

2007

Getting Started with Proxy Factory

On the road to WCF

This document presents you briefly the concept behind the ProxyFactory and walks you through four quick samples. The first one covers a web service dynamic proxy generation, the second covers a .Net Remoting dynamic proxy generation, the third and fourth shows you additional concepts such as asynchronous calls and proxy configuration using a mechanism similar to dependency properties.

Version 1.1 Rev b

RioterDecker
NetFxFactory.org
11/7/2006



Table of Contents

Presentation	3
About ProxyFactory	3
Concept.....	3
What can ProxyFactory do for you?.....	3
How will ProxyFactory change your life?	4
Walkthroughs	4
Setup	4
Sample1: Web Service	5
Step by step	6
Sample 2: .Net Remoting	7
Step by step	8
Sample 3: Going further – Asynchronous calls	11
Step by step	11
Sample 4: Going further: dependency properties	13
Step by step	14
Sample 5: WSE 3.0 secured credentials sample.....	15
Step by step	15
Sample 6: Configuration	17
Step by step	17
Take away	18
Tips and tricks.....	18
Interesting reading.....	18
Disclaimers & copyright	19

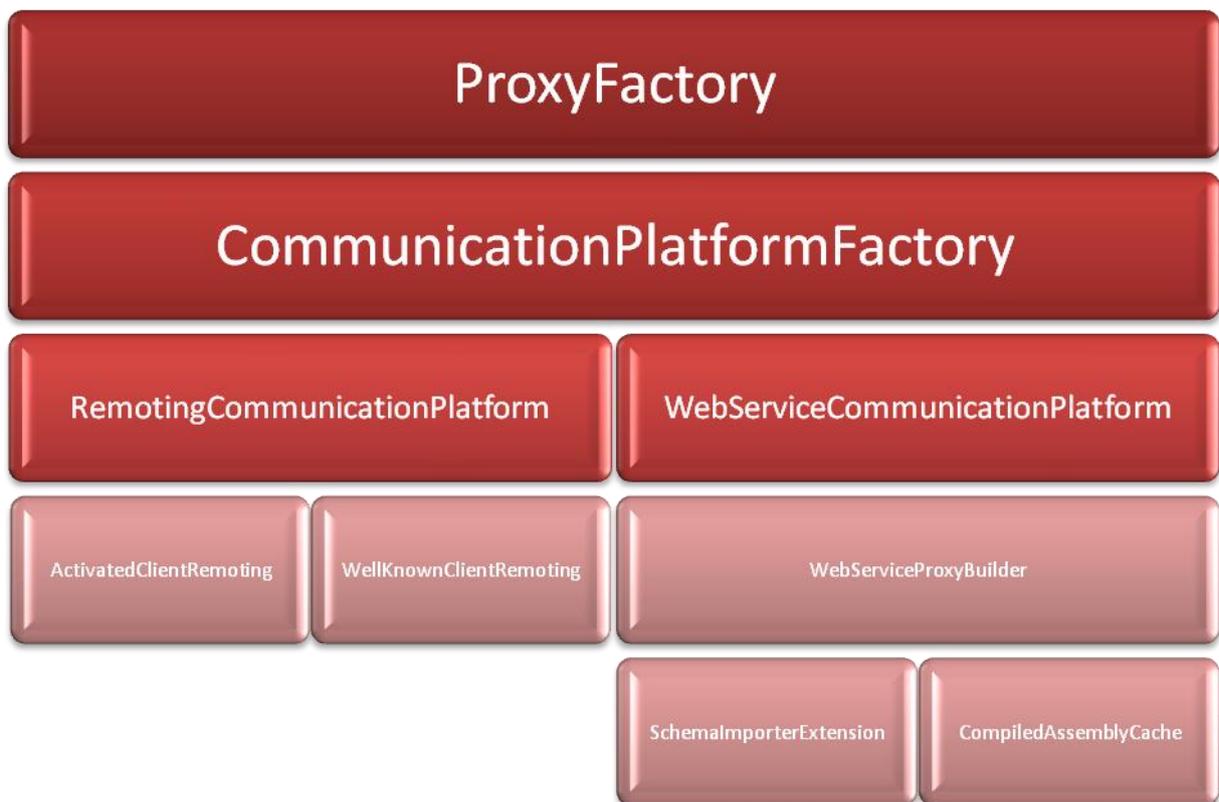
Presentation

About ProxyFactory

This library has been built based on Christian Weyer's Dynamic Web Service Library in order to provide an effective way of creating dynamic proxy's. Although, C. Weyer addressed a good range of problematic for web service proxy generation it does not take full advantage of .Net 2.0 features and leaves room for improvement. This is one of the points that we wanted to tackle with this library. Aside from the improvement does the WS road, we found interesting to include .Net Remoting proxy generation to the library thus transforming it into a true proxy factory that can be then used later on as a gateway to WCF.

