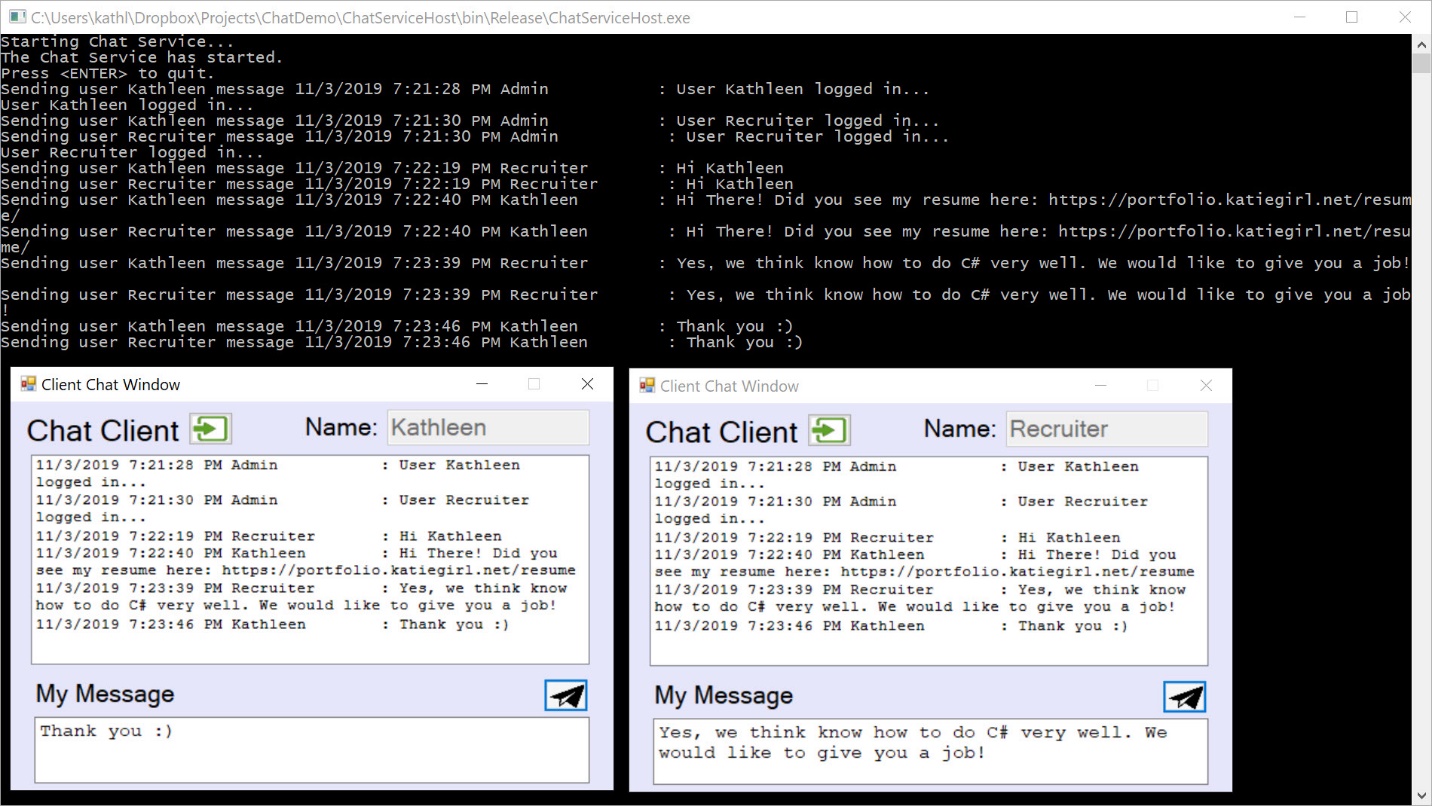
# Simple Chat Messenger – A Demo of a WCF Self-Hosted Service & Client "Tester" Windows Form Application Exchanging Messages

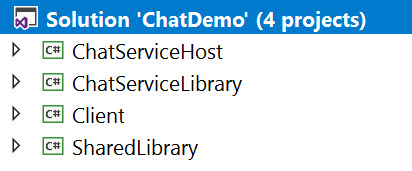
## 

## About

This project presents a simple Chat Message Service and Client Application demonstration. The Chat Service is a self-hosted (service host) WCF application launched and managed with a simple console interface. The service prints to the console window the user activities (log on, log off) and what messages are sent to users. The client “tester” has a simplified GUI user interface to quickly demo and test the service (Windows Form Application). The GUI has features to log on, log off, send a message, and receive messages from other users on the chat service.



