WCF Guidance for WPF Developers

Michele Leroux Bustamante, May 2009

Contents

Today, most applications comprise at least three tiers: a client application tier, a middle tier, and a database tier. Client applications, be they web applications or rich clients, rely on the middle tier to coordinate access to application resources including database assets. In a service-oriented architecture, this middle tier is composed of services – each exposing a well-defined chunk of business functionality. Windows Communication Foundation (WCF) is Microsoft's technology for developing services, and for building clients that consume services. WCF can be used to build classic client-server applications that rely on services hosted within a specific intranet domain, or to build and consume interoperable services or REST-based services over the Internet.

Most WCF documentation focuses on how to design, implement and host WCF services – with limited discussion of the client-side implications. This whitepaper will focus specifically on the client-side experience when building Windows Presentation Foundation (WPF) applications that consume WCF services over the intranet or Internet. Specifically, this paper will focus on issues that client developers frequently encounter when consuming WCF services including:

- Recommended practices for proxy generation
- Data binding considerations
- Guidance on sharing libraries between clients and services
- Managing proxy lifetime and dealing with exceptions and session timeouts
- Caching optimizations
- Considerations for multithreaded and duplex clients
- Hosting services at the client
- Consuming REST-based services

To provide some foundation for those new to WCF, the first section will provide a quick tour of the requirements to design, implement, host, and consume a WCF service. Following that, each section will dig in to concerns relevant to WPF client development and only where applicable summarize how a service should be implemented to support specific scenarios.

WCF 101

If you are new to WCF, this section will introduce you to WCF and explain the basics related to building and consuming WCF services. If you are already familiar with WCF, you can feel free to skip this section.

WCF is Microsoft's platform for building distributed service-oriented applications for the enterprise and the web that are secure, reliable, and scalable. It supersedes previous technologies such as .NET Remoting, Enterprise Services and ASP.NET Web Services (ASMX) by offering a unified object model for building applications that support the same distributed computing scenarios. WCF supports scenarios such as:

- Classic client-server applications where clients access functionality on remote server machines
- Distribution of services behind the firewall in support of a classic service-oriented architecture
- Asynchronous or disconnected calls implemented with queued messaging patterns
- Workflow services for long running business operations
- Interoperable web services based on SOAP protocol and advanced WS* protocols
- Web programming models with support for Plain-Old-XML (POX), Javascript Object Notation (JSON), Representational State Transfer (REST) and Syndication with RSS or Atom/Pub

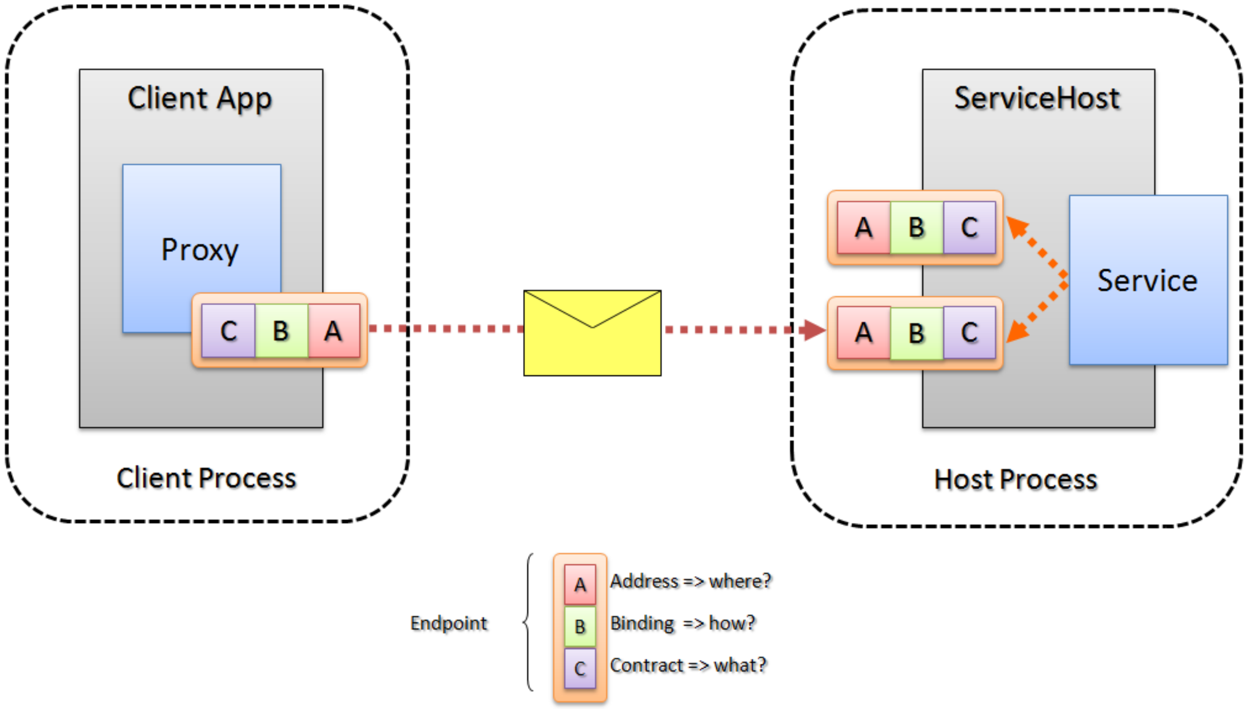
WCF was initially introduced with the .NET Framework 3.0 which released with Windows Vista in January 2007 –alongside with Windows Workflow Foundation (WF) and Windows Presentation Foundation (WPF). When the .NET Framework 3.5 released with Visual Studio 2008 in November 2007 additional WCF features were introduced including improvements to the web programming model and support for the latest WS* protocols.

It is impossible to sum up a platform as vast and feature-rich as WCF in a short section – but the sections to follow will explain some fundamental concepts that will provide you with a foundation for subsequent sections in this paper – including the steps required to create and host a service, and to call a service from a WPF client.

Services, Proxies and Endpoints

Figure 1 provides a visual perspective on the fundamental architecture of a WCF service and WCF client. A service is a type that exposes one or more service operations (methods) for remote clients to call. These operations usually coordinate calls to the business and data tier to implement their logic, while the service itself is a boundary for distributed communications. Services can be hosted in any managed process – for server deployments this is typically Internet Information Services (IIS) 6, the Windows Process Activation Service (WAS) (part of IIS 7), or a Windows Service. Part of the hosting process is to initialize a ServiceHost instance, tell it which service type should be instantiated for incoming requests, and configure endpoints to tell it where requests should be sent. An endpoint comprises the address where messages should be sent, a binding which describes a set of protocols supported at the address, and a contract describing which operations can be called at the address and their associated metadata requirements. The ServiceHost can be initialized programmatically or by declarative configuration – in either case it will be initialized with one or endpoints that will trigger initializing the service type when request are received. ServiceHost initialization is explicit for self-hosting scenarios (like a Windows Service or test Console) and automatic when hosting with IIS or WAS.

Figure 1: Fundamental architecture of a WCF implementation



In order for a client to call operations exposed by a hosted service it must have access to the service metadata (the contract) so that it can send messages with the required parameters, and process return values accordingly.

Clients typically rely on proxy generation to improve their productivity for writing code to call remote services. Services enable metadata exchange in order to support proxy generation. A proxy is a type that looks much like the service type, except that it does not include the service implementation. This type is configured much the same as the ServiceHost – it requires access to one of the service endpoint addresses, the binding for that endpoint, and the contract. This way the client and service agree on where the proxy should send messages, what protocols should be used, and what data should be serialized to call the operation, along with what data will be serialized upon its return. It doesn't matter if the client proxy looks identical to the service type, nor if the supporting types are implemented exactly as they are at the service. What matters is wire-compatibility between the two.

In the next sections I'll explain how to create, host and consume your first service. The example that will be used for many of the discussions in this whitepaper is a Todo List application – and so I will use that example to walk you through this process.

Creating a New Service

When you create a service you begin with defining the service contract and any associated data contracts required for serialization. The Todo List application uses the service contract shown in Figure 2 (ITodoListService) and the data contract shown in Figure 3 (TodoItem with associated enumerations).

Figure 2: ITodoListService service contract implementation

```
[ServiceContract(Namespace="http://wcfclientguidance.codeplex.com/2009/04")]
public interface ITodoListService
{
    [OperationContract]
    List<TodoItem> GetItems();

    [OperationContract]
    string CreateItem(TodoItem item);
    [OperationContract]
    void UpdateItem(TodoItem item);
    [OperationContract]
    void DeleteItem(string id);
}
```

Figure 3: TodoItem data contract implementation with dependent enumerations

```
[DataContract(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
public class TodoItem
{
    [DataMember(Order=1, IsRequired=false)]
    public string ID { get; set; }
    [DataMember(Order = 2, IsRequired = true)]
    public string Title { get; set; }
    [DataMember(Order = 3, IsRequired = true)]
    public string Description { get; set; }
    [DataMember(Order = 4, IsRequired = true)]
    public PriorityFlag Priority { get; set; }
    [DataMember(Order = 5, IsRequired = true)]
    public StatusFlag Status { get; set; }
    [DataMember(Order = 6, IsRequired = true)]
    public DateTime? CreationDate { get; set; }
    [DataMember(Order = 7, IsRequired = true)]
    public DateTime? DueDate { get; set; }
    [DataMember(Order = 8, IsRequired = true)]
    public DateTime? CompletionDate { get; set; }
    [DataMember(Order = 9, IsRequired = true)]
    public double PercentComplete { get; set; }
    [DataMember(Order = 10, IsRequired = true)]
    public string Tags { get; set; }
}
```

```
public enum PriorityFlag
{
    Low,
    Normal,
    High
}

public enum StatusFlag
{
    NotStarted,
    InProgress,
    Completed,
    WaitingOnSomeoneElse,
    Deferred
}
```

The service contract is implemented as a CLR interface decorated with the ServiceContractAttribute, each method decorated with the OperationContractAttribute so that it will be included in the service metadata. You typically supply a Namespace to the ServiceContractAttribute to disambiguate messages on the wire.

The data contract is a CLR type decorated with the DataContractAttribute, each member decorated with the DataMemberAttribute to include it in serialization. Again, a Namespace is supplied to the DataContractAttribute to disambiguate types during serialization. It is also a recommended practice that DataMemberAttributes include values for the Order and IsRequired properties. Order is alphabetical by default, but supplying explicit order improves clarity and maintainability. By default no values are required, so it is important to be explicit if any values are indeed required to fill out the type.

The service contract is implemented by a CLR type like the one shown in Figure 4 (TodoListService). The service type implements one or more service contract and coordinates calls to the business and data tier of the application. In this simple example the implementation of the service is embedded directly into the service type. The ServiceBehaviorAttribute is used to apply service behaviors that are tightly coupled to the implementation.

Figure 4: Implementation of TodoListService

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall,
ConcurrencyMode=ConcurrencyMode.Multiple)]
public class TodoListService: ITodoListService
{
    private static object _TodoListLock = new object();
    private static List<TodoItem> _SharedTodoList = new List<TodoItem>();

    public List<TodoItem> GetItems()
    {
        return _SharedTodoList;
    }

    public string CreateItem(TodoItem item)
    {
        lock (_TodoListLock)
        {
            item.ID = _SharedTodoList.Count.ToString();
            _SharedTodoList.Add(item);
        }
        return item.ID;
    }

    public void UpdateItem(TodoItem item)
    {
        lock (_TodoListLock)
        {
            TodoItem found = _SharedTodoList.Find(x => x.ID == item.ID);
            found.Title = item.Title;
            found.Description = item.Description;
            found.DueDate = item.DueDate;
            found.CompletionDate = item.CompletionDate;
            found.PercentComplete = item.PercentComplete;
            found.Priority = item.Priority;
            found.Status = item.Status;
            found.Tags = item.Tags;
        }
    }

    public void DeleteItem(string id)
    {
        lock (_TodoListLock)
        {
            _SharedTodoList.RemoveAll(x => x.ID == id);
        }
    }

}
```

Typically, services use InstanceContextMode.PerCall so that each call gets its own instance of the service, and ConcurrencyMode.Multiple so that multithreaded clients can execute multiple concurrent threads at the service. Since this example uses an in-memory collection to store TodoItem instances – a lock is used to protect access to the collection from multiple executing threads.

Configuring the ServiceHost

Assuming you are hosting on Windows Server 2003 or Windows Server 2008 machines – you will typically host services in Internet Information Services (IIS), Windows Process Activation Service (WAS) or self-host in a Windows Service. For testing purposes one usually uses a Console application to host services – graduating to the appropriate Windows Service or Web Site host to run final tests equivalent to production.

I'll start by explaining the hosting process with a Console and then compare to the other hosting environments. To host a service you will initialize the ServiceHost for that service type and supply one or more service endpoints, custom binding configurations if applicable, and configure runtime behaviors.

Figure 5 shows the code to initialize the ServiceHost for the TodoListService in the context of a Console test host. A few things are notable:

- Since the lifetime of the Console host is tied to the ServiceHost instance, it is ok to use the “using” statement. In the event that Close() throws an exception, when the process is shut down it will clean up any lingering channel resources. If this were a Windows Service, you would call Close() directly, and call Abort() in the face of an exception to ensure proper clean up if the process was to remain alive. You’ll see this pattern employed with proxies in this whitepaper.
- Handle the Faulted event so that you can notify administrators of any problems.

Figure 5: Initialization the ServiceHost for the TodoListService

```
static void Main(string[] args)
{
    using (ServiceHost host = new
ServiceHost(typeof(TodoList.TodoListService)))
    {
        host.Faulted += new EventHandler(host_Faulted);
        host.Open();

        Console.WriteLine("ServiceHost now running.");
        Console.ReadLine();
    }

    Console.ReadLine();
}

static void host_Faulted(object sender, EventArgs e)
{
    Console.WriteLine("ServiceHost has faulted. Restart the service.");
}
```

The code in Figure 5 assumes that endpoints will be configured in the app.config (this would be the web.config for IIS and WAS hosting). Figure 6 shows the <system.serviceModel> section for the TodoListService. A single endpoint is exposed over WSHttpBinding – which by default secures calls with a Windows credential. A metadata exchange endpoint is exposed over MexHttpBinding (a variation of WSHttpBinding without security) to support proxy generation. The associated service behavior (see the <serviceBehaviors> section) enables the ServiceMetadataBehavior to support the metadata exchange endpoint, and to enable HTTP browsing (so that you can view the WSDL document in the browser). The debug setting to includeExceptionDetailsInFaults should be set to false in production deployments.

Figure 6: Service model configuration for the TodoListService

```
<system.serviceModel>
  <services>
    <service name="TodoList.TodoListService"
behaviorConfiguration="serviceBehavior">
      <endpoint address="" binding="wsHttpBinding"
contract="Contracts.ITodoListService" />
      <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange" />
```

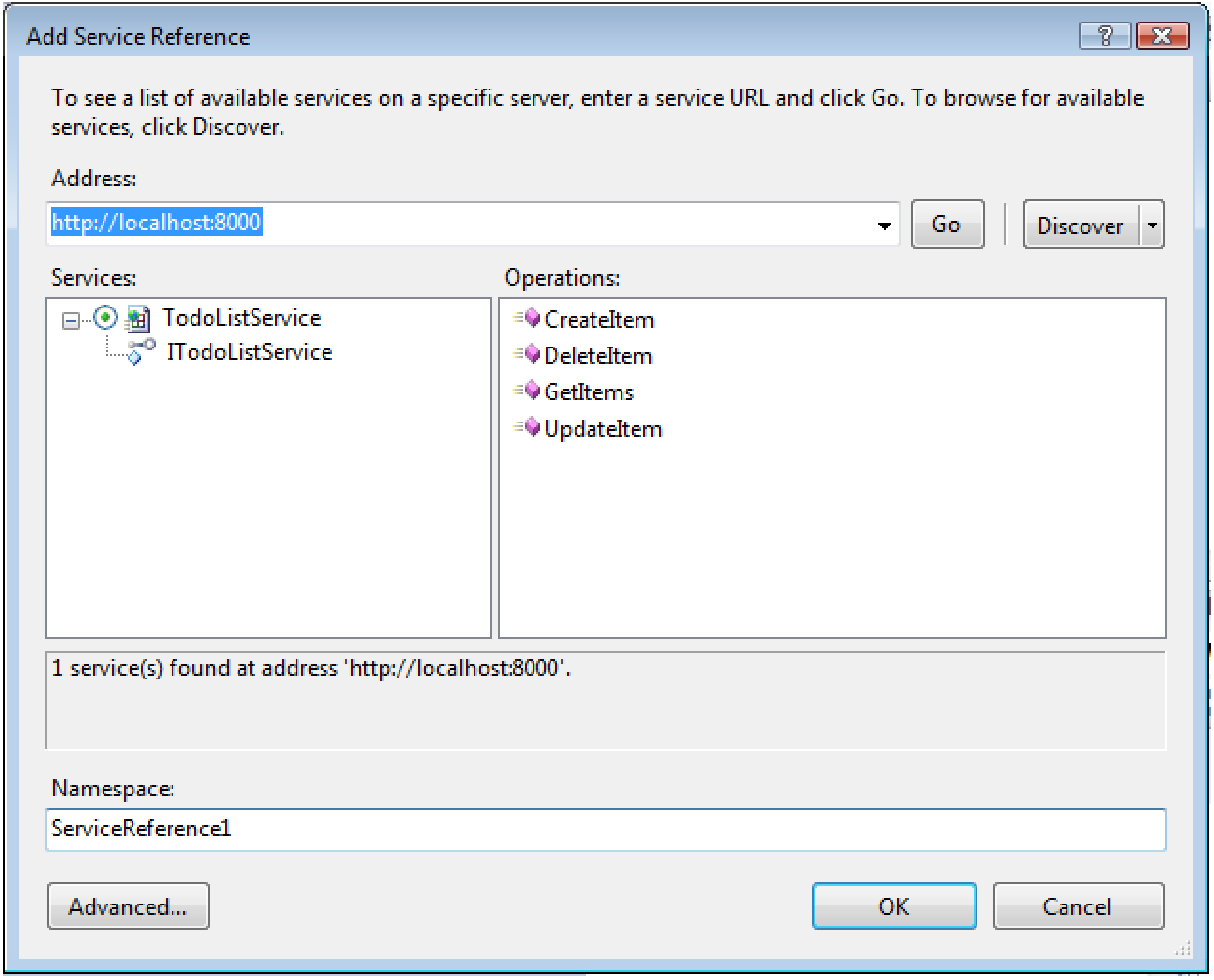
```
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8000"/>
        </baseAddresses>
      </host>
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="serviceBehavior">
        <serviceDebug includeExceptionDetailInFaults="false"/>
        <serviceMetadata httpGetEnabled="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

With this you now have a service to consume from your WPF client.

