....
```

HTTP GET 에 대한 설정이 끝났으면 이제 서비스는 클라이언트 요구에 의해 WSDL 을 제공할 수 있게 되었다. HTTP GET 방식에 의한 WSDL 은 서비스의 주소 뒤에 ?wsdl 이란 쿼리 문자열(query string)을 추가함으로써 얻을 수 있다. 우리의 간단한(?) Hello World 서비스에 대한 WSDL 주소는 다음과 같다.

http://localhost/wcf/example/helloworldservice?wsdl

이 URL 을 웹 브라우저에 직접 입력하면 현란한 XML 덩어리를 볼 수 있을 것이며 이 XML 은 다음에 설명할 프록시 자동 생성에 사용될 수 있는 WSDL 이다.

이렇게 자동적으로 서비스에 대한 메타 데이터를 생성하여 제공하면 되었지 왜 WCF 는 svctuil.exe 를 사용하여 .wsdl 파일과 .xsd 파일들을 생성하는 기능을 만들어 놓았을까 생각해 보자. 세상은 넓고 다양한 상황이 존재할 수 있다는 점을 잊지 말자. 인터넷 상의 웹 서비스는 불특정 다수의 클라이언트에게 서비스를 제공하는가 하면 특정 몇몇 클라이언트에게만 서비스를 제공하는 경우도 있다. **불특정 클라이언트에게 서비스를 제공한다 함은 클라이언트가 누가 될지 모른다는 얘기가 되므로 서비스에 대한 메타 데이터를 제공하기 위해서는 HTTP GET 이나 MEX(Metadata Exchange)를 통해 클라이언트가 능동적으로 서비스 메타 데이터를 조회하도록 하는 것이 좋을 것이다**.

반대로 특정 몇몇 클라이언트에게만 서비스를 제공할 때는 아무나 서비스의 메타 데이터를 조회해 가는 것은 여러모로 찜찜하다. 비록 인증 등의 방법으로 아무나 서비스를 호출하지 못하도록 스스로를 보호하겠지만 악의적인 사람이 WSDL 을 통해 서비스의 트랜스포트, 주소, 네트워크 포트와 같은 정보를 알아낼 수 있다. 또한 서비스의 메소드들에 대한 인터페이스 역시 공개되므로, HTTP GET 과 같이 공개적인 방법으로 메타 데이터를 제공하는 것은 "해킹하는데 필요한 정보를 무료로 가져가세요"라고 하는 것과 같은 꼴이 된다. 바로 이러한 이유에서 특정 클라이언트들에게만 WSDL 파일을 SDK 란 형식을 빌어 제공하거나 기타 다른 방법으로 WSDL 파일을 제공하는 방법을 사용하는 것이 좋을 수 있다. **이 때 svcutil.exe 유틸리티를 이용하여 WSDL 파일을 생성하고 생성된 WSDL 을 한정된 클라이언트에게만 제공할 수 있겠다**. 클라이언트가 WCF 를 사용한다면 아예 다음에 설명할 프록시 생성 방법을 이용하여 프록시를 생성하여 DLL 을 제공하는 것 역시 좋은 접근 방법이라 할 수 있겠다.

03. 프록시 자동 생성

클라이언트가 서비스에 대한 메타 데이터를 알아낼 수 만 있다면 서비스 프록시를 만드는데 필요한 서비스 인터페이스를 생성할 수 있다. 왜냐면 서비스 메타 데이터에는 프록시가 요구하는 서비스의 계약, 바인딩, 주소 정보가 모두 포함되어 있기 때문이다. Hello World 서비스를 예로 들어 설명하자면, 서비스 호출에 필요한 것은 IHelloWorld 인터페이스이다. 이를 위해 우리는 서비스의 구현이 포함되었던 HelloWorldService 어셈블리를 클라이언트에서 "참조" 했었다. 이러한 서버 측 구현을 참조하는 것은 서비스 지향적인 관점에 어긋날 뿐만 아니라 닷넷이 아닌 클라이언트를 사용할 수 없다는 단점이 있다는 것은 이미 지적한 바와 같다. 그렇다면 클라이언트가 IHelloWorld 인터페이스에 대해 이미 알고 있고 이를 클라이언트 코드 내에 선언한다면 어떻게 될까?

[리스트 7]의 클라이언트 코드에서 서비스의 구현에 대한 참조를 제거하는 작업을 직접 시도해 보자. 먼저 예제로 작성한 HelloWorldClient 프로젝트에서 HelloWorldService 참조를 제거한다. HelloWorldSerivce 프로젝트에 대한 참조가 제거 되었으므로 클라이언트 코드는 더 이상 IHelloWorld 인터페이스에 대해 알지 못한다. 당연히 컴파일 오류가 발생할 것이다. 따라서 IHelloWorld 인터페이스를 클라이언트 코드 내에 [리스트 15]과 같이 선언해 주도록 하자.

리스트 15. 서비스 구현에 대한 참조 없이 작성한 클라이언트 코드

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;

namespace HelloWorldClient2
{
    // 서비스 Contract 선언
   [ServiceContract(Namespace = "http://www.simpleisbest.net/wcf/helloworld")]
    public interface IHelloWorld
    {
        [OperationContract]
        string SayHello();
    }

    class Program
    {
        static void Main(string[] args)
        {
            Uri uri =
                new Uri("http://localhost/wcf/example/helloworldservice");
            ServiceEndpoint ep = new ServiceEndpoint(
                ContractDescription.GetContract(typeof(IHelloWorld)),
                new BasicHttpBinding(),
                new EndpointAddress(uri));

            ChannelFactory<IHelloWorld> factory =
                new ChannelFactory<IHelloWorld>(ep);
            IHelloWorld proxy = factory.CreateChannel();
            string result = proxy.SayHello();
            (proxy as IDisposable).Dispose();

            Console.WriteLine(result);
        }
```

[리스트 7]과 [리스트 15]가 다른 점은 HelloWorldService 어셈블리에 대한 참조가 사라졌기 때문에 네임스페이스를 참조한 using 문장이 사라진 것과 IHelloWorld 인터페이스에 대한 선언이 추가된 것일 뿐이다. IHelloWorld 인터페이스는 클라이언트 측의 WCF 런타임에서 사용할 것이므로 ServiceContract 특성과 OperationContract 특성이 서비스 측의 구현에서 선언된 것과 동일하게 추가되어 있음에 주목하기 바란다. [리스트 15]은 훌륭하게 잘 컴파일 될 뿐만 아니라 Hello World 서비스를 호출하는데 아무런 문제도 유발하지 않고 매우 잘 작동한다.

