PRESTO (*PRotocole d'Echanges Standard et Ouvert*) Starter Kit Samples Guide

Document Details

Version: 1.0
Published: June 2007

Microsoft

Table of Contents

Disclaimer

This document is provided for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, IN THIS DOCUMENT. Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

License

The PRESTO Starter Kit for Microsoft .NET Framework 3.0 is published under the CeCILL-B Free Software license agreement as described at the following Internet address: http://www.cecill.info/licences/Licence_CeCILL-B_V1-en.txt.

For commodity reasons, the CeCILL-B license agreement is reproduced hereafter.

CeCILL-B FREE SOFTWARE LICENSE AGREEMENT

Notice

This Agreement is a Free Software license agreement that is the result of discussions between its authors in order to ensure compliance with the two main principles guiding its drafting:

* firstly, compliance with the principles governing the distribution of Free Software: access to source code, broad rights granted to users,
* secondly, the election of a governing law, French law, with which it is conformant, both as regards the law of torts and intellectual property law, and the protection that it offers to both authors and holders of the economic rights over software.

The authors of the CeCILL-B (for Ce[a] C[nrs] I[nria] L[ogiciel] L[ibre])license are:

Commissariat à l'Energie Atomique - CEA, a public scientific, technical and industrial research establishment, having its principal place of business at 25 rue Leblanc, immeuble Le Ponant D, 75015 Paris, France.

Centre National de la Recherche Scientifique - CNRS, a public scientific and technological establishment, having its principal place of business at 3 rue Michel-Ange, 75794 Paris cedex 16, France.

Institut National de Recherche en Informatique et en Automatique - INRIA, a public scientific and technological establishment, having its principal place of business at Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay cedex, France.

Preamble

This Agreement is an open source software license intended to give users significant freedom to modify and redistribute the software licensed hereunder.

The exercising of this freedom is conditional upon a strong obligation of giving credits for everybody that distributes a software incorporating a software ruled by the current license so as all contributions to be properly identified and acknowledged.

In consideration of access to the source code and the rights to copy, modify and redistribute granted by the license, users are provided only with a limited warranty and the software's author, the holder of the economic rights, and the successive licensors only have limited liability.

In this respect, the risks associated with loading, using, modifying and/or developing or reproducing the software by the user are brought to the user's attention, given its Free Software status, which may make it complicated to use, with the result that its use is reserved for developers and experienced professionals having in-depth computer knowledge. Users are therefore encouraged to load and test the suitability of the software as regards their requirements in conditions enabling the security of their systems and/or data to be ensured and, more generally, to use and operate it in the same conditions of security. This Agreement may be freely reproduced and published, provided it is not altered, and that no provisions are either added or removed herefrom.

This Agreement may apply to any or all software for which the holder of the economic rights decides to submit the use thereof to its provisions.

Article 1 - DEFINITIONS

For the purpose of this Agreement, when the following expressions commence with a capital letter, they shall have the following meaning:

Agreement: means this license agreement, and its possible subsequent versions and annexes.

Software: means the software in its Object Code and/or Source Code form and, where applicable, its documentation, "as is" when the Licensee accepts the Agreement.

Initial Software: means the Software in its Source Code and possibly its Object Code form and, where applicable, its documentation, "as is" when it is first distributed under the terms and conditions of the Agreement.

Modified Software: means the Software modified by at least one Contribution.

Source Code: means all the Software's instructions and program lines to which access is required so as to modify the Software.

Object Code: means the binary files originating from the compilation of the Source Code.

Holder: means the holder(s) of the economic rights over the Initial Software.

Licensee: means the Software user(s) having accepted the Agreement.

Contributor: means a Licensee having made at least one Contribution.

Licensor: means the Holder, or any other individual or legal entity, who distributes the Software under the Agreement.

Contribution: means any or all modifications, corrections, translations, adaptations and/or new functions integrated into the Software by any or all Contributors, as well as any or all Internal Modules.

Module: means a set of sources files including their documentation that enables supplementary functions or services in addition to those offered by the Software.

External Module: means any or all Modules, not derived from the Software, so that this Module and the Software run in separate address spaces, with one calling the other when they are run.

Internal Module: means any or all Module, connected to the Software so that they both execute in the same address space.

Parties: mean both the Licensee and the Licensor.

These expressions may be used both in singular and plural form.

Article 2 - PURPOSE

The purpose of the Agreement is the grant by the Licensor to the Licensee of a non-exclusive, transferable and worldwide license for the Software as set forth in Article 5 hereinafter for the whole term of the protection granted by the rights over said Software.

Article 3 - ACCEPTANCE

3.1 The Licensee shall be deemed as having accepted the terms and conditions of this Agreement upon the occurrence of the first of the following events:

* loading the Software by any or all means, notably, by downloading from a remote server, or by loading from a physical medium;
* the first time the Licensee exercises any of the rights granted hereunder.

3.2 One copy of the Agreement, containing a notice relating to the characteristics of the Software, to the limited warranty, and to the fact that its use is restricted to experienced users has been provided to the Licensee prior to its acceptance as set forth in Article 3.1 hereinabove, and the Licensee hereby acknowledges that it has read and understood it.

Article 4 - EFFECTIVE DATE AND TERM

4.1 EFFECTIVE DATE

The Agreement shall become effective on the date when it is accepted by the Licensee as set forth in Article 3.1.

4.2 TERM

The Agreement shall remain in force for the entire legal term of protection of the economic rights over the Software.

Article 5 - SCOPE OF RIGHTS GRANTED

The Licensor hereby grants to the Licensee, who accepts, the following rights over the Software for any or all use, and for the term of the Agreement, on the basis of the terms and conditions set forth hereinafter.

Besides, if the Licensor owns or comes to own one or more patents protecting all or part of the functions of the Software or of its components, the Licensor undertakes not to enforce the rights granted by these patents against successive Licensees using, exploiting or modifying the Software. If these patents are transferred, the Licensor undertakes to have the transferees subscribe to the obligations set forth in this paragraph.

5.1 RIGHT OF USE

The Licensee is authorized to use the Software, without any limitation as to its fields of application, with it being hereinafter specified that this comprises:

1. permanent or temporary reproduction of all or part of the Software by any or all means and in any or all form.

2. loading, displaying, running, or storing the Software on any or all medium.

3. entitlement to observe, study or test its operation so as to determine the ideas and principles behind any or all constituent elements of said Software. This shall apply when the Licensee carries out any or all loading, displaying, running, transmission or storage operation as regards the Software, that it is entitled to carry out hereunder.

5.2 ENTITLEMENT TO MAKE CONTRIBUTIONS

The right to make Contributions includes the right to translate, adapt, arrange, or make any or all modifications to the Software, and the right to reproduce the resulting software.

The Licensee is authorized to make any or all Contributions to the Software provided that it includes an explicit notice that it is the author of said Contribution and indicates the date of the creation thereof.

5.3 RIGHT OF DISTRIBUTION

In particular, the right of distribution includes the right to publish, transmit and communicate the Software to the general public on any or all medium, and by any or all means, and the right to market, either in consideration of a fee, or free of charge, one or more copies of the Software by any means.

The Licensee is further authorized to distribute copies of the modified or unmodified Software to third parties according to the terms and conditions set forth hereinafter.

5.3.1 DISTRIBUTION OF SOFTWARE WITHOUT MODIFICATION

The Licensee is authorized to distribute true copies of the Software in Source Code or Object Code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,

2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the Object Code of the Software is redistributed, the Licensee allows effective access to the full Source Code of the Software at a minimum during the entire period of its distribution of the Software, it being understood that the additional cost of acquiring the Source Code shall not exceed the cost of transferring the data.

5.3.2 DISTRIBUTION OF MODIFIED SOFTWARE

If the Licensee makes any Contribution to the Software, the resulting Modified Software may be distributed under a license agreement other than this Agreement subject to compliance with the provisions of Article 5.3.4.

5.3.3 DISTRIBUTION OF EXTERNAL MODULES

When the Licensee has developed an External Module, the terms and conditions of this Agreement do not apply to said External Module, that may be distributed under a separate license agreement.

5.3.4 CREDITS

Any Licensee who may distribute a Modified Software hereby expressly agrees to:

1. indicate in the related documentation that it is based on the Software licensed hereunder, and reproduce the intellectual property notice for the Software,

2. ensure that written indications of the Software intended use, intellectual property notice and license hereunder are included in easily accessible format from the Modified Software interface,

3. mention, on a freely accessible website describing the Modified Software, at least throughout the distribution term thereof, that it is based on the Software licensed hereunder, and reproduce the Software intellectual property notice,

4. where it is distributed to a third party that may distribute a Modified Software without having to make its source code available, make its best efforts to ensure that said third party agrees to comply with the obligations set forth in this Article .

If the Software, whether or not modified, is distributed with an External Module designed for use in connection with the Software, the Licensee shall submit said External Module to the foregoing obligations.

5.3.5 COMPATIBILITY WITH THE CeCILL AND CeCILL-C LICENSES

Where a Modified Software contains a Contribution subject to the CeCILL license, the provisions set forth in Article 5.3.4 shall be optional.

A Modified Software may be distributed under the CeCILL-C license. In such a case the provisions set forth in Article 5.3.4 shall be optional.

Article 6 - INTELLECTUAL PROPERTY

6.1 OVER THE INITIAL SOFTWARE

The Holder owns the economic rights over the Initial Software. Any or all use of the Initial Software is subject to compliance with the terms and conditions under which the Holder has elected to distribute its work and no one shall be entitled to modify the terms and conditions for the distribution of said Initial Software.

The Holder undertakes that the Initial Software will remain ruled at least by this Agreement, for the duration set forth in Article 4.2.

6.2 OVER THE CONTRIBUTIONS

The Licensee who develops a Contribution is the owner of the intellectual property rights over this Contribution as defined by applicable law.

6.3 OVER THE EXTERNAL MODULES

The Licensee who develops an External Module is the owner of the intellectual property rights over this External Module as defined by applicable law and is free to choose the type of agreement that shall govern its distribution.

6.4 JOINT PROVISIONS

The Licensee expressly undertakes:

1. not to remove, or modify, in any manner, the intellectual property notices attached to the Software;

2. to reproduce said notices, in an identical manner, in the copies of the Software modified or not.

The Licensee undertakes not to directly or indirectly infringe the intellectual property rights of the Holder and/or Contributors on the Software and to take, where applicable, vis-à-vis its staff, any and all measures required to ensure respect of said intellectual property rights of the Holder and/or Contributors.

Article 7 - RELATED SERVICES

7.1 Under no circumstances shall the Agreement oblige the Licensor to provide technical assistance or maintenance services for the Software.

However, the Licensor is entitled to offer this type of services. The terms and conditions of such technical assistance, and/or such maintenance, shall be set forth in a separate instrument. Only the Licensor offering said maintenance and/or technical assistance services shall incur liability therefore.

7.2 Similarly, any Licensor is entitled to offer to its licensees, under its sole responsibility, a warranty, that shall only be binding upon itself, for the redistribution of the Software and/or the Modified Software, under terms and conditions that it is free to decide. Said warranty, and the financial terms and conditions of its application, shall be subject of a separate instrument executed between the Licensor and the Licensee.

Article 8 - LIABILITY

8.1 Subject to the provisions of Article 8.2, the Licensee shall be entitled to claim compensation for any direct loss it may have suffered from the Software as a result of a fault on the part of the relevant Licensor, subject to providing evidence thereof.

8.2 The Licensor's liability is limited to the commitments made under this Agreement and shall not be incurred as a result of in particular: (i) loss due the Licensee's total or partial failure to fulfill its obligations, (ii) direct or consequential loss that is suffered by the Licensee due to the use or performance of the Software, and (iii) more generally, any consequential loss. In particular the Parties expressly agree that any or all pecuniary or business loss (i.e. loss of data, loss of profits, operating loss, loss of customers or orders, opportunity cost, any disturbance to business activities) or any or all legal proceedings instituted against the Licensee by a third party, shall constitute consequential loss and shall not provide entitlement to any or all compensation from the Licensor.

Article 9 - WARRANTY

9.1 The Licensee acknowledges that the scientific and technical state-of-the-art when the Software was distributed did not enable all possible uses to be tested and verified, nor for the presence of possible defects to be detected. In this respect, the Licensee's attention has been drawn to the risks associated with loading, using, modifying and/or developing and reproducing the Software which are reserved for experienced users.

The Licensee shall be responsible for verifying, by any or all means, the suitability of the product for its requirements, its good working order, and for ensuring that it shall not cause damage to either persons or properties.

9.2 The Licensor hereby represents, in good faith, that it is entitled to grant all the rights over the Software (including in particular the rights set forth in Article 5).

9.3 The Licensee acknowledges that the Software is supplied "as is" by the Licensor without any other express or tacit warranty, other than that provided for in Article 9.2 and, in particular, without any warranty as to its commercial value, its secured, safe, innovative or relevant nature.

Specifically, the Licensor does not warrant that the Software is free from any error, that it will operate without interruption, that it will be compatible with the Licensee's own equipment and software configuration, nor that it will meet the Licensee's requirements.

9.4 The Licensor does not either expressly or tacitly warrant that the Software does not infringe any third party intellectual property right relating to a patent, software or any other property right. Therefore, the Licensor disclaims any and all liability towards the Licensee arising out of any or all proceedings for infringement that may be instituted in respect of the use, modification and redistribution of the Software. Nevertheless, should such proceedings be instituted against the Licensee, the Licensor shall provide it with technical and legal assistance for its defense. Such technical and legal assistance shall be decided on a case-by-case basis between the relevant Licensor and the Licensee pursuant to a memorandum of understanding. The Licensor disclaims any and all liability as regards the Licensee's use of the name of the Software. No warranty is given as regards the existence of prior rights over the name of the Software or as regards the existence of a trademark.

Article 10 - TERMINATION

10.1 In the event of a breach by the Licensee of its obligations hereunder, the Licensor may automatically terminate this Agreement thirty (30) days after notice has been sent to the Licensee and has remained ineffective.

10.2 A Licensee whose Agreement is terminated shall no longer be authorized to use, modify or distribute the Software. However, any licenses that it may have granted prior to termination of the

Agreement shall remain valid subject to their having been granted in compliance with the terms and conditions hereof.

Article 11 - MISCELLANEOUS

11.1 EXCUSABLE EVENTS

Neither Party shall be liable for any or all delay, or failure to perform the Agreement, that may be attributable to an event of force majeure, an act of God or an outside cause, such as defective functioning or interruptions of the electricity or telecommunications networks, network paralysis following a virus attack, intervention by government authorities, natural disasters, water damage, earthquakes, fire, explosions, strikes and labor unrest, war, etc.

11.2 Any failure by either Party, on one or more occasions, to invoke one or more of the provisions hereof, shall under no circumstances be interpreted as being a waiver by the interested Party of its right to invoke said provision(s) subsequently.

11.3 The Agreement cancels and replaces any or all previous agreements, whether written or oral, between the Parties and having the same purpose, and constitutes the entirety of the agreement between said Parties concerning said purpose. No supplement or modification to the terms and conditions hereof shall be effective as between the Parties unless it is made in writing and signed by their duly authorized representatives.

11.4 In the event that one or more of the provisions hereof were to conflict with a current or future applicable act or legislative text, said act or legislative text shall prevail, and the Parties shall make the necessary amendments so as to comply with said act or legislative text. All other provisions shall remain effective. Similarly, invalidity of a provision of the Agreement, for any reason whatsoever, shall not cause the Agreement as a whole to be invalid.

11.5 LANGUAGE

The Agreement is drafted in both French and English and both versions are deemed authentic.

Article 12 - NEW VERSIONS OF THE AGREEMENT

12.1 Any person is authorized to duplicate and distribute copies of this Agreement.

12.2 So as to ensure coherence, the wording of this Agreement is protected and may only be modified by the authors of the License, who reserve the right to periodically publish updates or new versions of the Agreement, each with a separate number. These subsequent versions may address new issues encountered by Free Software.