Concept

A picture is worth a thousand words so as an introduction to the concept of ProxyFactory here is a diagram that shows the main bricks under ProxyFactory's hood.



What can ProxyFactory do for you?

The emergence of the WCF platform announces the unification of all communications technologies provided by the .NET platform in a common foundation and so their decline. So it is still a safe bet to develop a library to leverage .NET 2.0 generics and SchemaImporterExtension mechanisms to offer a similar developer experience.

Such a library already exists and targets only web services communication, it was developed by Christian Weyer from Thinktechure but, as opposed to the WCF

ChannelFactory<TChannel>, his library doesn't provide an effective mechanism to handle specific types during the proxy generation process. This is what is tackled by the ProxyFactory by taking advantage of SchemaImporterExtension mechanisms. Furthermore, alternative communication technologies are targeted such as Remoting and WSE. This library offers a convenient way to provide an easy-to-use programming model that first, addresses in a common way two heterogeneous communication channels (Asmx and Remoting) but also help to drive existing applications towards the unification of the communication platforms proposed by Microsoft .Net 3.0 with WCF.

How will ProxyFactory change your life?

ProxyFactory provides a seamless programming experience whether you are addressing Web Services or .Net Remoting. Primarily, by taking full advantage of dynamic proxy generation as well as the generated assembly caching, ProxyFactory offers a transparent, performant and developer friendly way to handle the common communication channels. Furthermore, ProxyFactory drives you towards WCF by using a similar client side programming experience thus limiting the migration cost.

Walkthroughs

Setup

By now, you should have everything setup to run both Web service and .Net Remoting samples.

Here are the details of what has been installed by the Getting Started with ProxyFactory MSI package:

- In your system's program files:
 - This Getting started documentation
 - A folder containing the Web Service sample (with a completed solution and a follow through)
 - A folder containing the .Net Remoting sample (with a completed solution and a follow through)
 - A Reference Assemblies folder containing the ProxyFactory library as well as an Utils library (the samples reference these assemblies).
- Code snippets to quickly create ProxyFactory instances.
- Two projects template to easily create library or console application that use ProxyFactory.

Sample1: Web Service

The completed sample is available via the start menu at: Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Web Service Sample – Completed

The goal of this sample is to create a simple web service allowing interacting with someone's bank account and then interact with this web service using the ProxyFactory.

The following diagram summarizes the web service contract:

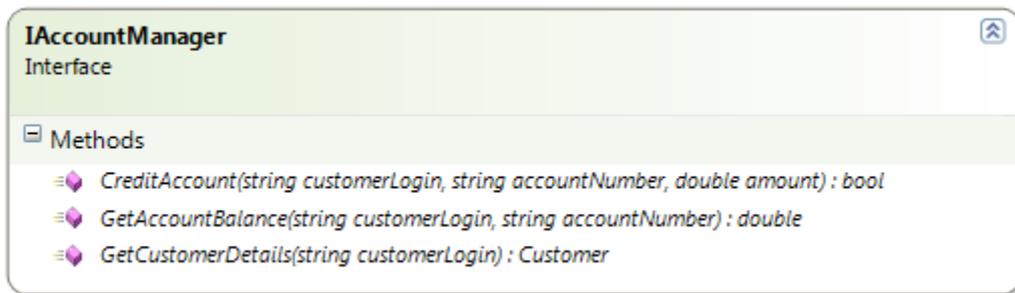


Figure 1: Web service contract.

The following diagram details the data structure exchanged between clients and WS:

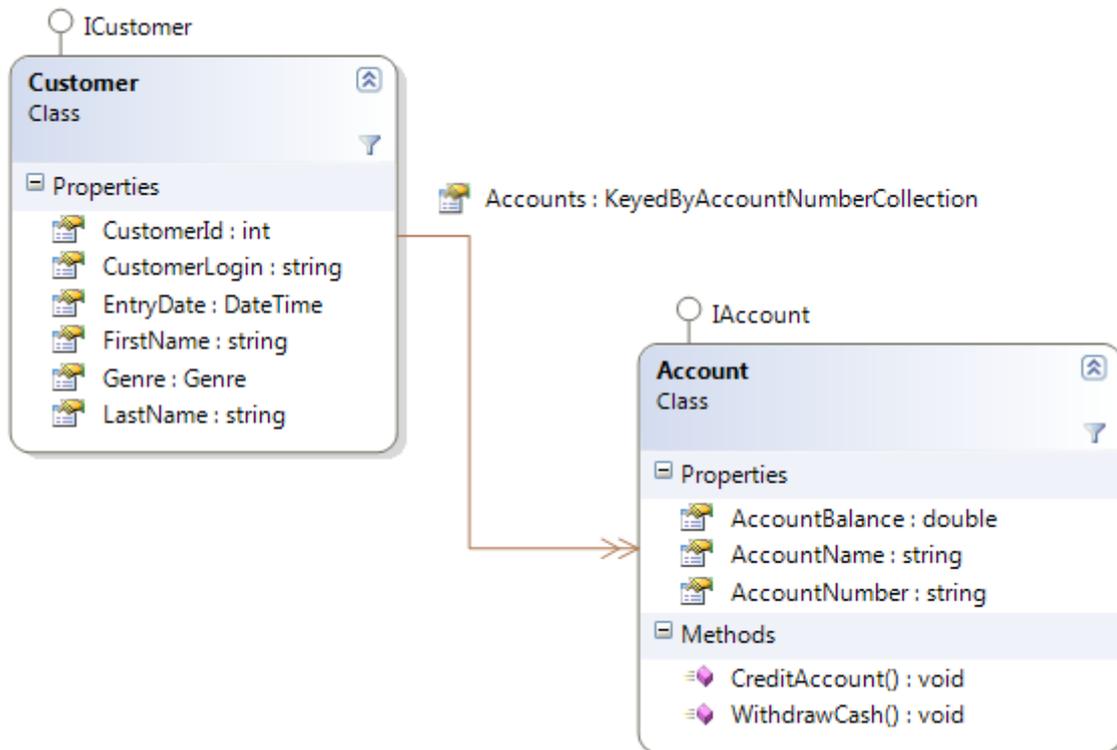


Figure 2: Data structures.

Step by step

Open the solution "Named Web Service Sample – Step by step" in your start menu Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Web Service Sample – Step by step This opens up a Visual Studio solution containing 3 projects: a web project hosting the web service, a library that contains the contracts and data structures and finally a console application that will be a client application. These projects are mostly empty shells that you will have to complete.

1. Implement the contracts and data structures required in this sample.
 - a. In the Sample.Contracts project, create the data structure described in figure 2.
 - b. Again in Sample.Contracts, create the web service contract described in figure 1
2. Create the web service.
 - a. In the web service project, add a reference to the Sample.Contracts project,
 - b. Then add a new web service named AccountManager to the project.
 - c. Finally implement the IAccountManager contract that you created in 1.b.

Trick: In the web service constructor insert a call to `CustomerRepository.LoadCustomers();` in order to simulate db access. Then use `CustomerRepository.Repository.TryGetValue(...)` to access customer's info in the web methods.

Now comes the most interesting part of this sample which is how to use the ProxyFactory.

3. Create the web service client
 - a. In the WSClient project, add references to :
 - ProxyFactory assembly located under:
<Program files>|NetFxFactory|Getting started with ProxyFactory|Referenced Assemblies
 - Sample.Contract project
 - b. Now that all the plumbing is done, you can start the interesting stuff, i.e. create the proxy and call the web methods. So, in the DoIt() method:
 - i. First define the Uri where the web service WSDL can be found in order to get all the information to generate the dynamic proxy.

```
Uri __wsHumanResourceUri = new  
Uri("http://localhost:1141/WebService/AccountManager.asmx?WSDL");
```
 - ii. Then instantiate a new ProxyFactory providing the web service's contract.

```
ProxyFactory<IAccountManager> __proxyFactory = new  
ProxyFactory<IAccountManager>(__wsHumanResourceUri);
```

- iii. Finally explicitly create the strongly typed proxy with the following statement:

```
IAccountManager __proxy = __proxyFactory.CreateProxy();
```

- iv. From now you can call any WebMethod for this web service like shown below:

```
ICustomer __customer = __proxy.GetCustomerDetails("PTarker");
```

Sample 2: .Net Remoting

The completed sample is available via the start menu at: Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Net Remoting Sample - Completed

The goal of this sample is the same as before, i.e. access bank account information. The only difference is that instead of using web services it uses .Net Remoting.