웹 서비스는 서비스에 대한 계약 인터페이스만 동일하다면 그 선언 및 구현이 닷넷 이건 아니건 닷넷의 어느 어셈블리에 존재하건 상관하지 않는다는 점이 핵심 요소가 되겠다. WCF 서비스와 WCF 클라이언트가 동일한 "인터페이스"를 바라보고 있기 때문에 클라이언트와 서비스가 SOAP 메시지를 주고 받는데 아무런 문제도 유발하지 않는다. 매우, 대단히, 정말로 중요한 개념이므로 잘 이해하고 넘어가도록 하자.

이제 남은 것은 서비스 계약인 IHelloWorld 인터페이스를 [리스트 15]과 같이 개발자가 직접 때려 넣어 주어야 하는 것인가 만이 숙제로 남았다. 웹 서비스는 WSDL 을 제공하고 WSDL 로부터 서비스 인터페이스를 알아낼 수 있다고 했으므로 **WSDL 로부터 IHelloWorld 인터페이스를 생성해 낼 수 있지 않을까**? 그렇다. 바로 그렇게 하면 되는 것이다. 이미 ASP.NET 웹 서비스를 사용해 본 독자라면 너무나도 당연하게 느끼는 웹 서비스 프록시 생성이란 것을 그대로 WCF 에도 적용할 수 있는 것이다. 하지만 WCF 는 ASP.NET 웹 서비스에 비해 훨씬 더 다양하고 유연한 방법들을 제공한다. 먼저 svcutil.exe 유틸리티를 이용하여 프록시를 생성하는 방법부터 살펴보자.

Svcutil.exe 이용

Svcutil.exe 유틸리티는 WSDL 을 생성하는 것보다는 WSDL 로부터 클라이언트 측의 프록시 코드를 생성하는데 더 많이 사용된다. 사용법은 그다지 어렵지 않다. 커맨드 라인 유틸리티라면 복잡한 옵션 때문에 어렵게만 생각하는 독자들이 많이 있겠지만 커맨드 라인 유틸리티를 잘 다루면 정말 '고수'란 소리를 들을 수 있을 뿐만 아니라 커맨드 라인 유틸리티만이 가능한 기능들도 있으므로 가급적 익숙해지도록 노력하는 것이 좋다. Svcutil.exe 을 사용하여 프록시를 생성하는 방법은 WSDL 을 svcutil.exe 입력으로 주고 출력으로 프록시 코드를 생성하도록 하면 된다. 이 때 WSDL 은 svcutil.exe 로 생성한 WSDL 파일이거나 WSDL 에 대한 URL 이어도 상관 없다. 앞서 WSDL 관련 파일들을 다음과 커맨드를 이용하여 생성했다면,

```
svcutil.exe HelloWorldService.dll
```

하나의 .wsdl 파일과 두 개의 .xsd 파일이 생성되었을 것이다. 이렇게 생성된 WSDL 관련 파일들을 다음과 같이 svcutil.exe 에 입력으로 제공하면 프록시를 위한 코드가 자동으로 생성된다.

```
svcutil.exe *.wsdl *.xsd /language:C# /out:Proxy.cs
```

위 명령은 폴더 내의 .wsdl 파일과 .xsd 파일을 입력으로 하여 C# 언어의 프록시 코드를 생성하되 코드의 파일명은 Proxy.cs 가 되도록 하는 것이다. 와일드 카드(*)를 사용하지 않고 구체적으로 파일명을 지정해 주어도 되며, /language 옵션 값에 C# 대신 VB 값을 주어 VB.NET 코드를 생성할 수도 있다.

만약 WSDL 파일 없이 WSDL 을 다운로드 받을 URL 을 알고 있다면 해당 URL 을 명시할 수 도 있다. 우리의 Hello World 서비스에 대한 WSDL 을 사용하여 프록시 코드를 생성하는 명령은 다음과 같다.

```
svcutil.exe http://localhost/wcf/example/helloworldservice?wsdl /language:C# /out:Proxy.cs
```

WSDL 에 대한 URL 을 사용하는 경우, WSDL 을 svcutil.exe 가 다운로드 받을 수 있어야 한다는 점에 주의하자. 다시 말해 WSDL 을 제공하는 서비스가 구동 중이어야 하며 WSDL 에 대한 HTTP GET 을 허용(HttpGetEnabled 속성)해야 한다는 것이다. WCF 서비스가 수행 중이지 않다면 svcutil.exe 는 WSDL 을 제공 받을 수 없을 것이며 당연히 오류를 유발할 것이다. 위 명령을 수행하기 위해서는 우리의 HelloWorldHost 프로젝트를 미리 수행시켜 WSDL 을 제공할 수 있도록 하자.

프록시 코드 탐험

Svcutil.exe 유틸리티를 통해 생성된 결과는 Proxy.cs 소스 코드와 configuration 설정을 담는 output.config 파일이다. Proxy.cs 파일은 직접 클라이언트 프로젝트 내에 포함시키면 되고, ouput.config 파일의 내용을 복사하여 app.config 에 적용하거나, app.config 파일이 이미 존재하지 않는다면 ouput.config 파일의 이름을 app.config 로 변경만 해도 된다. 파일 이름을 변경하여 app.config 파일을 생성한 경우 이 app.config 파일도 프로젝트 내에 포함시켜야 한다.

[리스트 16]에 표시한 자동 생성된 Proxy.cs 코드를 살펴보자. 생성된 프록시 코드에서 가장 먼저 눈에 띄는 것은 IHelloWorld 인터페이스에 대한 선언이다. 앞서 우리가 손으로 직접 만들어 넣은 IHelloWorld 인터페이스와 크게 다를 것 없이 ServiceContract 특성이 붙어 있으며 SayHello 메쏘드에도 OperationContract 특성이 붙어 있음을 알 수 있을 것이다. 수작업으로 작성한 인터페이스와 다른 부분은 ServiceContract 과 OperationContract 특성에 추가적으로 붙은 ConfigurationName, Action, ReplyAction 속성들인데 이들은 디폴트 값을 사용하므로 명시하지 않아도 되는 것들이다. 이들에 대해서는 4 장에서 서비스 계약에 대해 상세히 다룰 때 다시 설명하기로 하겠다. 중요한 점은 메타 데이터에서 서비스의 계약 인터페이스를 "생성"해 냈다는 점이다.