12.3 Any Software distributed under a given version of the Agreement may only be subsequently distributed under the same version of the Agreement or a subsequent version.

Article 13 - GOVERNING LAW AND JURISDICTION

13.1 The Agreement is governed by French law. The Parties agree to endeavor to seek an amicable solution to any disagreements or disputes that may arise during the performance of the Agreement.

13.2 Failing an amicable solution within two (2) months as from their occurrence, and unless emergency proceedings are necessary, the disagreements or disputes shall be referred to the Paris Courts having jurisdiction, by the more diligent Party.

Version 1.0 dated 2006-09-05.

Abstract

After a short presentation of the "PRotocole d'Echanges Standard et Ouvert" 1.1 (aka PRESTO) protocol, this document walks the reader through the process of implementing with the Windows Communication Foundation (WCF) the various PRESTO roles outlined in the above specification. It adds a number of additional examples that illustrate foreseeable future capabilities.

This document is intended for architects, developers or any people that are interested in consuming (from a .NET code) or exposing services (in .NET) that "speak" PRESTO. Whatever the role people are going to play, the sample source code provided in the PRESTO Starter Kit is intended to be interoperable with other implementations that conform to the PRESTO specification (please refer to the PRESTO PROTOCOL AT A GLANCE section below).

Feedback

Your feedback is important to us. Your participation and feedback through the PRESTO Starter Kit Feedback mailbox (mailto:prestosk@microsoft.com) is appreciated to make the PRESTO Starter Kit better.

Prerequisites

This PRESTO Starter Kit requires a PC running Windows XP Service Pack 2, Windows Server 2003 Service Pack 1 or Windows Vista, set up using the instructions in the INSTALLING PRESTO STARTER KIT SAMPLES document.

The reader should ideally be familiar with Web Service technology, the C# language, and the .NET Framework to easily follow the sample code.

PRESTO protocol at a glance

The "PRotocole d'Echanges Standard et Ouvert" (aka PRESTO) specification published by the DGME SDAE[1] (Direction Générale pour la Modernisation de l'Etat – Service pour le Développement de l'Administration Electronique) aims at providing a generic message exchange layer for exchanging potentially any eGovernement messages in the context of the ADELE initiative (http://www.adele.gouv.fr).

Such a specification falls under a step common to several countries and should also be used as a basis for the communications between European partners even also to be opened with other extra national partners. Germany, Sweden, Denmark and Estonia are each one in the course of creation of a protocol close to PRESTO. The European protocol eLink having been abandoned, the European Commission, within the framework of program IDABC[2] (Interoperable Delivery off Side-European eGovernment Services to Public Administrations, Business and Citizens), assists these countries to define a profile common resting on PRESTO, but based on international profiles of interworking.

To achieve its design objectives, the PRESTO protocol is built on the WS-* (STAR, which stands for Secured, Transacted, Asynchronous & Reliable) stack. For an understanding of the WS-* stack and how the different specifications compose with each other, you can consult the "AN INTRODUCTION TO THE WEB SERVICES ARCHITECTURE AND ITS SPECIFICATIONS" white-paper at the following URL: http://msdn.microsoft.com/library/en-us/dnwebsrv/html/introwsa.asp.

The Web Services specifications that constitute the WS-* stack are referenced at the following URL: http://msdn.microsoft.com/library/en-us/dnglobspec/html/wsspecsover.asp

It mainly consists of a set of Web services specifications, along with clarifications, amendments, and restrictions of those specifications that promote interoperability. In its initial 1.0 version, the following specifications constitute its foundation:

- Simple Object Access Protocol (SOAP) 1.2 [SOAP 1.2],
- Web Service Description Language (WSDL) 1.1 [WSDL 1.1]
- WS-Addressing (W3C Member Submission 10 August 2004) [WS-AddressingAugust2004],
- SOAP Message Transfer Optimization Mechanism (MTOM) [MTOM],
- Web Services Reliable Messaging (WS-ReliableMessaging) 1.0 [WS-RM1.0],
- Web Services Security: SOAP Message Security 1.0 (WS-Security) [WS-Security].

Version 1.1 of the PRESTO protocol additionally supports the following specifications:

- WS-Addressing 1.0 [WS-Addressing1.0] (in replacement of [WS-AddressingAugust2004])
- Web Services Reliable Messaging (WS-ReliableMessaging) 1.1 [WS-RM1.1] (optional)

A future version of the PRESTO protocol will use policies to define the metadata associated with the related endpoints and be able to dynamically request and consume these policies based on the foundation of the

[1] See http://synergies.modernisation.gouv.fr/rubrique.php3?id_rubrique=165.

[2] Interoperable Delivery of European eGovernment Services to public Administrations, Businesses and Citizens, see http://europa.eu.int/idabc.

WS-Policy [WS-Policy], WS-RMPolicy [WS-RMPolicy], WS-SecurityPolicy [WS-SecurityPolicy], and WS-MetadataExchange [WS-MetadataExchange] Web service specifications.

This foundation currently enables supporting the following key Message Exchange Patterns (MEP) by the PRESTO protocol:

- One-Way message exchange;
- Request-Reply message exchange with anonymous sender;
- Request-Reply message exchange with addressable sender;

The following extensibility points will be addressed in a future version of the PRESTO protocol:

- Message routing via a SOAP intermediary;
- Message exchange with a third party (via another protocol);

The PRESTO Starter Kit is intended to be used alongside both:

- The PRESTO Technical Reference [PRESTO-Ref] that provides a normative profile for the set of Web services specifications on which the PRESTO Starter Kit samples rely.
- The PRESTO Guide [PRESTO-Guide] which provides a non-normative description of the overall PRESTO message exchanges model.

For further information on the PRESTO protocol please refer to the documents referenced in the REFERENCES section.

Windows Communication Foundation (WCF) Programming Model Overview

This section provides a high-level view of WCF's architecture and the related programming model. It is intended to explain WCF's key concepts and how they fit together in order understand the code behind the PRESTO Starter Kit samples.

A WCF Service is a program that exposes a collection of *Endpoint*s. Each Endpoint is a portal for communicating with the world. A Client is a program that exchanges messages with one or more Endpoints.

Endpoints

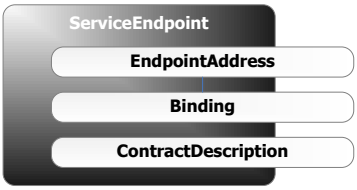
A service endpoint has an *Address*, a *Binding* and a *Contract* (or ABC):

A. The endpoint's address is a network address *where* the endpoint resides. The *EndpointAddress* class represents a WCF endpoint address.

B. The endpoint's binding specifies *how* the endpoint communicates with the world including things like transport protocol (e.g. HTTP), encoding (e.g. text, XOP/MTOM), and security requirements (e.g. SOAP message security). The *Binding* class represents a WCF binding.

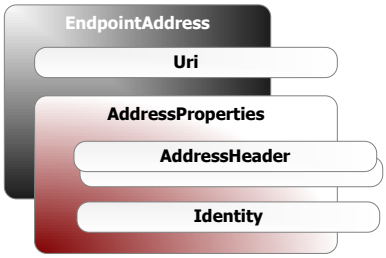
C. The endpoint's contract specifies *what* the endpoint communicates and is essentially a collection of messages organized in operations that have basic Message Exchange Patterns (MEPs) such as one-way and request/reply supported by the PRESTO protocol. The *ContractDescription* class represents a WCF contract.

The *ServiceEndpoint* class represents an endpoint and has an *EndpointAddress*, a *Binding* and a *ContractDescription* corresponding to the endpoint's address, binding and contract respectively:



Endpoint Address

An *EndpointAddress* is basically a URI, an identity and a collection of optional headers as shown hereafter.

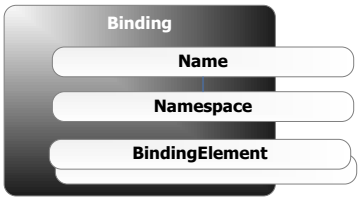


An endpoint's security identity is normally it's URI, however in some advanced scenarios the identity can be explicitly set independent of the URI using the Identity address property.

The optional headers are used to provide additional addressing information beyond the endpoint's URI. For example, address headers are useful for differentiating between multiple endpoints that share the same address URI.

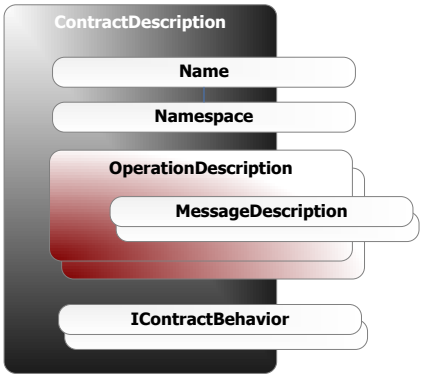
Bindings

A *Binding* has a name and namespace and a collection of composable binding elements. The binding's name and namespace uniquely identify it in the service's metadata. Each binding element describes an aspect of how the Endpoint communicates with the world.



Contracts

A *Contract* is a collection of *Operations* that specifies what the endpoint communicates to the outside world. Each operation is a simple message exchange, for example one-way or request/reply message exchange as supported by the PRESTO protocol.



The *ContractDescription* class is used to describe WCF Contracts and their operations. Within a *ContractDescription*, each contract operation has a corresponding *OperationDescription* that describes aspects of the operation such as whether the operation is one-way or request/reply. Each

OperationDescription also describes the messages that make up the operation using a collection of *MessageDescriptions*.

A *ContractDescription* is usually created from an interface or class that defines the contract using WCF's programming model. This type is annotated with *ServiceContractAttribute* and its methods that correspond to endpoint operations are annotated with *OperationContractAttribute*.

```
[assembly: ContractNamespaceAttribute("http://dgme.finances.gouv.fr/presto", ClrNamespace =
"dgme.finances.gouv.fr.presto")]
namespace dgme.finances.gouv.fr.presto
{
    [ServiceContractAttribute(Namespace = "http://dgme.finances.gouv.fr/presto")]
    public interface IPresto
    {
        [OperationContract(Action = "http://dgme.finances.gouv.fr/presto/submitOneWay", IsOneWay = true)]
        [XmlSerializerFormatAttribute()]
        void submitOneWay(submit1WayMessage message);
    }

    [ServiceContractAttribute(Namespace = "http://dgme.finances.gouv.fr/presto")]
    public interface IPresto2
    {
        [OperationContract(Action = "http://dgme.finances.gouv.fr/presto/submit",
              ReplyAction = "http://dgme.finances.gouv.fr/presto/submitResponse")]
        //[FaultContract(typeof(MessageSecurityException),ProtectionLevel=ProtectionLevel.EncryptAndSign)]
        [XmlSerializerFormatAttribute()]
        submitResponseMessage submit(submitRequestMessage request);
    }
}
```

Similar to bindings, each *Contract* has a *Name* and *Namespace* that uniquely identify it in the Service's metadata.

Each *Contract* also has a collection of *ContractBehaviors* which are modules that modify or extend the contract's behavior.

Behaviors

Behaviors are types that modify or extend Service or Client functionality. For example, the metadata behavior, implemented by *ServiceMetadataBehavior*, controls whether the Service publishes metadata. Similarly the security behavior controls impersonation and authorization while the transactions behavior controls enlisting in, and auto completing transactions.

Behaviors also participate in the process of building the channel and can modify that channel based on user-specified settings and/or other aspects of the Service or Channel.

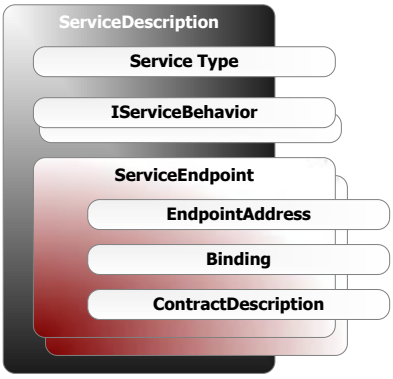
A service behavior is a type that implements *IServiceBehavior* and applies to Services. Similarly, a channel behavior is a type that implements *IChannelBehavior* and applies to Client channels.

Service and Channel Descriptions

The *ServiceDescription* class is an in memory structure that describes a WCF Service including the Endpoints exposed by the Service, the Behaviors applied to the Service and the type (a class) that implements the Service. *ServiceDescription* is used to create metadata, code/*App.config* file and channels.

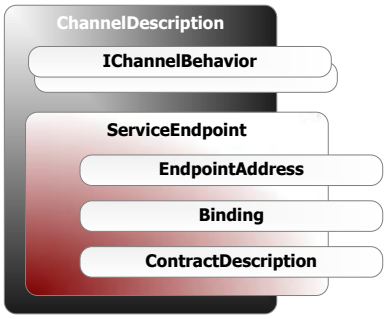
You can build this *ServiceDescription* object by hand. You can also create it from a type annotated with certain WCF attributes, which is the more common scenario. The code for this type can be written by hand or generated from a WSDL document using a WCF Service Model Metadata Utility tool called *svcutil.exe*.

Although *ServiceDescription* objects can be created and populated explicitly, they are often created behind the scenes as part of running the Service.



Similarly on the client side, a *ChannelDescription* describes a WCF Client's channel to a specific endpoint. The *ChannelDescription* class has a collection of *IChannelBehaviors* which are behaviors applied to the channel. It also has a *ServiceEndpoint* that describes the endpoint with which the channel will communicate.

Unlike *ServiceDescription*, *ChannelDescription* contains only one *ServiceEndpoint* that represents the target endpoint with which the channel will communicate.



WCF Runtime

The WCF runtime is the set of objects responsible for sending and receiving messages. For example, things like formatting messages, applying security and transmitting and receiving messages using a transport protocol as well as dispatching received messages to the appropriate operation all fall within the WCF runtime.

Message

The WCF *Message* is the unit of data exchange between a Client and an Endpoint Service. A *Message* is essentially an in-memory representation of a SOAP message InfoSet. Please note that *Message* is not tied

to text XML. Rather, depending on which encoding mechanism is used, a message can be serialized using WCF's text XML, XOP/MTOM or any other custom format.

Channels

Channels are the core abstraction for sending messages to and receiving messages from an endpoint. Broadly speaking, there are two categories of Channels:

1. Transport Channels handle sending or receiving opaque octet streams using some form of transport protocol such as HTTP.
2. Protocol Channels on the other hand implement a SOAP-based protocol by processing and possibly modify messages. For example, the security Channel adds and processes SOAP message headers and may modify the body of the message by encrypting it.

Channels are composable such that a Channel may be layered on top of another Channel that is in turn layered on top of a third Channel.

EndpointListener

An *EndpointListener* is the runtime equivalent of a *ServiceEndpoint*. The *EndpointAddress*, *Contract* and *Binding* of *ServiceEndpoint* (representing *where*, *what* and *how*), correspond to the *EndpointListener*'s listening address, message filtering and dispatch, and channel stack respectively. The *EndpointListener* contains the channel stack that is responsible for the sending and receiving messages.

ServiceHost and ChannelFactory

The WCF Service runtime is usually created behind the scenes by calling *ServiceHost.Open*. *ServiceHost* drives the creation of a *ServiceDescription* from on the Service type and populating the *ServiceDescription*'s *ServiceEndpoint* collection with endpoints defined in code or *App.config* file or both. *ServiceHost* then uses the *ServiceDescription* to create the channel stack in the form of an *EndpointListener* object for each *ServiceEndpoint* in the *ServiceDescription*.

Similarly, on the client side, the Client's runtime is created by a *ChannelFactory* which is the Client's equivalent of *ServiceHost*.

ChannelFactory drives the creation of a *ChannelDescription* based on a *Contract* type, a *Binding* and an *EndpointAddress*. It then uses this *ChannelDescription* to create the Client's channel stack.

Unlike the Service runtime, the Client runtime does not contain *EndpointListeners* because a Client always initiates connection to the Service so there is no need to "listen" for incoming connections.