Calling the Service

Once you have created a WPF application and the Console host is running you can generate a proxy for your WPF client to call the service. If the service, the host and the client are part of the same solution, you must run the Console host without debugging first in order to generate a proxy using Add Service Reference. From Solution Explorer you can right-click on the WPF project and select Add Service Reference which launches the Service Reference dialog. Supply the base address to the service as shown in Figure 7. If the service is hosted with IIS or WAS, the base address is the address to the .svc endpoint such as “*http://localhost/TodoListWebHost/TodoListService.svc*”.

Figure 7: Adding a service reference using a self-hosted service base address



To see a list of available services on a specific server, enter a service URL and click Go. To browse for available services, click Discover.

Address: http://localhost:8000

Services:
- TodoListService
  - ITodoListService

Operations:
- CreateItem
- DeleteItem
- GetItems
- UpdateItem

1 service(s) found at address 'http://localhost:8000'.

Namespace: ServiceReference1

This generates a proxy class named TodoListServiceClient, generates copies of the service metadata such as the service contract and data contract discussed earlier, and generates a client-side service model configuration for the app.config. You can now write code to construct the proxy and begin making calls to service operations such as GetItems() as in the following code:

```
TodoListServiceClient _Proxy = new TodoListServiceClient();
_ToDoItems = _Proxy.GetItems();
```

Of course there are some additional details we will now begin to discuss related specifically to client-side development – however this should give you the gist of how you can create, host and consume a WCF service.

Metadata and Proxy Generation

In theory, the client does not care how a service is implemented since it relies on published service metadata to generate a proxy that can call the service. This generates a client-side copy of the necessary metadata that results in a similar set of types to the service implementation with wire compatibility. The client application can optionally share metadata libraries with the service. The sections to follow will discuss the proxy generation features available to Add Service Reference and SvcUtil, compare ClientBase<T> and ChannelFactory<T>, compare proxy generation and shared libraries, and provide recommendations for client-side representations of types for data binding, change notifications, and version tolerance.

Proxy Generation Fundamentals

No matter how types are implemented at the service, WSDL definition, or the runtime ServiceDescription accessible through WS-MetadataExchange that is used to generate a proxy using SvcUtil. The ServiceHost initializes a ServiceDescription that, among other things, includes information for all service endpoints including related service contracts, message contracts, fault contracts, data contracts and other serializable types such as types marked with the SerializableAttribute, types that implement IXmlSerializable, and XmlSerializer types if the XmlSerializer is being used at the service. For those new to WCF, Figure 8 summarizes these contract types and their role in service development.

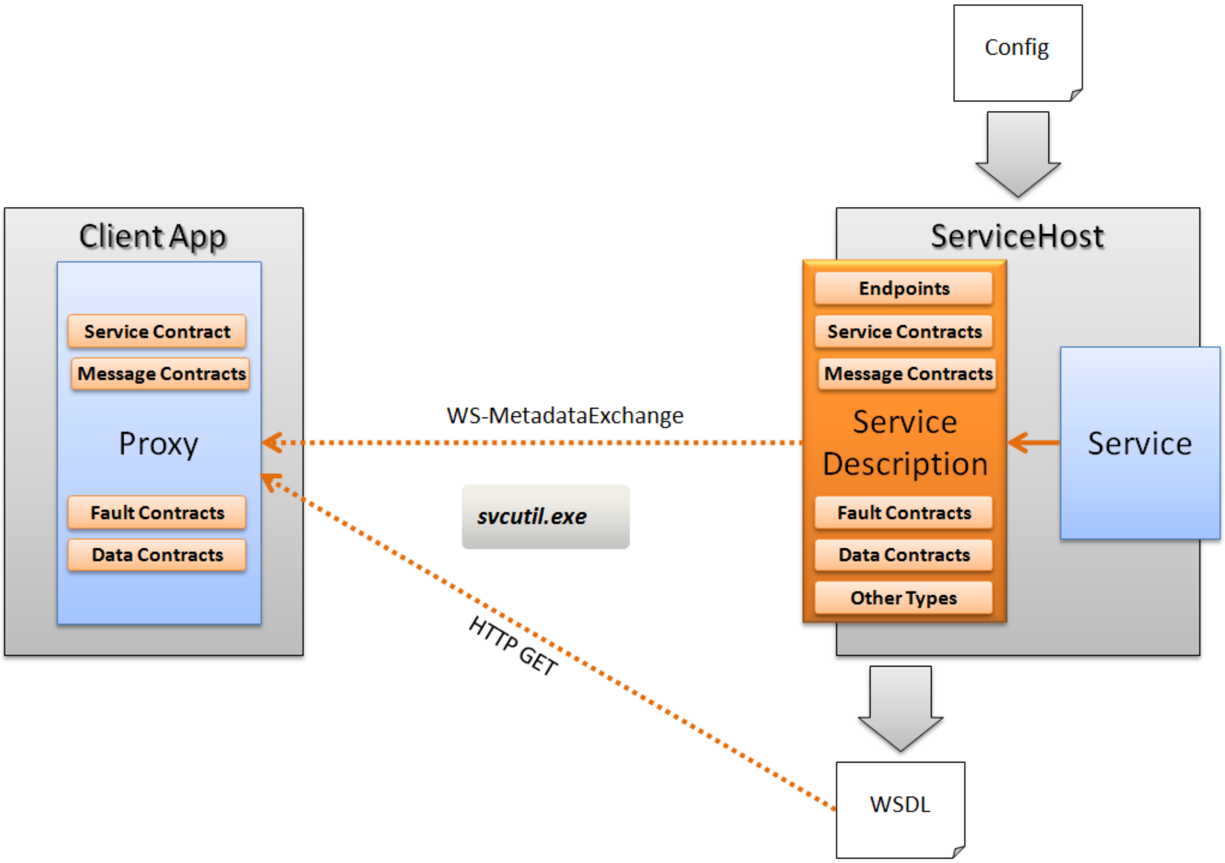
Figure 8: A summary of contract types that may be used at the service

Contract Type	Service Usage
Service Contract	Usually implemented as a CLR interface decorated with the ServiceContractAttribute. Defines service operations to be exposed for a particular endpoint.
Message Contract	Used to define SOAP messages as in parameters or return values from a service operation. Includes header and body elements that must be serializable types.
Fault Contract	Used to indicate in the service metadata the type of SOAP fault details that may be returned by a particular service operation if exception occurs and a fault is thrown.
Data Contract	The preferred way to define types that can be serialized by the DataContractSerializer for a WCF service. Forces you to opt-in all elements to be serialized according to preferred SOA semantics.
SerializableAttribute	Types marked with the SerializableAttribute are also serializable through the DataContractSerializer. This is useful for legacy types that are in pre-defined assemblies but results in wire-serialization (fields, not properties).
IXmlSerializable	Types that implement this interface are used handle custom XML serialization and deserialization for complex schemas that are not easily represented by other serializable types.
XmlSerializer	When the XmlSerializer is employed instead of the DataContractSerializer, these types can employ XmlSerializer attributes such as the XmlElementAttribute or XmlAttributeAttribute to control type serialization.
Plain Old CLR Object (POCO)	Types that are not marked with any attributes can also be serialized through the DataContractSerializer so long as they do not participate in a hierarchy that involves other serializable types nor reference any serializable types – in other words, all types must be POCO types.

Regardless of the types used to implement the service contract, proxy generation will create a service contract equivalent for the client. This will include a service contract, message contracts if applicable, and fault contracts if applicable. Typically, all serializable types are represented as data contracts at the client – even if they are represented by an alternate serializable type at the service. The exception to this is if the type cannot be represented as a data contract because of a complex schema, in which case the XmlSerializer may be configured for use at the client and all types decorated with XmlSerializer attributes. SvcUtil makes this decision as it inspects the metadata.

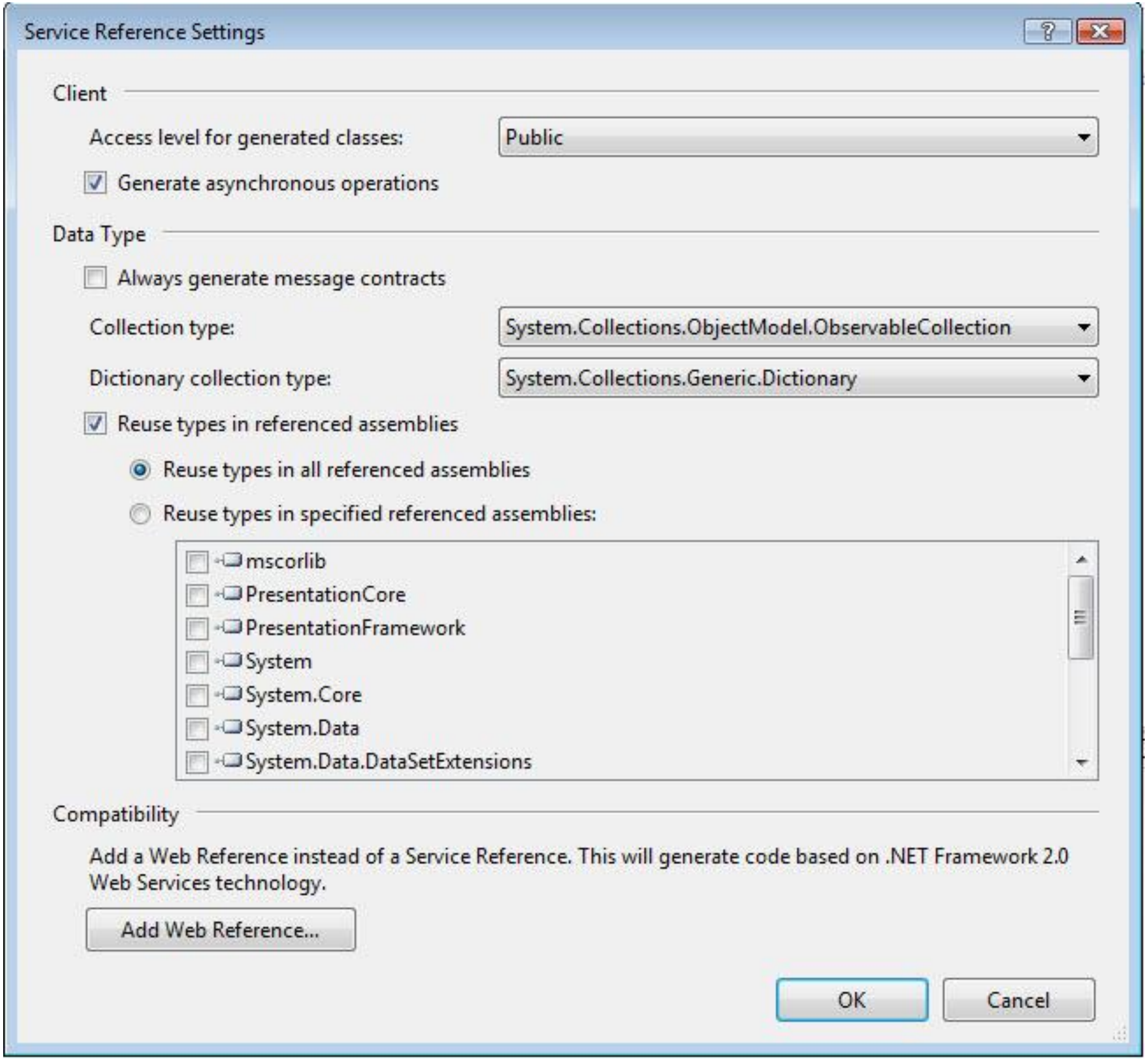
Figure 9 captures the process of proxy generation using SvcUtil assuming the default behavior – generating data contracts for all serializable types.

Figure 9: Proxy generation with SvcUtil



Visual Studio supplies this functionality through Add Service Reference as discussed earlier. In fact, all of the key features are available through the dialog, as shown in Figure 10, making it unnecessary to use SvcUtil from the command line to generate proxies unless you are automating the process or have a special circumstance.

Figure 10: Useful Service Reference settings for WPF clients



Service Reference Settings

Client

Access level for generated classes: Public

☑ Generate asynchronous operations

Data Type

☐ Always generate message contracts

Collection type: System.Collections.ObjectModel.ObservableCollection

Dictionary collection type: System.Collections.Generic.Dictionary

☑ Reuse types in referenced assemblies

● Reuse types in all referenced assemblies

○ Reuse types in specified referenced assemblies:

- ☐ mscorlib
- ☐ PresentationCore
- ☐ PresentationFramework
- ☐ System
- ☐ System.Core
- ☐ System.Data
- ☐ System.Data.DataSetExtensions

Compatibility

Add a Web Reference instead of a Service Reference. This will generate code based on .NET Framework 2.0 Web Services technology.

The following settings from Figure 10 are particularly useful when generating proxies for WPF clients:

- Select “Generate asynchronous operations” to simplify asynchronous calls to the service and improve perceived performance at the client
- Select ObservableCollection for client-side data binding – something I’ll talk more about
- By default the proxy will reuse types in referenced assemblies so don’t forget to add references to any shared libraries containing data contracts and serializable types before generating the proxy

ClientBase<T> and ChannelFactory<T>

The proxy generated by Add Service Reference inherits ClientBase<T> which exposes operations according to the service contract for a particular endpoint, and wraps calls to the inner communication channel. An alternative to this would be to directly construct the channel using ChannelFactory<T>, which provides access to the same service operations but does not provide the functionality of ClientBase<T>. Figure 11 compares the two approaches.

Figure 11: Comparison between ClientBase<T> and ChannelFactory<T>

Feature	ClientBase<T>	ChannelFactory<T>
Metadata	Service metadata including service contracts, data contracts, message contracts and fault contracts are automatically generated for the client. No need to share code.	Assumes service metadata including service contracts, data contracts, message contracts and fault contracts are already available to the client, possibly through shared assemblies.
Contracts and Object Model	Contracts are automatically generated by SvcUtil based on metadata and may not be identical to the object model employed by the service.	Contracts are often available through shared libraries if you own both sides – thus the object model is identical to that used by the service.
Extended Channel Features	Exposes ICommunicationObject and IContextChannel features without the need to cast the proxy reference.	Must cast the channel reference to access ICommunicationObject and IContextChannel features.
Channel Factory Caching	Automatically caches the channel factory based on a limited set of proxy constructors and initialization parameters.	Must manually cache the channel factory. This can yield more flexibility in caching options.
Asynchronous Pattern	Can generate a proxy that exposes asynchronous operations and implements an asynchronous pattern that exposes callback events that are invoked on the UI thread to simplify interaction with control properties.	Can generate a channel that exposes asynchronous operations, invoke operations asynchronously and implement callback events. Callback events are executed on a new thread which means interacting with control properties must be marshaled to the UI thread.

When working with WPF clients, ClientBase<T> yields the following distinct advantages:

- You can expose collections as ObservableCollection<T> directly from the proxy. If you are sharing metadata libraries, it is unlikely that the service contract implements ObservableCollection<T> as its return type for collections.
- The event-based asynchronous pattern marshals results to the UI thread removing the need to do this manually.

Generally speaking, the ClientBase<T> implementation, although sometimes verbose, should do the trick in a simple application. The only drawback is that the client-side object model is not identical to that of the service, and if you own both sides of the implementation it may be preferable for developers to use the same implementation. More complex applications usually need more streamlined exception handling implementations, more control over channel factory caching, and other custom functionality added to their proxies – which implies editing the generated proxy or rolling a custom proxy wrapper.

Sharing Types and Libraries

You have three main options to choose from for proxy generation or working with a custom proxy:

- If you don't own both sides, you are likely to generate a proxy using Add Service Reference (SvcUtil) which will create copies of all service contracts, data contracts and other related types from metadata. This is illustrated in Figure 12.
- You can optionally add a reference to some shared libraries, or pre-created libraries containing data contracts and other serializable types – before you generate the proxy. This way, the generated proxy will use the referenced types as shown in Figure 13.
- You can share libraries that include not only data contracts and serializable types, but also any service contracts, message contracts and fault contracts. In this case you would create a custom proxy using ClientBase<T> or ChannelFactory<T> as shown in Figure 14.