## Architecture



The demo project consists of these component topics:

* Shared Class Library Project “SharedLibrary”
  + IChatService (Interface for Service)
  + IChatServiceCallback (Interface for Service Callback)
  + DuplicateUserFault (Class describing a WCF Exception as a Fault)
  + ChatMessage (Class Describing a Chat Message Object)
* Service Class Library Project “ChatServiceLibrary”
  + ChatService (Code that Implements the Service Interface)
  + App.config (Configuration Reference for Service Host)
  + *Reference to the Shared Class Library*
* WCF Service (Host) Application Project “ChatServiceHost”
  + Program (Starts, Manages, Stops the Service)
  + App.config (Configuration for Service Host)
  + *Reference to the ChatServiceLibrary*
* Client “Tester to Service” Windows Form Application Project “Client”
  + *Reference to the Shared Class Library*
  + Main Form GUI User Interface
  + Form Code – Processes GUI User Interface

The service interface is defined not in the service application but in a *Shared Library*. This library defines the interface contracts for the chat message services (ex: Send Message) and is referenced by both the client and service host projects. The *SharedLibrary* also has a class definition that defines a chat message object with a name, timestamp, and message properties. Furthermore, a callback contract is defined for the client to implement so they can receive chat messages while connected to the service. Lastly, since exceptions (errors) are really faults in WCF services, a custom error (fault) class is defined in the shared library so that the client can understand and process the duplicate user faults (errors).

The *ChatServiceLibrary* implements the Chat Message service and contracts as defined in the *SharedLibrary*. The *ChatServiceHost* is a simple console application that is responsible for starting the Chat service, hosting, and managing the service (self-hosted).

The service behaviors were designed to allow all the clients to connect to a single instance of the service that sends messages to all connected clients. Further improvements and features such as thread locks should have been implemented to safeguard the shared resources, however that was not the purpose of this short demo application as I budgeted limited time for this project.

A client “tester” windows form application tests the service and provides output to the user in a simple GUI.

## Shared Class Library

A Class Library project (SharedLibrary) was added to my Visual Studio solution. This library is shared amongst the client and the service. The code is available on GitHub [here].

### IChatService (Interface for Service)

The ServiceContract for the Chat Message service contains the following operation contracts.

* void Login(string userName);
  + Logins the user and client and also registers a callback to this user/client to receive the messages
* void Logoff(string userName);
  + Logs off the user/client from the service and unregisters it to receive callbacks
* bool LogInState(string userName);
  + Returns the loggedin/connection state of the user
* void SendMessage(string userName, string message);
  + After the client sends a message to the service it EXPECTS a callback from this operation with the ChatMessage object that is created on the Service from it's request.
* List<ChatMessage> GetMessageHistory();
  + Returns the Message History limited to Last X (defined) Chat Messages. This is stored as a Queue on the Service.

#### Discussion

The Login method implements a Fault (Exception in WCF) contract for the client to be able to receive errors if it tries to log in with a duplicate user.

The *SendMessage* method implements bi-directional communication with its attribute (IsOneWay = false). After the client sends a message to the service it expects a callback from this operation with the ChatMessage object that is created on the Service from its request. This hinges on the Service Contract being configured to handle callbacks with a specific interface of instructions (ex: [ServiceContract(CallbackContract = typeof(IChatServiceCallback))]).

The code is available on GitHub [here].