PRESTO Starter Kit Samples Summary

The PRESTO Starter Kit provides samples for all the roles outlined in the PRESTO protocol specification. The samples are grouped so that you can easily locate the samples appropriate to your needs. All the samples are written in C#.

PRESTO Prototype Samples

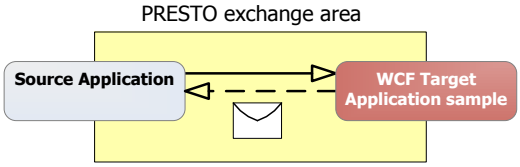
The PRESTO-enabled target application and source application samples below have been used for the several PRESTO interoperability tests conducted by the DGME SDAE.

Theses samples are located under the *Samples\Prototype* subdirectory under the directory location where you've installed the PRESTO Starter Kit. This subdirectory contains a solution file (.sln) Visual Studio 2005 that enables to build the SOAP intermediary samples.

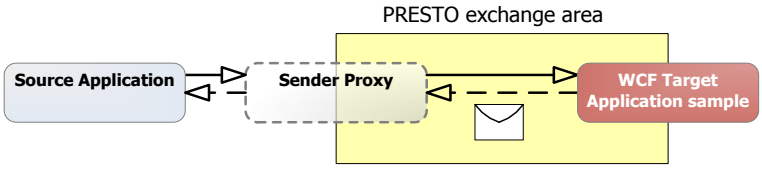
PRESTO-enabled Target Application (WCF)

A PRESTO-enabled target application is an application that can "speak" the PRESTO protocol to receive PRESTO-compliant messages from:

- Another PRESTO-enabled source application that exposes a PRESTO protocol endpoint:



- A PRESTO sender proxy:



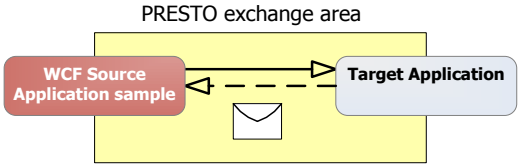
The PRESTO-enabled source application sample is provided as a Windows application. This sample is located under the *Samples\Prototype\TargetApplication* subdirectory.

It is indented to be used in conjunction with either the PRESTO-enabled source application or PRESTO add-in for Office system 2007 below. The sample can be easily modified to act as PRESTO receiver proxy for other applications.

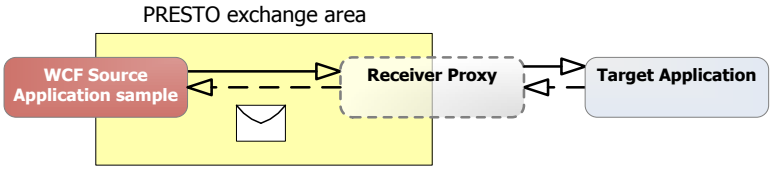
PRESTO-enabled Source Application (WCF)

A PRESTO-enabled source application is an application that can "speak" the PRESTO protocol to send PRESTO-compliant messages to:

- Another PRESTO-enabled application that exposes such PRESTO protocol endpoint:



- A PRESTO receiver proxy:



The PRESTO-enabled source application sample is a Windows application. This sample is located under the *Samples\Prototype\SourceApplication* subdirectory.

It is indented to be used in conjunction with above the PRESTO-enabled target application sample. The sample can be easily modified to act as sender proxy for other applications.

Notes/comments

The dichotomy introduced in the PRESTO specification between a PRESTO sender proxy and a PRESTO receiver proxy is purely logical to reflect their respective role they play in the message exchange. The former one is a service that generates and sends a PRESTO-compliant message to the latter one with regards to the PRESTO protocol. Subsequently, a PRESTO receiver proxy is a service that receives and consumes a PRESTO-compliant message with regards to the PRESTO protocol. As such, PRESTO proxies implement and expose a PRESTO protocol endpoint to the eGovernment partners and enable them to interoperate with any service that "talks" the PRESTO protocol. A PRESTO proxy implementation may play both roles.

A product like Microsoft BizTalk Server 2006 R2 can play simultaneously these two roles with a Windows Communication Foundation (WCF) adapter.

BizTalk Server 2006 R2 picks up where WCF leaves off and vice versa:

- WCF is the platform for building services on the Windows platforms and BizTalk Server 2006 R2 is the infrastructure for orchestrating and extending WCF services;
- BizTalk Server 2006 R2 is a Standards based Integration and Business Process Management Server for Windows and WCF supplies Standard communication protocols for services on the Windows platforms;

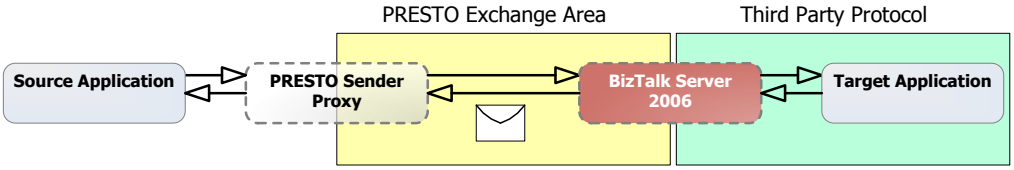
The PRESTO Adapters Starter Kit for Microsoft BizTalk Server 2006 R2 provides dedicated PRESTO adapters for Microsoft BizTalk Server R2 2006. The PRESTO Adapters Starter Kit for Microsoft BizTalk Server 2006 R2 published under the CeCILL-B[3] Free Software license agreement can be downloaded from

[3] See http://www.cecill.info/licences/Licence_CeCILL-B_V1-en.txt.

http://www.microsoft.com/downloads/details.aspx?FamilyID=826D4D2D-8E8C-439C-8104-B6DB89EEE626&displaylang=en.

Furthermore, when a third party application is the target destination of an initial PRESTO-compliant message, the Microsoft BizTalk Server environment natively offers the necessary logic to act as a gateway, i.e. brokered application to application integration and business to business integration, with the ability to:

i. Translate with complex mapping support PRESTO-compliant messages (from/)to third party messages;
ii. Route message onto the third party network, taking into account the third party protocol characteristics;



PRESTO add-in for Office system 2007 (WCF/VSTO)

The PRESTO add-in for Office system 2007 leverages the code of the PRESTO-enabled source application and provides protocol integration within Microsoft Word 2007.

This add-in sample is located under the *Samples\Prototype\WordAddIn4Presto* subdirectory. It is indented to be used in conjunction with above the PRESTO-enabled target application.

Beyond the PRESTO protocol: Message Chunking Samples

Theses samples are located under the *Samples\Beyond - Message Chunking* subdirectory under the directory location where you installed the PRESTO Starter Kit. This subdirectory contains a solution file (.sln) Visual Studio 2005 that enables to build the SOAP intermediary samples.

Important Note

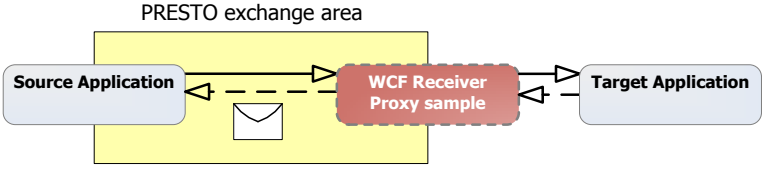
Chunking for message payload is NOT part of the current PRESTO specification. However, such an approach enables to have a smaller exchange unit for reliability recovering.

Hence, these samples are provided to illustrate potential foreseeable future capabilities for the PRESTO protocol.

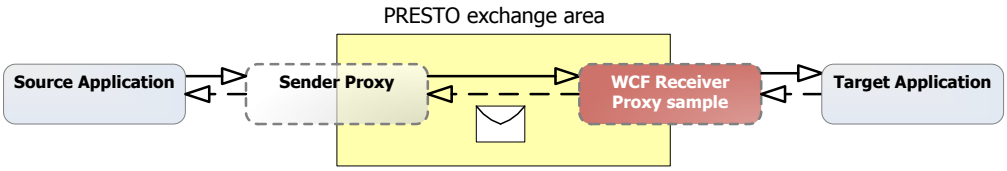
PRESTO-like Receiver Proxy with Chunking Support (WCF)

Generally speaking, a PRESTO receiver proxy is a service or an application that can "speak" the PRESTO protocol to receive PRESTO-compliant messages on behalf of a "legacy" application from:

- A PRESTO-enabled source application:



- A PRESTO sender proxy:



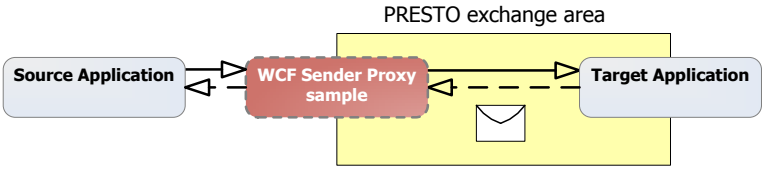
The receiver proxy (with chunking support) sample is provided as a console application. Please be aware that this proxy does NOT fully implement the PRESTO protocol specification for proxies.

This sample is located under the *Samples\Beyond - Message Chunking\ReceiverProxy* subdirectory. It is indented to be used exclusively in conjunction with below sender proxy sample. It does NOT work with any other Client sample.

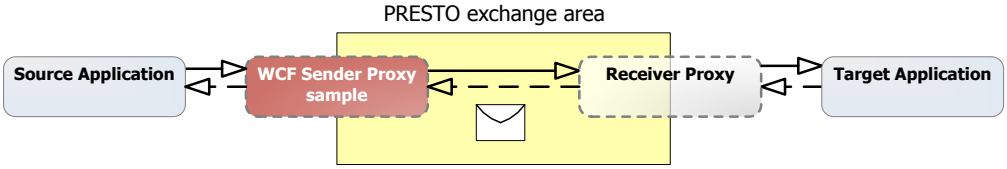
PRESTO-like Sender Proxy with Chunking Support (WCF)

Generally speaking, a PRESTO sender proxy is a service or an application that can "speak" the PRESTO protocol to send PRESTO-compliant messages on behalf of a "legacy" application to:

- A PRESTO-enabled target application:



- A PRESTO receiver proxy:



The sender proxy (with chunking support) sample is provided as a console application. Please be aware that this proxy does NOT fully implement the PRESTO protocol specification for proxies.

This sample is located under the *Samples\Beyond - Message Chunking\SenderProxy* subdirectory. It is indented to be used exclusively in conjunction with above receiver proxy sample. It does NOT work with any other Service sample.

Beyond the PRESTO protocol: SOAP intermediary Samples

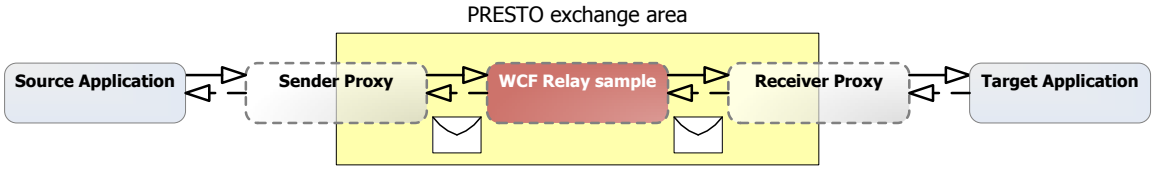
Theses samples are located under the *Samples\Beyond - Soap Intermediary* subdirectory under the directory location where you've installed the PRESTO Starter Kit. This subdirectory contains a solution file (.sln) Visual Studio 2005 that enables to build the SOAP intermediary samples.

Important Note

SOAP intermediary or relay is not part of the current PRESTO specification. However, this Message Routing pattern is an extensibility point that will be addressed in a future version of the PRESTO protocol. These samples are provided to illustrate foreseeable future capabilities.

PRESTO-like Relay (WCF)

The message path from a PRESTO-enabled source application or a PRESTO sender proxy to a PRESTO receiver proxy or PRESTO-enabled target application may involve one or multiple PRESTO relay(s). PRESTO relays are SOAP intermediaries in this message path. They are mainly intended to provide routing capabilities but nothing prevents them to provide additional services.



The PRESTO-enabled relay sample is provided as a console application. This sample is located under the *Samples\Beyond - Soap Intermediary\SoapRelay* folder. It is indented to be used exclusively in conjunction with below receiver proxy and sender proxy samples. It does NOT work with any other sample.

PRESTO-like Receiver Proxy (WCF)

The receiver proxy sample is provided as a console application. Please be aware that this proxy does NOT fully implement the PRESTO protocol specification for proxies.

This sample is located under the *Samples\Beyond - Soap Intermediary\ReceiverProxy* subdirectory. It is indented to be used exclusively in conjunction with both above SOAP relay sample and below sender proxy sample. It does NOT work with any other Client sample.

PRESTO-like Sender Proxy (WCF)

The sender proxy sample is provided as a Windows application. Please be aware that this proxy does NOT fully implement the PRESTO protocol specification for proxies.

This sample is located under the *Samples\Beyond - Soap Intermediary\SenderProxy* subdirectory. It is indented to be used exclusively in conjunction with above SOAP relay sample and receiver proxy sample. It does NOT work with any other Service sample.

Building the PRESTO Starter Kit Samples

The PRESTO Starter Kit samples can be built using Visual Studio 2005 or using the *msbuild* command from the command line. Both procedures are described in this topic.

Note

Before building or running any of the PRESTO Starter Kit Samples, please ensure you have performed the One-Time Setup Procedure in the INSTALLING PRESTO STARTER KIT SAMPLES document.

Building the samples using a command prompt

In order to build the samples using a command prompt, please perform the following steps:

1. Open the SDK command prompt and navigate to the *Samples*\<*SampleSet*> subdirectory under the directory location where you've installed the PRESTO Starter Kit, <*SampleSet*> is one of the followings:

   - *Prototype* (see section PRESTO PROTOTYPE SAMPLES);
   - *Beyond - Message Chunking* (see section BEYOND THE PRESTO PROTOCOL: MESSAGE CHUNKING SAMPLES);
   - *Beyond - Soap Intermediary* (see section BEYOND THE PRESTO PROTOCOL: SOAP INTERMEDIARY SAMPLES);

2. Type *msbuild* at the command line. All the PRESTO Starter Kit Samples are built to the *Samples*\<*SampleSet*>*bin**[Debug|Release]* subdirectory. The Debug vs. Release subdirectory depends on the eponym chosen target for the build.

Building the samples using Visual Studio 2005

In order to build the samples using Visual Studio 2005, please perform the following steps:

1. Open Windows Explorer and navigate to the *Samples*\<*SampleSet*> subdirectory under the directory location where you've installed the PRESTO Starter Kit, <*SampleSet*> is one of the followings:

   - *Prototype* (see section PRESTO PROTOTYPE SAMPLES);
   - *Beyond - Message Chunking* (see section BEYOND THE PRESTO PROTOCOL: MESSAGE CHUNKING SAMPLES);
   - *Beyond - Soap Intermediary* (see section BEYOND THE PRESTO PROTOCOL: SOAP INTERMEDIARY SAMPLES);

2. Double-click the .sln file icon located in the *Samples*\<*SampleSet*> subdirectory to open the file in the Visual Studio environment.

3. In the Build menu, select Rebuild Solution. All the PRESTO Starter Kit Samples are built to the *Samples*\<*SampleSet*>*bin**[Debug|Release]* subdirectory. The *Debug* vs. *Release* subdirectory depends on the eponym chosen target for the build.

Important Security Information about Metadata Endpoints

To prevent unintentional disclosure of potentially sensitive service metadata, the default configuration for Windows Communication Foundation (WCF) services should disable metadata publishing. Such a behavior is secure by default, but also means that you cannot use a metadata import tool (such as *Svcutil.exe*) to generate the client code required to call the service unless the service's metadata publishing behavior is explicitly enabled in configuration.