The following diagram summarizes the Remoting object:

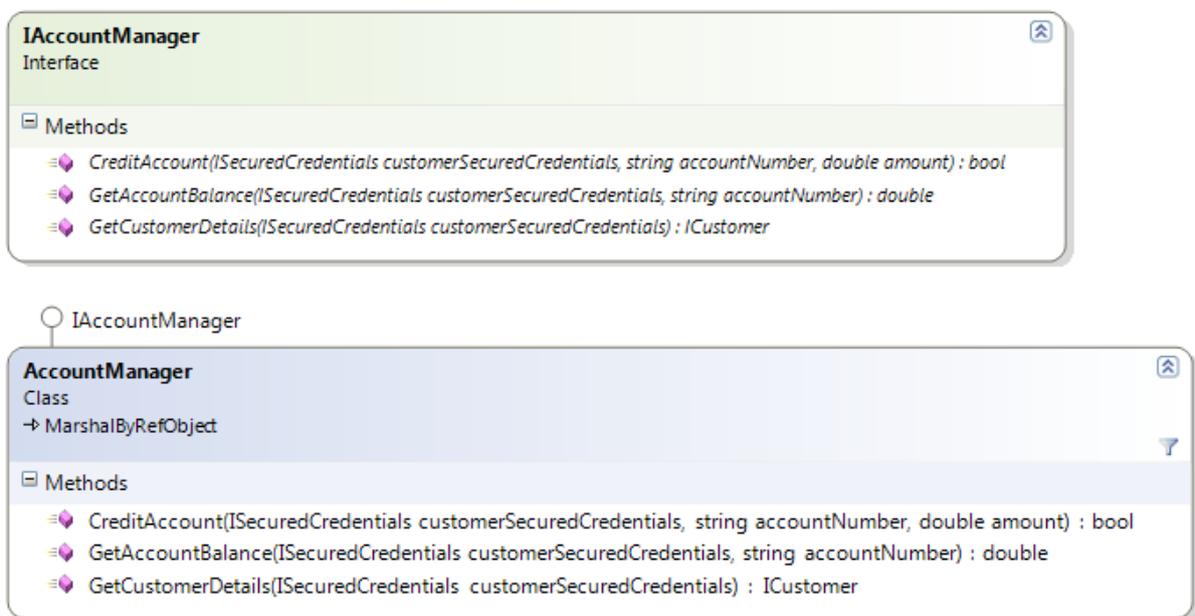


Figure 3: .Net Remoting contract.

The following diagram details the data structure used to interact with the remote object:

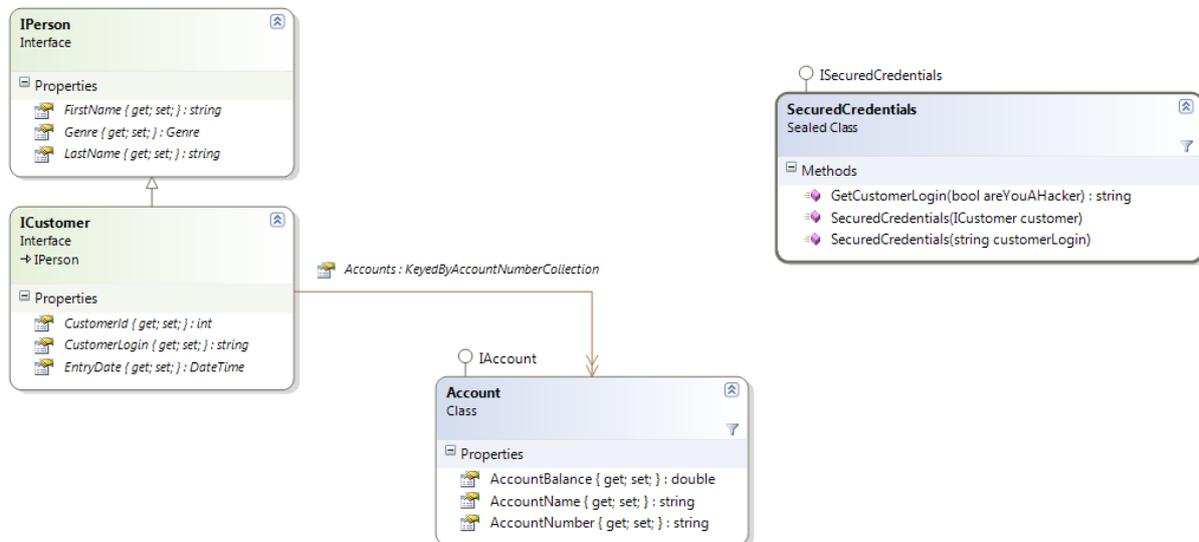


Figure 4: Data structures.