리스트 16. 자동 생성된 프록시 코드

```
[ServiceContract (Namespace="...", ConfigurationName="IHelloWorld")]
public interface IHelloWorld
{
    [OperationContractAttribute(Action="...", ReplyAction="...")]
    string SayHello();
}

......

public partial class HelloWorldClient : ClientBase<IHelloWorld>, IHelloWorld
{

    public HelloWorldClient()
    {
    }

    public HelloWorldClient(string endpointConfigurationName) :
            base(endpointConfigurationName)
    {
    }

    public HelloWorldClient(string endpointConfigurationName,
                            string remoteAddress) :
            base(endpointConfigurationName, remoteAddress)
    {
    }

    public HelloWorldClient(string endpointConfigurationName,
                            EndpointAddress remoteAddress) :
            base(endpointConfigurationName, remoteAddress)
    {
    }

    public HelloWorldClient(Binding binding, EndpointAddress remoteAddress) :
            base(binding, remoteAddress)
    {
    }

    public string SayHello()
    {
        return base.Channel.SayHello();
    }
}
```

그 다음에 주의 깊게 살펴보아야 할 것은 HelloWorldClient 란 클래스이다. 이 클래스는 ClientBase<T> 클래스에서 파생된 클래스로서 채널 팩토리를 직접 사용하는 프록시에 대한 래퍼(wrapper) 클래스로서 보다 편리하게 서비스를 호출하도록 해준다. HelloWorldClient 라는 클래스 이름은 서비스의 계약인 IHelloWorld 에서 자동으로 인터페이스 접두사(prefix) 인 I 를 제거하고 Client 접미사(postfix)를 붙인 것이다.

```
public abstract class ClientBase :
                    ICommunicationObject, IDisposable where TChannel : class
{
```

```
    protected ClientBase();
    protected ClientBase(string endpointConfigurationName);
    protected ClientBase(Binding binding, EndpointAddress remoteAddress);
    protected ClientBase(string endpointConfigurationName,
                         EndpointAddress remoteAddress);
    protected ClientBase(string endpointConfigurationName,
                         string remoteAddress);

    protected TChannel Channel { get; }
    ......
}
```

ClientBase<T> 클래스는 [리스트 7]이나 [리스트 9]와 같이 ChannelFactory<T> 클래스를 직접 사용하여 프록시를 생성하는 코드를 내부적으로 이미 가지고 있기 때문에 더 이상 이 클래스의 인스턴스를 생성하거나 CreateChannel 메쏘드를 호출할 필요가 없다. 예를 들어, configuration 의 서비스 종점 설정을 매개변수로 취하는 ClientBase<T>의 생성자(constructor)의 내부 코드는 다음과 같다.

인터페이스의 이름에는 이것이 인터페이스임을 밝히기 위해 I 접두사를 붙이는 것이 닷넷의 프로그래밍 관습이다.

```
protected ClientBase(string endpointConfigurationName)
{
    this.channel = null;
    if (endpointConfigurationName == null)
    {
        // Error Handling......
    }
    this.channelFactory =
        new ChannelFactory(endpointConfigurationName);
    ......
}

protected TChannel Channel
{
    get
    {
        if (this.channel == null) {
            // thread safe locking... (skip)
            this.channel = this.CreateChannel();
        }
        return this.channel;
    }
}
```

이 생성자는 [리스트 9]와 같이 매개변수로 주어진 configuration 이름을 사용하여 ChannelFactory<T> 객체를 초기화 하는 것을 알 수 있을 것이다. 또한 Channel 속성은 필요에 따라서 생성해 놓은 ChannelFactory 객체의 CreateChannel 메쏘드를 호출하여 프록시를 생성하고 있고 있다. [리스트 16]의 SayHello 메쏘드 내에서 Channel 속성을 통해 서비스를 호출하는 코드가 어떻게 작동하는지 이제 이해가 될 것이다.

Binding 과 EndpointAddress, 그리고 서비스의 주소를 매개변수로 취하는 ClientBase<T> 클래스의 다른 생성자들도 주어진 매개변수를 통해 다양한 방법으로 ChannelFactory<T> 객체를 초기화 하는 것일 뿐이다. ClientBase<T> 클래스는 ChannelFactory<T> 클래스를 직접적으로 사용하지 않고서도 서비스를 호출하게 해주는 편리한 프록시 래퍼 클래스로 생각하면 이해하기 쉽다. 이제 ClientBase<T>에서 파생된 HelloWorldClient 클래스를 사용하여 서비스 클라이언트 코드는 다음과 같이 작성할 수 있게 되었다.

리스트 17. 생성된 프록시 코드를 사용하는 클라이언트

```
string url = "http://localhost/wcf/example/helloworldservice";
BasicHttpBinding binding = new BasicHttpBinding();
EndpointAddress address = new EndpointAddress(url);
```

```
using (HelloWorldClient proxy = new HelloWorldClient(binding, address)) {
    string result = proxy.SayHello();
    Console.WriteLine(result);
}
```

[리스트 7]에 비해 [리스트 17]의 코드가 바뀐 부분은 ChannelFactory<T> 클래스를 전혀 사용하지 않았을뿐더러, using 문을 사용하여 보다 깔끔하게 코드를 작성할 수 있다는 것이다. 자..... 여기서 만족하지 말자. [리스트 9]와 같이 configuration 을 사용하는 코드 역시 HelloWorldClient 클래스를 통해 작성할 수 있지 않을까? 물론 가능하다. 그리고 더욱 즐거운 것은 svcutil.exe 가 configuration 을 작성하는데 쓰라고 output.config 라는 설정파일까지 이미 만들어 주었으니, 이 파일에 포함된 설정 내용은 [리스트 18]과 같다.

리스트 18. 자동으로 생성된 configuration 파일(output.config)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IHelloWorld" ..... >
        .....
        </binding>
      </basicHttpBinding>
      <netTcpBinding>
        <binding name="NetTcpBinding_IHelloWorld" ..... >
        .....
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint
            address="http://localhost/wcf/example/helloworldservice"
            binding="basicHttpBinding"
            bindingConfiguration="BasicHttpBinding_IHelloWorld"
            contract="IHelloWorld"
            name="BasicHttpBinding_IHelloWorld" />
      <endpoint
            address="net.tcp://localhost/wcf/example/helloworldservice"
            binding="netTcpBinding"
            bindingConfiguration="NetTcpBinding_IHelloWorld"
            contract="IHelloWorld"
            name="NetTcpBinding_IHelloWorld">
      .....
      </endpoint>
    </client>
  </system.serviceModel>
</configuration>
```

실제 output.config 의 내용이 상당히 많아서 기겁을 할 지도 모르겠다. 걱정할 필요 없다. 이 책을 읽어 가는 동안 금방 익숙해 질 것이며 설정 내용의 대부분은 바인딩의 디폴트 값을 명시적으로 언급한 것일 뿐이다. Svcutil.exe 입장에서 보았을 때 바인딩의 속성 값들이 어떤 것이 디폴트 값이고 어떤 것이 서비스에서 설정한 값인가를 판단하기 어렵기 때문에 서비스 메타 데이터에서 제시하는 모든 값들을 명시적으로 표현한 것일 뿐이다. 그래서 바인딩의 속성들이 복잡하게 보이지만 사실은 거의 대부분의 값들이 디폴트임을 알아두자.

주목할 부분은 <endpoint> 요소에 주어진 종점 설정의 이름이다. ClientBase<T> 클래스의 생성자에서 취하는 endpointConfigurationName 매개변수란 것이 바로 <endpoint> 요소에 나타난 name 속성의 값이란 얘기가 되겠다. Output.config 파일의 내용을 클라이언트의 app.config 파일에 복사해 넣으면, [리스트 17]의 코드는 다음과 같이 간략하게 코딩 할 수도 있다는 것이다.