In order to make experimenting with the samples easier and to illustrate the metadata exchange capabilities, almost all samples expose an unsecured metadata publishing endpoint. Such endpoints are potentially available to anonymous unauthenticated consumers and care must be taken before deploying such endpoints to ensure that publicly disclosing a service's metadata is appropriate. See the PRESTO target application sample for details on publishing service metadata.

Running the PRESTO Starter Kit Samples

The PRESTO Starter Kit samples can be run in a single-machine or cross-machine configuration. As supplied, the samples are ready for running on a single machine.

In a cross-machine configuration, it is necessary to modify a sample's *App.config* file settings. The following procedures explain how to run a sample in same-machine and cross-machine configurations.

Note

Before building or running any of the PRESTO Starter Kit Prototype Samples, please ensure you have performed the One-Time Setup Procedure in the INSTALLING PRESTO STARTER KIT SAMPLES document.

Running the samples on the same machine

Perform the following steps:

1. Run the Service from *Samples*\<*SampleSet*>*bin**[Debug|Release]*, from under the directory location where you've installed the PRESTO Starter Kit. <*SampleSet*> is one of the followings:

   - *Prototype* (see section PRESTO PROTOTYPE SAMPLES);
   - *Beyond - Message Chunking* (see section BEYOND THE PRESTO PROTOCOL: MESSAGE CHUNKING SAMPLES);
   - *Beyond - Soap Intermediary* (see section BEYOND THE PRESTO PROTOCOL: SOAP INTERMEDIARY SAMPLES);

   Service activity is displayed on the Service main window or console window.

2. Run the Client from *Samples*\<*SampleSet*>*bin**[Debug|Release]*, from under the directory location where you've installed the PRESTO Starter Kit. Client activity is displayed on the Client main window or console window.

3. If the Client and Service are not able to communicate, see the TROUBLESHOOTING TIPS section.

Running the samples across machines

Perform the following steps to run the samples across machines:

1. Copy the Client program files from *Samples*\<*SampleSet*>*bin**[Debug|Release]*, from under the directory location where you've installed the PRESTO Starter Kit, to the Client machine. <*SampleSet*> is one of the followings:

   - *Prototype* (see section PRESTO PROTOTYPE SAMPLES);
   - *Beyond - Message Chunking* (see section BEYOND THE PRESTO PROTOCOL: MESSAGE CHUNKING SAMPLES);

- *Beyond - Soap Intermediary* (see section BEYOND THE PRESTO PROTOCOL: SOAP INTERMEDIARY SAMPLES);

2. Open the client configuration file and change the address value of the endpoint definition to match the new address of the Service. Replace any references to "localhost" with a fully-qualified domain name in the address.

3. On the Client machine, launch the Client from a command prompt.

Specific instructions are given on a per Client sample basis whenever it is appropriate.

Debugging a sample

Perform the following steps to debug a sample:

1. Select the Debug mode (default) and build the related solution (using the Build menu or CTRL+SHIFT+B).

   You can now set breakpoints in the sample code and enable breakpoints on exceptions.

2. Right-click the sample project item and choose Debug, Start new instance.

Troubleshooting Tips

Running the samples on Windows Vista

One of the major changes in Windows Vista security is that most people are no longer going to be running with Administrator privileges by default like they were doing on earlier platforms. This impacts your ability to run HTTP web services because listening at a particular HTTP address is a restricted operation. By default, every HTTP path is reserved for use by the system administrator. Your services will fail to start with an *AddressAccessDeniedException* if you aren't running the service from an elevated account. The PRESTO Starter Kit sample codes make no exception of this.

Windows Server 2003 includes a tool called *httpcfg.exe* that lets the owner of an HTTP namespace delegate that ownership to another user. On Windows Vista, *httpcfg.exe* is no longer included and instead there's a new command set available through *netsh.exe*.

We walk through delegating part of the HTTP namespace to get a web service working that wants to both listen at:

- http://localhost:9999 for metadata exchange;
- http://localhost:8080 for accepting incoming requests;

Such a behavior is for instance the default for the PRESTO-enabled target application that shows how to quickly expose in a target application a WCF service endpoint that conforms to the PRESTO protocol.

Since you will probably not running as the Administrator when debugging in Visual Studio, the PRESTO-enabled target application will fail to start when you'll run it as follows:

```
HTTP could not register URL http://+:8080/. Your process does not have access rights to
this namespace (see http://go.microsoft.com/fwlink/?LinkId=70353 for details).
```

The plus sign in the URL just means that there's a wildcard being applied to the hostname.

To fix this problem, we first need to start a command prompt using "Run as administrator" so that we have elevated privileges. Then, we can use *netsh.exe* to give some of the Administrator's HTTP namespace to your user account. You can look at the existing HTTP namespace delegations by using the following command line at the prompt: *netsh http show urlacl*.

There should be several namespaces set up by default, including the default one that WCF uses for temporary addresses as an example.

```
Reserved URL            : http://+:80/Temporary_Listen_Addresses/
    User: \Everyone
        Listen: Yes
        Delegate: No
        SDDL: D:(A;;GX;;;WD)
```

Now, we are going to use the following two commands to assign some of the HTTP namespace to your user account use:

```
netsh http add urlacl url=http://+:9999/ user=<YOURMACHINE\YourUserName>
netsh http add urlacl url=http://+:8080/ user=<YOURMACHINE\YourUserName>
```

You can get the syntax for all of these commands by running the following command line without any arguments at the prompt: "*netsh http*".

Please note that we've matched the URL in this command to the URL that appeared in the error message. The wildcarding is important for getting the right reservation and you'll continue to be denied access if your reservation covers less than your service's attempted registration. Going back to Visual Studio, the service should now start up and run as expected.

PRESTO-enabled Target Application (WCF)

What this sample does

This sample shows how to quickly expose in a target application a WCF Service endpoint that conforms to the PRESTO protocol.

This sample includes support for the various PRESTO protocol settings as defined in its initial specification.

Key Concepts Illustrated

This sample includes the logic to dynamically build in code a WCF custom binding for the WCF Service endpoint exposed by the application. The binding can also be set in the service *App.config* file where several predefined relevant bindings are proposed by default. In both cases, the settings must match the one of its PRESTO counterpart: the source application or the PRESTO sender proxy. The UI enables to select how the binging is set.

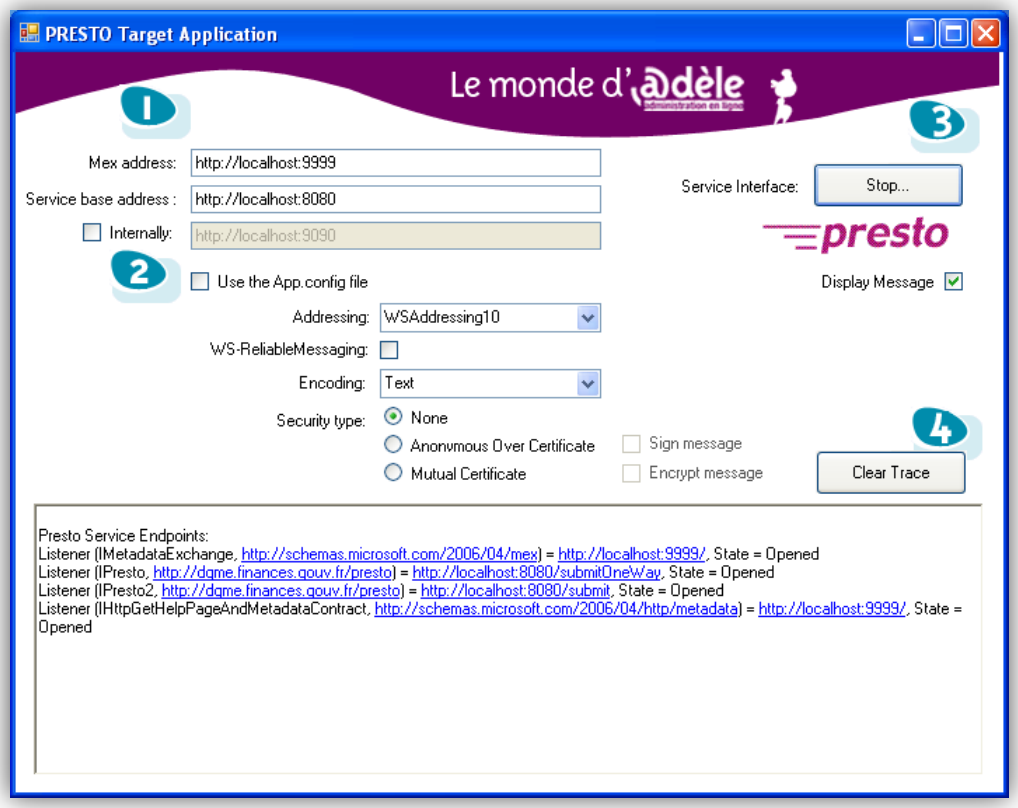
This sample also automatically exposes a metadata exchange endpoint so that its PRESTO counterpart can dynamically retrieve the protocol settings via WS-MetadataExchange [WS-MetadataExchange]. This metadata exchange endpoint is typically consumes by the PRESTO Source Application sample.

Please note that this is not part of the current PRESTO specification. This implementation is provided to illustrate foreseeable future capabilities.

How to run

Perform the following steps to run the sample:

1. Open Windows Explorer and navigate to the *Samples\Prototype\bin\[Debug|Release]* subdirectory under the directory location where you've installed the PRESTO Starter Kit.
2. Double-click the *TargetApplication.exe* Windows application.



Note

As illustrated above, the TargetApplication.exe listens on the service base address specified in the eponym field plus:

- the /submitOneWay suffix for the IPresto interface (OneWay);
- the /submit suffix) suffix for the IPresto2 interface (Request/Reply);

Theses suffixes are silently added to the specified service base address when the Start button is clicked.

3. Double-click the *SourceApplication.exe* Windows application.

Defining and Implementing a Contract

As previously above, the easiest way to define a contract is creating an interface or a class and annotating it with *ServiceContractAttribute* allowing the system to easily create from it a *ContractDescription*.

When using interfaces or classes to define contracts, each interface or class method that is a member of the contract must be annotated with *OperationContractAttribute*:

```
using System.ServiceModel;

[assembly: ContractNamespaceAttribute("http://dgme.finances.gouv.fr/presto", ClrNamespace =
"dgme.finances.gouv.fr.presto")]
namespace dgme.finances.gouv.fr.presto
{
    // A WCF contract defined using an interface (IPresto)
    [ServiceContractAttribute(Namespace = "http://dgme.finances.gouv.fr/presto")]
    public interface IPresto
```

```
    {
        [OperationContract(Action = "http://dgme.finances.gouv.fr/presto/submitOneWay", IsOneWay = true)]
        [XmlSerializerFormatAttribute()]
        void submitOneWay(submit1WayMessage message);
    }

    [ServiceContractAttribute(Namespace = "http://dgme.finances.gouv.fr/presto")]
    public interface IPresto2
    {
        [OperationContract(Action = "http://dgme.finances.gouv.fr/presto/submit",
               ReplyAction = "http://dgme.finances.gouv.fr/presto/submitResponse")]
        //[FaultContract(typeof(MessageSecurityException),ProtectionLevel=ProtectionLevel.EncryptAndSign)]
        [XmlSerializerFormatAttribute()]
        submitResponseMessage submit(submitRequestMessage request);
    }
}
```

Implementing the contract in this case is simply a matter of creating a class that implements both IPresto and IPresto2. That class becomes the WCF Service class:

```
// the service class implements the IPresto and IPresto2 interfaces
[ServiceBehavior(IncludeExceptionDetailInFaults = true)]
public class PrestoService : AbstractPrestoService, IPresto, IPresto2
{

    public void submitOneWay(submit1WayMessage messageIn)
    {
        TargetApplication.OutputTrace.Log("Processing PrestoService(IPresto)::submitOneWay");

        if ( (messageIn != null) && (messageIn.submitOneWay != null) )
            SaveMessage(messageIn.submitOneWay.testDocIn);
    }

    public submitResponseMessage submit(submitRequestMessage messageIn)
    {
        try
        {
           TargetApplication.OutputTrace.Log("Processing PrestoService(IPresto2)::submit");

           submitResponseMessage responseMsg = new submitResponseMessage();
           responseMsg.submitResponse = new submitResponse();
           responseMsg.submitResponse.testDocOut = messageIn.submit.testDocIn;

           if ( (messageIn != null) && (messageIn.submit != null) )
               SaveMessage(messageIn.submit.testDocIn);

               return responseMsg;
         }
         catch (Exception ex)
         {
           System.Diagnostics.Trace.Write(ex.Message);
           if (ex.InnerException != null)
               System.Diagnostics.Trace.Write(ex.InnerException.Message);

               return null;
         }
    }
}
```

Defining a Custom binding for the service

This sample enables to create a custom binding and related binding elements either programmatically or in the Service *App.config* to assemble a binding using system defined binding elements that fulfill the PRESTO protocol requirements.

Each binding element represents a processing step when receiving (or sending) messages. At runtime, binding elements create the listeners (and channels) necessary to build incoming (as well as outgoing) channel stacks.

The binding element collection for the PRESTO protocol contains possibly for three main types of binding elements. The presence of each binding element describes part of the *how* of communicating with the endpoint:

1. The protocol binding elements – These elements represent higher-level processing steps that act on messages. Listeners (and channels) created by these binding elements can add, remove, or modify the message content. A given binding may have an arbitrary number of the protocol binding elements, each inheriting from the *BindingElement* class. WCF includes several protocol binding elements, including the *ReliableSessionBindingElement* and the *SymmetricSecurityBindingElement* **,** which are very useful technical enablers for the implementation of a message exchange solution conform to the PRESTO specification.

   The *ReliableSessionBindingElement* indicates that the Endpoint uses reliable messaging [WS-ReliableMessaging] to provide message delivery assurances. The *SymmetricSecurityBindingElement* indicates that the Endpoint uses SOAP message security [WS-Security].

   Each binding element usually has properties that further describe the specifics of the how of communicating with the Endpoint. For example, the *ReliableSessionBindingElement* has an *Assurances* property that specifies the required message delivery assurances, such as none, at least once, at most once or exactly once;

2. The encoding binding elements – These elements represent transformations between a message and an encoding ready for transmission on the wire. Typical WCF bindings include exactly one encoding binding element. The following two encoding binding elements are used to confirm with the PRESTO specification: *MtomMessageEncodingBindingElement* vs. *TextMessageEncodingBindingElement*.

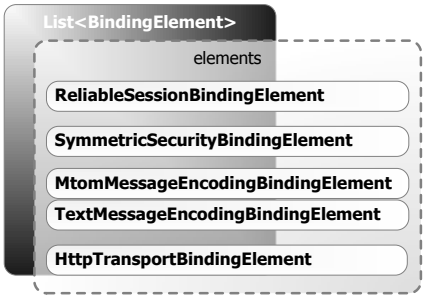
   The *MtomMessageEncodingBindingElement* indicates that the Endpoint uses XOP/MTOM [MTOM] for the message encoding scheme. The *TextMessageEncodingBindingElement* indicates that the Endpoint uses text encoding instead;

3. And finally the transport binding elements – These elements represent the transmission of an encoding message on a transport protocol. Typical WCF bindings include exactly one Transport Binding Element, which inherits from *TransportBindingElement*.

   Please note that the *Prototype* and *Beyond - Message Chunking* (see section BEYOND THE PRESTO PROTOCOL: MESSAGE CHUNKING SAMPLES) solution samples make exclusive use of the *HttpTransportBindingElement*. The *HttpTransportBindingElement* indicates that the Endpoint will communicate with the world using HTTP as the transport protocol. The *Beyond - Soap Intermediary* (see section BEYOND THE PRESTO PROTOCOL: SOAP INTERMEDIARY SAMPLES) illustrates in addition other transport binding elements such the *TcpTransportBindingElement*;

When creating a specific binding, the order and types of binding elements in Bindings are significant: The collection of binding elements is used to build a communications stack ordered according to the order of binding elements in the binding elements collection. The last binding element to be added to the collection corresponds to the bottom component of the communications stack while the first one corresponds to the top component. Incoming messages flow through the stack from the bottom upwards while outgoing messages flow from the top downwards. Therefore the order of binding elements in the collection directly affects the order in which communications stack components process messages.