Step by step

Open the solution named “Net Remoting Sample – Step by step” in your start menu Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Net Remoting Sample – Step by step. This opens up a Visual Studio solution containing 3 projects: a console application hosting the remote object, a library that contains the contracts and data structures and finally a console application that will be a client application. These projects are mostly empty shells that you will have to complete.

1. Implement the contracts and data structures required in this sample.
 - a. In the Sample.Contracts project, create the data structure described in figure 4.
 - b. Again in Sample.Contracts, create the remote object along with its interface described in figure 3.
2. Create the remote object.
 - a. In the **Server** project, add a reference to the Sample.Contracts project,
 - b. Then add the following configuration information in the application’s config file

```

<system.runtime.remoting>
  <application>
    <service>
      <wellknown
        mode="SingleCall"
        type="Sample.Contracts.AccountManager, Sample.Contracts"
        objectUri="AccountManager.rem"/>
      <activated type="Sample.Contracts.AccountManager, Sample.Contracts"/>
    </service>
  </application>
</system.runtime.remoting>

```

```

</service>
<channels>
  <!--<channel ref="tcp" secure="true" port="9000"/>-->
  <channel ref="tcp" port="9000"/>
</channels>
</application>
</system.runtime.remoting>.

```

c. Finally implement the `IAccountManager` contract that you created in 1.b.

```

Configuration __configuration =
ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
RemotingConfiguration.Configure(__configuration.FilePath, false);

```

That's all server side.

Now comes the most interesting part of this sample which is how to use the `ProxyFactory` with `Remoting`.

1. Create the client

a. In the **RemotingClient** project, add references to :

- `ProxyFactory` assembly located under:

```

<Program files>|NetFxFactory|Getting started with ProxyFactory|Referenced
Assemblies

```

- **Sample.Contracts** project

b. Now that all the plumbing is done, you can start the interesting stuff, i.e. create the proxy and call the remote object. So, in the `DoIt()` method:

i. The first step is to define the remote object configuration in the config file:

```

<system.runtime.remoting>
  <application name="RemotingClient">
    <client url="tcp://localhost:9000">
      <wellknown
        type="Sample.Contracts.IAccountManager, Sample.Contracts"
        url="tcp://localhost:9000/AccountManager.rem"/>
      <!--<activated
        type="Sample.Contracts.AccountManager, Sample.Contracts"/>-->
    </client>
  </application>
  <channels>
    <channel ref="tcp" port="0" />
    <!--<channel
      ref="tcp"
      secure="true"
      port="0"
      useDefaultCredentials="true"/>-->
  </channels>
</application>
</system.runtime.remoting>

```

- ii. Then in the Program.DoIt(...) method, instantiate a new ProxyFactory providing the remote object contract.

```
ProxyFactory<IAccountManager> __proxyFactory = new  
ProxyFactory<IAccountManager>();
```

- iii. Finally explicitly create the strongly typed proxy with the following statement:

```
IAccountManager __proxy = __proxyFactory.CreateProxy();
```

- iv. From now you can call directly the remote object:

```
ICustomer __customer = __proxy.GetCustomerDetails("PTarker");
```

Well, if you have previously completed the web service sample you probably noticed that, except for ProxyFactory ctor, client side code this is the exact same steps () whether you are accessing web services or .Net Remote objects.

Sample 3: Going further – Asynchronous calls

The goal of this third sample is to dig into more details of what can be done with ProxyFactory. Even if ProxyFactory is based on a business contract it doesn't break any functionalities available to asmx. The well-known asynchronous support still works; furthermore using interface inheritance it is trivial to add the methods allowing asynchronous calls to the original contract client side. This is what will be demonstrated in the first section of this sample.

The completed sample is available via the start menu at:

Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Going Further – Asynchronous calls - Completed

The goal of this sample is the same as before, i.e. access bank account information. The only difference is that the call to the GetCustomerDetails method is done asynchronously.

Step by step

Open the solution named "Asynchronous Web Service Sample - Step by step" in your start menu Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Asynchronous Web Service Sample - Step by step. This opens up a Visual Studio solution containing 3 projects: a web project hosting the web service, a library that contains the contracts and data structures and finally a console application that will be a client application. These projects are exactly the same as what you get at the end of the Sample 1. The only thing you have to include in this step by step is the interface exposing IAccountManager asynchronously.

