# A WCF Stock Service (Singleton) Library, Service Host Console Application, and a “Tester” Client (ChannelFactory) Console Application

## About

This project presents a WCF Stock Service Library (*StockServiceLib*) that mimics a stock exchange. The service is implemented as a “singleton” and maintains persistent data between client calls and can handle multiple client sessions. The service is hosted via a console application (*StockServiceHost*). The client and service participate in a bi-directional/callback relationship. The client (*StockClient)* uses the ChannelFactory pattern as opposed to “Add Service Reference” with SVCUTIL. The client and service share a common assembly (*SharedLib*) that contains the key contract and data model information. Furthermore, a *Utilities* project is used by the client console application to facilitate user data entry and the complicated details of building and managing the WCF ChannelFactory connection implementation. The *ProxyGen* class inside the Utilities project abstracts the details of implementing and managing a generic ChannelFactory connection to a generic service for a client. Note: The Utilities project library was included as base code for my lab project to facilitate speedy completion; we were not expected to code this Utilities project ourselves due to complexity and time constraints. The remaining projects in the solution (SharedLib, StockClient, StockServiceHost, and StockServiceLib), I completed individually per requirements for the lab project.

## Architecture Overview

The StockServiceDemo Visual Studio solution application consists of five project assemblies:

* *SharedLib* (Class Library) “Service and Data Contracts”
* *StockClient* (Console Application) “Client Tester to Service”
* *StockServiceHost* (Console Application) “Hosts the Service”
* *StockServiceLib* (WCF Service Library) “Implements the Service”
* *Utilities* (Class Library) “Helper classes”

## SharedLib

This library project contains the service contract, callback contract, and two data contracts required by this service. You can see the project code here []

### Data Model Contracts

The data model contracts assure understanding and agreement of data objects flowing to/from the service and clients. In this example, it is implemented in a *SharedLib* assembly that will be shared amongst both the service and clients. The clients will utilize this class library to serialize/deserialize data for over the wire communications to consume the service. There are two data model objects represented in the service model: *Stock* code here [] and *StockTransaction* code here[].

#### Stock

This is the data model blueprint for a Stock object in the service. This class is the primary data model for the application. It contains a stock's ticker symbol and the current trading price.

This Data Contract has two public automatic properties:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| **Symbol** | string | Stock symbol, like "MSFT" for Microsoft |
| **Price** | decimal | Current trading price |

It has a default and parameterized constructor for object initialization. The class is marked with the [DataContract] and the properties each have [DataMember] attributes to explicitly state this class and its defined properties are available for WCF (WCF is an opt-in and if you do not explicitly define these attributes, it will be invisible to the WCF service/client capabilities).

Lastly, the class overrides the generic object ToString() method to better format the object’s properties for consumption.

The ToString() override returns the following formatted stock symbol and price:

*return string.Format("{0,-6} {1,10:N2}", Symbol, Price);*

#### StockTransaction

This class represents a change in a *StockTransaction* object. Basically, when a stock has a price change the change is stored in a StockTransaction object and it is sent to all clients that are monitoring the stocks.

This Data Contract will have four public automatic properties:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| **Stock** | Stock | Stock object |
| **Time** | DateTime | Date/time of the transaction |
| **Change** | decimal | Amount that the stock price changed |
| **Shares** | int | Number of shares traded at the new price |

Just like the Stock class, it also has a default and parameterized constructor for object initialization. The class is marked with the [DataContract] and the properties each have [DataMember] attributes to explicitly state this class and its defined properties are available for WCF (WCF is an opt-in and if you do not explicitly define these attributes, it will be invisible to the WCF service/client capabilities).

Lastly, the class overrides the generic object ToString() method to better format the object’s properties for consumption.

The ToString() override returns the following formatted StockTransaction that includes a special character to indicate the direction that the stock is moving (Note: If you cannot see the special symbol on your console application, you might have to change character to something like V for down and ^ for up like I had to do in my project implementation):

*char direction = '=';*

*if (Change < 0)*

*{*

*direction = '\u25BC';*

*}*

*else if (Change > 0)*

*{*

*direction = '\u25B2';*

*}*

*return string.Format("{0:yyyy-MM-dd HH:mm:ss} {1} {2} {3,10:N2} [{4,8:N0}]",*

*Time, Stock, direction, Change, Shares);*

### Service and Operations Contract

There are two service interfaces defined: *IStockService* (interface for the client calling the service) and *IStockCallback* (interface for the service calling back the client). You can see the IStockService code here [] and the IStockCallback code here [].