Binding elements must always be added in the following order: reliability, security (optional), encoding (text vs. MTOM) and transport (HTTP). For more information, please refer to "CREATING USER-DEFINED BINDINGS AND BINDING ELEMENTS" at the following address: http://msdn2.microsoft.com/en-us/library/ms733893.aspx.



Considering the above, the custom binding is created programmatically through the *CustomBinding* class as follows:

```
public Binding GenerateBinding()
{
    List<BindingElement>elements = new List<BindingElement>();


    // WS-ReliableMessaging
    if (this.useReliableMessaging)
    {
       ReliableSessionBindingElement reliableSessionBindingElement = null;
       reliableSessionBindingElement = new ReliableSessionBindingElement();
       reliableSessionBindingElement.AcknowledgementInterval = new TimeSpan(0, 0, 0, 2);
       reliableSessionBindingElement.MaxTransferWindowSize = 32;
       reliableSessionBindingElement.InactivityTimeout = new TimeSpan(0, 0, 10);
       reliableSessionBindingElement.MaxPendingChannels = 32;
       reliableSessionBindingElement.MaxRetryCount = 8;
       reliableSessionBindingElement.FlowControlEnabled = true;
       reliableSessionBindingElement.Ordered = true;
       elements.Add(reliableSessionBindingElement);
    }

    // WS-Security
    if ((this.useDigitalSignature) || (this.UseEncryption))
    {
       SymmetricSecurityBindingElement messageSecurity =
                SecurityBindingElement.CreateAnonymousForCertificateBindingElement();
       // Create supporting token parameters for the Client X509 certificate
       X509SecurityTokenParameters clientX509SupportingTokenParameters = new X509SecurityTokenParameters();
       // Specify that the supporting token is passed in message send by the Client to the Service
       clientX509SupportingTokenParameters.InclusionMode = SecurityTokenInclusionMode.AlwaysToRecipient;
       // Turn off derived keys
       clientX509SupportingTokenParameters.RequireDerivedKeys = false;
       // Augment the binding element to require the client's X509 certificate as an endorsing token in the message
       messageSecurity.EndpointSupportingTokenParameters.Endorsing.Add(clientX509SupportingTokenParameters);
       elements.Add(messageSecurity);
    }

    // XOP/MTOM or text encoding
    MessageEncodingBindingElement encodingBindingElement = null;
    if (this.EncodingMode == PrestoSoapNodeConfig.PrestoEncodingMode.Text)
    {
        TextMessageEncodingBindingElement textEncodingBindingElement = new TextMessageEncodingBindingElement();
        textEncodingBindingElement.WriteEncoding = System.Text.Encoding.UTF8;
        textEncodingBindingElement.ReaderQuotas.MaxArrayLength = maxMessageSize;
        encodingBindingElement = textEncodingBindingElement;
    }
    else
    {
        MtomMessageEncodingBindingElement mtomEncodingBindingElement = new MtomMessageEncodingBindingElement();
        mtomEncodingBindingElement.WriteEncoding = System.Text.Encoding.UTF8;
        mtomEncodingBindingElement.ReaderQuotas.MaxArrayLength = maxMessageSize;
        encodingBindingElement = mtomEncodingBindingElement;
    }

    // WS-Addressing
    if (this.AddressingVersion == PrestoSoapNodeConfig.PrestoAddressingVersion.WSAddressing10)
    {
        encodingBindingElement.MessageVersion = MessageVersion.Soap12WSAddressing10;
    }
    else
    {
        encodingBindingElement.MessageVersion = MessageVersion.Soap12WSAddressingAugust2004;
    }
    elements.Add(encodingBindingElement);
```

```
    HttpTransportBindingElement httpTransportBindingElement = new HttpTransportBindingElement();
    httpTransportBindingElement.ManualAddressing = false;
    httpTransportBindingElement.MaxReceivedMessageSize = maxMessageSize;
    httpTransportBindingElement.AllowCookies = true;
    httpTransportBindingElement.AuthenticationScheme = System.Net.AuthenticationSchemes.Anonymous;
    httpTransportBindingElement.HostNameComparisonMode = HostNameComparisonMode.StrongWildcard;
    httpTransportBindingElement.ProxyAuthenticationScheme = System.Net.AuthenticationSchemes.Anonymous;
    httpTransportBindingElement.Realm = "";
    httpTransportBindingElement.TransferMode = TransferMode.Buffered;
    httpTransportBindingElement.UnsafeConnectionNtlmAuthentication = false;
    httpTransportBindingElement.UseDefaultWebProxy = true;
    elements.Add(httpTransportBindingElement);

    CustomBinding currentBinding = new CustomBinding(elements.ToArray());
    return currentBinding;
}
```

Conversely, such a binding configuration can also be declaratively defined in the Service *App.config* file:

```
<!--configuration file used by above code -->
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <customBinding>
           <binding name="prestoMtomReliableBinding">
             <reliableSession acknowledgementInterval="00:00:1"
                              maxTransferWindowSize="32"
                              inactivityTimeout="00:10:00"
                              maxPendingChannels="128"
                              maxRetryCount="8"
                              ordered="false" />
             <mtomMessageEncoding messageVersion="Soap12WSAddressingAugust2004" writeEncoding="utf-8">
                <readerQuotas maxArrayLength="5000000" />
             </mtomMessageEncoding>
             <httpTransport manualAddressing="false"
                              maxReceivedMessageSize="5000000"
                              maxBufferPoolSize="524288"
                              maxBufferSize="5000000"
                              allowCookies="false"
                              authenticationScheme="Anonymous"
                              bypassProxyOnLocal="false"
                              hostNameComparisonMode="StrongWildcard"
                              proxyAuthenticationScheme="Anonymous"
                              realm=""
                              transferMode="Buffered"
                              unsafeConnectionNtlmAuthentication="false"
                              useDefaultWebProxy="true" />
           </binding>
           …
       </customBinding>
    </bindings>
    <service name="prestoService">
      <endpoint name="prestoMtomReliableBinding"
                address="submit"
              binding="customBinding"
                bindingConfiguration="prestoMtomReliableBinding "
                contract="dgme.finances.gouv.fr.presto.IPresto"/>
      …
      <endpoint contract="IMetadataExchange"
                binding="mexHttpBinding"
               address="mex" />
     </service>
    </services>
  </system.serviceModel>
</configuration>
```

Setting the Service Identity

A Service must identify itself using an X.509 certificate. This Starter Kit makes use of the X.509 certificates installed when following the instructions in the INSTALLING PRESTO STARTER KIT SAMPLES document. Of course, if you want to, you can substitute your own certificates.

This certificate is accessed from both the Service and Client sides. WCF uses this certificate to optionally encrypt the exchanged PRESTO-compliant messages on the Client side. Note that this means that the Source Application sample must have access to this certificate (not its private key), usually via its LocalMachine *Personal* ("My") store. It can also be downloaded from the metadata endpoint.

The Target Application sample and the *App.config* file are modified as follows so that WCF can locate and use the *PRESTO Target Application* certificate. Let's start with the *App.config*. The thumbprint of the Service certificate is specified in the Application settings as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    …
    <!-- The Service Certificate for the Identity of the IPresto endpoints -->
    <add key="serviceCertificateThumbprint" value="DC07A1A56E0E6F4E0809999149ACC98C8A6E7479" />
    …
  </appSettings>
…
</configuration>
```

Here you can see we are using the *PRESTO Target Application* certificate in the LocalMachine *Personal* ("My") store. You could store the certificate in any of the certificate stores but this is an appropriate place for a Service.

```
 // Set the Service X.509 certificate
 serviceCredentials.ServiceCertificate.SetCertificate(StoreLocation.LocalMachine, StoreName.My,
 X509FindType.FindByThumbprint, config.ServiceCertificateThumbprint);
```

Validating the Client signature if any

WCF needs the untrusted issuer's flag if not using a fully trustable Client certificate. This might be for example, because the certificate has expired, or a certificate revocation list (CRL) is inaccessible.

Currently certificate validation will fail since CRL entries are missing. Please remember that our certificates for development/testing purposes only have been generated with *Makecert.exe* and consequently, do not have a CDP (*CRL Distribution Point*) extension. Placing the certificate in *Trusted People* circumvents this.

The *X509RevocationMode.NoCheck* flag enables not to check the revocation state of the certificate (*revocationMode* fields in WCF *App.config*).

```
 // Setting the CertificateValidationMode to PeerTrust or PeerOrChainTrust means that if the
 // certificate is in the Trusted People store, then it will be trusted without performing a validation
 // of the certificate's issuer chain. Such a setting is used here for convenience so that the
 // sample can be run without having to have certificates issued by a certificate authority (CA).
 // This setting is less secure than the default, ChainTrust. The security implications of this
 // setting should be carefully considered before using PeerTrust or PeerOrChainTrust in production code.
 serviceCredentials.ClientCertificate.Authentication.CertificateValidationMode =
                     X509CertificateValidationMode.ChainTrust;

// The PRESTO Starter Kit X.509 Certificates have been generated with the Certificate Creation tool
// (makecert.exe). These certificates are for development/testing purposes only and do NOT have a CDP
// extension. As a result, the revocation test has to been turned off, otherwise, such certificates are
// considered expired. In production code, the following line MUST be commented.
serviceCredentials.ClientCertificate.Authentication.RevocationMode = X509RevocationMode.NoCheck;
```

Exposing a MEX endpoint for the service

Please note that the MEX endpoint for a Service is not exposed by default. You have to have the following in the Service *App.config*:

```
<endpoint contract="IMetadataExchange" binding="mexHttpBinding" address="mex" />
```

In the Target Application sample, the MEX endpoint is exposed in code instead.

```
// Create a service host
Uri baseAddress = new Uri(config.ServiceEndpointUri);
serviceHost = new ServiceHost(typeof(PrestoService), baseAddress);

// Add a MEX endpoint for the service
ServiceMetadataBehavior metadataBehavior = new ServiceMetadataBehavior();
metadataBehavior.HttpGetEnabled = true;
metadataBehavior.HttpGetUrl = new Uri(config.ServiceEndpointMexUri);
serviceHost.Description.Behaviors.Add(metadataBehavior);
serviceHost.AddServiceEndpoint(typeof(IMetadataExchange), MetadataExchangeBindings.CreateMexHttpBinding(),
    config.ServiceEndpointMexUri);
```

Defining Endpoints and Starting the Service

Endpoints can be defined in code or in *App.config* file.

The *DefineEndpointImperatively* method below shows the easiest way to define endpoints in code and start the Service: one for the IPresto interface for One-Way and another one for the IPresto2 interface for Request-Reply.

A WCF Service exposes a collection of endpoints where each endpoint is a portal for communicating with the world. Each endpoint has an Address, a Binding and a Contract (ABC). The address is *where* the Endpoint resides, the binding is *how* the Endpoint communicates and the contract is *what* the endpoint communicates.

On the Service, a *ServiceDescription* holds the collection of *ServiceEndpoints* each describing an endpoint that the Service exposes. From this description, *ServiceHost* creates a runtime that contains an *EndpointListener* for each *ServiceEndpoint* in the *ServiceDescription*. The endpoint's address, binding and contract (representing the *where*, *what* and *how*), correspond to the *EndpointListener*'s listening address, message filtering and dispatch, and channel stack respectively.

```
public void DefineEndpointImperatively()
{
   // Create a service host
   Uri baseAddress = new Uri(config.ServiceEndpointUri);
   ServiceHost serviceHost = new ServiceHost(typeof(PrestoService), baseAddress);

   // Add a MEX endpoint for the service
   ServiceMetadataBehavior metadataBehavior = new ServiceMetadataBehavior();
   metadataBehavior.HttpGetEnabled = true;
   metadataBehavior.HttpGetUrl = new Uri(config.ServiceEndpointMexUri);
   serviceHost.Description.Behaviors.Add(metadataBehavior);
   serviceHost.AddServiceEndpoint(typeof(IMetadataExchange), MetadataExchangeBindings.CreateMexHttpBinding(),
       config.ServiceEndpointMexUri);

   // Add an endpoint for the IPresto Interface (Reliable One-Way MEP) using the AddEndpoint helper method
   // to create the ServiceEndpoint and add it to the ServiceDescription
   ServiceEndpoint serviceEndpoint;
   if (config.UseSoapIntermediary)
   {
      serviceEndpoint = serviceHost.AddServiceEndpoint(typeof(IPresto),      // Contract type
                                usedBinding,                                 // Custom bindings
                                EndpointUri.oneWayRelativeAddress,           // Relative address
                                new Uri(config.SoapIntermediaryEndpointUri)); // SOAP intermediary endpoint's address
   }
   else
   {
      serviceEndpoint = serviceHost.AddServiceEndpoint(typeof(IPresto), usedBinding,
                                       EndpointUri.oneWayRelativeAddress);
   }

   // Add an endpoint for the IPresto2 Interface (Reliable Request-Reply MEP)
   ServiceEndpoint serviceEndpoint2;
   if (config.UseSoapIntermediary)
   {
      serviceEndpoint2 = serviceHost.AddServiceEndpoint(typeof(IPresto2), usedBinding,
                                      EndpointUri.requestReplyRelativeAddress,
                                       new Uri(config.SoapIntermediaryEndpointUri));
   }
   else
   {
      serviceEndpoint2 = serviceHost.AddServiceEndpoint(typeof(IPresto2), usedBinding,
```

```
                                        EndpointUri.requestReplyRelativeAddress);
    }

    serviceHost.UnknownMessageReceived += new
        EventHandler<UnknownMessageReceivedEventArgs>(serviceHost_UnknownMessageReceived);