The interface is described below:

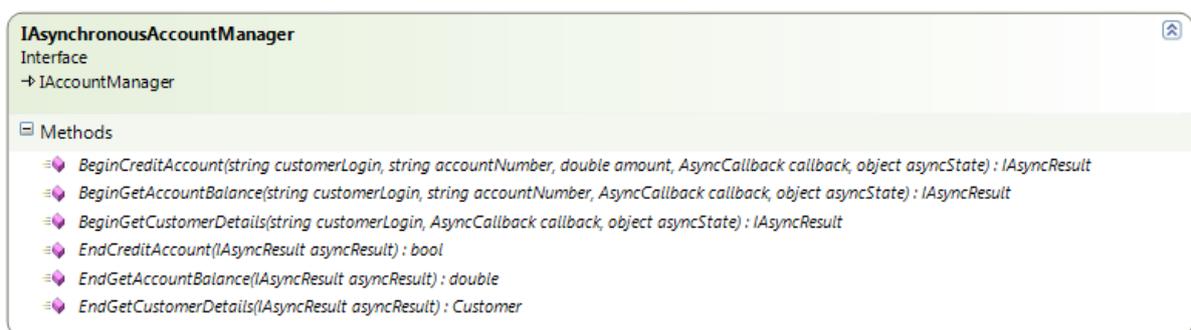


Figure 5: IAccountManager asynchronous interface

1. First, create the interface described figure 5 in the WSClient project, this interface is directly derived from the IAccountManager and apply the asynchronous call pattern defined by asmx (i.e. for each method of the interface you need to add a method with prefix Begin + method name along with the original parameters and two

additional ones specific to async calls (callback and asyncState) and another method prefixed with End + methodname with a AsyncResult parameter).

2. In WSCient.Program.DoIt() method, create a new ProxyFactory templated with the interface you created in the previous step instead of using directly IAccountManager. (remember, you can use the snippets deployed with this getting started package)
3. Create the GetCustomerDetail callback method like described below in pseudo code:

Method signature:

```
public static void GetCustomerDetailsCallback(IAsyncResult asyncResult)
```

Logic:

Get the proxy out of the asyncResult parameter (AsyncState property)

Get the completed result by calling the EndGetCustomerDetails method.

4. In the WebService project, add a 10s Thread.Sleep in the **AccountManager**. GetCustomerDetails in order to simulate a high latency.
5. Back in the WSCient project, Instantiate an AsyncCallback delegate after the ProxyFactory's CreateProxy method,
6. Finally, call the BeginGetCustomerDetails and pass in the AsyncCallback,
7. You are now ready to test your code!

Sample 4: Going further: dependency properties

Where the third sample demonstrated that using ProxyFactory no functionalities were lost, the goal of this last sample is to demonstrate the same level of parameterization of each CommunicationPlatform is still available. It is important to keep in mind that the ProxyFactory federates where WCF unifies; therefore each CommunicationPlatform has its own parameters. These are exposed via a [Dependency Properties](#) mechanism inspired by .Net 3.0's. Furthermore, this mechanism allows to define the cache policy associated to the dynamically generated assembly.

Here are the cache policies available:

Member name	Description
CheckWsdIForInvalidation	(Default Value). Clears the assembly cached locally only if the wsdl do not match the one used to generate this assembly.
LocalCacheOnly	Satisfies a request using the locally cached proxy assembly; does not send a request for an item that is not in the cache. When this cache policy level is specified, a TemporaryCacheException exception is thrown if the proxy assembly is not in the client cache.
LocalCacheIfAvailable	Satisfies a request using the locally cached proxy assembly but sends a request for an item that is not in the cache.
ClearAllCachedAssemblies	Clears all the assemblies cached locally before satisfying a request.
ClearMatchingCachedAssembly	Clears only the assembly cached locally that matches the request's wsdl.

Note: this is the way you can set the credentials or the web proxy information when using WSE 3.0.

Here is how to do set credentials in a WSE 3.0 scenario:

```
ProxyFactory<IService> __serviceFactory = new
ProxyFactory<IService>("MyService");
ICredentials __credentials = new NetworkCredential("username", "password");
__serviceFactory.SetPropertyValue(WebServicesPropertyMetadata.CredentialsPro
perty, __credentials);
IService __service = __serviceFactory.CreateProxy();
string __text = __service.HelloWorld();
```

The completed sample is available via the start menu at:

Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Going Further –
Dependency property - Completed

The goal of this sample is the same as sample 1, i.e. access bank account information. The only difference is that now you set explicitly the caching policy.