Figure 12: Classic proxy generation using SvcUtil

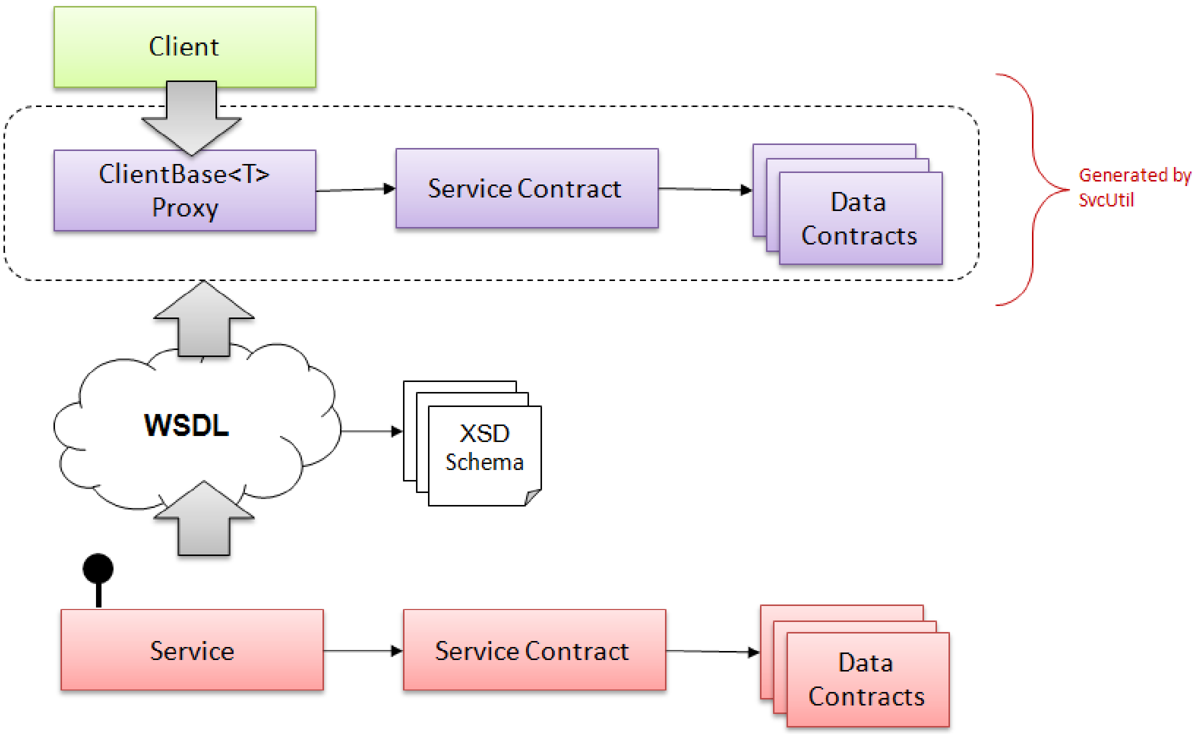


Figure 13: Generating proxies with shared types

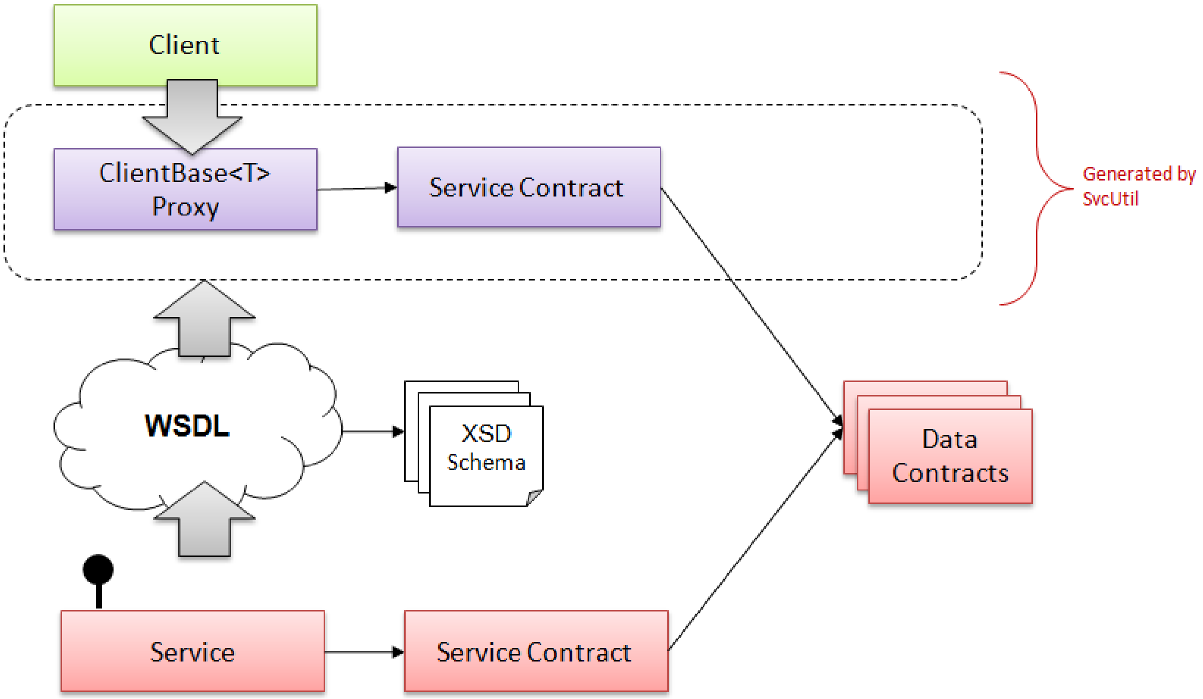
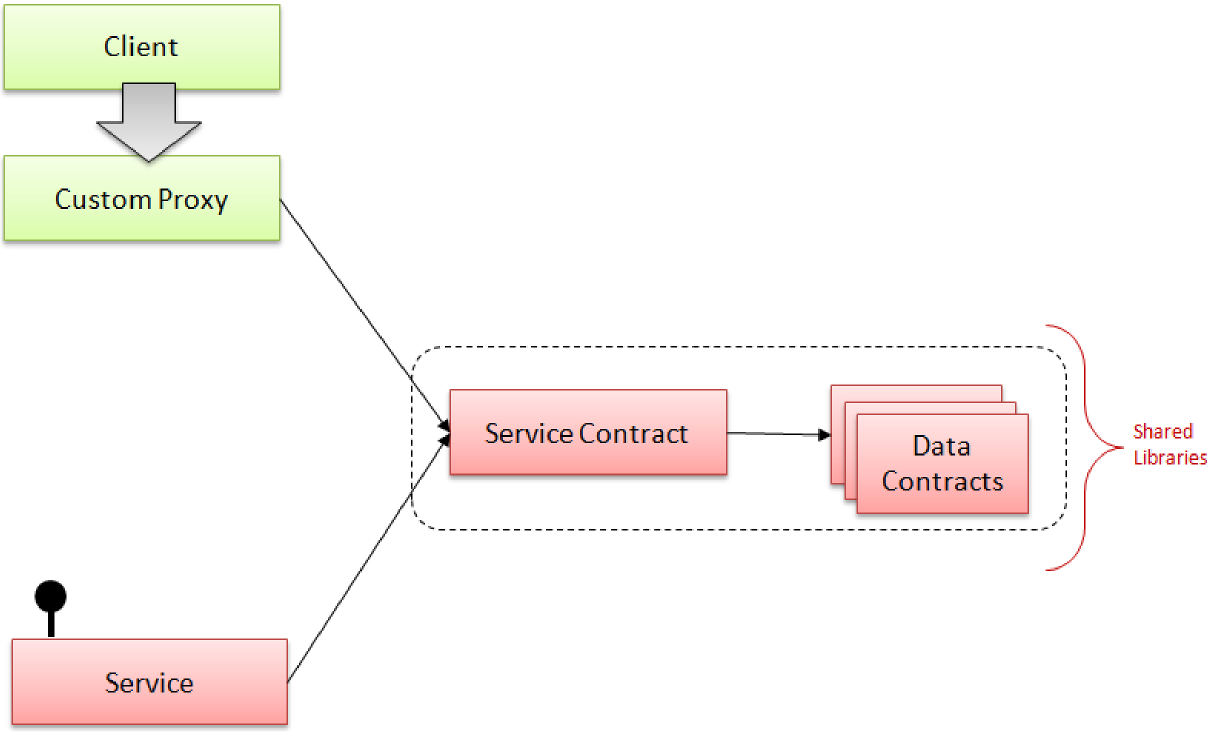


Figure 14: Creating a custom proxy using shared service contracts and related types



As mentioned earlier, proxy generation is usually good enough for simple examples. If you share libraries containing complex types you can generate a proxy using those types. On advantage of this might be sharing validation code on both sides, for example, while still leveraging the benefits of generating the ClientBase<T> implementation.

Chances are, however, that if you own the libraries with complex types you probably also own the service implementation which means you can share all metadata in separate libraries. This approach is good for productivity and version control – if libraries are shared all developers are sure to be using the same contracts.

If you decide to share libraries between service and client developers, you should isolate assemblies containing metadata from those with the service and business implementation. For example, create an assembly for contracts and another for entities. The contracts assembly should contain the service contract and any message contracts or fault contract types. The entities assembly should contain all data contracts or other forms of serializable types used by service contracts and message contracts.

Some of the topics to be discussed in this whitepaper should also be considered if you are sharing metadata libraries between clients and services:

- You should create a synchronous and asynchronous version of the service contract so that clients can use the latter even if the service uses the synchronous version.
- Service contracts will expose List<T> which means clients will have to convert results to ObservableCollection<T> for data binding – instead of deserializing into that collection type.
- Implementations of INotifyPropertyChanged and IDataErrorInfo, frequently used by clients, should only be available to the client code. Use partial classes to achieve this.
- Implement IExtensibleDataObject at the client only. If you implement this interface on types shared by the client and service – be sure to set use the ServiceBehaviorAttribute to disable support for this at the service to prevent denial of service attacks.

Arrays and Data Binding

Collections, lists and arrays are always represented in service metadata as XSD schema arrays. By default, SvcUtil (and Add Service Reference) generates a proxy that represents arrays as **System.Array** – but this is not a particularly friendly programming model compared to other collection types, nor is it helpful for data binding activities. Typically you would choose from **List<T>**, **BindingList<T>** or **ObservableCollection<T>** as the collection type (from Figure 10) as you generate your proxy. Each of these types implements several interfaces that are common to collections such as **IList**, **IEnumerable<T>**, **IEnumerable**, **ICollection<T>**, and **ICollection** – but they differ in the following respects:

- If you aren't doing any data binding, **List<T>** is an obvious choice as it provides an easy to use object model for interacting with collections. It does not, however, supply and benefits for data binding activities.
- **BindingList<T>** is typically chosen to support data binding in Windows Forms applications or for shared libraries that will be used by both WPF and Windows Forms applications.
- **ObservableCollection<T>** is the preferred data binding collection for WPF clients.

As an example, if you bind a WPF DataGrid control to a collection of type **List<T>** the UI is not automatically updated when the list or the items in the list are updated. However, if you bind to **BindingList<T>** or **ObservableCollection<T>**, you need only assign the collection to the ItemsSource property and the UI is updated with the collection. The following code binds an **ObservableCollection<T>** to a DataGrid:

```
ObservableCollection <TodoItem> _TodoItems = new
ObservableCollection<TodoItem>();
_TodoItems = _Proxy.GetItems();
TodoDataGrid.ItemsSource = _TodoItems;
```

Any changes to _TodoItems will be reflected in the DataGrid, for example:

```
_TodoItems.Add(new TodoItem {…}); // UI updated!
```

Change Notifications

When you generate a proxy with Add Service Reference, data contract types implement the **INotifyPropertyChanged** interface. As shown in Figure 15, the set operation for each property raises the PropertyChanged event so that subscribing clients will be notified of the change.

Figure 15: A partial view of the INotifyPropertyChanged implementation in the TodoItem data contract

```
[System.Runtime.Serialization.DataContractAttribute(Name="TodoItem",
Namespace="http://wcfclientguidance.codeplex.com/2009/04/schemas")]
[System.SerializableAttribute()]
public partial class TodoItem : object, IExtensibleDataObject,
INotifyPropertyChanged
{
    // fields

    [System.Runtime.Serialization.DataMemberAttribute(IsRequired=true)]
    public string Title {
        get {
            return this.TitleField;
        }
        set {
            if ((object.ReferenceEquals(this.TitleField, value) != true)) {
                this.TitleField = value;
                this.RaisePropertyChanged("Title");
            }
        }
    }

    // additional members

    public event PropertyChangedEventHandler PropertyChanged;

    protected void RaisePropertyChanged(string propertyName) {
        System.ComponentModel.PropertyChangedEventHandler propertyChanged = 
this.PropertyChanged;
        if ((propertyChanged != null)) {
            propertyChanged(this, new 
System.ComponentModel.PropertyChangedEventArgs(propertyName));
        }
    }
}
```

One way that you can leverage this implementation is to provide a PropertyChanged handler for each item in a collection. If the collection is bound to a WPF DataGrid, as the user makes changes you can handle it by sending updates to the service if applicable. The following code illustrates this:

```
// subscribing to PropertyChanged
foreach(TodoItem item in _TodoItems)
{
  item.PropertyChanged += Item_PropertyChanged;
}

// sending updates to the service
void Item_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    TodoItem item = (TodoItem) sender;
    _Proxy.UpdateItem(item);
}
```

Of course, for very large lists a more appropriate approach might be to send updates in batch rather than as changes are made to individual items.

If you are sharing libraries between client and service, it is best not to use notifications at the service. It is best to keep the implementation in shared libraries to that which both client and service can share – as in the core data contract implementation – and use partial classes to implement client-specific features like INotifyPropertyChanged at the client.

Version Tolerance

Proxies generated with Add Service Reference also create data contract types that implement **IExtensibleDataObject** to support version tolerance. The implementation adds a single member, ExtensionData, to the type as shown here:

```
[DataContract(Namespace = 
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
public class TodoItem : IExtensibleDataObject
{
    // data members

    #region IExtensibleDataObject Members

    public ExtensionDataObject ExtensionData { get; set; }

    #endregion
}
```

The implementation supports scenarios where the client accesses a service that has updated its data contracts to add new data members unknown to earlier versions at the client. Unknown XML elements are preserved in the deserialized instance of the type at the client, so that if the client saves updates to other members, when serializing messages to the service elements in the ExtensionData dictionary will be serialized along with it.

You typically do not want to support this at the service to prevent Denial-of-Service (DoS) attacks, so if you are sharing libraries between clients and services you can implement this interface and suppress the behavior at the service by applying the ServiceBehaviorAttribute with IgnoreExtensionDataObject set to true.

```
[ServiceBehavior(IgnoreExtensionDataObject=true)]
```

Proxy Lifetime Management

You will typically keep each proxy instance around for the lifetime of your WPF application – but, that means your code must also handle recreating the proxy when the communication channel is no longer usable, and look for optimizations such as channel factory caching. This section will focus on techniques for managing proxy lifetime for both **ClientBase<T>** and **ChannelFactory<T>**, in addition to channel factory caching optimizations.

ClientBase<T> Proxy Creation and Disposal

Proxies based on ClientBase<T> hide the specifics of channel factory and channel creation. A reference to a ClientBase<T> proxy therefore is a reference to both the underlying channel factory and channel. To align proxy lifetime with that of the application you typically create the proxy when the main window is created and dispose of it when the main window is closed. This is contrary to the plethora of WCF samples that illustrate creating a proxy and disposing of it during each method call like the following code illustrates:

```
using (TodoListServiceClient proxy = new TodoListServiceClient())
{
    proxy.UpdateItem(item);
}
```

There are a few reasons why the using statement is not recommended for proxy lifetime management. First, it is costly to create the client channel and doing this for every call carries a lot of overhead. In a multithreaded client if each thread creates a new proxy it can severely impact application performance. Second, the call to Dispose() actually calls the proxy’s Close() method – which may fail if the client channel is in a faulted state. This can happen when the channel has a transport session and a problem occurs such as an uncaught exception at the service.

NOTE: Transport sessions are present with TCP and Named Pipe bindings, and with HTTP bindings that have either reliable sessions or secure sessions enabled.

It is better practice to create the proxy and scope it to the application, and when closing the proxy place a try…catch around the call to Close(). If Close() fails, the code should call Abort() to clean up remaining channel resources the proxy has allocated. Figure 16 illustrates this. The proxy is created when the main Window is constructed, and disposed during the Closed event. In the event something causes the client channel to fault, the Faulted event is handled – at which time the proxy is aborted to clean up, and a new instance created. This ensures there is always a proxy available for use. This code makes a few assumptions about the application's requirements:

- That any exceptions causing the channel to fault have already been reported to the user
- That the service does not use in-memory sessions and thus it is acceptable to create a new proxy without data loss from the user's perspective
- That when the application is shutting down, any exceptions thrown when Close() is called are not useful to report

Figure 16: Proxy lifetime aligned with the lifetime of the main application window

```
public partial class MainWindow : Window
{

    private TodoListServiceClient _Proxy;

    public MainWindow()
    {

        InitializeComponent();

        if (!DesignerProperties.GetIsInDesignMode(new DependencyObject()))
        {
            _Proxy = new TodoListServiceClient();
            _Proxy.InnerChannel.Faulted += new
      EventHandler(InnerChannel_Faulted);

            _TodoItems = _Proxy.GetItems();
        }
    }

    void InnerChannel_Faulted(object sender, EventArgs e)
    {
        _Proxy.Abort();
        _Proxy = new TodoListServiceClient();
        _Proxy.InnerChannel.Faulted+=new EventHandler(InnerChannel_Faulted);

    }

    private void AddTodoButton_Click(object sender,
System.Windows.RoutedEventArgs e)
    {
        _Proxy.CreateItem(_NewTodo);
    }

    void Item_PropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        _Proxy.UpdateItem(item);
    }

  private void DeleteTodoItem_Click(object sender,
System.Windows.RoutedEventArgs e)
    {

         _Proxy.DeleteItem(item.ID);
    }

    private void window_Closing(object sender, CancelEventArgs e)
    {
        try
        {
            _Proxy.Close();
        }
        catch
        {
            _Proxy.Abort();
        }

    }
}
```

You may see examples that check to see if the channel is in a faulted state and call Abort() if it is, otherwise call Close():

```
if (_Proxy.State == System.ServiceModel.CommunicationState.Faulted)
    _Proxy.Abort();
else
    _Proxy.Close();
```

Although this approach may work most of the time, there is always a chance that the underlying channel will change to faulted state after you check the state. In that case, Close() will still throw an exception and resource cleanup left incomplete. For these reasons the approach in Figure 16 is better in practice.

If you are using ChannelFactory<T> to create a proxy, rather than the generated type based on ClientBase<T>, the equivalent code to create and dispose of the channel is shown in Figure 17. The key differences in this approach are as follows:

- The channel factory is explicitly cached so that, in the event the channel is faulted, the same channel factory can be used to recreate the channel
- The Faulted event is exposed by the channel
- The proxy must be cast to ICommunicationObject to close or abort the channel.

Calls to service operations using the proxy remain identical to that shown in Figure 16.