    // Create and open the service runtime
    serviceHost.Open();
}
```

PRESTO-enabled Source Application (WCF)

What this sample does

This sample shows how to quickly build in a source application a Client in WCF that can send PRESTO-compliant messages to any PRESTO Service endpoint.

This sample includes support for the various PRESTO protocol settings as defined in its initial specification.

Key Concepts Illustrated

This sample includes the logic to dynamically set in code a WCF custom binding. The binding can also be set in the Client *App.config* file where several predefined bindings are proposed by default. In both cases, the settings must match the one of its PRESTO counterpart: the target application or the PRESTO sender proxy. The UI enables to select how the binging is set.

This sample provides a way to dynamically retrieve the settings expected by its PRESTO counterpart by interrogating through WS-MetadataExchange [WS-MetadataExchange] a metadata exchange endpoint exposed (if any) by it.

Please note that this is not part of the current PRESTO specification. This implementation is provided to illustrate foreseeable future capabilities.

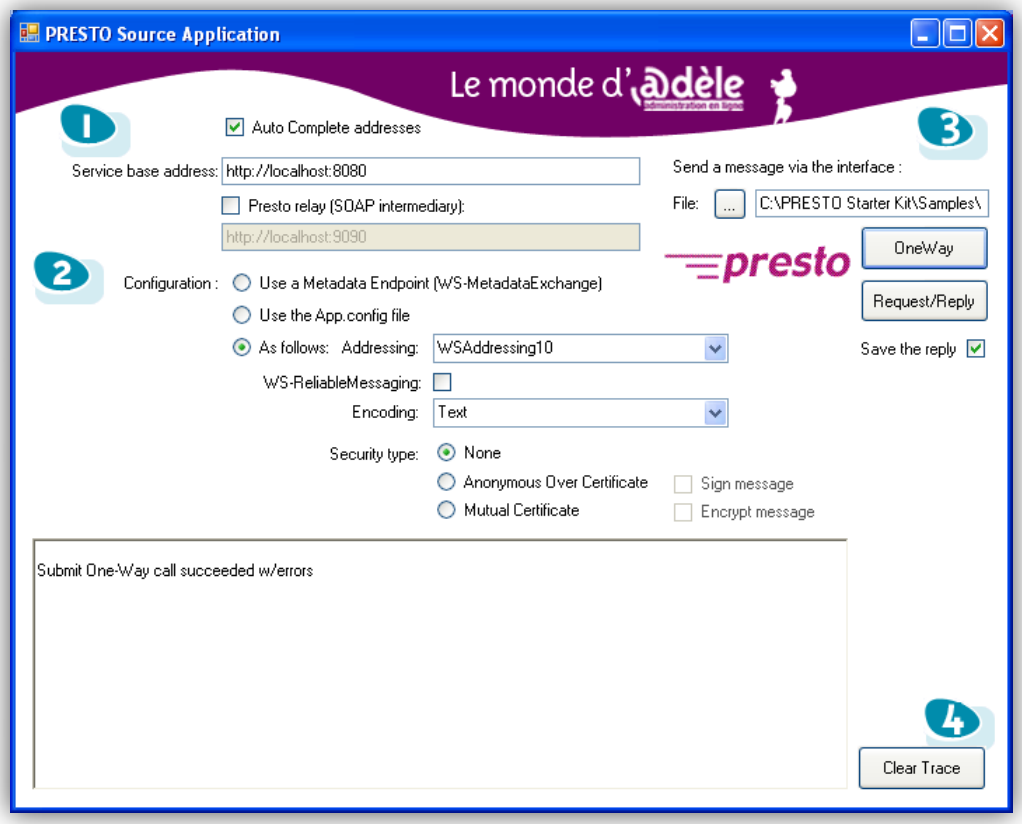
This sample includes the necessary calls to select a file on the local disk, build a PRESTO-compliant message that will vehicle the selected file and finally send the message with a One-Way or Request-Reply (with anonymous sender) message patterns.

How to run

Perform the following steps to run the sample:

1. Open Windows Explorer and navigate to the *Samples\Prototype\bin\[Debug|Release]* subdirectory under the directory location where you installed the PRESTO Starter Kit.
2. Double-click the *TargetApplication.exe* Windows application (see above).

3. Double-click the *SourceApplication.exe* Windows application.



Important

When 'Auto Complete addresses' is checked (default), the */submitOneWay* suffix (vs. */submit* suffix) is silently added to the specified service base address and, if selected, to the PRESTO relay address for an OneWay operation (vs. a Request/Reply operation) is checked.

This checkbox MUST be unchecked whenever you don't want to use the *TargetApplication.exe* on the other side. This enables you to fully control the several endpoint addresses.

How to run from a different machine

Perform the following steps to run the sample from a different machine:

1. Copy the sample program files from *Samples\Prototype\bin\[Debug|Release]*, from under the directory location where you've installed the PRESTO Starter Kit, to the Client machine.

   The program files are *SourceApplication.exe*, *SourceApplication.exe.config*, *ClientChannel.dll* and *PrestoContract.dll*;

   This sample makes use of the X.509 certificates installed when following the instructions in the INSTALLING PRESTO STARTER KIT SAMPLES document. You need to follow the same instructions for the client machine. Of course, if you want to, you can substitute your own certificates.

2. Open the *SourceApplication.exe.config* configuration file and change the address value of the endpoint definition to match the new address of the Service. Replace any references to "localhost" with a fully-qualified domain name in the address.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- The Base address of the IPresto endpoints -->
    <add key="baseAddress" value="http://localhost:8080" />
    <!-- The Base address of the MEX endpoint -->
    <add key="baseMexAddress" value="http://localhost:9999" />
    <!-- The Base address of the SOAP intermediary endpoint -->
    <add key="baseListenToAddress" value="http://localhost:9090" />
    …
  </appSettings>
  …
</configuration>
```

Sending Messages to an Endpoint

The code below shows two ways to send a message to the IPresto endpoint. *SendMessageToEndpoint* hides the Channel creation which happens behind the scenes while *SendMessageToEndpointUsingChannel* example does it explicitly.

The first example in *SendMessageToEndpoint* uses the Service Model Metadata Utility tool (*Svcutil.exe*) and the Service's metadata to generate a contract (IPresto for One-Way or IPresto2 for Request-Reply), a proxy class (PrestoProxy in this example) that implements the contract, and associated *App.config* (not shown here). Again, the contract defined by IPresto specifies the *what* (i.e. the operations that can be performed) while the generated config contains a binding (the *how*) and an address (the *where*).

Using this proxy class is simply a matter of instantiating it and calling the Submit method. Behind the scenes, the proxy class will create a channel and use that it to communicate with the endpoint.

The second example in *SendMessageToEndpointsUsingChannel* below shows communicating with an endpoint using *ChannelFactory* directly.

Instead of using a proxy class and an *App.config* file, a channel is created directly using *ChannelFactory<IPresto>.CreateChannel*. A *ChannelDescription* holds the one *ServiceEndpoint* with which the Client communicates. From this *ChannelDescription*, *ChannelFactory* creates the channel stack that can communicate with the Service's endpoint. Also, instead of using the *App.config* file to define the endpoint's address and binding, the *ChannelFactory<IPresto>* constructor takes those two pieces of information as parameters. The third piece of information required to define an endpoint, namely the contract, is passed in as the type T.

```
using System.ServiceModel;

// This contract is generated by svcutil.exe from the service's metadata
[ServiceContractAttribute(Namespace = "http://dgme.finances.gouv.fr/presto")]
public interface IPresto
{
    [OperationContract(Action = "http://dgme.finances.gouv.fr/presto/submitOneWay", IsOneWay = true)]
    [XmlSerializerFormatAttribute()]
    void submitOneWay(submit1WayMessage message);
}

// This class is generated by svcutil.exe from the service's metadata generated config is not shown here
public class PrestoProxy : IPresto
{
    ...
}
```

```
public class WCFClientApp
{
    public void SendMessageToEndpoint()
    {
        submit1WayMessage sr = new submit1WayMessage();

        // This uses a proxy class that was created by svcutil.exe from the service's metadata
        PrestoProxy proxy = new PrestoProxy();
        proxy.submitOneWay(sr);
    }

    public void SendMessageToEndpointUsingChannel()
    {

        submit1WayMessage sr = new submit1WayMessage();

        EndpointAddress endPointAddress = new EndpointAddress(new Uri(config.ServiceEndpointUri + "/" +
                                                EndpointUri.requestReplyRelativeAddress));

        // This uses ChannelFactory to create the channel. The address, the binding and  the contract type (IPresto)
        // have to be specified.
        ChannelFactory<IPresto> factory = new ChannelFactory<IPresto>(config.GenerateBinding(), endPointAddress);

        IPresto channel = factory.CreateChannel();
        channel.submitOneWay(sr);
        factory.Close();
    }
}
```

Signing the PRESTO message

A client may use an X.509 certificate to sign PRESTO-compliant messages. Likewise, a client may use in addition the Service X.509 certificate used to expose its identify to encrypt PRESTO-compliant messages.

This Starter Kit makes use of the X.509 certificates installed when following the instructions in the INSTALLING PRESTO STARTER KIT SAMPLES document. Of course, if you want to you can substitute your own certificates.

These certificates are accessed from both the Client side and the Service side. WCF uses them to sign and optionally encrypt the exchanged PRESTO-compliant messages. Note that this means that the Source Application sample must have access to its certificate, usually via its certificate store.

The Source Application sample and the *App.config* file are modified as follows so that WCF can locate and use the *PRESTO Source Application* certificate. Let's start with the *App.config*. The thumbprint of the Client and the Service certificates are specified in the Application settings.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    …
    <!-- The Service Identity of the IPresto endpoints -->
    <add key="identity" value="PRESTO Target Application" />
    <!-- The Service Certificate for the Identity of the IPresto endpoints -->
    <add key="serviceCertificateThumbprint" value="DC07A1A56E0E6F4E0809999149ACC98C8A6E7479" />
    <!-- The Client Certificate -->
    <add key="clientCertificateThumbprint" value="F31416E1D645367AC2DB89E31B8EF89A4E77CF21" />
  </appSettings>
…
</configuration>
```

Here you can see we are using the *PRESTO Source Application* certificate in the LocalMachine *Personal* ("My") store. You could store the certificate in any of the certificate stores but this is an appropriate place for a client.

```
// Set the X.509 Client Signing Certificate as Credential for the factory
factory.Credentials.ClientCertificate.SetCertificate(StoreLocation.LocalMachine, StoreName.My,
        X509FindType.FindByThumbprint, config.ClientCertificateThumbprint);

// Set the X.509 Service Certificate as Credential for the factory
factory.Credentials.ServiceCertificate.SetDefaultCertificate(StoreLocation.LocalMachine, StoreName.My,
        X509FindType.FindByThumbprint, config.ServiceCertificateThumbprint);
```

Please note that if you'll select the use of a Metadata Endpoint, the Service certificate is dynamically retrieved by the Client. It's part of the channel being dynamically built

Whatever the way the Service certificate is retrieved, i.e. downloaded from the metadata endpoint vs. read from the LocalMachine *Personal* ("My") store) , WCF needs the untrusted issuers flag if not using a fully trustable Service. This might be for example, because the certificate has expired, or a certificate revocation list (CRL) is inaccessible.

Currently certificate validation will fail since CRL entries are missing. Please remember that our certificates for development/testing purposes only have been generated with *Makecert.exe* and consequently, do not have a CDP (*CRL Distribution Point*) extension. Placing the certificate in *Trusted People* circumvents this.

The *X509RevocationMode.NoCheck* flag enables not to check the revocation state of the certificate (*revocationMode* fields in WCF *App.config*).

```
// Setting the CertificateValidationMode to PeerTrust or PeerOrChainTrust means that if the
// certificate is in the Trusted People store, then it will be trusted without performing a validation
// of the certificate's issuer chain. Such a setting is used here for convenience so that the
// sample can be run without having to have certificates issued by a certificate authority (CA).
// This setting is less secure than the default, ChainTrust. The security implications of this
// setting should be carefully considered before using PeerTrust or PeerOrChainTrust in production code.
factory.Credentials.ServiceCertificate.Authentication.CertificateValidationMode =
X509CertificateValidationMode.ChainTrust;

// The PRESTO Starter Kit X.509 Certificates have been generated with the Certificate Creation tool
// (makecert.exe). These certificates are for development/testing purposes only and do NOT have a CDP
// extension. As a result, the revocation test has to been turned off, otherwise, such certificates are
// considered expired. In production code, the following line MUST be commented.
factory.Credentials.ServiceCertificate.Authentication.RevocationMode = X509RevocationMode.NoCheck;
```

For convenience, in this starter kit we are accessing the certificate in the LocalMachine "My" store. In a real client deployment you are likely to put the certificate into the CurrentUser's "Trusted People" store. The Client app should be running using the logged-in user account so the Current User store is a naturally accessible place for the certificate to be installed.

WCF needs access permission to the Client certificate's private key since it will use it for signing and decryption.

Using a Metadata Resolver

Everything you can put in an *App.config* file you can choose to implement in code instead. In fact, with code there are a few things that you *cannot* do in config. One of the things you can do with code on the client is to avoid using pre-defined client configuration entirely or building in code a custom binding that reflects the user settings in the UI, and instead generate a client proxy dynamically using the *MetadataResolver* class. It is this class that the Service Model Metadata Utility Tool (*Svcutil.exe*) uses to generate client proxies.

Whereas *App.config* files are useful for being able to make changes to a WCF client or service configuration without having to recompile, this code enables us to do away with client configuration entirely. The only thing we need to know *a priori* is the MEX endpoint reference for the service exposed by a PRESTO receiver proxy or a target application we want to consume and its type. In this case, the MEX endpoint reference is maintained in the *ServiceEndpointMexUri* of the static config class. The type can be either *IPresto* for One-Way or *IPresto2* for Request-Reply.

```
ChannelFactory<IPresto> factory = null;

Uri mexUri = new Uri(config.ServiceEndpointMexUri);
ContractDescription contract = ContractDescription.GetContract(typeof(IPresto));

EndpointAddress mexEndpointAddress = new EndpointAddress(mexUri);
ServiceEndpointCollection endpoints = MetadataResolver.Resolve(contract.ContractType, mexEndpointAddress);

foreach (ServiceEndpoint endpoint in endpoints)
{
   if (endpoint.Contract.Namespace.Equals(contract.Namespace) && endpoint.Contract.Name.Equals(contract.Name))
   {
      factory = new ChannelFactory<IPresto>(endpoint.Binding, endpoint.Address);

      // Get the security requirements for sending message and configure the factory accordingly
      ISecurityCapabilities isc = endpoint.Binding.GetProperty<ISecurityCapabilities>
                                        (new BindingParameterCollection());
      if (isc.SupportedRequestProtectionLevel != ProtectionLevel.None)
      {
         // Set the X.509 Client Certificate as Credential for the factory
         factory.Credentials.ClientCertificate.SetCertificate(StoreLocation.LocalMachine, StoreName.My,
                                          X509FindType.FindByThumbprint, config.ClientCertificateThumbprint);

         // The PRESTO Starter Kit X.509 Certificates have been generated with the Certificate Creation tool
         // (makecert.exe). These certificates are for development/testing purposes only and do NOT have a CDP
         // extension. As a result, the revocation test has to been turned off, otherwise, such certificates are
         // considered expired. In production code, the following line MUST be commented.
         factory.Credentials.ServiceCertificate.Authentication.RevocationMode = X509RevocationMode.NoCheck;
      }