[insert code here]

### IChatServiceCallback (Interface for Service Callback)

The ServiceContract for the Chat Message service contains only one callback to the client. Whenever a client is connected to the service, it will receive callbacks from the service with the recent chat messages happening at that instant in time. The client is required to implement the service callback interface in order to receive the instant chat messages. In addition, the operation behavior is set to one-way so that the service does not wait on a response from the client, which could cause performance issues. Therefore, this is a one-way message feature.

The code is available on GitHub [here].

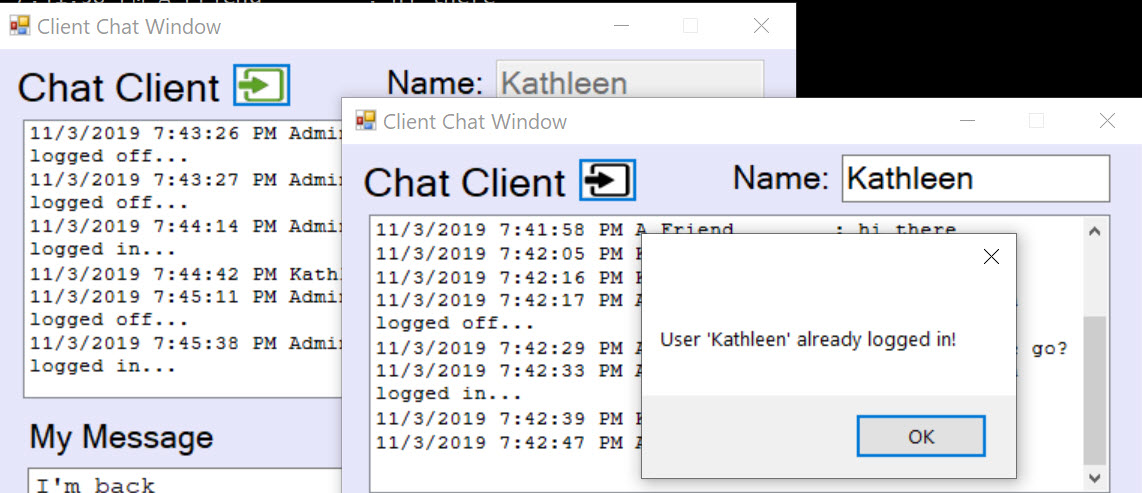
[insert code here]

### ChatMessage (Class for Chat Message Object)

The *ChatMessage* class represents a chat message object in the application. It describes the message ChatMessage originator (sender’s name as a string), the DateTime that it was sent (universal time reference on the service), and the message (string). In addition, there is a custom override of its *ToString()* object representation for display in the client form application list window of the chat messages. The object string representation formats the ChatMessage to display as one long string, but the client does some additional formatting to show the message “word wrap”. The code is available on GitHub [here].

[insert code here]

### DuplicateUserFault (Class for Error/Exception/Fault Object)



The *DuplicateUserFault* class represents a specific error condition in the application that is an Exception (error) on the service and communicated as a fault in WCF to the client. It describes the specific error condition of a client trying to login with a username that is already logged in to the chat service. The fault is communicated to the client to handle and understand that it could not log in and why. The code is available on GitHub [here].

[insert code here]

## Chat Message Service Library

A class library describing the service implementation of the interfaces was added to my Visual Studio solution. Separating the service implementation from the host application project, provided one more level of abstraction and separation to better organize the solution. The code is available on GitHub [here].

### ChatService (Code that Implements the Service Interface)

The service implementation code has service behaviors configured to allow for a single instance while each service call connects to the same instance. The service library references a shared class library called “*SharedLibrary*” that contains the interface definition for the service and the class definition for a file object (*ChatMessage*) holding the chat message information as well as the Fault (Exception) “*DuplicateUserFault*” class model. The code is available on GitHub [here].

#### Fields

The fields section contains a few notable private fields. The first field “maximumMessages” limits the chat message history to a rolling of X number of messages. The chat messages (last X maximumMessages) are stored in a Queue container that represents the message history. Lastly, a Dictionary contains the logged in users (string) and their callback information (IChatServiceCallback).

[insert code here]

#### Service Interface Implementation Methods

The service implementations are described below.

