Windows Communication Framework

Windows Communication Framework (WCF) is Microsoft's solution for distributed computing and Service-Oriented Applications (SOA). With WCF you can spread the processing load over one or more servers, allowing you to put the presentation logic on one server and the data access / business logic on another. There are a couple of issues inherent in such a separation, but there are also some compelling reasons to follow this type of deployment model.

Benefits

You may want to use this to better support connection pooling to the database, which is especially useful with desktop applications.

You may want to do this to improve security, such that only the data servers need to be able to connect to the database, and the presentation servers can be blocked. This is especially useful with Internet-facing applications, where the presentation servers are in the Demilitarized Zone (DMZ), and everything else is safely behind the firewalls.

You may want to do this to lower the cost of initializing an ORM system. Initializing an ORM such as Entity Framework or NHibernate can be expensive, and initializing all of the mapping/configuration information on the data server can lower the cost of starting up the presentation server. Instead of having to load the metadata for every page request, or even in the application start, you can load the metadata when the WCF host is started.

Finally, you may want to do this to help solve bandwidth concerns; in some settings, you may find that the data passed back and forth between the database and the data server is more than the data passed back and forth between the data server and the presentation servers. With web applications, this may not be a major concern because they may well both be on the same network segment, and so the net effect is negligible. But with a desktop application, the database-to-data-server link may be a local network hop, while the data-server-to-presentation link may be a slow network hop over a lower bandwidth.

Drawbacks

The most obvious drawback is that method calls must now be marshaled across server boundaries. This may mean that something that *looks* like a simple method will actually be an expensive service method call. Essentially, we want to get all of the data we need in as few calls as possible, which also means we want to avoid a "chatty" interface.

Suppose you have a UI screen that needs information from one particular business entity, as well as a collection of other related business entities. Instead of making a call to retrieve the `User` object, and a separate call to retrieve the list of `Permission` objects, you want to retrieve all of this information with one call. Depending on the pattern you are following, this may be known as a **composite DTO (Data Transfer Object)**, or it may be known as `ViewModel` , or you may have your own name for it. The important thing is to reduce the number of service calls.

```
public interface IService
{
   User GetUser(int userId);
   IList<Permission> GetPermissionsByUserId(int userId);
}
```

Listing 5.26: A simple "chatty" service interface.

```
public class UserPermissions : CompositeDtoBase
{
   public IList<Permission> Permissions { get; set; }
   public User User { get; set; }
}
```

Listing 5.27: A simple composite DTO.

```
public interface IConsolidatedService
{
   UserPermissions GetUserPermissions(int userId);
}
```

Listing 5.28: A more streamlined service interface.

Disposable

The objects in the WCF namespace implement the `IDisposable` interface but, unfortunately, it is not implemented properly. Unlike with other objects that implement the disposable pattern, you should avoid the `Using` statement with a service proxy. If you *do* try it, then the `Dispose()` method of a proxy will call `the Close()` method, which will cause an exception if a fault has occurred on a channel.

Relying on the `Using` construct may also result in orphaned service clients and unmanaged resources. The best practice for properly managing and cleaning up service clients is to follow a pattern similar to the one in Listing 5.29.

```
var clientproxy = new UnderTheHoodClient();
try
{
   clientproxy.Method();
   clientProxy.Close();
}
catch(CommunicationException)
{
   clientProxy.Abort();
}
catch (TimeoutException)
{
   clientProxy.Abort();
}
```

Listing 5.29: WCF call pattern.

You can also simplify this into a single method accepting a lambda statement. A `UseAndClose` method will run the action that you specify, and also properly handle the closing of the connection.

```
public static RETURN UseAndCloseService<SERVICE, RETURN>
     (Func<SERVICE, RETURN> block) where SERVICE : class,
           ICommunicationObject, new()
{
   SERVICE service = null;
   try
   {
     service = new SERVICE();
     return block.Invoke(service);
   }
   finally
   {
     if (service != null)
        CloseWCFService(service);
   }
}
```

Listing 5.30: A simple implementation of the `UseAndClose` method.

In this implementation, we create the service, call the specified method, and then immediately close it. In this example, `SERVICE` and `RETURN` are generic type parameters. We pass these two data types in when we make the method call. This allows us to specify the data type for the service but also the return type for the method that we will call. The `CloseWCFService` may look similar to that in Listing 5.31.

```
public static void CloseWCFService(ICommunicationObject wcfService)
{
    bool success = false;
    if (wcfService != null)
    {
     try
     {
       if (wcfService.State != CommunicationState.Faulted)
       {
         wcfService.Close();
         success = true;
       }
     }
       catch (Exception ex)
       {
            // log with error
       }
       finally
       {
         if (!success)
           wcfService.Abort();
         wcfService = null;
       }
     }
  }
```

Listing 5.31: A centralized `CloseService` method.

Using such a method is also relatively straightforward, and you don't even have to sacrifice Intellisense (Listing 5.32).

```
 var result = Utility.UseAndCloseService<WcfServiceClient,
      PageResponseDto>(srv => srv.DoWork(param1, param2));
```

Listing 5.32: Using the `UseAndCloseService` method.

Configuration

WCF provides several configuration settings that can influence availability and scalability. The configuration settings can also ultimately influence the resources used under load.

For example, the `serviceThrottling` element has three attributes (see Table 5.3). There is no magic combination that should be used. You need to test your settings under your own load to determine which combination yields the best throughput.

`maxConcurrentCalls`	This is a positive integer that determines the maximum number of messages that can be processed concurrently. The default value is 16 times the number of processors on the server, meaning that if there are more than 32 requests on a dual processor server, the extras will be queued up. This also helps prevent the server from being flooded with requests during a usage spike or a DOS attack. If you find that you have a lot of requests queuing up and creating delays in the presentation servers, but resources are not maxed out on the data server, you may want to consider raising this limit.
`maxConcurrentInstances`	This is a positive integer that specifies the maximum number of `InstanceContext` objects in the service. This value is influenced by the `InstanceContextMode` attribute: **PerSession** - this value will determine the total number of sessions **PerCall** - this value will determine the total number of concurrent calls **Single** - this value is irrelevant since only one `InstanceContext` will ever be used. Once this threshold is reached, new requests will be queued until an instance context is closed. The default is the sum of the default value of `MaxConcurrentSessions` and the default value of `MaxConcurrentCalls.`
`maxConcurrentSessions`	This is a positive integer that specifies the maximum number of sessions a `ServiceHost` object can accept. The service will accept more requests, but they will not be read until the current number of active sessions fall below this threshold. The default value is 100 times the number of processors.

Table 5.3: `serviceThrottling` attributes and their descriptions.