      break;
   }
}
```

PRESTO add-in for Office system 2007 (WCF/VSTO)

What this sample does

This sample shows how to send a Word document to a PRESTO recipient from Word 2007.

Key Concepts Illustrated

This sample shows how easy it is to build a strongly-typed, robust Visual Studio Tools for Office add-in in Visual Studio for PRESTO based on the PRESTO Source Application sample.

The project wizard generates all the "plumbing" code that is required, leaving you to focus on the business-specific custom code that you want to add; in this case, The PRESTO protocol support for sending documents.

This sample illustrates how to create a ribbon and a custom task pane for Word 2007 with Visual Studio Tools for Office. The custom task pane acts as the PRESTO Source Application sample and shares, for that purpose most of its code with the PRESTO Source Application sample. Consequently, the code description for the PRESTO Source Application sample also applies here.

How to granting Full Trust to the add-in assemblies

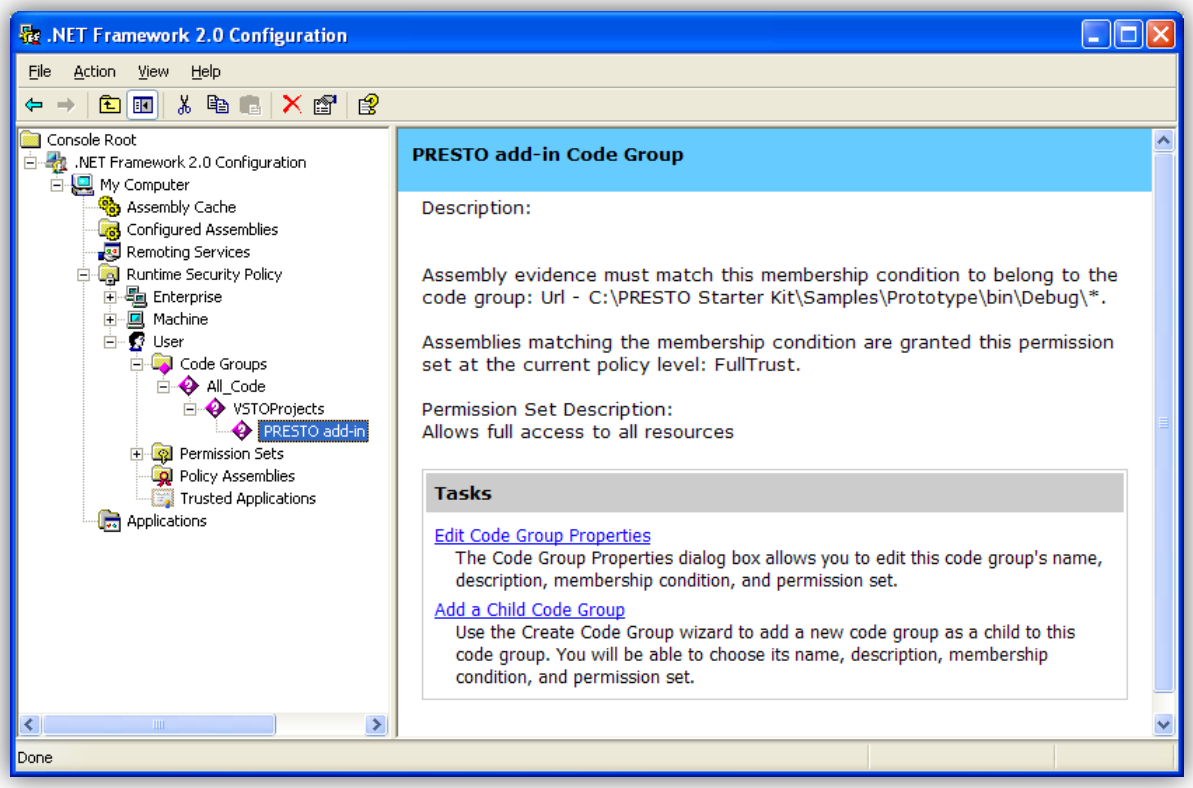
Before running the add-in sample, you must grant trust to the add-in assemblies so that the .NET Framework allows them to execute. You must grant the customization assembly *FullTrust* permissions. In addition, you must trust any referenced or satellite assemblies with the appropriate level of permissions.

Perform the following steps:

1. In Control Panel, open Administrative Tools.
2. Run Microsoft .NET Framework 2.0 Configuration.
3. In the tree view on the left side, expand .NET Framework 2.0 Configuration, expand My Computer, expand Runtime Security Policy, expand User, expand Code Groups, expand All_Code, and then expand VSTOProjects.

   Please note that if you have NOT compiled the add-in project before, you will not have the VSTOProjects folder. You can add the new code group to the All_Code root node, or you can compile the add-in project to have the VSTOProjects folder created automatically.
4. On the right is the VSTOProjects Code Group description, which has a Tasks section at the bottom of the page. In the Tasks section, click Add a Child Code Group. The Create Code Group wizard starts.

5. Select Create a new code group, and enter a name (Word add-in for PRESTO) and description that will help you identify the project. Click Next.
6. In the Choose the condition type for this code group list, click URL.
7. In the URL box, type the full path to the path to the *Samples\Prototype\bin\[Debug|Release]* subdirectory under the directory location where you've installed the PRESTO Starter Kit followed by an asterisk; for example, *C:\<starter kit path>\Samples\Prototype\bin\Debug\ **.
8. Click Next.
9. Select Use existing permission set, and then select *FullTrust* from the list.
10. Click Next.
11. Click Finish.



How to run from Visual Studio 2005

Perform the following steps:

1. Select the Debug mode (default) and build the related solution (using the Build menu or CTRL+SHIFT+B).
2. Right-click the sample project item and choose Debug, Start new instance. Word 2007 starts.

You can now set breakpoints in the sample code and enable breakpoints on exceptions.

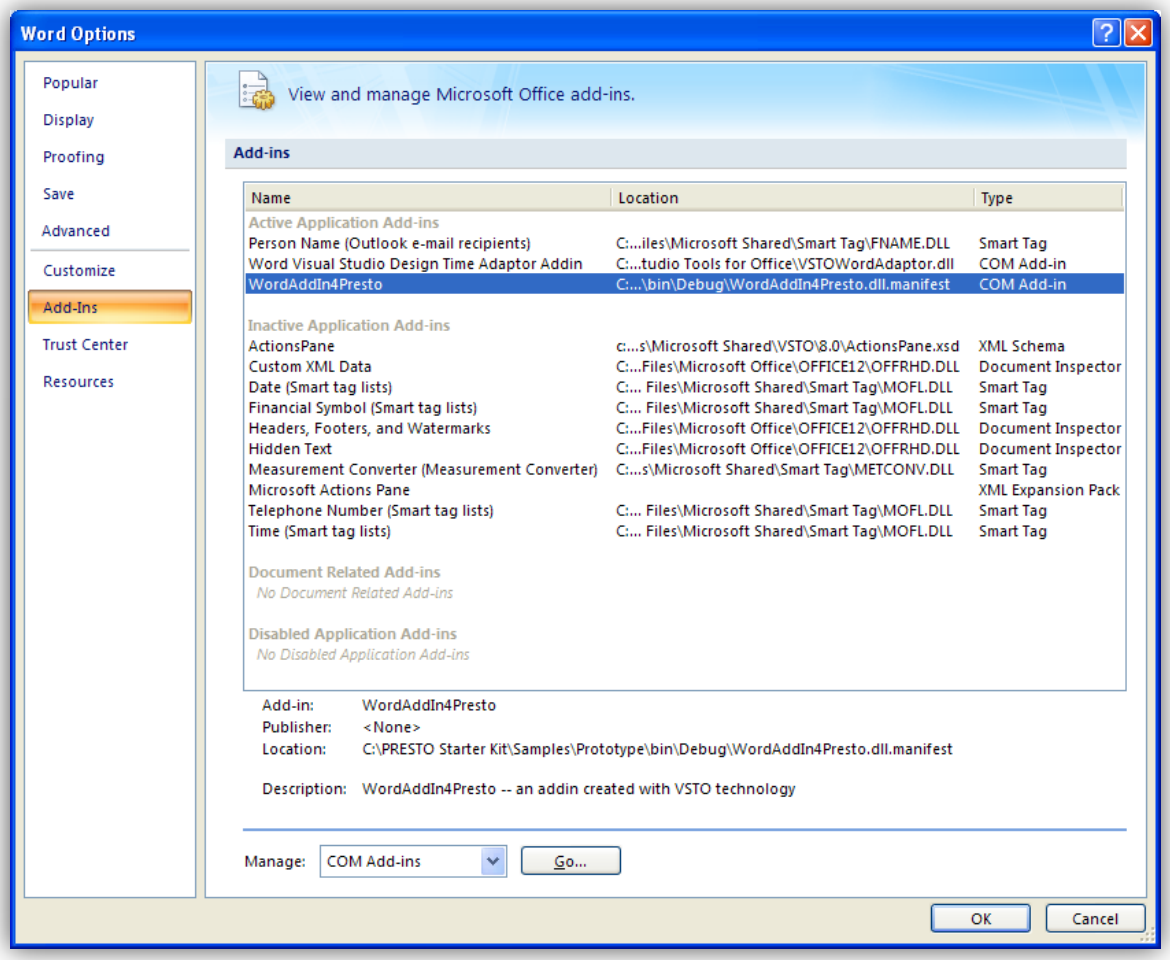
How to check the Word Add-in installation

Perform the following steps:

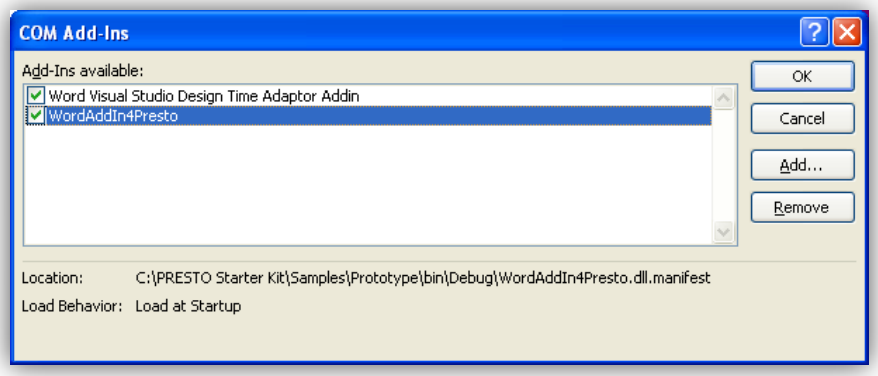
1. In Word 2007, click the Microsoft Office Button.

2. Click the Word Options button.

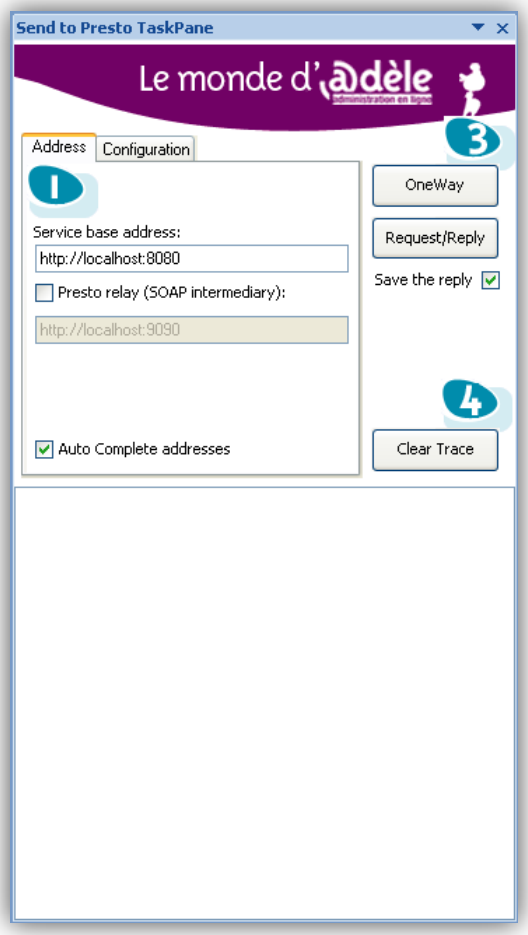
3. In the Word Options dialog box, click Add-Ins in the navigator bar on the left. A list of installed add-ins appears.



4. At the bottom of this dialog box, in the Manage list, click COM Add-ins, and then click Go. The traditional COM Add-ins dialog box appears.



5. After you confirm that your add-in is indeed being loaded, close the dialog boxes by clicking OK twice.



Adding a Custom Task Pane for the PRESTO UI

The new custom task pane model in Office 2007 opens up a wide range of opportunities for providing a better user experience than the doc-level *ISmartDocument*-based task pane.

Indeed, Office provides the basic framework for hooking up the task pane, and it leaves the developer to implement the task pane in whatever way the developer sees fit. This add-in sample displays in the task pane the same UI as the one in the PRESTO Source Application sample.

The PRESTO add-in's main class, i.e. the *ThisAddIn* host item base class, provides the standard Visual Studio Tools for Office *Startup* and *Shutdown* methods. You can put any initialization code you want into the *Startup* method, and any termination clean-up code you want into the *Shutdown* method.

Note that an add-in can implement one or more custom task panes by implementing the *ICustomTaskPaneConsumer* interface. However, Visual Studio Tools for Office provides a default implementation of this interface to streamline development.

The PRESTO task pane is added with on line of code, specifying the user control for PRESTO UI interface and the caption to use.

```
internal CustomTaskPane ctp;
```

```
private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
   // Create the Presto User Control and the related TaskPane
   ctp = this.CustomTaskPanes.Add(new WordAddIn4Presto.UCPresto(), "Send to Presto TaskPane");
   ctp.Visible = false;
}
```

Adding Ribbon Customization

Note that an add-in can implement Ribbon customization by implementing the *IRibbonExtensibility* interface. However, Visual Studio Tools for Office provides a default implementation of this interface to streamline development.

When adding a custom ribbon for controlling the PRESTO UI, Visual Studio Tools for Office generates a new class for the ribbon, called in this case *PrestoRibbon*, and an eponym XML file, which contains markup for the ribbon customization:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui" onLoad="OnLoad">
  <ribbon>
    <tabs>
      <tab idMso="TabHome">
        <group id="Presto"
               label="DGME">
          <toggleButton id="togglePrestoButton"
                        size="large"
                        label="Send to Presto"
                        screentip="Send a copy of the document in a Presto message"
                        onAction="OntogglePrestoButton"
                        getPressed="GetPressed"
                        getImage="GetImage" />
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

This file is actually not part of the *PrestoRibbon* class. Rather, it is an override of the *RequestService* virtual method in the *ThisAddIn* host item base class. This method hooks up the *ServiceRequest* event on the Visual Studio Tools for Office Application object for this application. This is a standard mechanism that is also used by custom task panes, custom form regions, and potentially other new Office programmability interfaces.

Here's how this works: Office loads the add-in, and queries the add-in to see if it implements any of the programmability interfaces. In this add-in we do implement the *IRibbonExtensibility* interface, so we give Office back the PrestoRibbon object that implements *IRibbonExtensibility*, so that Office can later call back on this object when it needs to.

```
// This is an override of the RequestService method in the ThisAddIn class.
// To hook up your custom ribbon uncomment this code.
public partial class ThisAddIn
{
   private PrestoRibbon ribbon;

   protected override object RequestService(Guid serviceGuid)
   {
      if (serviceGuid == typeof(Office.IRibbonExtensibility).GUID)
      {
         if (ribbon == null)
             ribbon = new PrestoRibbon();
         return ribbon;
      }

         return base.RequestService(serviceGuid);
   }
}
```

The *PrestoRibbon* class implements the *IRibbonExtensibility* interface defined by Office specifically for customizing the Ribbon. This is implemented in a class that is separate from the main *ThisAddIn* class for two reasons. First, it is good practice to factor out discrete functionality into separate classes. Second, the class that implements *IRibbonExtensibility* must be made visible to COM, and the *ThisAddIn* class cannot be made COM-visible because it is derived from classes where this would make no sense, and which are explicitly *not* COM-visible.

```
[ComVisible(true)]
public class PrestoRibbon : Office.IRibbonExtensibility
{
```

The *PrestoRibbon* class declares a private field for the underlying *IRibbonUI* interface.

```
private Office.IRibbonUI ribbon;
```

There is a default public constructor (required for COM instantiation), but this does nothing.

```
public PrestoRibbon()
{
}
```

The *GetCustomUI* method is the only method defined on the *IRibbonExtensibility* interface. Its purpose is to return the XML string for the Ribbon customization markup back to Office when Office calls it. By default, in a Visual Studio Tools for Office implementation, this XML string is an embedded resource.

```
public string GetCustomUI(string ribbonID)
{
   return Properties.Resources.Ribbon1;
}
```

The *OnLoad*, *OntogglePrestoButto* n, *GetPressed*, and *Get Image* methods are not defined on the *IRibbonExtensibility* interface. However, *IRibbonExtensibility* is a dispatch interface, and dispatch interfaces can have any number of additional methods that are discoverable at run time (through late binding). Therefore, *OnLoad*, *OntogglePrestoButton*, *GetPressed*, and *Get Image* are not predefined—the PRESTO ribbon customization might not need such methods at all. Nonetheless, whatever additional methods we provide must satisfy two conditions:

1. They must match the reference in the XML string;
2. Each one must match a specific signature, depending on how you declare that it will be used;

If we refer back to the ribbon XML above, the starter code generated by the Add Ribbon Item wizard declares the *OnLoad* to be called when the entire custom Ribbon markup is loaded. For Office to call this method successfully, it must be a public method with a void return value, and it must take an *IRibbonUI* parameter.

```
public void OnLoad(Office.IRibbonUI ribbonUI)
{
   this.ribbon = ribbonUI;
}
```

```
…
```

Similarly, the markup declares that *OnTogglebutton* should be called when the user clicks the toggle button. The callback for a toggle button must be a public method with a void return, and must take two parameters: an *IRibbonControl* reference and a Boolean reference.