```
using (HelloWorldClient proxy =
```

```
                new HelloWorldClient("BasicHttpBinding_IHelloWorld")) {
    string result = proxy.SayHello();
    Console.WriteLine(result);
}
```

위 코드는 BasicHttpBinding_IHelloWorld 란 이름을 가진 <endpoint> 요소를 찾아, 필요한 바인딩, 바인딩 설정, 종점 주소 등의 정보를 사용하여 프록시를 초기화하고 프록시 인스턴스를 생성하게 된다. 물론 이 때 사용하는 바인딩은 BasicHttpBinding 이며, 만약 NetTcpBinding 을 사용하고자 한다면 configuration 이름을 BasicHttpBinding_IHelloWorld 가 아닌 NetTcpBinding_IHelloWorld 이란 이름을 사용하면 될 것이다.

설정된 Configuration 을 오버라이드 하는 것 역시 어렵지 않게 가능하다. 다음 예제 코드는 configuration 에서 설정된 메시지 인코딩(만약 존재 한다면)을 오버라이드 하여 항상 MTOM 이 메시지를 인코딩 하는데 사용하도록 강요하고 있다.

```
using (HelloWorldClient proxy =
                 new HelloWorldClient("BasicHttpBinding_IHelloWorld")) {
    BasicHttpBinding binding = proxy.Endpoint.Binding as BasicHttpBinding;
    if (binding != null) {
        binding.MessageEncoding = WSMessageEncoding.Mtom;
    }
    string result = proxy.SayHello();
    Console.WriteLine(result);
}
```

또 하나, HelloWorldClient 클래스를 생성할 때 생성자에 매개변수를 전혀 주지 않고 디폴트 생성자(default constructor)를 사용하는 방법도 있다. 이 디폴트 생성자를 사용하면 클라이언트가 호출하고자 하는 서비스의 계약 인터페이스를 사용하는 종점을 configuration 에서 찾아 적용하고자 시도한다. 즉, IHelloWorld 인터페이스를 계약으로 사용하는 <endpoint> 요소를 찾는다는 말이 되겠다. 만약 IHelloWorld 인터페이스를 계약으로 사용하는 종점 설정을 발견하지 못하거나, 이 인터페이스를 계약으로 사용하는 종점이 2 개 이상이 발견되면 InvalidOperationException 이 발생하므로, 2 개의 종점을 제공하는 우리의 Hello World 예제에는 적용할 수 없다.

Visual Studio 의 서비스 참조 기능

지금까지 svcutil.exe 유틸리티를 사용하여 클라이언트 프록시 코드를 생성하고 관련된 configuration 까지 사용하는 방법을 살펴봤다. 하지만 이 방법은 상당히 손이 많이 가는 귀찮은 작업이다. 우리 개발자들이 Copy & Paste 와 같은 단순한 반복은 곧 잘 하지만, 서비스 프록시 코드를 생성하고 생성된 코드를 프로젝트에 추가하는 등의 작업은 상당히 싫어하는 편이다. 커맨드 프롬프트를 구동시키고, svcutil.exe 명령을 때려 넣고, 생성된 코드를 다시 프로젝트에 추가해야 하니 척 봐도 좀 귀찮아 보이지 않는가?

그래서..... Visual Studio 내부에서 직접 프록시 코드와 configuration 설정을 해주는 "서비스 참조 추가(Add Service Reference)" 기능을 사용할 수 있다([그림 2] 참조). 이 메뉴를 선택하면 [그림 3]과 같은 서비스 참조 추가 대화 상자가 나타나고 여기에 서비스의 주소를 명시할 수 있다. Visual Studio 2008 의 서비스 참조 기능은 서비스의 주소에서 경험적인 방법을 통해 서비스의 메타 데이터를 구하는 URL 을 검색한다. 즉, 서비스 주소로부터 MEX(Metadata Exchange) 종점을 유추해 낸다던가 서비스 주소 뒤에 ?wsdl 을 붙인 주소로부터 WSDL 을 읽으려고 시도한다던가 하는 시도를 여러 차례 반복하여 메타데이터를 구하려고 시도한다는 것이다.