Step by step

Open the solution named "Asynchronous Web Service Sample - Step by step" in your start menu Start Menu\Programs\NetFxFactory \Getting started with ProxyModel\ Dependency Property Sample - Step by step. This opens up a Visual Studio solution containing 3 projects: a web project hosting the web service, a library that contains the contracts and data structures and finally a console application that will be a client application. These projects are exactly the same as what you get at the end of the Sample 1. The only thing you have to include in this step by step is a call to the proxy's SetPropertyValue in order to define the cache policy associated with this proxy.

1. Open WSClient then program.cs
2. In order to set the ProxyFactory cache policy, you only need the following:

```
__proxyFactory.SetPropertyValue(WebServicesPropertyMetadata.CachePolicyProperty  
, cachePolicy);
```

Where cachePolicy is passed as parameter to the Dolt method. This allows you to clear the cache on the first call to Dolt and then use the cached assembly for the next calls to improve performances.

3. You're done!

Sample 5: WSE 3.0 secured credentials sample

The goal of this sample is to demonstrate how to use ProxyFactory in a WSE 3.0 scenario by implementing a secured connection via WSE 3.0 credentials.

Step by step

Open the solution named "Web service WSE Sample – Step by step" in the start menu \ Programs \ NetFxFactory \ Getting started with ProxyModel\ Samples\ 5. WSE. The solution contains 3 projects: a web project hosting the WSE web service, a library containing contracts and data structures and finally a console application acting as a client.

The first step consists of deploying the X509 certificates from the Quickstart WSE sample by running setup.bat located in the following folder: <Program files>\Microsoft WSE\3.0\Samples. Those certificates will be used to authenticate the secured connection.

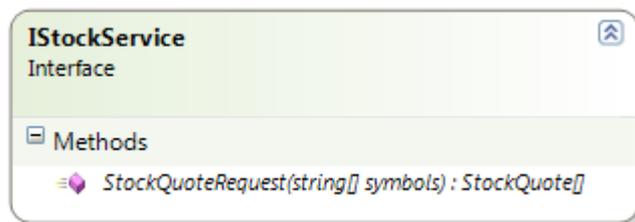


Figure 6: IStockService Interface

1. In the SampleBase project,
 - a. Code the interface described figure 6,
 - b. Have a look at Documents.cs describing the data structures involved,
2. In the SecureConversationPolicyService web project
 - a. Open wse3policycache.config and add the following configuration inside the `<usernameForCertificateSecurity>` element:

```
<serviceToken>
  <x509 storeLocation="LocalMachine"
    storeName="My"
    findValue="CN=WSE2QuickStartServer"
    findType="FindBySubjectDistinguishedName" />
</serviceToken>
```

- This configuration allows to use the certificates deployed earlier.
 - b. Have a look at the SecureConversationService.cs file

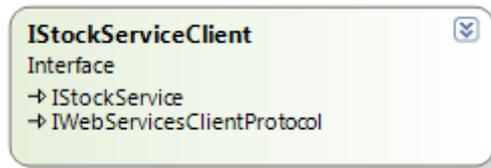


Figure 7 : IStockServiceClient Interface

3. In the SecureConversationPolicyClient project,
 - a. Create the IStockServiceClient interface described figure 7,
 - b. Open Program.cs and instantiate a new PrxoFactory in Program.Main un as described below:

```

ProxyFactory<IStockServiceClient> __proxyFactory = new
    ProxyFactory<IStockServiceClient>(new Uri(WSDL));
IStockServiceClient __proxy = __proxyFactory.CreateProxy();

```

- c. Then type the following code to create the token and set the policy to apply and finally call the web method:

```

// This is the username security token
//which is used by the service proxy
UsernameToken token =
    new UsernameToken(Environment.UserName, GetPassword(true));

// Username and password are set through code.
//X509 is set through policy.
__proxy.SetClientCredential(token);

// Set the ClientPolicy onto the proxy
// NB: Think to install the WSE QuickStart Sample X509 certificates.
__proxy.SetPolicy("ClientPolicy");

// Call the service
Console.WriteLine("Calling {0}", WSDL);
String[] symbols = { "FABRIKAM", "CONTOSO" };
StockQuote[] quotes = __proxy.StockQuoteRequest(symbols);

```

- d. Finally, open up app.config and add the following configuration in the <microsoft.web.services3> section:

```

<policy fileName="wse3policyCache.config" />

```

Sample 6: Configuration

The goal of this sample is to demonstrate how to take advantage of the configuration flexibility when using ProxyFactory..

Step by step

Open the solution named "Configuration – Step by step" in the start menu \ Programs \ NetFxFactory \ Getting started with ProxyModel\ Samples\ 6. Configuration. The solution contains 4 projects: a web project hosting the asmx web service, a server project hosting the remoting object, a library containing contracts and data structures and finally a console application acting as a client.