```
public void OntogglePrestoButton(Office.IRibbonControl control, bool isPressed)
{
    …
}
```

The method is implemented to control the task pane visibility in conjunction with the *GetPressed* method (see below).

At the bottom of the class is the definition of a helper method, called *GetResourceText*. All this method does is parsing the embedded resources in the current assembly to extract a text resource specified by name. In this case, this is used to extract the custom markup string.

Creating a basic custom Ribbon with Visual Studio Tools for Office is clearly trivial, as it provides all the code you need to get started.

One of the interesting aspects of building managed solutions based on Office is that we sometimes have to deal with COM types directly. The ribbon is a case in point with the *GetImage* method declared in ribbon XML above.

The traditional Office *CommandBarButton* class uses *IPictureDisp* objects for its images. The Ribbon also uses *IPictureDisp* for the same purpose. The problem is how to get a COM *IPictureDisp* object from a managed Image or Bitmap object.

There are a couple of alternatives here. One approach is to write a custom class to implement *IPictureDisp* directly, but the simplest approach consists in using the *AxHost* class defined in *System.Windows.Forms*. This class is used by the *AxImp* tool to wrap ActiveX controls and expose them as Windows Forms controls. Crucially, for our purposes, it offers a method called *GetIPictureDispFromPicture*. This method takes in an Image and converts it to an *IPictureDisp*. Use *AxHost* to convert the icon resource into an *IPictureDisp* so that you can assign it to the button.

IPictureDisp is an interface defined in the stdole type library and interop assembly. You'll find the stdole interop assembly is already added as a reference in the project.

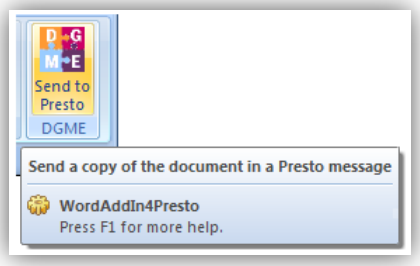
The next issue is that the *GetIPictureDispFromPicture* method is a protected static method in the *AxHost* class. Being static is not a problem, but being protected means that you can only access it from a class derived from *AxHost*. So, we need to write a class that derives from *AxHost*—we can then write a custom method that internally calls *GetIPictureDispFromPicture*.

```
using System.Windows.Forms;
…
private static string GetResourceText(string resourceName)
{
internal class PictureDispMaker : AxHost
{
   private PictureDispMaker() : base("") { }
```

```
    static public stdole.IPictureDisp ConvertImage(Image image)
    {
        return (stdole.IPictureDisp)GetIPictureDispFromPicture(image);
    }

    static public stdole.IPictureDisp ConvertIcon(Icon icon)
    {
        return ConvertImage(icon.ToBitmap());
    }
}
```

We then have to write the callback method that Office will use to get the converted image. This method must take an *IRibbonControl* parameter and return an *IPictureDisp*.

```
public stdole.IPictureDisp GetImage(Office.IRibbonControl control)
{
    stdole.IPictureDisp image = null;

    switch (control.Id)
    {
        case "togglePrestoButton":
            image = PictureDispMaker.ConvertImage(Properties.Resources.dgme);
            break;
    }

    return image;
}
```



Synchronizing the Ribbon and the custom Task Pane

The PRESTO custom task pane should not be visible on startup, and the user should always be able to make it visible or hidden at will. The simplest approach is to provide a ribbon button to allow the user to toggle the task pane visibility.

By implication, it should never be assumed in the add-in code that the task pane is visible at any given time. The *VisibleChangedEvent* needs to be sinking on the task pane in order know its state – the user might open/close the task pane via the custom ribbon button conveniently supplied, but they might also simply hit the X box to close it directly. By further implication, we are responsible for synchronizing the toggled state of the ribbon button to the actual state of the task pane.

```
[ComVisible(true)]
public class PrestoRibbon : Office.IRibbonExtensibility
{
    private Office.IRibbonUI ribbon;

    public PrestoRibbon()
    {
    }

    #region IRibbonExtensibility Members

    public string GetCustomUI(string ribbonID)
    {
        return Properties.Resources.PrestoRibbon;
    }

    #endregion
```

```
    #region Ribbon Callbacks

    public void OnLoad(Office.IRibbonUI ribbonUI)
    {
        this.ribbon = ribbonUI;
    }

    private bool isTaskPaneVisible;
    public bool IsTaskPaneVisible
    {
        get { return isTaskPaneVisible; }
        set
        {
           isTaskPaneVisible = value;
           ribbon.InvalidateControl("togglePrestoButton");
        }
    }

    public bool GetPressed(Office.IRibbonControl control)
    {
        switch (control.Id)
        {
           case "togglePrestoButton":
              return isTaskPaneVisible;
           default:
              return false;
        }
    }


    public void OntogglePrestoButton(Office.IRibbonControl control, bool isPressed)
    {
       Globals.ThisAddIn.ctp.Visible = isPressed;
    }

    public stdole.IPictureDisp GetImage(Office.IRibbonControl control)
    {
       stdole.IPictureDisp image = null;

       switch (control.Id)
       {
          case "togglePrestoButton":
              image = PictureDispMaker.ConvertImage(Properties.Resources.dgme);
              break;
       }

       return image;
    }

    #endregion

    …
}
```

The boolean *isTaskPaneVisible* field in the *PrestoRibbon* class is exposed via a property which invalidates the ribbon togglePrestoButton inside the setter. Invalidating the control will make Office call back to the *GetPressed* method. In the GetPressed method, we return the current value of the flag we're using to cache the visible state of the PRESTO task pane. When the user clicks the *togglePrestoButton*, we set the visible state of the task pane.

Over in the *AddIn* main class, we make sure the *CustomTaskPane* field is accessible to the *PrestoRibbon* class. We create the task pane in the Startup method, and at the same time we hook up an event handler for the *VisibleChanged* event. When we get this event, we toggle the state of the boolean flag in the *PrestoRibbon* class:

```
internal CustomTaskPane ctp;

private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
    // Create the Presto User Control and the related TaskPane
    ctp = this.CustomTaskPanes.Add(new WordAddIn4Presto.UCPresto(), "Send to Presto TaskPane");
    ctp.Width = 364;
    ctp.Visible = false;
    ctp.VisibleChanged += new EventHandler(ctp_VisibleChanged);
}

void ctp_VisibleChanged(object sender, EventArgs e)
{
    this.ribbon.IsTaskPaneVisible = !this.ribbon.IsTaskPaneVisible;
}
```

Beyond the PRESTO protocol: Message Chunking Support (WCF)

What this solution sample does

This sample shows how to implement a "chunking channel". Such a channel allows fragmenting a message (with its payload) into a number of smaller messages. These chunks will get reassembled on the Service receiving side.

Key Concepts Illustrated

By fragmenting/reassembling at the SOAP layer, each SOAP "chunk message" can be in conjunction reliably sent using WS-ReliableMessaging [WS-ReliableMessaging] and even secured using WS-Security. This approach works over any arbitrary transport.

As a quick reminder, what WS-ReliableMessaging does among other things is grouping a set of messages. A message group is identified by an unique sequenceID and each message is given a number as well (1, 2, 3, etc.). At the Service side, these sequenceIDs and message numbers can be used to invoke the messages in order. So actually WS-ReliableMessaging does not chunk messages, it only numbers them. So, the "chunking channel" is fully complementary.

How to run

Perform the following steps to run the sample:

1. Open Windows Explorer and navigate to the *Samples\Beyond - Message Chunking\bin\[Debug|Release]* subdirectory under the directory location where you've installed the PRESTO Starter Kit.
2. Double-click the ReceiverProxy.exe console application.
3. Double-click the *SenderProxy.exe* Windows application. A new *Outgoing* folder is created underneath the *Samples\Beyond - Message Chunking\bin\[Debug|Release]* subdirectory to drop payloads to be sent to the Receiver proxy.

A new *Incoming* folder is created underneath the *Samples\Beyond - Message Chunking\bin\[Debug|Release]* subdirectory to store the payloads when received.

How to run the Client from a different machine

Perform the following steps to run the client from a different machine:

1. Copy the Client program files from *Samples\Beyond - Message Chunking\bin\[Debug|Release]*, from under the directory location where you've installed the PRESTO Starter Kit, to the Client machine. The program files are *SenderProxy.exe*, *SenderProxy.exe.config*.

2. Open the *SenderProxy.exe.config* configuration file and change the address value of the endpoint definition to match the new address of the Service. Replace any references to "localhost" with a fully-qualified domain name in the address.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    …
    <client>
      <endpoint name="prestoChunking"
                address="http://localhost:8080"
                binding="customBinding"
                bindingConfiguration="presto"
                contract="Microsoft.Presto.StarterKit.NonNormative.IChunking"/>
    </client>
  </system.serviceModel>
</configuration>
```

Beyond the PRESTO protocol: SOAP intermediary (WCF)

What this solution sample does

This sample shows how quickly build a "PRESTO-enabled" SOAP intermediary that may contain the basic logic to route a PRESTO-compliant message to its final destination.

Key Concepts Illustrated

This sample illustrates basic PRESTO-compliant messages routing through a SOAP intermediary.

This is a simplified router that sits between one sender proxy sample and one PRESTO Receiver proxy sample.

This sample accepts and returns generic *Message* objects and will not modify them to and from the final endpoint. In this sample, the physical address of the final endpoint is given by the *FinalDestination* parameter of the *SoapRelayServiceBehavior* attribute, although this could easily be modified to read the *App.config* file or some other source as well.

The PRESTO Sender proxy sample will send its message to the PRESTO Receiver proxy "*urn:PrestoProxyEndpoint*" via the physical router address "http://localhost:9090/soaprelay".

The PRESTO Relay accepts messages destined for any endpoint (see the custom *MatchAllEndpoints IContractBehavior*)and the *ProcessMessage* method will operate on calls with any action (see *OperationContract* Action and *ReplyAction ="*"* attributes).

Please note that this is not part of the current PRESTO specification. Indeed, as previously mentioned, the Message Routing pattern is an extensibility point that will be addressed in a future version of the PRESTO protocol.

This sample is provided to illustrate foreseeable future capabilities.

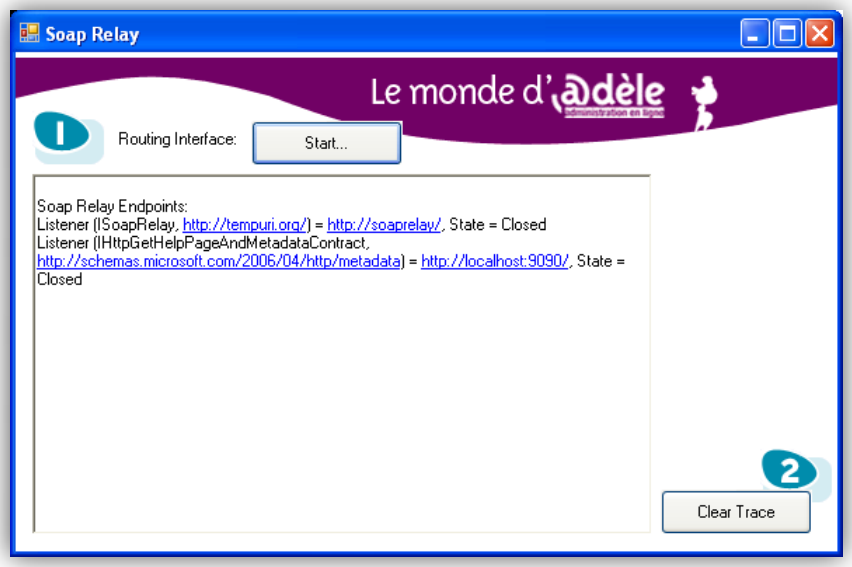
How to run

Perform the following steps to run the sample:

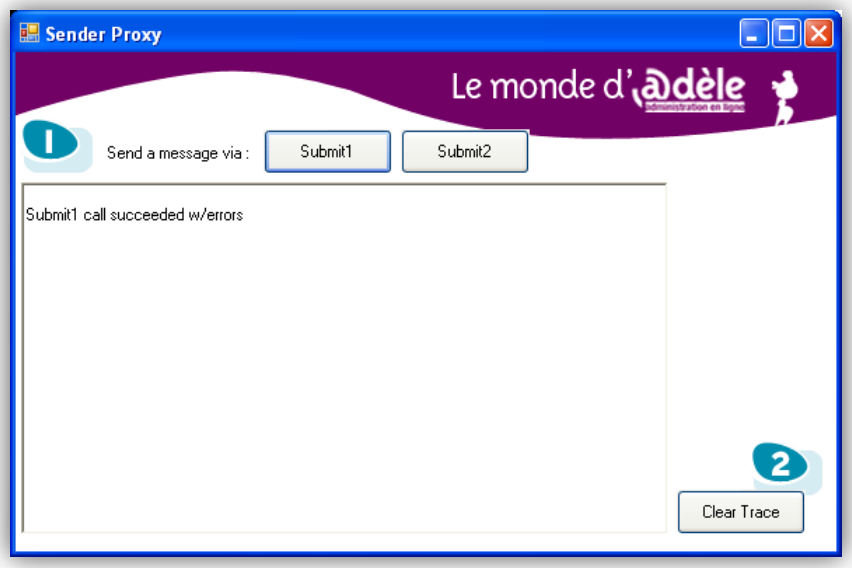
1. Open Windows Explorer and navigate to the *Samples\Beyond - Soap Intermediary\bin\[Debug|Release]* subdirectory under the directory location where you've installed the PRESTO Starter Kit.

2. Double-click the *ReceiverProxy.exe* console application.

3. Double-click the *SoapRelay.exe* Windows application.



4. Double-click the *SenderProxy.exe* Windows application.



How to run the Client from a different machine

Perform the following steps to run the client from a different machine:

1. Copy the Client program files from *Samples\Beyond - Soap Intermediary\bin\[Debug|Release]*, from under the directory location where you've installed the PRESTO Starter Kit, to the Client machine. The program files are *SenderProxy.exe*, *SenderProxy.exe.config*.

2. Open the *SenderProxy.exe.config* configuration file and change the address value of the endpoint definition to match the new address of the Service. Replace any references to "localhost" with a fully-qualified domain name in the address.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- The SOAP relay address for the INonNormativePresto endpoint -->
    <add key="viaAddress" value="http://localhost:9090/soaprelay" />
  </appSettings>
  …
</configuration>
```

References

[PRESTO-Guide]
"A guide to supporting PRESTO", June 2006.

[PRESTO-Ref]
"A Technical Reference for the PRESTO protocol", June 2006.

[MTOM]
"SOAP Message Transfer Optimization Mechanism", January 2005.

[SOAP 1.2]
"SOAP Version 1.2 Part 1: Messaging Framework," June 2003.

[WSDL 1.1]
"Web Service Description Language (WSDL) 1.1", March 2001.

[WS-AddressingAugust2004]
"Web Services Addressing (WS-Addressing)", August 2004.

[WS-Addressing10]
Web Services Addressing (WS-Addressing)", May 2006.

[WS-MetadataExchange]
"Web Services Metadata Exchange (WS-MetadataExchange)", September 2004.

[WS-Policy]
"Web Services Policy Framework (WS-Policy)", September 2004

[WS-RM1.0]
"Web Services Reliable Messaging (WS-ReliableMessaging) 1.0", February 2005.

[WS-RM1.1]
"Web Services Reliable Messaging (WS-ReliableMessaging) 1.1", April 2007.

[WS-RMPolicy]
"Web Services Reliable Messaging Policy Assertion (WS-RM Policy) 1.1", April 2007.

[WS-Security]
"Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)", March 2004.

[WS-SecX509]
"Web Services Security: X.509 Token Profile V1.0", March 2004.

[WS-SecurityPolicy]
"Web Services Security Policy Language (WS-SecurityPolicy)", April 2006.

[XMLDSIG]
"XML-Signature Syntax and Processing", March 2002.

[XMLENC]
"XML Encryption Syntax and Processing", August 2002.

[XML Schema, Part 1]
"XML Schema Part 1: Structures", May 2001.

[XML Schema, Part 2]
"XML Schema Part 2: Datatypes", May 2001.

[XOP]
"XML-binary Optimized Packaging (XOP) 1.0", January 2005.