주어진 서비스의 주소로부터 다양한 시도를 수행하여 WSDL 을 구했다면 [그림 3]과 같이 서비스 종점에서 제공하는 서비스 계약(들)과 서비스 계약 내의 메쏘드들이 표시된다. Visual Studio 를 사용하여 서비스 프록시를 생성할 때는 항상 네임스페이스를 입력하도록 되어 있는데, 이 네임스페이스는 생성될 서비스 프록시 클래스 및 인터페이스의 네임스페이스가 되며 프로젝트 내에서 참조한 서비스를 다시 재 참조할 때도 사용된다. Visual Studio 에서 생성되는 프록시는 항상 이 네임스페이스를 갖는 클래스로서 표현됨을 기억해 두자.

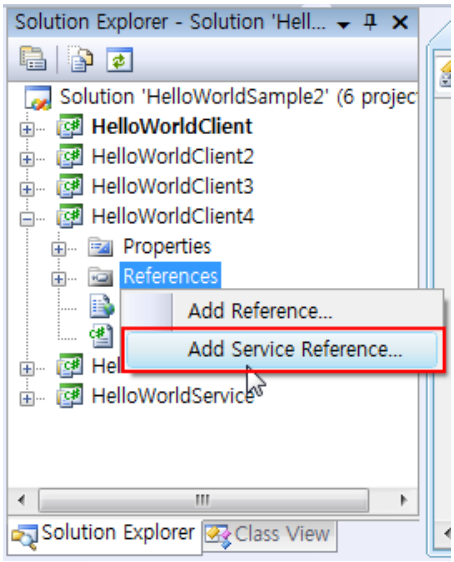


그림 2. Visual Studio 2008 의 서비스 참조 추가

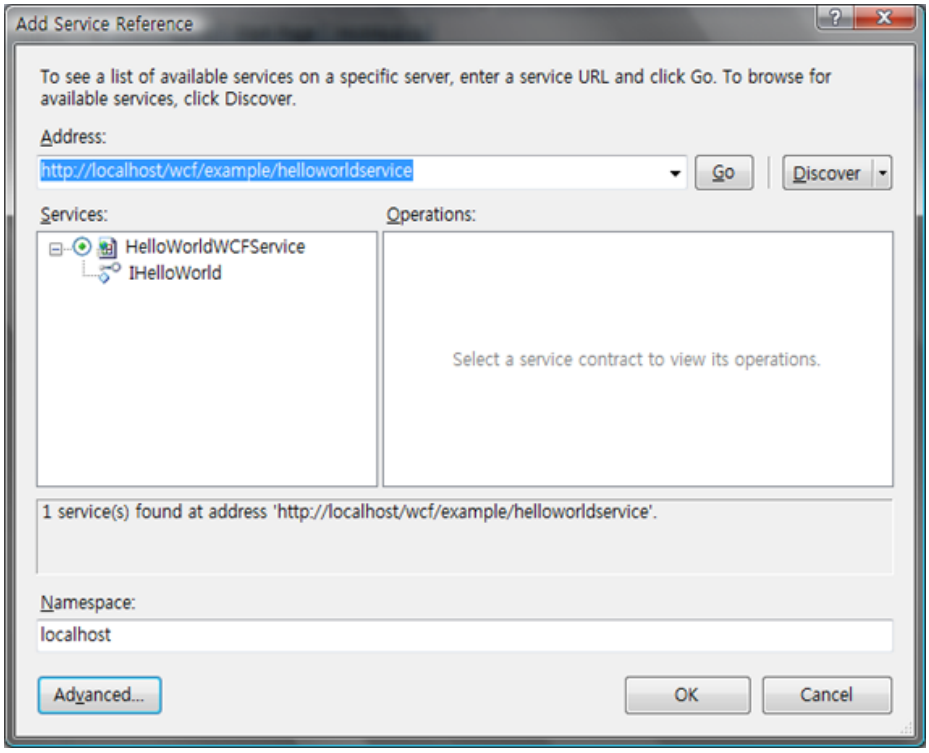


그림 3. 서비스 참조 추가 대화 상자

서비스 참조가 성공적으로 추가되면 [그림 4]와 같이 프로젝트에는 필요한 폴더 구조와 몇몇 파일들이 생성된다. 먼저, 클라이언트 프록시 코드를 담는 소스 코드(Reference.cs)가 생성되는데 이 코드는 svcutil.exe 을 통해 생성된 코드와 거의 동일 하다([리스트 16] 참조). 또한 자동으로 어플리케이션 설정 파일인 app.config 가 생성되어 필요한 WCF 설정들이 포함되거나, 이미 app.config 파일이 존재하는 경우 이 파일의 내용을 업데이트 하기도 한다. App.config 파일이 새로이 추가 되었건 기존 파일이 변경되었건 상관 없이 설정 파일의 내용은 [리스트 18]의 설정 내용과 다를 것이 없다. 이외에도 .wsdl 파일과 .xsd 파일들 그리고 몇몇 파일들이 생성되는데, 이 파일들은 서비스의 WSDL 및 XML 스키마에 대한 캐시 파일이며 서비스의 메타데이터를 읽은 URL 등의 정보들이 포함되어 있다. 이들 파일들은 서비스 참조를 갱신(update) 할 때 사용되어 서비스 메타데이터가 어떻게 변경되었는가를 추적하는데도 사용된다. 이 정보가 훼손되면 서비스 참조는 업데이트될 수 없다. 따라서 이들 파일의 내용을 수정하려고 시도해서는 안 된다.

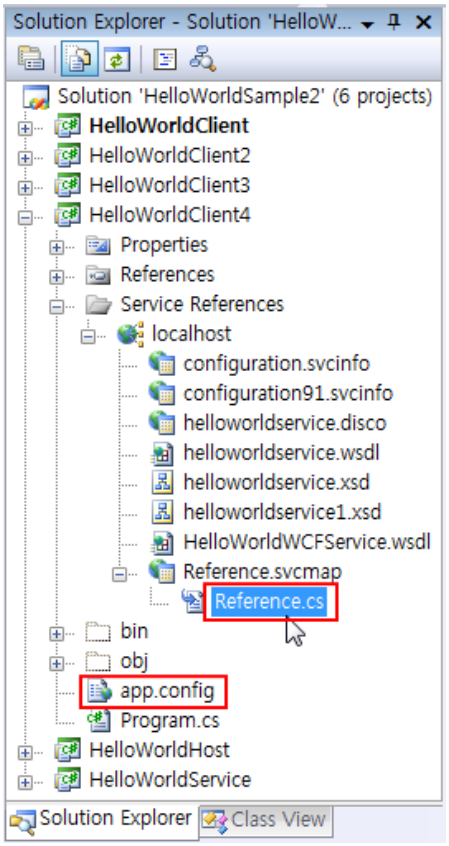


그림 4. 서비스 참조 후 생성되는 폴더 구조와 관련 파일들

자, 이렇게 서비스에 대한 프록시를 자동 생성했다면 이젠 클라이언트 코드는 Visual Studio 가 생성해 준 클래스를 사용하기만 하면 된다. [리스트 19]는 클라이언트 코드 전체를 보여주고 있다. Visual Studio 가 생성하는 프록시 코드는 항상 네임스페이스를 사용하여 코드를 생성하므로 using 문장이 추가되었음에 유의하자. 물론 svcutil.exe 를 사용할 때도 /namespace 옵션을 통해 다음과 같이 생성되는 코드의 네임스페이스를 설정할 수 있다. 이 옵션은 WSDL 상에 명시된 XML 네임스페이스를 닷넷 네임스페이스에 어떻게 매핑 시킬 것인가를 명시하는데, 다음 예는 WSDL 상의 모든 네임스페이스를 localhost 라는 닷넷 네임스페이스에 매핑 시키는 것을 보여주고 있다. 이러한 방식의 네임스페이스 매핑은 Visual Studio 의 기본 행동이기도 하다.

svcutil.exe http://localhost/wcf/example/helloworldservice?wsdl /language:C# /namespace:*,localhost /out:Proxy.cs

리스트 19. Visual Studio 가 생성해 준 프록시를 사용하는 클라이언트 전체 코드

```
using System;
using HelloWorldClient4.localhost;