#### IStockService

This is the interface called by the client to the service and has the [ServiceContract] attribute applied to it. The [ServiceContract] attribute also required the following configuration:

* SessionMode set to SessionMode.Required
* CallbackContract set to typeof(IStockCallback)

The following operation contracts were added to the IStockService interface:

|  |  |  |
| --- | --- | --- |
| **Name** | **Parameters** | **Returns** |
| **Login\*** | None | void |
| **Logout\*\*** | string sessionID = null | void |
| **StartTickerMonitoring** | None | void |
| **StopTickerMonitoring** | None | void |
| **GetStockQuote** | string symbol | Stock |
| **AddNewStock** | string symbol, decimal price | void |

\*This OperationContract must have its IsInitiating property set to true. \*\*This OperationContract must have its IsTerminating property set to true.

#### IStockCallback

The second interface is called IStockCallback. This interface acts as the contract that the client must implement to receive callback messages. It has one method and does not require a ServiceContract attribute.

*[OperationContract(IsOneWay = true)]*

*void StockUpdated(StockTransaction tx);*

## StockServiceLib

*StockServiceLib* contains the service code that implements the IStockService interface. It has a project reference to *SharedLib*.

### StockService Class

This class implements the IStockService interface and has the ServiceBehavior attribute applied to it. The ServiceBehavior attribute has its InstanceContextMode property set to *InstanceContextMode.Single* to ensure there will be only one service running on the server for all clients (Singleton pattern). You can see the StockService class here [].

This class has four private fields:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| **m\_Clients** | ConcurrentDictionary<string, ClientContainer> | Maintains a list of clients connected to the service, keyed by the client session ID |
| **m\_Stocks** | ConcurrentDictionary<string, Stock> | List of Stock objects, keyed by the stock’s ticker symbol |
| **m\_Rnd** | Random | Random number generator |
| **m\_Timer** | Timer | Timer that will fire periodically, changing a stock’s value and notifying clients |

The *ConcurrentDictionary* objects behave very much like standard Dictionary objects except they are tolerant to multiple threads accessing the data.

The default constructor of this class does the following:

* Initialize m\_Clients, m\_Stocks and m\_Rnd to new default objects of their respective types.
* Create a set of default stocks
* Initialize the timer for the callback

#### Login()

Provides the implementation for the Login method. It retrieves the client’s session ID and the client’s callback channel. These values are used to cache the list of clients into the m\_Clients field. If the user has already logged in, then a fault will be thrown to the client.

#### Logout(string sessionID = null)

This method performs the necessary logic to logoff a client. Essentially, the client with the given session ID is removed from m\_Clients. If the session ID is not provided (which is likely coming from the client) then it is discovered using the same technique as in Login.

#### AddNewStock(string symbol, decimal price)

AddNewStock does pretty much what the name says – it adds a new Stock object to the m\_Stocks dictionary. This is also the method called in the constructor when adding the list of default stocks. The symbol parameter is the ticker symbol for the new Stock, which also acts as its key in the m\_Stocks dictionary. The price parameter is the starting price for the new Stock. If the stock symbol already exists the new stock is rejected and a fault is thrown to the client.

#### GetStockQuote(string symbol)

This method searches for the given Stock in the m\_Stocks dictionary (remember, the key in the dictionary is the stock symbol). If the Stock object exists, return it. Otherwise, it throws a new FaultException.

#### StartTickerMonitoring()

This method is called from the client when the client wishes to be notified about changes to stocks over time. All this method does is modify the appropriate ClientContainer (*see next section*) object in m\_Clients to set the IsActive property to true. The timer looks for this value when deciding if a client should be notified.

#### StopTickerMonitoring()

This does essentially the same thing as the StartTickerMonitoring method except it will set the IsActive property to false to stop the client from receiving updates.

#### StockTimerCallback(object state)

This method acts as the timer callback event handler for m\_Timer. When this event is triggered, a Stock object in m\_Stocks will be randomly modified and notifications will be sent to any subscribing clients. The method does not to allow a Stock to have a negative price. Once a Stock object has been picked and modified, each client with the “IsActive” property set to true will be notified of the change.