Everything is wired up so the only thing to complete in this sample is the client configuration and the client's calls.

1. Open app.config located in the client project and add the following configuration information inside the <client></client> elements as described below:

```
<endpoint
  binding="Remoting"
  contract="thinktecture.ProxyModel.Test.IStockService, Contract" />
<endpoint
  name="RemotingStockServiceOverWS"
  binding="WebServices"
  contract="thinktecture.ProxyModel.Test.IStockService, Contract"
  address="http://localhost:51640/WebService/StockService.asmx?WSDL" />
<endpoint
  name="RemotingStockServiceOverRemoting"
  binding="Remoting"
  contract="thinktecture.ProxyModel.Test.IStockService, Contract" />
```

2. Open program.cs located in the client project. In the Main method, add the following code to the corresponding place:

```
TestIt(new ProxyFactory<IStockService>());
```

This statement uses the first configuration element matching the IStockService contract found in the configuration file.

```
TestIt(new ProxyFactory<IStockService>("RemotingStockServiceOverWS"));
```

This statement uses the endpoint configuration named "RemotingStockServiceOverWS".

```
TestIt(new ProxyFactory<IStockService>("RemotingStockServiceOverRemoting"));
```

This statement uses the endpoint configuration named "RemotingStockServiceOverRemoting".

```
TestIt(new ProxyFactory<IStockService>(CommunicationTechnology.WebServices,
new Uri("http://localhost:51640/WebService/StockService.asmx?WSDL")));
```

This statement does not rely on configuration information but uses explicit parameters.

Take away

Tips and tricks

Be aware that when using the ProxyFactory constructor with the wsdl location, the wsdl is generated *each and every* time if accessed at <http://<machine>/mywebservice.asmx?wsdl> where it is generated only once, timestamped and cached when accessed at <http://<machine>/mywebservice.wsdl>. So be careful when setting the cache policy on the ProxyFactory using the dependency property `WebServicesPropertyMetadata.CachePolicyProperty`.

Interesting reading

[From .NET Remoting to the Windows Communication Foundation \(WCF\)](#)

[ASP.NET Web services to the Windows Communication Foundation](#)

[WSE->WCF](#)

[Migrating .NET Remoting to WCF \(and even ASMX!\)](#)

[Remoting Style Distributed Objects with WCF](#)

[Security Practices: .NET Framework 2.0 Security Practices at a Glance](#)

Disclaimers & copyright

Copyright (c) 2002-2006, thinktecture (<http://www.thinktecture.com>).

All rights reserved.

Copyright (c) 2006, netxfactory (<http://www.netxfactory.org>).

All rights reserved.

Copyright

The software and all other content of the distributed package, if not otherwise stated, is Copyright 2002-2006 thinktecture (<http://www.thinktecture.com/>).

All rights reserved.

2006 netxfactory (<http://www.netxfactory.org/>).

All rights reserved.

Terms of Use

Permission is hereby granted to use this software, for both commercial and non-commercial purposes, free of charge. Permission is hereby granted to copy and distribute the software for non-commercial purposes. A commercial distribution is NOT allowed without prior written permission of the author(s).

Further, redistribution and use in source and binary forms, with or without modification, are permitted only provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following warranty disclaimer.

(2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following warranty disclaimer in the documentation and/or other materials provided with the distribution.

(3) Neither the name of thinktecture, netxfactory, rioterdecker nor the names of its author(s) may be used to endorse or promote products derived from this software without specific prior written permission.

Warranty

This software is supplied "AS IS". The author(s) disclaim all warranties, expressed or implied, including, without limitation, the warranties of merchantability and of fitness for any purpose. The author(s) assume no liability for direct, indirect, incidental, special, exemplary, or consequential damages, which may result from the use of this software, even if advised of the possibility of such damage.

Submissions

The author(s) encourage the submission of comments and suggestions concerning this software. All suggestions will be given serious technical consideration. By submitting material to the author(s), you are granting the right to make any use of the material deemed appropriate, i.e. any communication or material that you transmit to the author(s) by electronic mail or otherwise, including any data, questions, comments, suggestions or the like, is, and will be treated as, non-confidential and nonproprietary information. The author(s) may use such communication or material for any purpose whatsoever including, but not limited to, reproduction, disclosure, transmission,

publication, broadcast and further posting. Further, the author(s) are free to use any ideas, concepts, know-how or techniques contained in any communication or material you send for any purpose whatsoever, including, but not limited to, developing, manufacturing and marketing products.