namespace HelloWorldClient4
{
    class Program
    {
        static void Main(string[] args)
        {
            using (HelloWorldClient proxy =
                    new HelloWorldClient("NetTcpBinding_IHelloWorld")) {
                string result = proxy.SayHello();
                Console.WriteLine(result);
            }
        }
    }
}
```

수동으로 프록시 만들기

Svcutil.exe 나 Visual Studio 를 통해 ClientBase<T> 클래스 기반의 프록시 코드를 생성하는 방법 외에도 이 클래스를 직접 코딩 할 수도 있다. 유틸리티나 개발도구가 생성해주는 코드는 고정적인 코드를 생성하기 때문에 어떤 변환 작업을 수행하기 어려울 때가 많다. 예를 들어 클라이언트가 서비스를 호출하기 전에 로그를 남겨야 한다고 가정해 보자. 자동으로 생성된 프록시 코드(Reference.cs 혹은 proxy.cs)를 직접 수정할 수도 있겠다. 하지만 나중에 서비스의 인터페이스가 바뀌게 되어 서비스 참조를 업데이트 해야만 한다면 프록시 코드가 다시 생성되기 때문에 이전에 작성해 놓은 코드는 모조리 사라져 버리게 된다. 매우 극단적인 예지만 이러한 상황은 실제 프로젝트 상에서 종종 발생하는 일이므로 직접 수작업으로 프록시 코드를 생성해야 하는 경우도 생긴다는 말이다.

[리스트 20]은 수작업으로 작성한 프록시 코드를 보여주고 있다. 어셈블리 참조에 의해서건, 직접 코드를 작성했건, 자동으로 생성된 프록시 코드에서 빌려오건, 일단 서비스에 대한 계약 인터페이스를 구할 수만 있다면 ClientBase<T> 클래스에서 파생된 프록시 클래스를 생성할 수 있으며 이 클래스는 구미에 맞게 마음대로 작성할 수 있다. [리스트 20]은 configuration 을 통해 바인딩과 서비스 주소를 초기화 하는 생성자 만을 선언하였으며, 서비스 호출 전에 디버그 메시지를 출력하고 있다.

리스트 20. 수작업으로 작성한 프록시 코드 예제

```
public class MyProxy : ClientBase<IHelloWorld>, IHelloWorld
{
    public MyProxy(string configurationName)
        : base(configurationName)
    {
    }

    public string SayHello()
    {
        System.Diagnostics.Debug.WriteLine(“Invoke SayHello()");
        return base.Channel.SayHello();
    }
}
```

04. IIS에 서비스 호스팅

지금까지 WCF 의 가장 기본적인 개념들인 서비스 종점, 계약, 주소, 바인딩, 호스트, 채널, 클라이언트 프록시 등을 살펴보았다. 또한 관련된 예제 코드들 역시 WCF 의 기본적인 사항들을 이해할 수 있을 정도로 충분히 살펴보았다. 이 정도만 알고 있다면 WCF 에 관련된 기본적인 프로그래밍 기초는 닦았다고 할 수 있다. 하지만 이것으로는 충분하지 않다. WCF 기반의 실제 프로젝트를 진행할 때 서비스는 많은 클라이언트에 의해 호출될 것이며 또한 오랫동안 안정적으로 서비스를 제공해야 할 것이다. 다시 말하자면 서비스 호스트 프로세스는 안정적으로 다수의 클라이언트로부터 동시 호출을 수용해야만 한다. **이러한 서비스의 가용성(availability)과 안정성(stability)에 직결되는 것은 서비스 호스트 프로그램을 얼마나 안정적으로 작성할 것 인가이다**.

바로 IIS(Internet Information Service)가 바로 그런 호스팅 환경을 제공한다. IIS 는 오랫동안 웹 어플리케이션의 호스트로서 빠른 성능과 안정된 수행환경을 자랑해 왔다. 특히, IIS 6.0 이 제공하는 웹 가든(web garden), 프로세스 재생(recycling) 등의 기능은 최상의 웹 어플리케이션의 안식처로 평가 받아 왔다. 이러한 최상의 호스팅 환경을 WCF 에서도 사용할 수 있다. 즉, WCF 서비스의 호스트로서 IIS 를 사용할 수 있다는 말이다.

WCF 가 설치되면 IIS 에 ISAPI Extension 및 ASP.NET 확장자(extension) 매핑을 통해 .svc 확장자를 가진 URL 이 특별히 처리되도록 설정을 수행한다. 즉, 확장자가 .svc 인 파일이 액세스되면 이 파일이 WCF 의 서비스로서 인식되도록 한다는 말이다. WCF 에서 제공하는 ASP.NET 의 HTTP 핸들러는 .svc 파일의 내용과 web.config 파일의 내용을 참조하여 ServiceHost 객체를 스스로 생성하고 서비스를 구동시켜 클라이언트 호출을 받을 수 있도록 구성되어 있다. 이제부터 구체적으로 WCF 서비스를 IIS 에 호스팅 하는 방법을 살펴보도록 하자.

먼저 [그림 5]과 같이 솔루션에 ASP.NET 웹 어플리케이션 프로젝트를 추가하도록 한다. 물론 "웹 사이트 추가" 기능을 사용해도 되지만, 공동 작업, 배포, 빌드 등 여러 가지 이유로 웹 어플리케이션 프로젝트가 더 많이 사용되곤 한다. 프로젝트 생성 후, Default.aspx 는 필요하지 않으므로 과감히 지워버리자. 그리고 [그림 6]과 같이 서비스의 계약이 선언되고 구현된 HelloWorldService 프로젝트를 "참조" 하도록 하자. 기억할지 모르겠지만 1 장에서 Hello World 서비스의 계약을 선언하고 구현할 때 굳이 별도의 DLL 프로젝트를 만든 이유가 여기에 있다. 달랑 서비스 호스트 EXE 프로젝트에 서비스를 구현해 버렸다면 IIS 에 호스팅 하기 위해 소스 코드를 복사하거나 코드를 다시 구현해 주어야 했을 것이다. 서비스를 별도의 어셈블리로 만들어 놓으면 이처럼 독립 EXE 어플리케이션이나 IIS 상에서 호스팅 하거나 Windows 서비스에서도 호스팅 하는 다양한 시나리오를 적용할 수 있어서 유리하다.

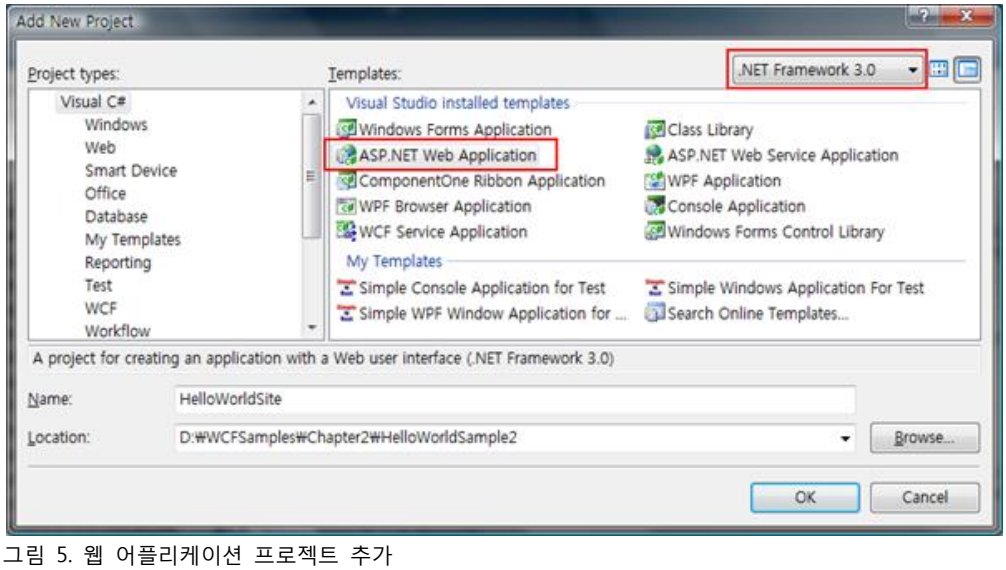


그림 5. 웹 어플리케이션 프로젝트 추가

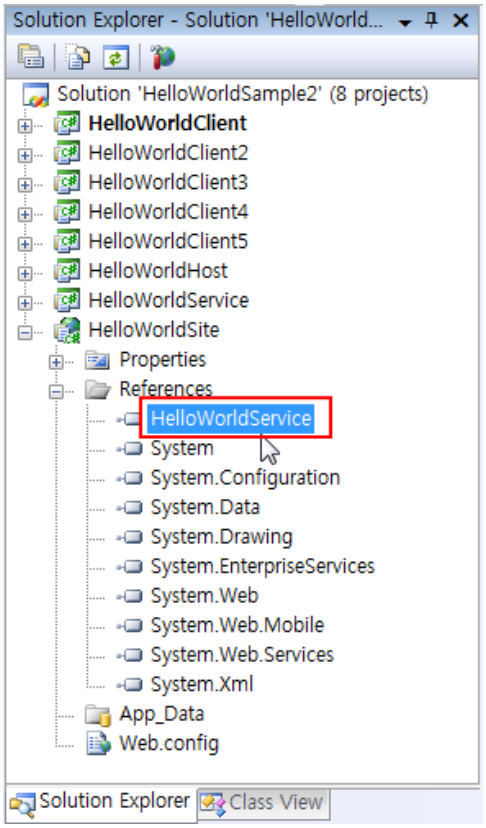


그림 6. HelloWorldService 프로젝트 참조

이제 [그림 7]과 같이 텍스트 파일 아이템 템플릿을 이용하여 WCF 서비스에 대한 정보가 포함된 HelloWorld.svc 파일을 생성한다. Visual Studio 에는 WCF 서비스를 위한 아이템 템플릿을 제공하지만 이 아이템 템플릿은 .svc 파일 뿐만 아니라 서비스 구현에 관련된 .cs 파일들도 생성해 버린다. 웹 프로젝트 내에 서비스 구현을 포함시킬 것이라면 이 아이템 템플릿이 편리하지만 우리의 Hello World 서비스 예제처럼 외부 어셈블리에 서비스 구현을 가지고 있는 경우 적합한 아이템 템플릿이 아니다. 따라서 텍스트 파일 아이템 템플릿을 사용하도록 하자.

HelloWorld.svc 파일은 ASP.NET 웹 서비스의 .asmx 파일과 매우 비슷하며 그 용도와 역할마저 같다. HelloWorld.svc 파일에는 ServiceHost 객체가 호스팅 할 서비스 타입을 명시하는 것이 가장 핵심적이다. 그리고 .asmx 파일과 마찬가지로 디버깅 여부 등의 잡다한 속성을 명시할 수 있다. 다음은 HelloWorld.svc 파일의 전체 내용이다.