### ClientContainer Class

This class called ClientContainer acts as a container or “wrapper” for the client callback objects that will be called when a stock price changes. It has both a default and parameterized constructor for object initialization. You can see the ClientContainer class code here []

This class has two public automatic properties:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| **ClientCallback** | IStockCallback | Reference to a client callback object (ie. handle to a client object) |
| **IsActive** | bool | Indicates if the given client is set to receive callback messages. |

## StockServiceHost

The StockServiceHost is a typical Console Application being used to host the StockService service class. For the hosting configuration, I used the net.tcp protocol. The <system.serviceModel> portion of the config file was customized to the solution address, bindings, and contract. See the XML configuration here[] and project code here [].

*<system.serviceModel>*

*<services>*

*<service name="StockServiceLib.StockService">*

*<endpoint address="" binding="netTcpBinding" contract="SharedLib.IStockService" />*

*<endpoint address="mex" binding="mexTcpBinding" contract="IMetadataExchange" />*

*<host>*

*<baseAddresses>*

*<add baseAddress="net.tcp://localhost:977/StockService/"/>*

*</baseAddresses>*

*</host>*

*</service>*

*</services>*

*<behaviors>*

*<serviceBehaviors>*

*<behavior>*

*<serviceMetadata/>*

*<serviceDebug includeExceptionDetailInFaults="False"/>*

*</behavior>*

*</serviceBehaviors>*

*</behaviors>*

*</system.serviceModel>*

## StockClient “Tester Client Console Application”

The stock client project is a Console Application that consumes the service through a ChannelFactory pattern with the IStockService interface. It tests the Stock Service by adding, retrieving, and monitoring stocks. You can see the project code here [].

### StockMonitor

The class acts as the target for callbacks from the service. Also, it implements IStockCallback (the client callback contract). That interface contains only one method, the implementation of that interface method. Click here [] to see the class code.

### StockUpdated(StockTransaction tx)

This method is called when the service wishes to notify the client of a stock price change. This method prints the “tx” parameter to the Console in preferred formatted string overridden in the StockTransaction class. Click here [] to see the class code.

### Implementing the UI

Since this is a Console app, the UI is simplistic. First, an enum (MenuChoicesEnum) to the Program class is called so that client application user can select a menu of choices to interact with the service. Click here [] to see the UI client code.

The Program class implements two private static fields in Program:

static IStockService m\_Proxy - *m\_Proxy will contain a reference to the server proxy for access by all methods in this class.*

static StockMonitor m\_Monitor - *m\_Monitor is the object that will receive callback events from the service.*

### Main()

The Main method is just a loop with prompts asking the user what action to perform. It utilizes the ConsoleHelpers class in the Utility project to help the user choose one of the values in MenuChoicesEnum. Depending on the choice made by the user, a different method is called.

Also in this method is the call to ProxyGen.GetChannel which passes a reference to m\_Monitor to indicate what object should be contacted for callbacks from the service. The overload of GetChannel that is called will create the proxy using DuplexChannelFactory instead of the normal ChannelFactory. This is what enables bi-directional communication between client and service.

### AddStock()

AddStock prompts the user for a string called symbol and a decimal called price. Then the AddNewStock method is called upon the m\_Proxy object, passing in the given variables. If successful (no exception) it notifies that the user that the stock was successfully added. Otherwise, it displays the error.

### GetStockQuote()

GetStockQuote prompts the user for a string called symbol. Then the GetStockQuote method is called upon the m\_Proxy object, passing in the given variable. If successful (no exception) then output the returned Stock object. Otherwise, it displays the error.

### MonitorStocks()

This method does the following:

* Notify the user that stock monitoring has begun and that they should press enter to stop.
* Call the StartTickerMonitoring method of the m\_Proxy object.
* Wait with a Console.ReadLine call.
* When the user hits enter and passes the Console.ReadLine call from the previous step, call StopTickerMonitoring on m\_Proxy.

### Exceptions

Note: When handling exceptions on the client, it catches a "FaultException" as that is what the service will send. FaultException is defined within System.ServiceModel. The property that is displayed on the client is the "Reason".