Figure 17: Managing proxy lifetime with ChannelFactory<T>

```
public partial class MainWindow : Window
{

    ChannelFactory<ITodoListService> _Factory;
    private ITodoListService _Proxy;

    public MainWindow()
    {

        InitializeComponent();

        if (!DesignerProperties.GetIsInDesignMode(new DependencyObject()))
        {
            _Factory = new ChannelFactory<ITodoListService>("");
            _Proxy = _Factory.CreateChannel();
            _Proxy.Faulted += new EventHandler(InnerChannel_Faulted);

        }
    }

    void InnerChannel_Faulted(object sender, EventArgs e)
    {
        ICommunicationObject co = _Proxy as ICommunicationObject;
        co.Abort();

        _Proxy = _Factory.CreateChannel();
        _Proxy.Faulted += new EventHandler(InnerChannel_Faulted);
    }


    private void window_Closing(object sender, CancelEventArgs e)
    {
        try
        {
            _Proxy.Faulted -= InnerChannel_Faulted;
            ICommunicationObject co = _Proxy as ICommunicationObject;
            co.Close();
        }
        catch
        {
              try
              {
                  ICommunicationObject co = _Proxy as ICommunicationObject;
                  co.Abort();
              }
              catch {}
        }

        try
        {
            _Factory.Close();
        }
        catch
        {
              try
              {
                  _Factory.Abort();
```

```
                }
                catch {}
            }
        }

}
```

Although attaching the lifetime of the proxy to the application is ideal, there could be reason to create a new instance of the proxy before the application is closed, including:

- As mentioned earlier, the underlying channel could be put into a faulted state causing the proxy to be unusable
- If your application supports logging in an alternate user – this requires constructing a new proxy since the channel is immutable
- For the same reason, if any other facet of the channel must be modified during the lifetime of the application, a new proxy with those features must be created

Figure 16 and Figure 17 illustrate recreating the channel when the Faulted event is handled, but you may also explicitly recreated the channel for any of these reasons. If this is the case, you must remember to close or abort the previously created proxy, which means both the factory and channel if using ChannelFactory<T>, to clean up those resources.

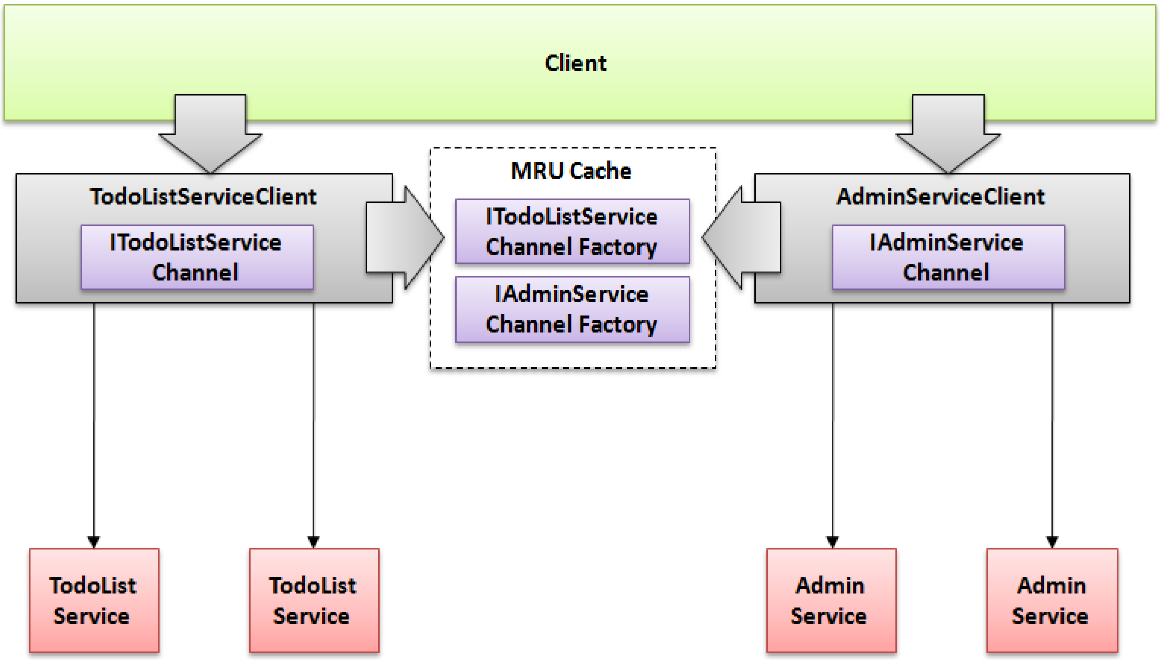
ChannelFactory<T> and MRU Caching

Although you may try to hold on to each proxy for the lifetime of the application, you can optimize the cost to recreate the channel by caching the channel factory. Figure 17 showed the code to do this for a single proxy by maintaining a reference to the factory separate from the actual channel (proxy) reference. In .NET 3.5 SP1 a new feature was introduced to automatically cache the channel factory in the Most Recently Used (MRU) channel cache when you create proxies using ClientBase<T>.

The MRU cache automatically stores channel factories for each ClientBase<T> proxy you create in the application domain based on common initialization properties. Thus, you can write code to recreate the proxy and the underlying code will search for a channel factory that matches those initialization parameters – creating a new channel factory only if necessary.

Suppose the Todo List application also exposed an AdminService – Figure 18 illustrates how the client, while creating the generated ClientBase<T> proxy for each service, also generates an entry in the MRU cache for each factory type.

Figure 18: Populating the MRU cache while generating ClientBase<T> proxies



In reality, caching the channel (proxy) is what yields the best performance for a client application – in particular if it is multithreaded. Where channel factory caching comes in handy is when proxies must be recreated for timeouts or faulted channels in a multithreaded application. The MRU cache feature removes the need to manually cache the channel factory and build a custom proxy – when ClientBase<T> is doing the job just fine.

Exception Handling

When you want to communicate with a WCF service you will typically construct the proxy, keep it alive for the lifetime of the application, and make calls to service operations using the same proxy instance. If the operation throws an exception, it is converted into a SOAP fault – the interoperable standard for returning exceptions from a service. In a perfect situation you can use the same proxy to make calls to services operations even after an exception is thrown. Depending on the nature of the communication channel, and the type of exception thrown, you may have to contend with faulted channels and proxy recreation semantics.

This section will review the types of exceptions that a service can throw, the client-side semantics for handling service exceptions, and issues related to session expiry and faulted channels that impact proxy lifetime.

Exceptions and Faults

Services have a few options for reporting exceptions to client applications:

- They can allow CLR exceptions to flow up to the service channel
- They can trap CLR exceptions and throw a simple fault exception in its place
- They can trap CLR exceptions and throw a detailed fault exception

Allowing CLR exceptions to propagate to the client is not recommended. By default this results in a very general fault being reported to the client, and is not useful for the application or the end user to act on. Furthermore, uncaught exceptions are considered a risk to the service channel and thus the channel is faulted. In the presence of a transport session (TCP, Named Pipes or HTTP with secure sessions or reliable sessions enabled) this renders the proxy unusable.

Throwing a simple fault implies using FaultException to throw the fault as follows:

```
throw new FaultException("Invalid operation.");
```

This is useful for reporting a simple error message to the client, without faulting the channel. The client can catch this as a FaultException as follows:

```
catch (FaultException faultex)
{
  MessageBox.Show(faultex.Message);
}
```

Throwing a detailed fault implies using the generic type FaultException<T> to throw the fault from the service – where T is a data contract or other serializable type describing additional details to be returned to the client beyond a simple error message. Figure 19 illustrates a simple example of a data contract used as the fault detail, and the code to throw the fault.

Figure 19: Throwing FaultException<T> at the service

```
[DataContract(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
public class FaultDetail
{
    public string OriginalExceptionType { get; set; }
    public string OriginalExceptionMessage { get; set; }
}


throw new FaultException<FaultDetail>(new FaultDetail { OriginalExceptionType
= "InvalidOperationException", OriginalExceptionMessage = "Unable to update
item." }, "Invalid item ID.", FaultCode.CreateSenderFaultCode("SenderFault",
"http://wcfclientguidance.codeplex.com/2009/04"));
```

Typically, when the service implements an exception handling model that includes throwing detailed faults, operations will include a FaultContractAttribute for each type of FaultException<T> that can be thrown by the operation. Fault contracts are included in the WSDL description so that clients can generate proxies that include the fault contract and a copy of the data contract matching the fault detail type. The client can then catch detailed faults as follows:

```
catch (FaultException<FaultDetail> faultex)
{
  string errorMessage = faultex.Message;
  FaultDetail detail = faultex.Detail;

  MessageBox.Show(string.Format("Error message: {0}, Detail:  {1}",
errorMessage, detail));
}
```

If the service throws exceptions with details, this probably means that the service designer thinks the detail is important – beyond a simple error message. At a minimum, services should throw a simple FaultException instead of allowing exceptions to propagate to fault the communication channel.

Exceptions and Proxy Lifetime

If the communication channel between clients and services does not involve a transport session, proxy lifetime is not impacted by certain types of exceptions. The following scenarios involve session, and therefore require special consideration:

- Intranet communications over TCP (NetTcpBinding) or Named Pipes (NetNamedPipeBinding) involve a socket or named pipe – both being transport sessions.
- Internet communications over HTTP that include either secure sessions or reliable sessions. By default WSHttpBInding and WS2007HttpBinding enables secure sessions, WSDualHttpBInding requires reliable sessions, and WSFederationHttpBInding and WS2007FederationHttpBinding requires secure sessions.

BasicHttpBinding and WebHttpBinding are not included in this list since they do not support advanced web service protocols.

In the presence of sessions there are two key concerns: session timeouts and uncaught CLR exceptions. By default, sessions have a lifetime of ten minutes, after which time the service channel will timeout and be put into a faulted state. Figure 20 illustrates what happens when the service channel times out. When the proxy accesses the service the timer begins (1). If the client does not call the service within the session timeout period the service channel is faulted (2) which means the session no longer exists at the service. On the next call the service, the call will fail since the session is gone, the client channel will be faulted, and a communication exception reported to the client (3) indicating the source of the timeout. There isn't a specific timeout exception – instead the channel that owned the session is the source of the fault. From here on in if the client tries to use the proxy a communication exception will be reported indicating that the channel is faulted (4).

Figure 20: The impact of timeout exceptions on the client proxy

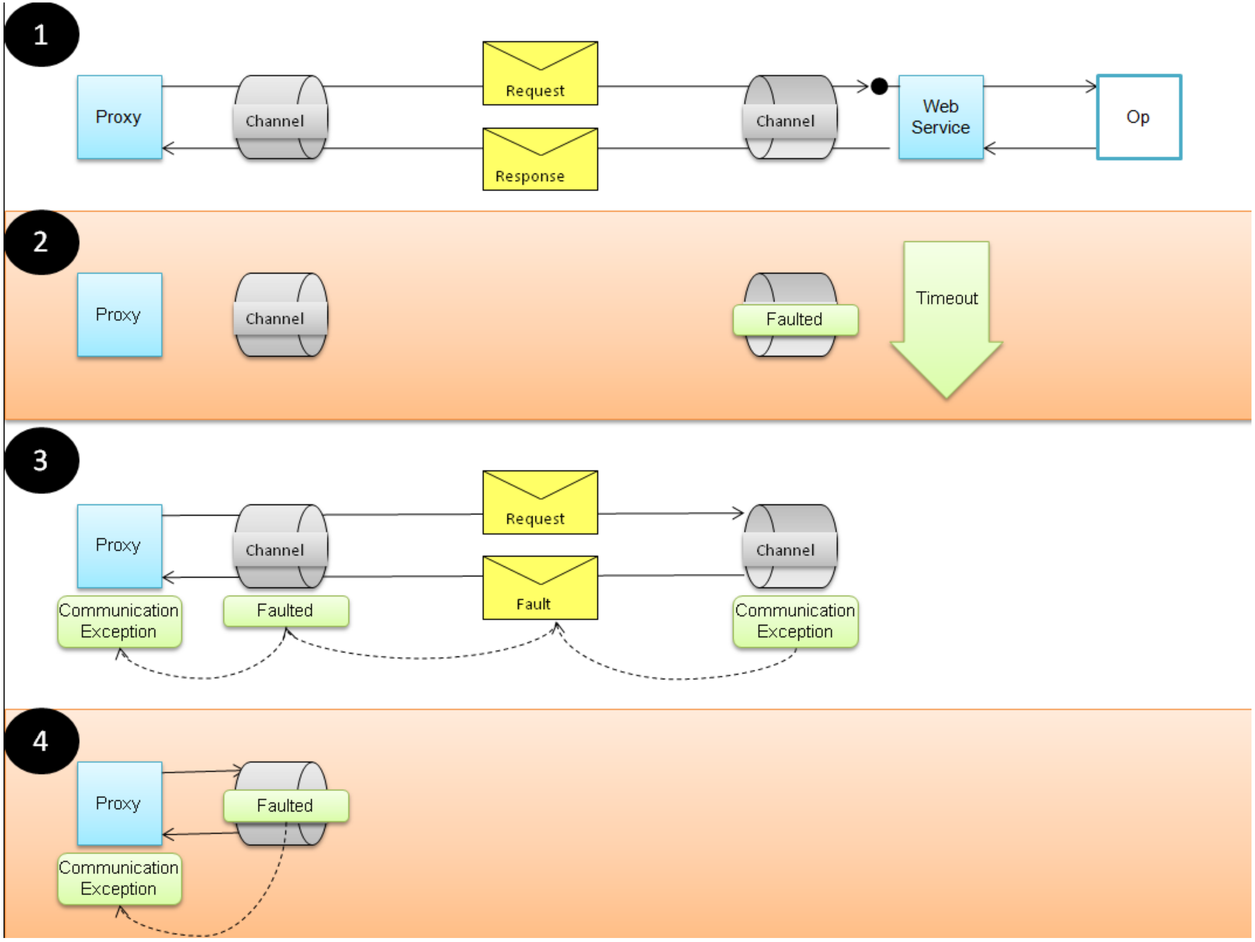
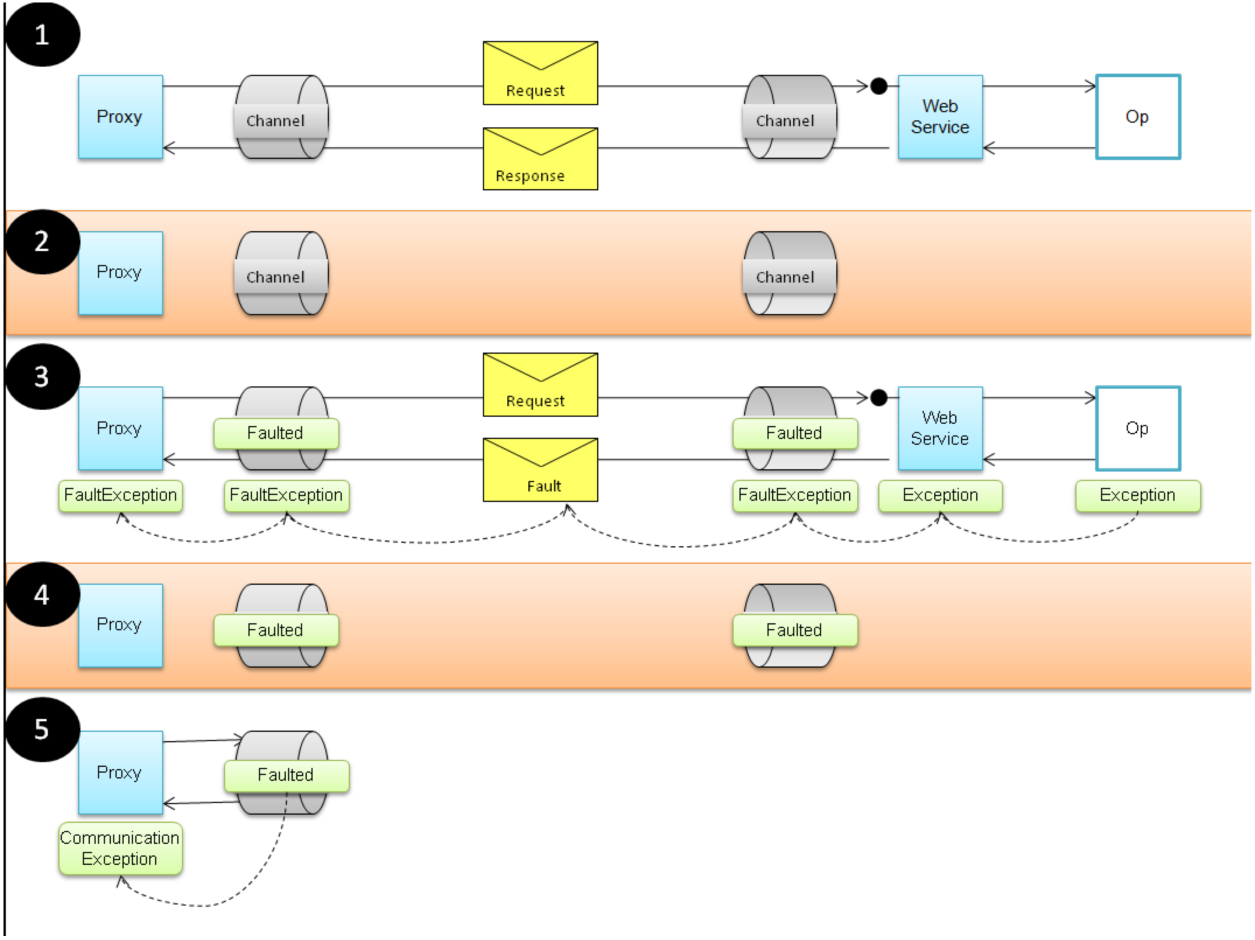


Figure 21 captures a similar flow for the case where an uncaught exception is thrown at the service. The session is established when the proxy makes the first call (1) and both ends of the session remain intact so long as there isn't a session timeout (2). If the proxy calls a service operation that throws an uncaught exception, the service channel is faulted and a generic fault is reported to the client channel, which in turn reports the general exception to the calling client code (3). Both the service and client channel are in a faulted state at this point (4), thus subsequent calls to the proxy result in a communication exception (5).

Figure 21: The impact of uncaught exceptions on the client proxy



If the client wants to provide a pleasant (and useful) experience for the end-user, the following should be done when sessions are present. When making a call with a proxy, if a CommunicationException is thrown – and it is not a FaultException – in order to hide the session timeout from the end-user create a new proxy and try the call again. If it still fails – you have a bigger problem that requires the user's attention. If it succeeds, you may have had a session timeout or a faulted channel resulting from a one-way call – the latter of which would not be reported until the next call. An example of how this should work is shown here:

```
try
{
   // use the proxy here
}
catch (CommunicationException comEx)
{
   FaultException faultEx = comEx as FaultException;
   if (faultEx != null)
   {
       throw;
   }

   // use the proxy again here, if it throws, let it
}
```

You should also handle the client channel's Faulted event and recreate the proxy when the event is triggered. The user will still be presented with any exceptions that occur at the service – causing the faulted channel – but they will seamlessly be presented with a new channel to continue working with the service. This approach is particularly important when calling services that unfortunately do not convert uncaught exceptions to faults. In the previous section of this whitepaper, Figure 16 and Figure 17 illustrate how to handle the Faulted event and recreate the proxy using ClientBase<T> and ChannelFactory<T> scenarios respectively.