```
<%@ServiceHost language=c# Debug="true"
    Service="HelloWorldService.HelloWorldWCFService" %>
```

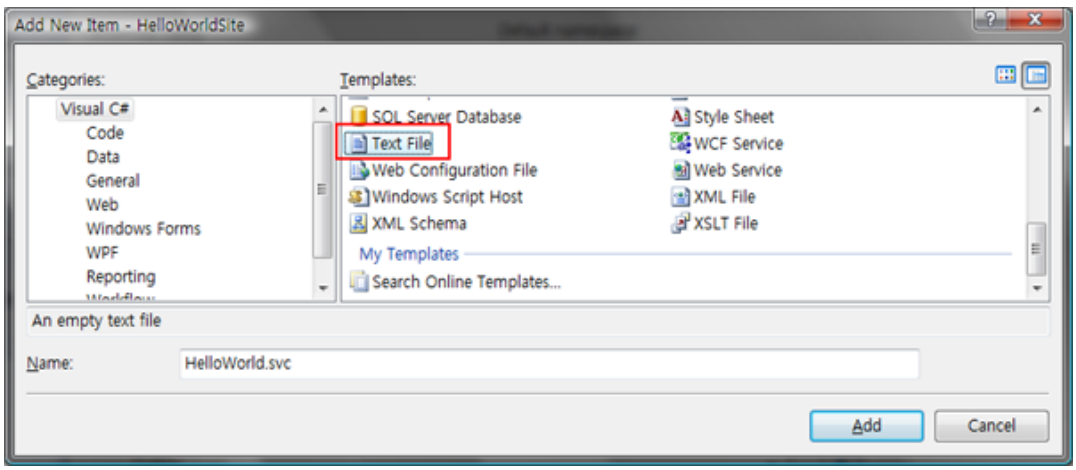


그림 7. HelloWorld.svc 파일 추가

HelloWorld.svc 파일에 명시된 @ServiceHost 지시자(directive)는 WCF 런타임이 생성할 ServiceHost 객체를 초기화 하는데 사용한다. 앞서 여러 번 살펴본 서비스 호스트 코드들([리스트 5]와 같이)에서 ServiceHost 클래스의 인스턴스를 생성할 때 생성자의 매개변수로 사용한 서비스 타입 정보를 기억하는가? 이 서비스 타입 정보를 @ServiceHost 지시자의 Service 속성을 통해 명시하는 것이다. 위 .svc 파일의 @ServiceHost 지시자 내용을 억지로 코드로서 표현해 보자면 다음 정도가 되겠다. 여기에서 서비스의 베이스 주소가 .svc 파일의 주소가 되었음을 주목하자. 이러한 코드가 WCF 런타임에 의해 자동적으로 수행된다고 보면 된다 .