##### Get Message History

Returns a list of the most recent chat messages (List<ChatMessage>)

[insert code here]

##### Login

Logins the user and registers the callback into a dictionary of logged in users. Sends a chat message that the user has logged on.

[insert code here]

##### Login State

Returns the login state (true if user is logged in, false otherwise) of a user name from the client (string).

[insert code here]

##### Logoff

Logs off the user by removing the user from the logged in user list and sends a message that the user has logged off.

[insert code here]

##### Send Message

The service implementation that handles the client request to send a chat message

[insert code here]

#### Helper Methods

The service helper method implementations are described below.

##### Add Message

Adds the chat message object (ChatMessage) to the queue collection

[insert code here]

##### Send Message to Users

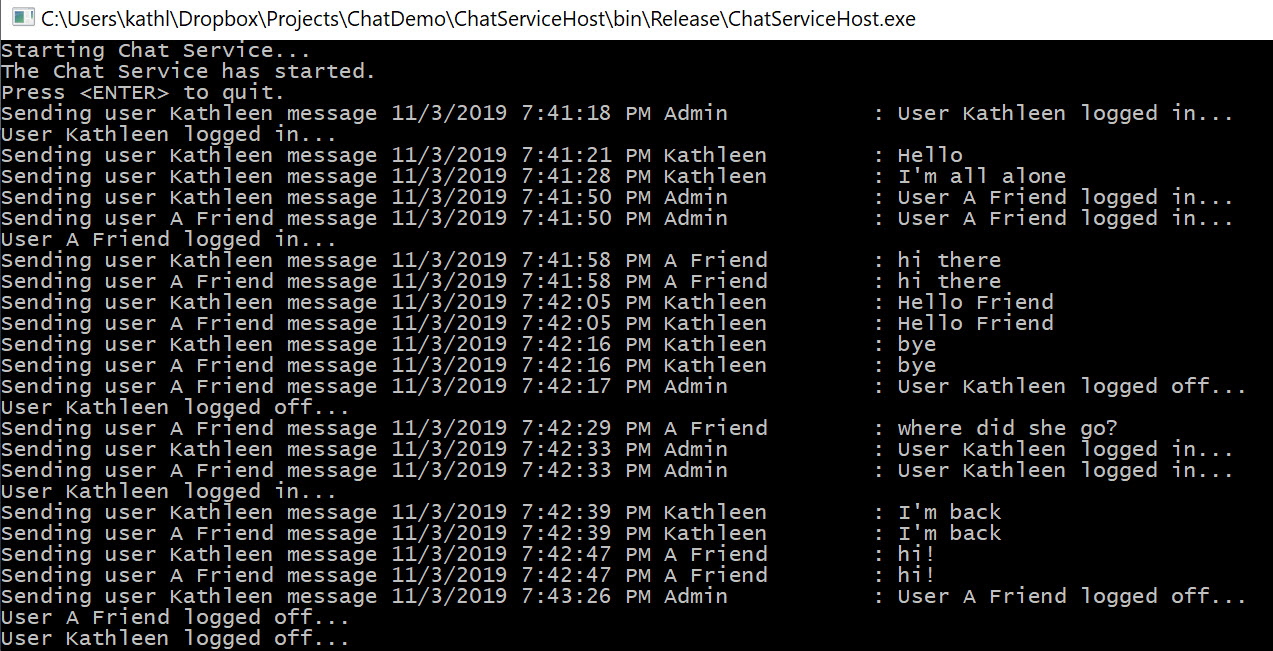
Sends the chat message object (ChatMessage) to registered users & callbacks. Logs off any disconnected clients.

[insert code here]

### Application Configuration App.Config

The application configuration simple sets up the service model configuration including the endpoint, interface contract name, tcp protocol, and the base address for this demo. This configuration *must be included where the application is hosted* *(run)* (see next section) and is displayed in this service class library project for reference. The code is available on GitHub [here].

## Chat Message Service Host (ChatServiceHost)



The *ChatServiceHost* project is a simple console application responsible for starting, managing, and stopping the Chat Message service. This demo hosts the service using tcp protocol on a custom port address as described in the application configuration file. The code is available on GitHub [here].