As you can imagine, all of this exception handling can cause significant clutter in the client code. Fortunately, you can create an exception handling proxy wrapper that handles recreating the channel when it is faulted. A sample has been provided to illustrate this.

Asynchronous Calls and Multithreading

Client applications can be optimized with asynchronous calls and multithreading. Delegating work to threads other than the UI thread can improve perceived performance since the user is able to continue interacting with the UI while other work is being done. The downside, of course, is that this requires some knowledge of multithreading and synchronization techniques. This section will not become a crash course in multithreading as that would require another whitepaper, however it will provide you with some tips directly related to client that call WCF services including the most effective way to call services asynchronously, and considerations for multithreaded clients that consume services.

Asynchronous Proxies

Although individual calls to service operations may not impact the perceived performance of your WPF client applications, applications that must retrieve large amounts of data or make frequent service calls can benefit from making those calls asynchronously. Asynchronous calls free up the UI thread to process Windows messages so that users can keep working while as remote calls are in progress.

Proxies generated with Add Service Reference support two modes of asynchronous operations: delegate-based and event-based. Delegate-based asynchronous operations follow the traditional asynchronous delegate pattern whereby each call is broken into two: a begin operation, and an end operation. The following illustrates such an implementation for the TodoListService proxy for the GetItems() operation:

```
public IAsyncResult BeginGetItems(AsyncCallback callback,
object asyncState)
{
  return base.Channel.BeginGetItems(callback, asyncState);
}

public ObservableCollection<TodoItem> EndGetItems(IAsyncResult result)
{
  return base.Channel.EndGetItems(result);
}
```

When the client code calls BeginGetItems() the service call is executed on a separate thread. Typically, you will pass a delegate to the AsyncCallback parameter to be notified when the call is complete – at which time you will call EndGetItems() to retrieve the result. Figure 22 illustrates this. The callback will execute on the thread that executed the call – which is not the UI thread. As such, to interact with the UI any work should be posted to the UI thread. One effective way to do this is to expose access to the synchronization context of the main Window so that the callback can use that to send or post messages for execution. As shown in Figure 22, the main Window can expose a member called _SyncContext that represents the Window's synchronization context. The SynchronizationContext instance exposes a Send() and Post() method – the former executes synchronously, the latter, asynchronously.

Figure 22: Calling service operations using the asynchronous delegate pattern

```
// Synchronization context for the main Window
private SynchronizationContext _SyncContext = SynchronizationContext.Current;

// invoke the service asynchronously
_Proxy.BeginGetItems(OnGetItemsCallback, null);

// Asynchronous callback
private void OnGetItemsCallback(IAsyncResult result)
{
    if (result.IsCompleted)
    {

        this._SyncContext.Send(state =>
            {
                _TodoItems = _Proxy.EndGetItems(result);
                foreach (TodoItem item in _TodoItems)
                {
                    item.PropertyChanged += Item_PropertyChanged;
                }
                TodoDataGrid.ItemsSource = _TodoItems;

            }, null);
    }
}
```

The drawback of the delegate-based asynchronous pattern is that every callback must synchronize access to UI components, and to other instance members belonging to the main Window. This requires developers to write synchronization logic which quickly adds to code clutter, and can become hard to follow. An alternative to this is to use the event-based approach.

Asynchronous proxies also expose an asynchronous call for each service operation – with the suffix "Async" – and an event handler which is fired when the call is completed. For the GetItems() operation two implementations of GetItemsAsync() provided – one that takes a state parameter in the event the client is multithreaded and must associate responses with a particular call:

```
public void GetItemsAsync() {…}
public void GetItemsAsync(object userState) {…}
```

The default implementation fires the GetItemsCompleted event, which passes a strongly typed set of event arguments that includes the expected result of the call as a parameter:

```
public event EventHandler<GetItemsCompletedEventArgs> GetItemsCompleted;
```

Figure 23 illustrates the relevant client code require to use this event-based pattern. Note that the synchronization context is not longer required since the GetItemCompleted event is executed on the UI thread. This logic is encapsulated in the proxy.

Figure 23: Calling service operations using the event-based asynchronous pattern

```
// Suscribing to the completed event
_Proxy.GetItemsCompleted += new
EventHandler<GetItemsCompletedEventArgs>(_Proxy_GetItemsCompleted);

// Calling the operation asynchronously
_Proxy.GetItemsAsync();

// Handling the completed event
void _Proxy_GetItemsCompleted(object sender, GetItemsCompletedEventArgs e)
{
    _TodoItems = e.Result;
    foreach (TodoItem item in _TodoItems)
    {
        item.PropertyChanged += Item_PropertyChanged;
    }
    TodoDataGrid.ItemsSource = _TodoItems;
}
```

Clearly the second approach greatly simplifies the client experience. In fact, there is very little reason to use the former delegate-based approach. So why does SvcUtil generate both patterns in the proxy? In some very special cases, you may want to execute code that does NOT interact with the UI on the callback thread first, before posting code to execute on the UI thread. Any time work that does not involve the UI can perform on another thread, UI performance is improved. In this case you can leverage the delegate-based pattern to exercise more control.

Multithreading Considerations

Sometimes making asynchronous calls to services isn't sufficient to improve the perceived performance of your WPF client applications. Client applications sometimes keep make numerous concurrent calls to keep the UI up-to-date by polling services, and have other processing logic that must be performed alongside service calls. In short, sometimes the client must manage its own threads and perform work to communicate with services from those non-UI threads. This usually means that synchronous operations exposed by the proxy can be used, instead of either of the asynchronous patterns discussed in the previous section – after all, the code is running on a separate thread.

You have some options for creating custom threads including asynchronous delegates, custom threads, and new features of the Task Parallel Library and PLINQ – available in CTP format at the time of this writing for .NET 3.5 SP1. These approaches are summarized in Figure 24.

Figure 24: A short comparison of multithreading approaches

Multithreading Approach	Characteristics
Asynchronous Delegates	Asynchronous delegates execute on a thread from the thread pool and are throttled up to the thread pool limit (the default is usually 25). All code is executed on that thread, including any callback notifications when the thread has finished executing. Calls to update UI elements must be executed on the UI thread, and code that interacts with shared resources such as member variables must be synchronized while the thread is executing and during callback.
Custom Threads	Custom threads execute on a new thread that is not taken from the thread pool – thus the throttle limit is not applicable. All code is executed on that thread, and you must write custom code, usually involving wait handles, to interact with the thread and discover when the thread is has completed execution. There are no callbacks built-in. This approach provides a much finer grained control over threads but also requires additional expertise.
Task Parallel Library and PLINQ	This library simplifies multithreading by reducing the number of lines of code necessary to kick off a new thread, and manage the thread pool and number of concurrent calls. In addition, the underlying code automatically takes advantage of multi-core and multiprocessor systems by distributing work among them. PLINQ is a special aspect of this library that enables you to iterate of a collection of items and all code related to each iteration is executed in parallel on separate threads. The iteration is blocking until all threads are complete. This is a really clean way to do things in parallel that might involve a remote service call.

Regardless of your approach to multithreading you must consider the following optimizations where WCF services are concerned:

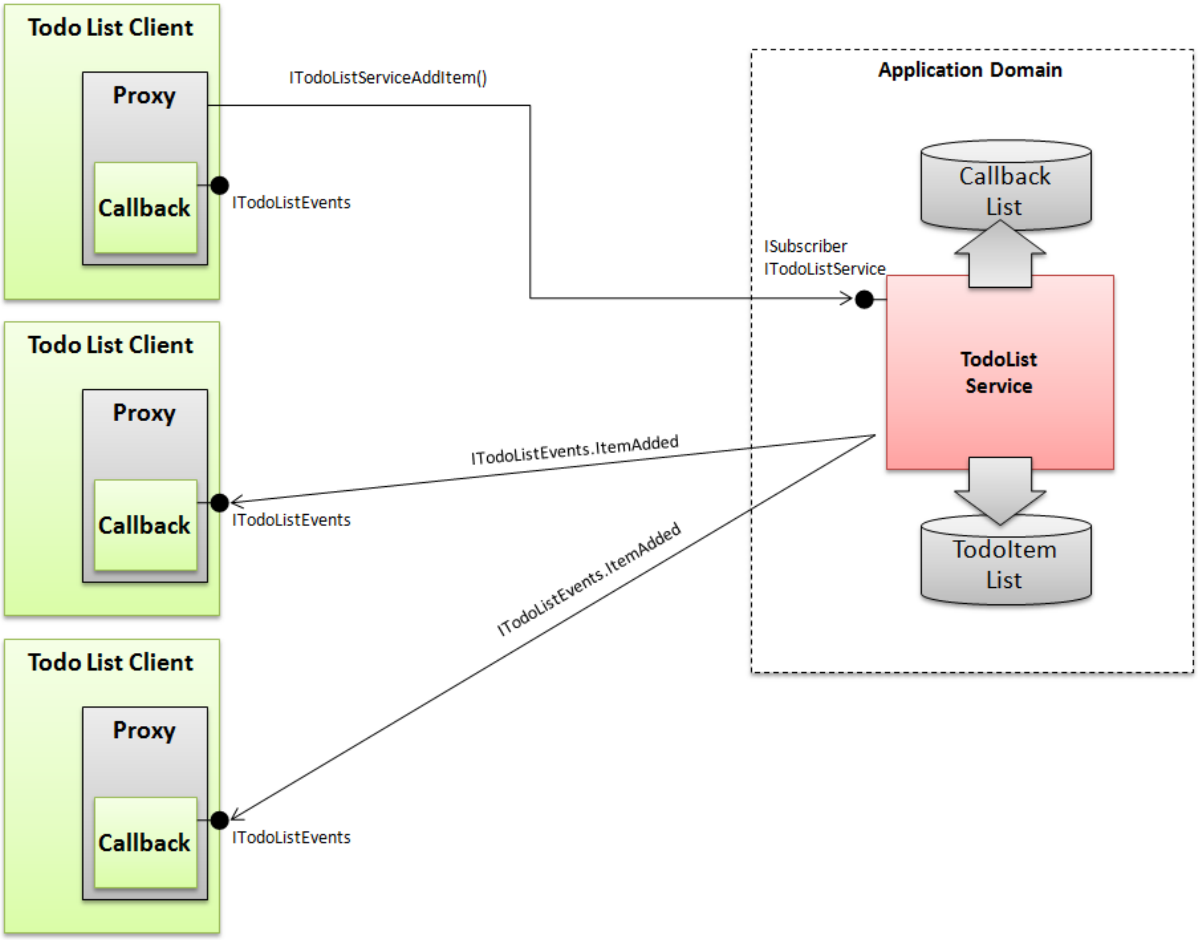
- You should share the same proxy instance with all threads – so long as each thread doesn't require different channel settings such as address or credentials. This reduces the overhead of creating a new channel on each thread – which can really slow performance in a multithreaded client!
- Use the MRU cache or provide a custom channel factory caching solution to reduce overhead on each thread in the event a new channel must be created to recover from session expiry or uncaught exceptions at the service. Remember that this is only likely to be an issue if a transport session is being used.
- Always call Open() on the proxy or channel before any of the threads begins making calls to service operations. The first call to a service operation will call Open() for you, however it will cause all other threads to queue, waiting for the first call to complete. If you call Open() ahead

of time, calls can execute concurrently using the same proxy instance. Be sure to lock the call to Open() and maintain a variable indicating if the channel is already open. Don't rely on the channel's State property since this can change immediately after you check the value.

Duplex Communications and Callbacks

Services may define callback contracts when service operations are long running and warrant out-of-band notifications, or when implementing a transient publish and subscribe (pub-sub) scenario. Consider the example shown in Figure 25. In this scenario the Todo List application incorporates a transient pub-sub design using duplex communication. If one of the Todo List clients adds, updates, or deletes an item - other subscribing clients are notified of the change.

Figure 25: Transient pub-sub implementation using callbacks



In this section I'll describe the relevant characteristics and recommendations for implementing callbacks between WPF clients and WCF services – in the context of this scenario.

Service Design

In this transient pub-sub implementation the service maintains a list of subscribers in the application domain. Since duplex communication relies a transport session, the service can count on each subscribing client communicating with the same service channel – so in a distribute environment with multiple application domains each will maintain its own subscriber list.

A transport session is provided for all endpoints that support duplex communication including NetTcpBinding, NetNamedPipeBinding or WSDualHttpBinding. The latter simulates a transport session using reliable sessions.

In the TodoListService implementation there are three contracts:

- ITodoListService defines core service functionality
- ISubscriber defines subscribe and unsubscribe functionality
- ITodoListEvents defines events for the callback contract

These contracts are shown in Figure 26. Note that the ITodoListEvents contract is associated with ITodoListService as the callback contract. Also note that ITodoListService inherits ISubscriber – so that a single duplex endpoint can be exposed for the service as follows:

```
<endpoint address="TodoListService" binding="wsDualHttpBinding"
contract="Contracts.ITodoListService" />
```

Figure 26: Service contracts for the duplex TodoListService

```
[ServiceContract(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04")]
public interface ISubscriber
{
    [OperationContract]
    void Subscribe();

    [OperationContract]
    void Unsubscribe();
}


[ServiceContract(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04")]
public interface ITodoListEvents
{
    [OperationContract(IsOneWay = true)]
    void ItemAdded(TodoItem item);

    [OperationContract(IsOneWay = true)]
    void ItemChanged(TodoItem item);

    [OperationContract(IsOneWay = true)]
    void ItemDeleted(string id);
}

[ServiceContract(Namespace="http://wcfclientguidance.codeplex.com/2009/04",
CallbackContract=typeof(ITodoListEvents))]
public interface ITodoListService: ISubscriber
{
    [OperationContract]
    List<TodoItem> GetItems();

    [OperationContract]
    string CreateItem(TodoItem item);
    [OperationContract]
    void UpdateItem(TodoItem item);
    [OperationContract]
    void DeleteItem(string id);
}
```

The TodoListService implementation of ISubscriber is shown in Figure 27. Access to the subscriber list is protected with a lock since multiple threads can access the static member. Very simply this implementation adds or removes the calling subscriber's callback contract from the list.

Figure 27: TodoListService implementation of the ISubscriber contract

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall,
ConcurrencyMode=ConcurrencyMode.Multiple)]
public class TodoListService: ITodoListService, ISubscriber
{
    public static object _SubscriberLock = new object();
    public static List<ITodoListEvents> _TodoListEventSubscribers = new
List<ITodoListEvents>();

    public void Subscribe()
    {
        ITodoListEvents callback =
OperationContext.Current.GetCallbackChannel<ITodoListEvents>();

        lock (this.m_subscriberLock)
        {
            if (!_TodoListEventSubscribers.Contains(callback))
            {
                _TodoListEventSubscribers.Add(callback);
            }
        }
    }

    public void Unsubscribe()
    {
        ITodoListEvents callback =
OperationContext.Current.GetCallbackChannel<ITodoListEvents>();
        lock (this.m_subscriberLock)
        {
            if (_TodoListEventSubscribers.Contains(callback))
            {
                _TodoListEventSubscribers.Remove(callback);
            }
        }
    }
}
```

When a particular client calls one of the service operations to add, update or delete an item it notifies the remaining subscribers of the change. A simple way to do this is to traverse the subscriber list and invoke the appropriate callback operation for each callback channel – omitting that of the calling client. The following code illustrates this code for the CreateItem() operation:

```
ITodoListEvents callback =
OperationContext.Current.GetCallbackChannel<ITodoListEvents>();
foreach (ITodoListEvents cb in _TodoListEventSubscribers)
{
    if (cb != callback)
    {
        cb.ItemAdded(item);
    }
}
```

This code can be improved by doing the following:

- The service should publish callback events on a separate thread so that the current call can return
- Any exceptions thrown while publishing should be caught since the service shouldn't care about exceptions thrown during a callback operation
- In the event one of the callback channels is no longer available, the service should remove it from the subscriber list
- Traversal of the subscriber list should be protected from concurrent access in the event the list is changed during publishing

These improvements are shown in the Publish method shown in Figure 28.

Figure 28: Publishing on a separate thread at the service

```
private void Publish(TodoListEvent todoListEvent, TodoItem item)
{
    ITodoListEvents callback =
OperationContext.Current.GetCallbackChannel<ITodoListEvents>();

    Thread t = new Thread(x =>
    {
        List<ITodoListEvents> toremove = new List<ITodoListEvents>();

        lock (_SubscriberLock)
        {
            foreach (ITodoListEvents cb in _TodoListEventSubscribers)
            {
                if (cb.GetHashCode() != x.GetHashCode())
                {
                    Console.WriteLine("Sending event to {0}",
cb.GetHashCode());
```

```
                    try
                    {
                        if (todoListEvent == TodoListEvent.ItemAdded)
                            cb.ItemAdded(item);
                        else if (todoListEvent == TodoListEvent.ItemAdded)
                            cb.ItemChanged(item);
                        else if (todoListEvent == TodoListEvent.ItemDeleted)
                            cb.ItemDeleted(item.ID);
                    }
                    catch (Exception ex)
                    {
                        FaultException faultex = ex as FaultException;
                        if (faultex == null)
                        {
                            Console.WriteLine("Callback failed, removing 
{0}", cb.GetHashCode());
                            toremove.Add(cb);
                        }
                    }
                }
            }
            if (toremove.Count > 0)
            {
                foreach (ITodoListEvents cb in toremove)
                {
                    _TodoListEventSubscribers.Remove(cb);
                }
            }
        }

    });
    t.Start(callback);

}
```

Implementing the Callback Contract

When the client generates a proxy for a duplex endpoint it generates a copy of the service contract in addition to the callback contract. The proxy inherits DuplexClientBase<T> instead of ClientBase<T> and a bunch of functionality is provided therein to facilitate processing callbacks at the client. An important part of this is the implementation of the callback contract.

It is usually best to implement the callback contract on a separate type for manageability. The following is a partial view of a simple callback contract implementation for the TodoListService:

```
public class TodoListCallback: ITodoListServiceCallback
{
    public void ItemAdded(TodoItem item)
    {…}

    public void ItemChanged(TodoItem item)
    {…}

    public void ItemDeleted(string id)
    {…}
}
```

To initialize the proxy, the callback contract type is constructed and wrapped in an InstanceContext type – the latter supplies the host site for the client to receive callbacks. The InstanceContext is passed to the proxy constructor when it is initialized as follows:

```
TodoListCallback callback = new TodoListCallback(this);
_Proxy = new TodoListServiceClient(new InstanceContext(callback));


private DuplexChannelFactory<ITodoListService> _Factory;
private ITodoListService _Proxy;
```

That summarizes the basics: construct a callback object and pass it to the duplex proxy instance. From there the lifetime of the callback object is tied to that of the proxy – as you would expect. There are, however, some recommended practices to consider for the callback type definition:

- Access to the application Window
- Execution on a separate thread from the UI
- Handling UI updates from another thread
- Exception handling

Access to the main window, or some other application Window, may be necessary if callback operations update the UI. One pattern for this is to pass a reference to the Window that the callback will interact with in the constructor:

```
private MainWindow _MainWindow;

public TodoListCallback(MainWindow window)
{
  _MainWindow = window;
}
```

Callbacks should not execute on the UI thread. Although this removes concurrency issues it has a negative impact on UI performance since callback messages are processed through the message pump along with other windows messages. It is best if the callback object throttles messages and forwards relevant calls to the UI thread as needed. For this reason, the callback object should use the **CallbackBehaviorAttribute** and set **UseSynchronizationContext** to false. By default, **ConcurrencyMode** is set to **ConcurrencyMode.Single**. The **CallbackBehaviorAttribute** is applied to the callback type definition:

```
[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Single,
UseSynchronizationContext=false)]
public class TodoListCallback: ITodoListServiceCallback
```

Callback operations must, however, interact with UI elements on the UI thread. One effective way to do this is to expose access to the synchronization context of the main Window as discussed earlier with asynchronous calls. The callback object can use the synchronization context to send or post messages for execution. For this example, the main Window of the Todo List Client application includes this member to expose the synchronization context:

```
public SynchronizationContext _SyncContext = SynchronizationContext.Current;
```

When the callback object receives an event it can access this _SyncContext member and send a delegate to execute on its thread. The following illustrates how the AddItem event sends code to execute on the main Window's thread:

```
public void ItemAdded(TodoItem item)
{
  _MainWindow._SyncContext.Send(state =>
  {
    _MainWindow._TodoItems.Add(item);
  }, null);

}
```

Another important consideration for callback contracts is exception handling. If the callback operation throws an uncaught exception, this faults the communication channel – even if the callback operation is one-way – thus the client's proxy and callback object will no longer be usable. It is always best practice to throw faults instead of exceptions from services and from callback objects. The following illustrates an update to the ItemAdded event that catches exceptions and converts them to a simple FaultException:

```
public void ItemAdded(TodoItem item)
{
    try
    {
        _MainWindow._SyncContext.Send(state =>
        {
            _MainWindow._TodoItems.Add(item);
        }, null);

    }
    catch (Exception ex)
    {
        throw new FaultException(ex.Message);
    }
}
```

Managing Duplex Proxy Lifetime

In a perfect world, when you create a duplex proxy in support of two-way communications between client and service – the client and server channel would remain active until the user terminates the application. Of course, this is not the case. Any of the following problems can play havoc on your communication channel:

- A server machine fails or the application domain with the server channel is shut down or recycled.
- The service or callback instance throws an uncaught exception which ultimately faults the channel on both ends.

The client must therefore not only manage the lifetime of the duplex proxy, but handle subscription to the service and collection of any server data created when communications were down. The typical steps for duplex proxy lifetime management are as follows:

- Create the proxy when the application starts and keep it alive for the duration of the application which usually means tying lifetime to the main application Window
- Close or abort the proxy when the application Window is closed
- Recreate the proxy when the client channel is faulted, which includes either providing a new callback instance or passing the same instance to the proxy once again
- Each time the proxy is created or recreated, subscribe once again to the service and retrieve the latest data

Figure 29 illustrates these practices. InitializeProxy() and GetData() are called when the Window is created, and if the proxy is faulted or closed unsuspectingly. The proxy handles both the Faulted and Closed events because with a duplex channel, there are times that the Faulted event is not fired but the proxy is closed due to an uncaught exception from a callback operation. Also notable is that both the Faulted and Closed handlers may execute on the callback thread if a problem originates from that thread - and so both handlers post their work to the UI thread.

Figure 29: Duplex proxy lifetime management with DuplexClientBase<T>

```
public partial class MainWindow : Window
{

    private TodoListServiceClient _Proxy;

    public MainWindow()
    {
        if (!DesignerProperties.GetIsInDesignMode(new DependencyObject()))
        {
            InitializeProxy();
            GetData();
        }
    }

    private void GetData()
    {
        try
        {
            _TodoItems = _Proxy.GetItems();
            foreach (TodoItem item in _TodoItems)
            {
                item.PropertyChanged += Item_PropertyChanged;
            }

            TodoDataGrid.ItemsSource = _TodoItems;
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }

    private void InitializeProxy()
    {
            try
            {
                TodoListCallback callback = new TodoListCallback(this);

                _Proxy = new TodoListServiceClient(new
InstanceContext(callback));
                _Proxy.InnerDuplexChannel.Closed += new
EventHandler(InnerDuplexChannel_Closed);
                _Proxy.InnerDuplexChannel.Faulted += new
EventHandler(InnerDuplexChannel_Faulted);

                _Proxy.Subscribe();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }

    }

    void InnerDuplexChannel_Closed(object sender, EventArgs e)
    {

        this._SyncContext.Post(state =>
        {
            InitializeProxy();
            GetData();

        }, null);

    }

    void InnerDuplexChannel_Faulted(object sender, EventArgs e)
    {

        this._SyncContext.Post(state =>
        {
            _Proxy.Abort();

            InitializeProxy();
            GetData();
        }, null);

    }

    private void window_Closing(object sender, CancelEventArgs e)
    {

        try
        {
            _Proxy.InnerDuplexChannel.Closed -= InnerDuplexChannel_Closed;
            _Proxy.InnerDuplexChannel.Faulted -= InnerDuplexChannel_Faulted;
            _Proxy.Unsubscribe();
            _Proxy.Close();
        }
        catch (Exception)
        {
            _Proxy.Abort();
        }
    }
}
```

If you don't rely on the generated proxy or DuplexClientBase<T> for duplex channels, you can use DuplexChannelFactory<T> directly. This provides a similar experience to ChannelFactory<T> in terms of channel factory caching and casting required during cleanup – but it accepts the callback type to its constructor as does DuplexClientBase<T>. In fact, DuplexChannelFactory<T> will wrap the callback type in the InstanceContext for you so you can save a step there. Figure 30 illustrates the differences from Figure 29 for duplex proxy lifetime management with DuplexChannelFactory<T>.

Figure 30: Duplex proxy lifetime management with DuplexChannelFactory<T>

```
public partial class MainWindow : Window
{

    private DuplexChannelFactory<ITodoListService> _Factory;
    private ITodoListService _Proxy;

    private void InitializeProxy()
    {
            try
            {
                TodoListCallback callback = new TodoListCallback(this);

                if (_Factory == null)
                    _Factory = new DuplexChannelFactory<ITodoListService>(new 
InstanceContext(callback), "");

                _Proxy = _Factory.CreateChannel();
                ICommunicationObject co = _Proxy as ICommunicationObject;
                co.Faulted += new EventHandler(InnerDuplexChannel_Faulted);
                co.Closed += new EventHandler(InnerDuplexChannel_Closed);

                _Proxy.Subscribe();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }

    }

    void InnerDuplexChannel_Faulted(object sender, EventArgs e)
    {
        this._SyncContext.Post(state =>
        {
            ICommunicationObject co = _Proxy as ICommunicationObject;
            co.Abort();

            InitializeProxy();
            GetData();

        }, null);

    }

    private void window_Closing(object sender, CancelEventArgs e)
    {
        try
        {
            ICommunicationObject co = _Proxy as ICommunicationObject;
            co.Faulted -= InnerDuplexChannel_Faulted;
            co.Closed -= InnerDuplexChannel_Closed;
        }
        catch
        {
            try
            {
                ICommunicationObject co = _Proxy as ICommunicationObject;
                co.Abort();
            }
            catch {}
        }

        try
        {
            _Factory.Close();
        }
        catch (Exception)
        {
            try
            {
                _Factory.Abort();
            }
            catch { }
        }
    }
}
```

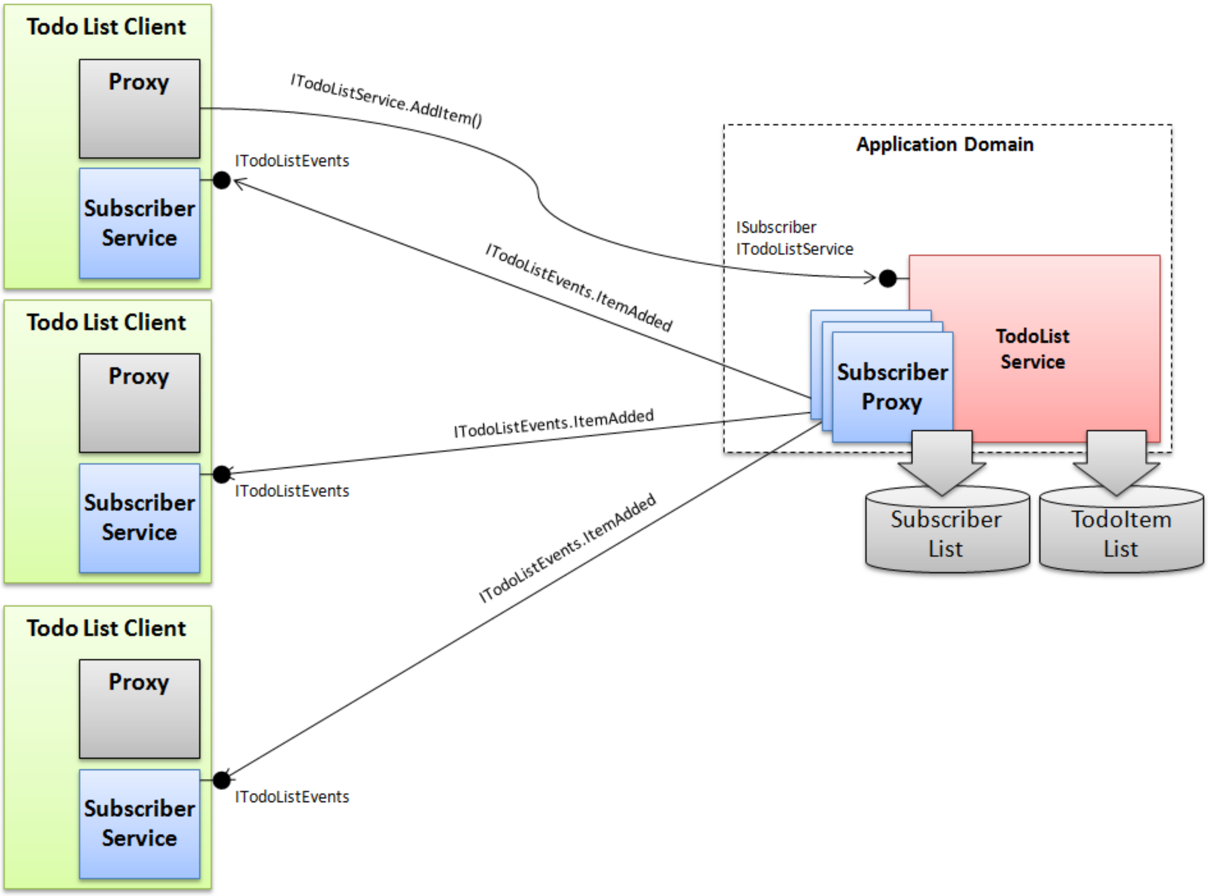
Hosting Services with WPF Clients

As mentioned earlier, any managed executable can host a WCF service. Generally, services hosted on server machines do not involve a user interface since they are unattended – but applications can find uses for hosting services as part of a client application. Some examples include:

- Using a router service at the client to determine which services to forward messages to, perhaps to handle online/offline situations
- Hosting services at the client to be shared by multiple client applications, perhaps to manage local statistics and handle notifications
- Publish and subscribe (pub-sub) scenarios that use durable subscriptions instead of transient duplex channels

Figure 31 illustrates the latter pub-sub scenario for the Todo List application discussed in other sections of this whitepaper.

Figure 31: The Todo List application implemented as a durable pub-sub scenario



The sections to follow will explore considerations relevant to hosting services in a WPF client application using the scenario illustrated in Figure 31 as an example.

Client-Side Service Design

Services hosted in a client application have similar design considerations related to throttling, concurrency and exception handling as do callback instances in a duplex scenario. Consider the TodoListSubscriberService definition shown in Figure 32. This service has the following characteristics:

- The ServiceHost instance will not be synchronized with the UI thread, as Indicated by the setting for UseSynchronizationContext on the ServiceBehaviorAttribute. That means that all calls will execute on a new thread at the client.
- The service uses InstanceContextMode.Singleton which slightly reduces the overhead of each request as only one instance of the service will be created.
- The service uses ConcurrencyMode.Single which prevents multiple threads from executing concurrently on the service type. Since requests do not execute on the UI thread, they are processed one by one and as needed post instructions to the UI thread.
- All service operations catch any exceptions and throw faults to prevent the communication channel with the TodoListService (the publisher) from being faulted.

Figure 32: Implementation of a client-side subscriber service based on ITodoListEvents

```
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Single, InstanceContextMode
= InstanceContextMode.Single, UseSynchronizationContext=false)]
public class TodoListSubscriberService: ITodoListEvents
{
    private MainWindow _MainWindow;

    public TodoListSubscriberService(MainWindow window)
    {
        _MainWindow = window;
    }

    public void ItemAdded(TodoItem item)
    {
        try
        {
            _MainWindow._SyncContext.Send(state =>
            {
                _MainWindow._TodoItems.Add(item);
            }, null);

        }
        catch (Exception ex)
        {
            throw new FaultException(ex.Message);
        }
    }

    public void ItemChanged(TodoItem item)
    {
        try
        {
            _MainWindow._SyncContext.Send(state =>
            {
                TodoItem found = _MainWindow._TodoItems.First<TodoItem>(x =>
x.ID == item.ID);

                found.PropertyChanged -= _MainWindow.Item_PropertyChanged;

                found.CreationDate = item.CreationDate;
                found.CompletionDate = item.CompletionDate;
                found.Description = item.Description;
                found.DueDate = item.DueDate;
                found.Title = item.Title;
                found.Priority = item.Priority;
                found.PercentComplete = item.PercentComplete;
                found.Status = item.Status;
                found.Tags = item.Tags;

                found.PropertyChanged += _MainWindow.Item_PropertyChanged;
            }, null);
        }
        catch (Exception ex)
        {
            throw new FaultException(ex.Message);
        }

    }

    public void ItemDeleted(string id)
    {
        try
        {
            _MainWindow._SyncContext.Send(state =>
            {
                TodoItem found = _MainWindow._TodoItems.First<TodoItem>(x =>
x.ID == id);
                _MainWindow._TodoItems.Remove(found);
            }, null);
        }
        catch (Exception ex)
        {
            throw new FaultException(ex.Message);
        }
    }
}
```

By default, when you construct a ServiceHost instance on the UI thread, the service will join the UI thread. That's why setting UseSynchronizationContext to false is an important part of configuring a client-side hosted service. This ensures that concurrent calls to the service do not place too much pressure on the message pump which is better served handling Windows messages that update the UI for end-users. Of course, as needed, the service thread will post message to the UI thread as well - but this should be done in a controlled manner. That is why it is also necessary to throttle at the service, and this is handled with using ConcurrencyMode.Single.

If you want to increase throughput at the client, you can optionally use ConcurrencyMode.Multiple and rely on the ServiceThrottleBehavior to control how many concurrent threads should execute at the client. The service behavior settings in code and in configuration are shown here assuming that four concurrent calls are allowed to process at the client:

```
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Multiple,
InstanceContextMode = InstanceContextMode.Single,
UseSynchronizationContext=false)]
public class TodoListSubscriberService: ITodoListEvents
```

```
<serviceThrottling maxConcurrentCalls="4" />
```

With ConcurrencyMode.Multiple any state managed by the singleton service instance will also require concurrency protection.

Regardless of the number of concurrent calls, since operations are not called on the UI thread any interactions with the UI must sent or posted to that synchronization context - as discussed earlier with callbacks. In fact, the implementation of each service operation on the TodoListSubscriberService type is almost identical to the callback object implementation discussed earlier.

One difference, however, is in how the main Window is passed to the service type. Since the service is a singleton, you can construct the service type and initialize it prior to constructing the ServiceHost as shown here:

```
TodoListSubscriberService subscriberService = new
TodoListSubscriberService(this);
_SubscriberServiceHost = new ServiceHost(subscriberService);
```

The Window exposes a SynchronizationContext member as with the duplex scenario to support sending updates to the UI thread.

As for exception handling, just as any good service should - faults are thrown instead of exceptions to preserve the communication channel in the presence of sessions.

Managing ServiceHost Lifetime

The lifetime of the ServiceHost instance for a client-side service is usually tied to the application lifetime – although this is not a strict requirement. Assuming it is you will likely use the following lifetime management guidelines:

- Open the ServiceHost when the main Window is created
- Dispose of the ServiceHost when the main Window is closing
- If the ServiceHost should be faulted or closed during the lifetime of the application – recreate it

As mentioned before, since the service type is likely to be a Singleton in a client-side host you should construct the service type first, and then pass it to the ServiceHost during initialization. In addition, before opening the ServiceHost handle the Closing and Faulted events to be sure that the application is aware of such failures and can respond accordingly. The resulting code to initialize the ServiceHost would then be as follows:

```
private void InitializeSubscriberService()
{

  TodoListSubscriberService subscriberService = new
TodoListSubscriberService(this);
  _SubscriberServiceHost = new ServiceHost(subscriberService);

  _SubscriberServiceHost.Closing += new
EventHandler(_SubscriberServiceHost_Closing);
  _SubscriberServiceHost.Faulted += new
EventHandler(_SubscriberServiceHost_Faulted);

  _SubscriberServiceHost.Open();
}
```

There are a few possible approaches to endpoint configuration for client-side services:

- You can used a fixed endpoint configuration with an agreed upon port for the application, usually configured on installation.
- You can find a free port to use each time the ServiceHost is initialized.

In either case, the client must subscribe to the service sending an address where it can be reached. The prior example for ServiceHost initialization assumes the address is configured as the service endpoint in the app.config for the application as follows:

```
<system.serviceModel>
  <services>
    <service
name="TodoList.WpfClient.SubscriberService.TodoListSubscriberService">
      <endpoint address="http://localhost:9000/TodoListSubscriberService"
binding="basicHttpBinding" contract="Contracts.ITodoListEvents"/>
    </service>
  </services>
</system.serviceModel>
```

If you go the dynamic route and look for a free port (assuming that administrators of client machines haven't locked down ports) you can use the FindFreePort() function shown in Figure 33, and initialize the ServiceHost with a base address and endpoint in code like so:

```
int freeport = FindFreePort();
Uri httpBase = new
Uri(string.Format("http://localhost:{0}/SubscriberService", freeport));

TodoListSubscriberService subscriberService = new
TodoListSubscriberService(this);
_SubscriberServiceHost = new ServiceHost(subscriberService, httpBase);
```

```
_SubscriberServiceHost.AddServiceEndpoint(typeof(ITodoListEvents), new
BasicHttpBinding(), "");
```

You can also omit the call to AddServiceEndpoint() and use the following relative endpoint configuration for the service, relying on the dynamic base address as the service endpoint:

```
<system.serviceModel>
  <services>
    <service
name="TodoList.WpfClient.SubscriberService.TodoListSubscriberService">
      <endpoint address="" binding="basicHttpBinding"
contract="Contracts.ITodoListEvents"/>
    </service>
  </services>
</system.serviceModel>
```

Figure 33: Implementation of FindFreePort()

```
private int FindFreePort()
{
    int port = 0;

    IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 0);
    using (Socket socket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp))
    {
        socket.Bind(endPoint);
        IPEndPoint local = (IPEndPoint)socket.LocalEndPoint;
        port = local.Port;
    }

    if (port == 0)
        throw new InvalidOperationException("Unable to find a free port.");

    return port;
}
```

When the application Window is closed cleanup should include disposing of the ServiceHost. Typically Close() is called instead of Abort() in order to allow any messages in the pipeline to complete processing prior to shutdown. For client-side services this is usually not a concern, since if the application is shutting down the UI won’t need to respond to messages. Assuming this, the following shutdown code would be used:

```
    try
    {
        _SubscriberServiceHost.Faulted -= _SubscriberServiceHost_Faulted;
        _SubscriberServiceHost.Closing -= _SubscriberServiceHost_Closing;
        _SubscriberServiceHost.Abort();
    }
    catch
    {}
```

If you want to gracefully shutdown, for example if the client will save the results of any outstanding messages somewhere locally, here is an alternative:

```
try
{
    _SubscriberServiceHost.Faulted -= _SubscriberServiceHost_Faulted;
    _SubscriberServiceHost.Closing -= _SubscriberServiceHost_Closing;
    _SubscriberServiceHost.Close();
}
catch
{
    _SubscriberServiceHost.Abort();
}
```

That sums up how to create and dispose of the ServiceHost instance, but what happens in the event the ServiceHost is faulted or closed unsuspectingly during the lifetime of the application? In this case, the client will no longer receive notifications from the service. Although a very rare occurrence, this case should be handled none-the-less. In the event the ServiceHost Faulted or Closing events are fired unexpectedly you can, and should try to recreate the ServiceHost instance. If there is a bigger issue, such as incorrect configuration, it will fail again and the issue should be reported. Since these events execute on the ServiceHost thread – not the UI thread – you can use the main Window's synchronization context to execute necessary recovery code on the UI thread.

The following code illustrates posting a call to the UI thread that executes InitializeSubscriberService() – the function described earlier:

```
this._SyncContext.Post(state =>
{
    InitializeSubscriberService();
}, null);
```

Managing Subscriptions

With duplex communications the client and service channel maintain a session and the callback object lifetime is tied to the lifetime of the proxy. The service can maintain a list of callback objects for subscribing clients – but this list cannot be persisted so if anything goes wrong, the channel must be recreated by the client to reestablish communications.

When the client application hosts a service in a pub-sub scenario, a subscription process is necessary to notify remote services of the location of that service. In this case, the client holds a proxy to the remote service, and the remote service holds a proxy to the client-side service. There are many approaches to pub-sub in terms of the layers of coupling between clients and services – and this discussion won't address the various approaches. The goal here is to provide a summary of considerations for the client for maintaining that subscription in the face of various possible failures.

In the case of the TodoListService from the scenario shown in Figure 31, the lifetime of this subscription is managed by the proxy. Since the TodoListService exposes the ISubscriber interface, the client can call Subscribe() when the application starts, and Unsubscribe() when the application is closed. The question is of what information must be passed to successfully subscribe? This can be handled in a number of ways:

- If you pass only the address of the service, there is an assumption that the client and service agree on the appropriate binding and contract for the SubscriptionService endpoint.
- As an alternative, the service can infer the binding configuration from the scheme of the endpoint. For example, if the client exposes an HTTP endpoint at its service, the binding may default to WSHttpBinding with a specific configuration.
- Another option might be for the service to infer the same binding as the client used when invoking Subscribe(). The service could look at the service endpoint hit by the client and use the same binding configuration.
- Another option, if more flexibility is required, is to provide all necessary binding configuration details in the call to Subscribe(). The amount of detail depends on the application's requirements of course.

Assuming that the service requires only the address, it may implement Subscribe() and Unsubscribe() as shown in Figure 34. The Subscribe() implementation in this example creates a channel using the address of the subscriber and a fixed binding and events contract – storing the channel in a dictionary keyed by the address. Unsubscribe() removes the dictionary entry and closes the channel. Distributed implementations will normally store subscribers in a database table, for durability. In that case the proxy might be cached in a particular application domain to save overhead where possible, and otherwise be created on demand.

Figure 34: Implementation of Subscribe() and UnSubscribe() for the TodoListService

```
public void Subscribe(string address)
{

    ITodoListEvents subscriber = 
ChannelFactory<ITodoListEvents>.CreateChannel(new BasicHttpBinding(), new 
EndpointAddress(address));
    lock (_SubscriberLock)
    {
        if (!_TodoListEventSubscribers.ContainsKey(address))
        {
            _TodoListEventSubscribers.Add(address, subscriber);
        }
    }
}

public void Unsubscribe(string address)
{
    lock (_SubscriberLock)
    {
        if (_TodoListEventSubscribers.ContainsKey(address))
        {
            ITodoListEvents subscriber = _TodoListEventSubscribers[address];
            _TodoListEventSubscribers.Remove(address);

            ICommunicationObject obj = subscriber as ICommunicationObject;
            try
            {
                obj.Close();
            }
            catch
            {
                obj.Abort();
            }
        }
    }
}
```

If the remote service is unavailable for any reason, calls to the service from the proxy will fail. This is an indication that notifications from the service are not being sent to the client-side subscriber service – and in the case of the Todo List application, the client must assume it is no longer up to date on the latest list of TodoItem entries. The client must therefore do the following to recover when the service is unavailable:

- If calls from the proxy fail, and a session is present, the proxy will be faulted at some point and this can be the trigger to recreate the proxy, subscribe to the service again, and request the latest data.
- If calls from the proxy fail, and there is no session, the proxy will NOT be faulted. In that case the proxy must use the communication exception to trigger subscribing again and requesting the latest data.
- In both cases, the client may have to make a few attempts to recreate the client proxy if the service is unavailable for some time. If the channel uses reliable sessions, this is handed on your behalf – and so failure indicates retries have already fails and the application should shut down.

If the ServiceHost fails at the client (again, a rare occasion at best) the following steps should be taken:

- Reconstruct the ServiceHost instance
- Use the proxy to resubscribe to the remote service since it could have removed the subscription if a notification failed
- Use the proxy to request the latest data since the application is surely out of sync if notifications were lost

Consuming REST-Based Services

REST-based services differ from traditional SOAP-based services primarily in the following ways:

- You use HTTP GET, POST, PUT or DELETE requests to communicate with service operations available at a particular Uri.
- Messages are formatted as Plain-Old-XML (POX) or JSON.

- REST-based services are not documented by WSDL, nor can proxies be generated using metadata exchange. Clients therefore do not use typical proxies to communicate with REST-based services.

In this section I will provide you with a crash course in creating and hosting a REST-based service for consumption by a WPF client, to provide context when I discuss your options at the client.

A Crash Course in REST-Based Services

You can REST-enable a SOAP-based service contract by decorating service operations with WebGetAttribute and WebInvokeAttribute. These attributes are used to describe the following:

- The Uri template that will invoke each operation.
- The HTTP verb used to invoke each operation – which is typically GET, POST, PUT or DELETE.
- The request and response format which can be XML or JSON.
- Whether the request or response will include a wrapper element or not.

Figure 35 shows the ITodoListService contract as a REST-based contract.

Figure 35: The REST-based version of the ITodoListService contract

```
[ServiceContract(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04")]
public interface ITodoListService
{
    [OperationContract]
    [WebGet(BodyStyle = WebMessageBodyStyle.WrappedRequest, RequestFormat =
WebMessageFormat.Xml, ResponseFormat = WebMessageFormat.Xml, UriTemplate =
"Items")]
    List<TodoItem> GetItems();

    [OperationContract]
    [WebInvoke(BodyStyle = WebMessageBodyStyle.WrappedRequest, RequestFormat
= WebMessageFormat.Xml, ResponseFormat = WebMessageFormat.Xml, UriTemplate =
"Items", Method = "POST")]
    string CreateItem(TodoItem item);
    [OperationContract]
    [WebInvoke(BodyStyle = WebMessageBodyStyle.WrappedRequest, RequestFormat
= WebMessageFormat.Xml, ResponseFormat = WebMessageFormat.Xml, UriTemplate =
"Items/{id}", Method = "PUT")]
    void UpdateItem(string id, TodoItem item);
    [OperationContract]
    [WebInvoke(BodyStyle = WebMessageBodyStyle.WrappedRequest, RequestFormat
= WebMessageFormat.Xml, ResponseFormat = WebMessageFormat.Xml, UriTemplate =
"Items/{id}", Method = "DELETE")]
    void DeleteItem(string id);
}
```

Assuming the base address for this service is "*http://localhost/TodoListService.svc*" then Figure 36 summarizes the expected interaction over HTTP.

Figure 36: Interacting with the TodoListService over HTTP

HTTP Verb	Relative Uri	Description
GET	/Items	Requests all items from the service. An array of TodoItem types is returned.
POST	/Items	Sends a new TodoItem to create at the service. The new item id is returned.
PUT	/Items/{id}	Sends a new TodoItem to update a particular item at the service.
DELETE	/Items/{id}	Indicates to the service an item to delete by its id.

To host a REST-based service you must enable the endpoint behavior, WebHttpBehavior, in order to bypass the default SOAP message processing supplied by the service model. This is typically done by initializing the WebServiceHost for your service type, instead of using the ServiceHost type. If you are self-hosting (as in a Console test host or Windows Service) this is done manually as follows:

```
WebServiceHost host = new WebServiceHost(typeof(TodoListService));
host.Open();
```

For services hosted in IIS 6 or IIS 7 with the WAS – you configure the WebServiceHostFactory for the .svc endpoint as follows:

```
<%@ ServiceHost Service="TodoList.TodoListService"
Factory="System.ServiceModel.Activation.WebServiceHostFactory" %>
```

Of course you must provide at least one endpoint to reach the service, and for REST-based services endpoints use the binding WebHttpBinding as shown here:

```
<service name="TodoList.TodoListService">
  <endpoint address="" binding="webHttpBinding"
contract="Contracts.ITodoListService" />
</service>
```

Using WebChannelFactory<T>

As mentioned earlier, there isn't a proxy generation experience for REST-based services. If you have access to the metadata libraries as in the shared libraries approach discussed earlier, you can easily create a proxy using WebChannelFactory<T> - the equivalent of ChannelFactory<T> for WCF's web programming model. This requires you to have access to the libraries that define the service contract and any data contracts and other serializable types it relies on. It is always good practice to separate metadata libraries from service implementations – and so if you follow this practice you should already be well positioned to work with WebChannelFactory<T> at the client.

In fact, the developer experience working with WebChannelFactory<T> is like that of ChannelFactory<T>. First, your client should reference libraries that contain the service contract and data contracts. Second, you create a client channel for the service as follows:

```
WebChannelFactory<ITodoListService> _ChannelFactory = new
WebChannelFactory<ITodoListService>(
new Uri("http://localhost:8000/TodoListService"));

ITodoListService _Proxy = _ChannelFactory.CreateChannel();
```

By default WebChannelFactory<T> uses a WebHttpBinding endpoint for the proxy at the address specified in the constructor. You can also explicitly provide the binding configuration as follows:

```
_ChannelFactory = new WebChannelFactory<ITodoListService>(new
WebHttpBinding(), new Uri("http://localhost:8000/TodoListService"));

_Proxy = _ChannelFactory.CreateChannel();
```

Proxy generation for classic SOAP-based services produces an application configuration file with the appropriate endpoint configuration including the address, binding and contract. It is usually desirable to externalize the endpoint address since this may change as you move from development to production machines. The following configuration section and code produces the equivalent results:

```
_ChannelFactory = new WebChannelFactory<ITodoListService>("default");
_Proxy = _ChannelFactory.CreateChannel();