### Configuration

The client app.config file was updated with the correct serviceModel configuration. See the XML configuration file here []

*<system.serviceModel>*

*<client> <endpoint address="net.tcp://localhost:977/StockService/"*

*binding="netTcpBinding"*

*contract="SharedLib.IStockService"*

*name="netTcpBinding\_StockService"/>*

*</client>*

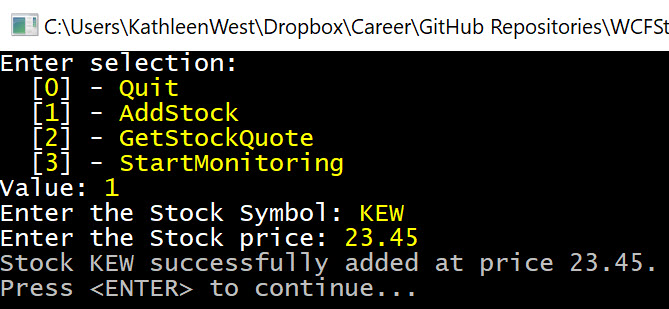
*</system.serviceModel>*

## Utilities

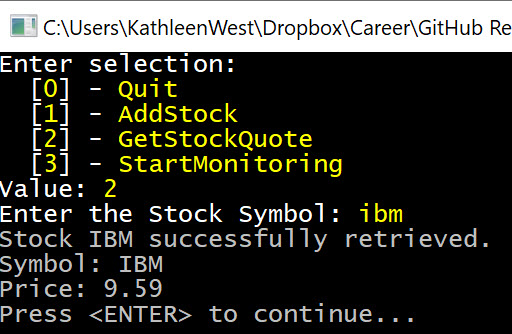
Furthermore a "Utilities" project is used by the client console application to facilitate user data entry and the complicated details of building and managing the WCF ChannelFactory connection implementation. The "ProxyGen" class inside the Utilities project abstracts the details of implementing and managing a generic ChannelFactory connection to a generic service for a client. Note: The Utilities library was included as base code for my lab project to facilitate my speedy completion; we were not expected to code this ourselves due to complexity and time constraints. The remaining projects in the solution (SharedLib, StockClient, StockServiceHost, and StockServiceLib), I completed individually per requirements for the lab project. Click here [] to see the Utilities code.

## Demo

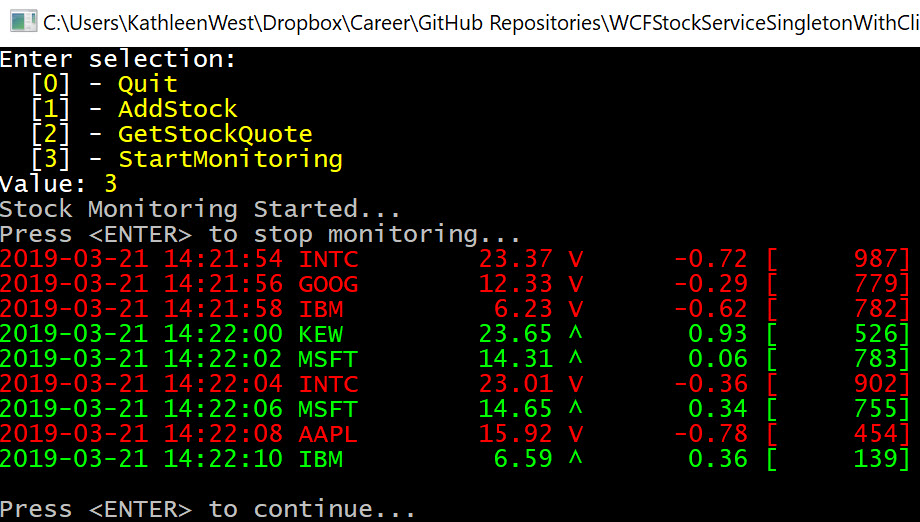
Add Stock



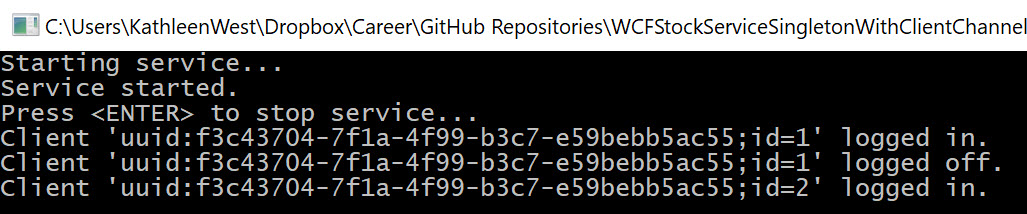
Get Stock Quote



Monitor Stocks



Service Host Console Application with Client Session Monitoring



## Code

The entire project code repository is available on GitHub here.