```
ServiceHost host =
    new ServiceHost(Type.GetType("HelloWorldService.HelloWorldWCFService"),
    new Uri("http://localhost:1687/HelloWorld.svc"));
```

.svc 파일의 내용에 의해 서비스 호스트를 생성할 준비가 되었으므로 이제 남은 것은 이 서비스가 사용할 서비스 종점(endpoint)에 대한 사항이 남았다. 즉, @ServiceHost 의 Service 속성에 의해 명시된 서비스 타입이 구현하는 서비스 계약(contract), 그리고 서비스의 주소와 서비스가 사용할 바인딩을 WCF 에게 알려야 하는 것이다. 서비스 종점에 대한 설정은 web.config 에 <system.ServiceModel> 섹션을 추가하면 된다. Configuration 을 사용하는 방법은 이미 살펴 보았으므로 web.config 의 내용을 곧바로 살펴보도록 하자.

리스트 21. WCF 서비스 호스팅에 관련된 web.config 파일 내용

```
<?xml version="1.0"?>
<configuration>
    <appSettings/>
    <connectionStrings/>
    <system.web>
        <compilation debug="true" />
        <authentication mode="Windows" />
    </system.web>
    <system.serviceModel>
        <services>
            <service name="HelloWorldService.HelloWorldWCFService"
                    behaviorConfiguration="HelloWorldService">
                <endpoint contract="HelloWorldService.IHelloWorld"
                        address=""
                        binding="basicHttpBinding" />
            </service>
        </services>
        <behaviors>
            <serviceBehaviors>
                <behavior name="HelloWorldService">
                    <serviceMetadata httpGetEnabled="true"/>
                </behavior>
            </serviceBehaviors>
        </behaviors>
    </system.serviceModel>
</configuration>
```

[리스트 21]의 web.config 설정은 ASP.NET 을 위한 섹션들이 존재할 뿐이지 [리스트 2]의 configuration 과 크게 다를 바 없다. 서비스의 계약은 당연히 IHelloWorld 인터페이스가 될 것이며, 서비스 주소는 베이스 주소를 그대로 사용하도록 설정하였다. IIS 에

호스팅 되는 서비스의 베이스 주소는 .svc 파일의 주소가 됨을 명심하자. 마지막으로 바인딩은 BasicHttpBinding 을 사용하였다.

IIS 에 호스팅 되는 서비스의 바인딩은 트랜스포트로서 반드시 HTTP 프로토콜을 사용해야만 한다. 이는 IIS 가 웹 서버이기 때문에 너무나도 당연한 제약으로서 받아 들여야 할 것이다. 이 제약은 Windows 2000 및 Windows XP 에 포함된 IIS 5.x 버전과 Windows Server 2003 에 포함된 IIS 6.0 에만 해당된다. Windows Vista 와 Windows Server 2008 에 포함되는 **IIS 7.0 은 Windows Process Activation Service(WAS)를 통해 TCP 혹은 명명된 파이프 트랜스포트를 사용하는 WCF 서비스도 IIS 의 작업 프로세스(worker process) 내에 호스팅이 가능하다.**

자..... 이제 .svc 파일을 생성하였고 web.config 파일에 관련 설정도 했으면 테스트를 해볼 차례이다. 솔루션에 생성한 웹 프로젝트는 기본 설정 상 ASP.NET Development Server 라 불리는 웹 서버 상에서 수행된다. 이 개발용 웹 서버를 이용하여 테스트를 수행 할 수도 있고 개발된 웹 사이트를 IIS 상에 배포하여 테스트할 수도 있다. 혹은 웹 프로젝트의 설정을 바꾸어 가상 디렉터리를 통해 항상 IIS 를 사용하도록 할 수도 있다. 이 책이 ASP.NET 에 대한 책이 아니므로 더 이상 길게 설명하진 않겠다. 관련 서적이나 관련 자료를 MSDN 도움말에서 찾아보기 바란다.

필자는 개발용 웹 서버를 통해 간단히 테스트 해 보았고, 그 결과를 [그림 8]에 보인다. 웹 서비스의 URL 을 주목할 필요가 있으며 또한 WSDL 을 위한 주소 역시 눈길 한 번 줘야 한다. 서비스의 URL 은 앞서 언급한 대로 .svc 파일의 URL 이 되었으며 WSDL 을 위한 URL 은 쿼리 문자열로서 ?wsdl 이 추가 되었음을 확인 하자.

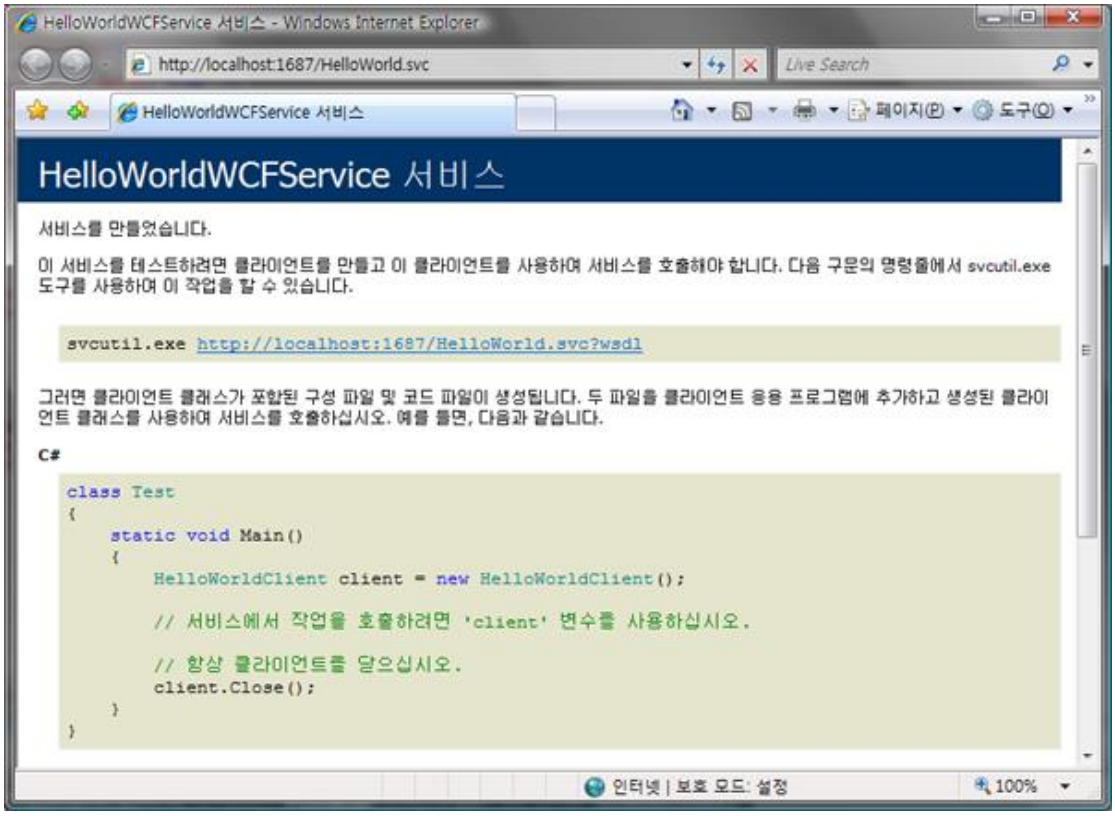


그림 8. IIS 에 의해 호스팅 된 Hello World 서비스

테스트가 끝났으니 이제 WCF 서비스는 IIS 상에서 호스팅 되어 서비스를 제공할 수 있게 되었다. 앞서 작성해 놓은 클라이언트의 코드 혹은 app.config 파일에서 서비스의 주소를 적절히 바꾸어 주면 클라이언트는 IIS 상에서 호스팅 되는 서비스를 호출 할 수 있게 된다.