<system.serviceModel>
  <client>
    <endpoint address="http://localhost:8000/TodoListService"
binding="webHttpBinding" contract="Contracts.ITodoListService"
name="default"/>
  </client>
</system.serviceModel>
```

You can also create the channel in a single call, however in order to cache the channel factory it is better to create the channel factory first and then the channel.

WCF REST Starter Kit

Microsoft published the WCF REST Starter Kit to assist WCF developers in building HTTP services based on the web programming model which includes REST-based services as well as syndication feeds. The REST Starter Kit includes the following:

- Visual Studio templates to improve developer productivity producing REST-based services and syndication feeds
- Improved help page when browsing to HTTP services to facilitate discovery of functionality

- Better error reporting, improved extensibility and caching for HTTP services
- Support for metadata generation using Paste XML as Types
- HttpClient type to simplify client code

The goal of this paper is not to elaborate on ways that the REST Starter Kit affects the service developer's approach to REST-based services. I will, however, explain how the kit helps the client developer to consume REST-based services.

NOTE: At the time of this writing the WCF REST Starter Kit Preview 2 is available on CodePlex at http://aspnet.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=24644. This section is subject to change as the kit evolves and as features from the kit are introduced into Visual Studio 2010.

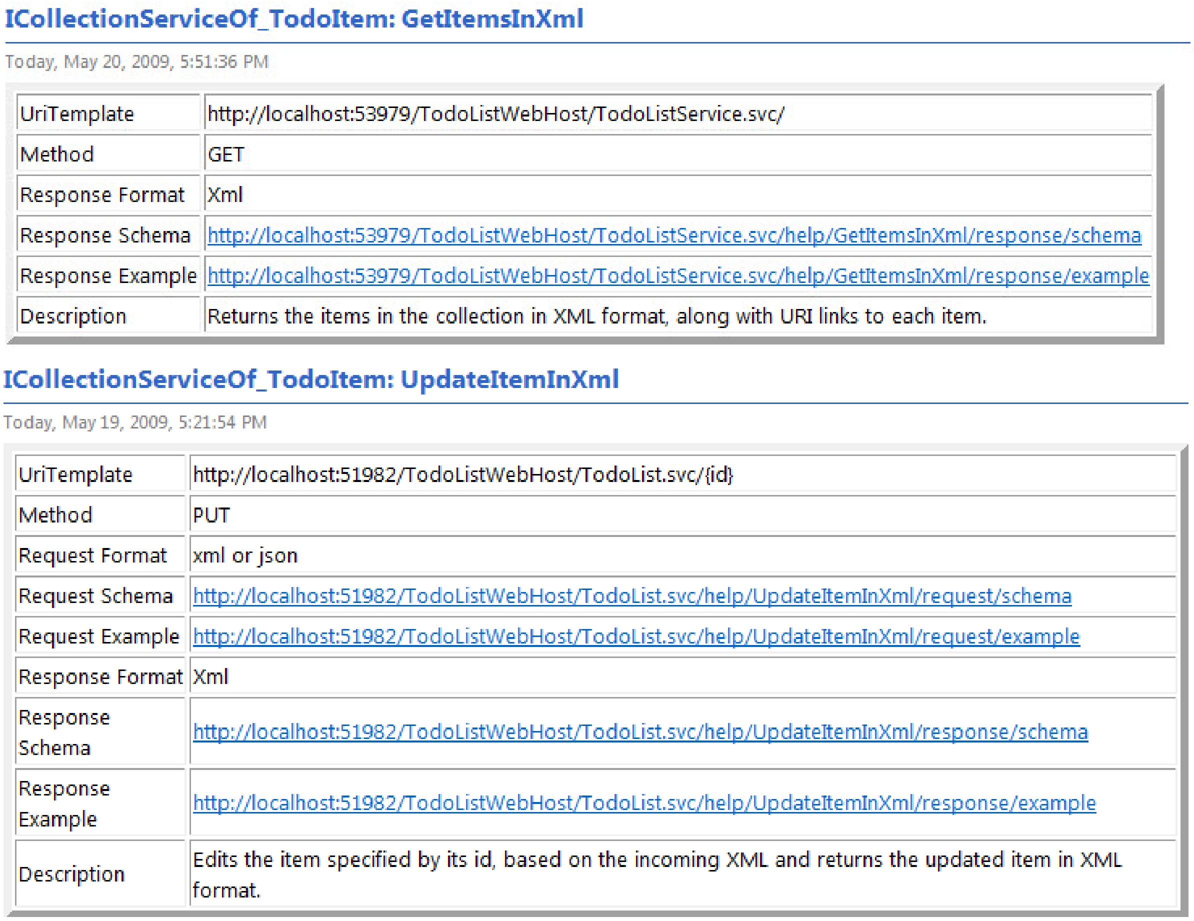
Probably one of the most common types of REST-based services you can consume is a resource collection like the list of TodoItem types used in other examples for this paper. This discussion assumes this type of service will be consumed from your REST-based client using the REST Starter Kit. The steps to consume such a service are as follows:

- Browse to the help page to learn how you can interact with the service over HTTP
- Generate types from the schemas available via the help page
- Reference the appropriate REST Starter Kit assemblies in the client project
- Use the HttpClient type to communicate with service Uris over HTTP including GET, PUT, POST and DELETE instructions

Viewing the Help Page

Figure 37 provides an example of what the browser help page for a REST-based service looks like. Shown in the figure is the description for GetItems() and UpdateItems() methods. Each method description includes the Uri to which HTTP requests should be sent, the HTTP verb (method) to use, the request and response schema, a sample request and response message format in XML or JSON. You can use this information to build the HttpClient code necessary to interact with the service, and to generate types for serialization at the client.

Figure 37: A partial view of the browser help page produced for a REST-based service



ICollectionServiceOf_TodoItem: GetItemsInXml

Today, May 20, 2009, 5:51:36 PM

UriTemplate	http://localhost:53979/TodoListWebHost/TodoListService.svc/
Method	GET
Response Format	Xml
Response Schema	http://localhost:53979/TodoListWebHost/TodoListService.svc/help/GetItemsInXml/response/schema
Response Example	http://localhost:53979/TodoListWebHost/TodoListService.svc/help/GetItemsInXml/response/example
Description	Returns the items in the collection in XML format, along with URI links to each item.

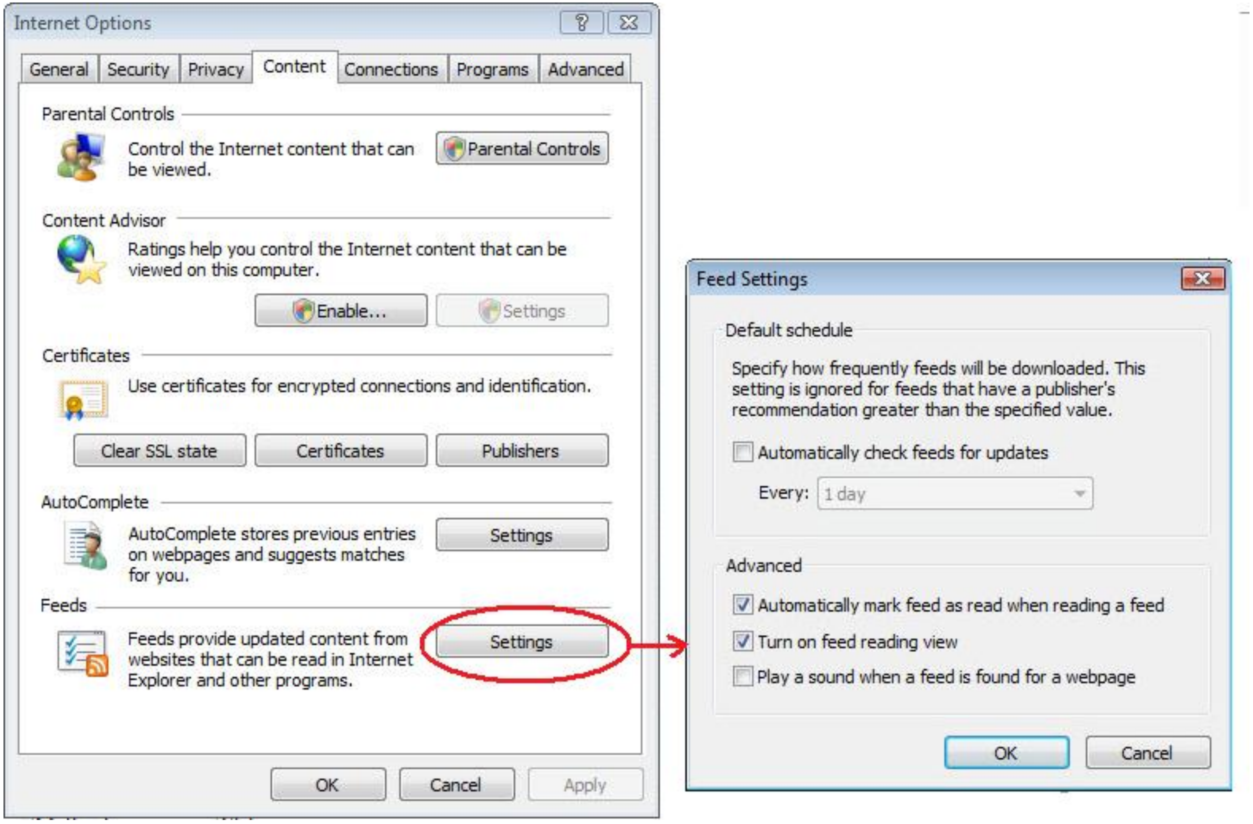
ICollectionServiceOf_TodoItem: UpdateItemInXml

Today, May 19, 2009, 5:21:54 PM

UriTemplate	http://localhost:51982/TodoListWebHost/TodoList.svc/{id}
Method	PUT
Request Format	xml or json
Request Schema	http://localhost:51982/TodoListWebHost/TodoList.svc/help/UpdateItemInXml/request/schema
Request Example	http://localhost:51982/TodoListWebHost/TodoList.svc/help/UpdateItemInXml/request/example
Response Format	Xml
Response Schema	http://localhost:51982/TodoListWebHost/TodoList.svc/help/UpdateItemInXml/response/schema
Response Example	http://localhost:51982/TodoListWebHost/TodoList.svc/help/UpdateItemInXml/response/example
Description	Edits the item specified by its id, based on the incoming XML and returns the updated item in XML format.

Note that the browser help page only appears friendly as shown in Figure 37 if you turn on feed reading view as illustrated in Figure 38.

Figure 38: Enabling feed reading view for IE 7



Parental Controls

Control the Internet content that can be viewed.

Content Advisor

Ratings help you control the Internet content that can be viewed on this computer.

Certificates

Use certificates for encrypted connections and identification.

AutoComplete

AutoComplete stores previous entries on webpages and suggests matches for you.

Feeds

Feeds provide updated content from websites that can be read in Internet Explorer and other programs.

Default schedule

Specify how frequently feeds will be downloaded. This setting is ignored for feeds that have a publisher's recommendation greater than the specified value.

☐ Automatically check feeds for updates

Every: 1 day

Advanced

☑ Automatically mark feed as read when reading a feed

☑ Turn on feed reading view

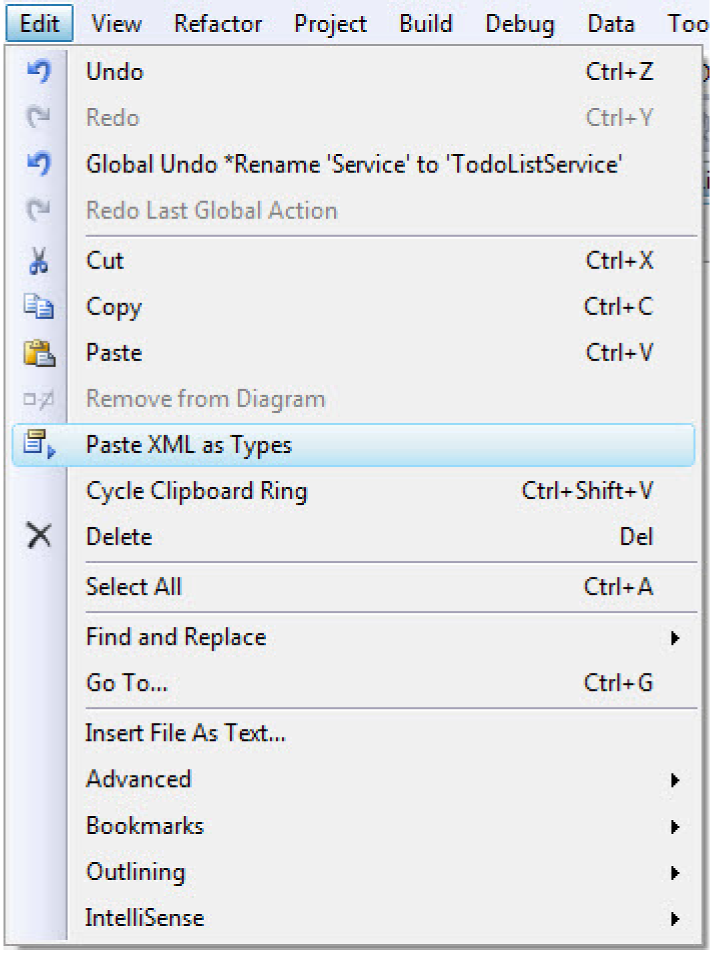
☐ Play a sound when a feed is found for a webpage

Generating Types with Paste XML as Types

If you click on the request or response schema link from the help page you will be shown a schema for the message. Using this you can generate types for the client project. First, view the source of the schema shown in the browser (Page->View Source). Copy this to the clipboard and from Visual Studio select Edit->Paste XML as Types (see Figure 39). This feature is installed with the WCF REST Starter Kit. If your types depend on enumerations you will have to paste the schema first and then instead of pasting the schema for the type – paste the sample XML.

NOTE: At the time of this writing Paste XML as Types does not handle schemas with enumeration dependencies. When you paste the sample XML for the type you'll have to wire up properties that use the enumerations by hand as they will be pasted as string types.

Figure 39: Paste XML as Types feature



For Uri operations that return a collection of types, such as a call to GetItems() – a collection of items is returned. If the response is XML, it looks similar to the following:

```
<ItemInfoList>
  <ItemInfo>
    <EditLink>...</EditLink>
    <Item>...</Item>
  </ItemInfo>
  <ItemInfo>
    <EditLink>...</EditLink>
    <Item>...</Item>
  </ItemInfo>
</ItemInfoList>
```

The collection actually includes a wrapper type for the main resource, which includes an <EditLink> element associated with each resource <Item>. The <EditLink> property supplies a Uri where clients can drill down to interact with the resource. The actual resource type is contained inside the <Item> element. For example, properties of the TodoItem type discussed earlier would be serialized inside this element.

Using Paste XML as Types based on the sample XML for the TodoListService collection response, the types in Figure 40 are generated (edited for brevity). This result works great for retrieval of the collection using the following code:

```
HttpClient _Proxy = new
HttpClient("http://localhost:53978/TodoListWebHost/TodoListService.svc/");
HttpResponseMessage response = _Proxy.Get();
response.EnsureStatusIsSuccessful();

ItemInfoList itemsCollection =
response.Content.ReadAsXmlSerializable<ItemInfoList>();
```

This code constructs the HttpClient type supplied with the WCF REST Starter Kit, supplies it with the base Uri to the service, issues an HTTP GET using the Get() method of the HttpClient proxy and deserializes the results using the XmlSerializer. From there you can interact with the deserialized *itemsCollection* instance.

Figure 40: Types generated for a collection response from the TodoListService

```
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
[System.Xml.Serialization.XmlRootAttribute(Namespace = "", IsNullable =
false)]
public partial class ItemInfoList
{

  [System.Xml.Serialization.XmlElementAttribute("ItemInfo")]
  public ItemInfoListItemInfo[] ItemInfo {get; set;}
}

[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
public partial class ItemInfoListItemInfo
{

  public string EditLink {get; set;}

  public ItemInfoListItemInfoItem Item {get; set;}

}

[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
public partial class ItemInfoListItemInfoItem
{

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public string ID {get; set;}


  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public string Title {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public string Description {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
 public string Priority {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public string Status {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public System.DateTime CreationDate {get; set;}

 [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public System.DateTime DueDate {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public System.DateTime CompletionDate {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public float PercentComplete {get; set;}

  [System.Xml.Serialization.XmlElementAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
  public string Tags {get; set;}
}
```

In addition to generating types for the collection result, you should generate types required as input to a request, such as an HTTP POST to add a new item. The result from Paste XML as Types is slightly different as shown in Figure 41 for adding a new TodoItem. The main difference is that the type name matches the resource type name as defined for the service ("TodoItem") and a root namespace is defined for serialization of elements, instead of a namespace defined for individual properties. This is important because if you try to use the type defined for the collection in Figure 40 ("ItemInfoListItemInfoItem") even if the CLR type is renamed correctly it won't work because the service requires a root namespace (this might change as the WCF REST Starter Kit evolves).

Figure 41: Types generated for a single item response from the TodoListService

```
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true, Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
[System.Xml.Serialization.XmlRootAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas", IsNullable = false)]
public partial class TodoItem
{
  public string ID {get; set;}
  public string Title {get; set;}
  public string Description {get; set;}
  public string Priority {get; set;}
  public string Status {get; set;}
  public System.DateTime CreationDate {get; set;}
  public System.DateTime DueDate {get; set;}
  public System.DateTime CompletionDate {get; set;}
  public float PercentComplete {get; set;}
  public string Tags {get; set;}
}
```

To address the incongruent results from types generated by Paste XML as Types for collections versus individual calls I recommend the following:

- Generate types for the response from an HTTP GET for the collection.
- Generate the type used by the request for an HTTP POST (add feature). Use this type instead of the type generated for the collection, and refactor accordingly.

In addition, you may have to do any of the following in order to get serialization to work properly:

- If your types rely on enumerations, paste those from their respective schema representations separately.
- After pasting the type that depends on enumerations, you will have to rewire it to use the enum definition.
- Add XmlElementAttribute to type properties to control order in serialization to match that of the service type expectations.
- If you refactor the CLR type names be sure and supply the correct ElementName property to the XmlRootAttribute for the type.

Figure 42 illustrated the resulting type definitions (without their internal details) used for the TodoListService.

Figure 42: Refactored XmlSerializer types generated for the TodoListService

```
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
[System.Xml.Serialization.XmlRootAttribute(Namespace = "", IsNullable =
false, ElementName="ItemInfoList")]
public partial class TodoItemList
{…}

[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
[System.Xml.Serialization.XmlRootAttribute(Namespace = "", IsNullable =
false)]
public partial class ItemInfo
{…}

[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true, Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas")]
[System.Xml.Serialization.XmlRootAttribute(Namespace =
"http://wcfclientguidance.codeplex.com/2009/04/schemas", IsNullable = false)]
```

```
public partial class TodoItem
{…}
```

Using HttpClient to Issue HTTP Requests

As mentioned earlier, you use the HttpClient type from Microsoft.Http to communicate with your REST-based services. This type exposes functionality to execute HTTP GET, POST, PUT and DELETE commands via Get(), Post(), Put() and Delete() methods. You initialize the HttpClient instance with the correct base Uri for the service and supply any relative Uri requirements for each command. To pass content to a request, as in a POST to create a new item, you can use the HttpContent type – more specifically, the HttpContentExtensions type which exposes static methods such as CreateXmlSerializable<T>() to construct content from a serializable type. Likewise, to read content from an HTTP response HttpContentExtensions supplies extension methods such as ReadAsXmlSerializable<T>() to rehydrate types. Figure 43 shows a partial listing of the code used to issue GET, POST, PUT and DELETE requests to the TodoListService.

Figure 43: HttpClient code to issue HTTP GET, POST, PUT and DELETE requests to the TodoListService

```
HttpClient _Proxy = new
HttpClient("http://localhost:53978/TodoListWebHost/TodoListService.svc/");

// HTTP GET
HttpResponseMessage response = _Proxy.Get();
response.EnsureStatusIsSuccessful();

TodoItemList itemsCollection =
response.Content.ReadAsXmlSerializable<TodoItemList>();

// HTTP POST
HttpContent content =
HttpContentExtensions.CreateXmlSerializable<TodoItem>(todoItem);

HttpResponseMessage response = _Proxy.Post("", content);
response.EnsureStatusIsSuccessful();

ItemInfo newItemWithId = response.Content.ReadAsXmlSerializable<ItemInfo>();


// HTTP PUT
HttpContent content =
HttpContentExtensions.CreateXmlSerializable<TodoItem>(todoItem);

 HttpResponseMessage response = _Proxy.Put(todoItem.ID, content);
 response.EnsureStatusIsSuccessful();

// HTTP DELETE
HttpResponseMessage response = _Proxy.Delete(todoItem.ID);
response.EnsureStatusIsSuccessful();
```

DataContracts and Shared Types

If you own the service and the client, you are likely going to prefer sharing your data contract libraries with your client application. In this case you will need to create data contracts for the item collection and type wrapper discussed earlier, and you will also change some of the methods used for serialization and deserialization. Figure 44 shows the data contract equivalent of the item collection and type wrapper used when retrieving a collection result from the TodoListService. The CLR type name is not important, but the DataContractAttribute must supply a Name property matching the name of the wrapper elements returned by the service, and an empty namespace since that is what the service returns.

Figure 44: Data contract equivalents for the item collection and type wrapper from Figure x

```
[DataContract(Name="ItemInfoList", Namespace="")]
public partial class TodoItemList
{
  [DataMember(Name="ItemInfo")]
  public TodoItemInfo[] ItemInfo {get; set;}
}

[DataContract(Name="ItemInfo", Namespace="")]
public partial class TodoItemInfo
{
  [DataMember]
  public string EditLink {get; set;}

  [DataMember]
  public TodoItem Item {get; set;}
}
```

As for the HttpClient code used to interact with the service, the main difference is in how the content for each request is serialized and deserialized. Instead of using XmlSerializer methods, data contract methods are used such as the ReadAsDataContract<T>() and CreateDataContract<T>() – as shown in Figure 45.

Figure 45: Data contract equivalents for the item collection and type wrapper from Figure x

```
HttpClient _Proxy = new
HttpClient("http://localhost:53979/TodoListWebHost/TodoListService.svc/");

// HTTP GET
HttpResponseMessage response = _Proxy.Get();
response.EnsureStatusIsSuccessful();

TodoItemList itemsCollection =
response.Content.ReadAsDataContract<TodoItemList>();

// HTTP POST
HttpContent content =
HttpContentExtensions.CreateDataContract<TodoItem>(todoItem);

HttpResponseMessage response = _Proxy.Post("", content);
response.EnsureStatusIsSuccessful();

TodoItemInfo newItemWithId =
response.Content.ReadAsDataContract<TodoItemInfo>();
```

```
// HTTP PUT
HttpContent content =
HttpContentExtensions.CreateDataContract<TodoItem>(todoItem);

HttpResponseMessage response = _Proxy.Put(todoItem.ID, content);
response.EnsureStatusIsSuccessful();

// HTTP DELETE
HttpResponseMessage response = _Proxy.Delete(todoItem.ID);
response.EnsureStatusIsSuccessful();
```

Acknowledgements

I'd really like to thank the many product team members at Microsoft who are very responsive to answering questions related to WCF and the WCF REST Starter Kit! You guys rock!