### Main Program

The main program is the entry point for the service host application. It creates a ServiceHost object of type described in the *ChatServiceLibrary*, opens the host up for connections, and waits for the user to manually close the service (pressing enter in the console window). Also it receives console commands from the service implementation that logs the activity going on the server (user login, logoff, messaging, etc.).

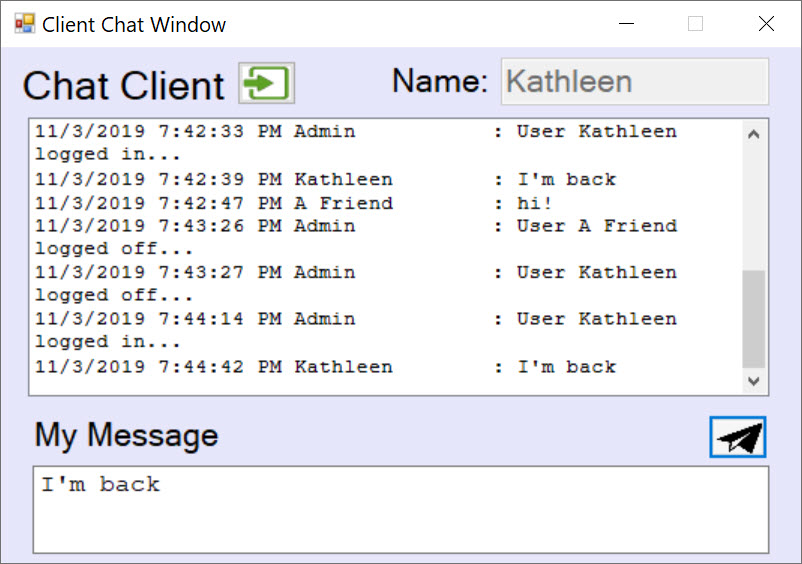
[insert code here]

### Application Configuration App.Config

The application configuration simple sets up the service model configuration including the endpoint, interface contract name, tcp protocol, and the base address for this demo. The code is available on GitHub [here].

[insert code here]

## Client “Tester to Service” Windows Form Application



The client “tester to service” is a simple windows form application project in the same solution that connects to the “Chat Service” by use of a ChannelFactory proxy to a specified service name that is defined in the client application configuration. The client program will use this proxy to test the OperationContract or methods available in the ServiceContract and return the results to the user on the form window.

The client application shares and references the “Shared Library” – a class library project. It has access to the Chat Message service interface to know exactly how it can consume the service with the help of a ChannelFactory proxy (discussed in the next section). The project code is available on GitHub [here].

### Features

The client application was meant to only demonstrate consumption of the Chat Message service and has limited scope of features. Some of the client application features are:

#### Simple UI Design

The user interface is simplified to allow the log in and log off of the chat message service with one button. The connection action button turns green when connected and black when disconnected. The UI adapts to the connection state of the client to the service. When connected, the user has the ability to write and send messages to the chat service. When disconnected, those features are disabled.

#### Validation

The client validates the login state before sending messages or performing complex actions. The username is validated locally and at the service. If there is an existing user logged in, the service sends a detailed fault that the client then processes and alerts the user through a message dialog window. Additional validation to the right state works in tandem with the simple UI design features.

#### Chat History & Formatting

The chat history is retrieved from the chat message service and formatted specially to fit in a word-wrap text style in the Windows Form control list box. A special thanks to a stack-flow article [here] that I used to make sure the text was formatted correctly to wrap in the list box.

### Form Code

The code behind file is for the client tester and manages the application in one file. A separate business logic/process layer could have been added but the focus of the demo was on the service and consumption, not a UI/UX design. The code behind the form is available on GitHub [here].

#### Fields

* Private fields allow the client to work with the following:
* Logged into service state
* User name on the client (may be truncated on service)
* If the chat message history is loaded into the client

[insert code here]

#### Constructors

When the form is instantiated, it does some house keeping on the form GUI (ex: buttons enable/disable). Send message capability is disabled until the user successfully connects to the service.

[insert code here]

#### Events

Connection Button Click

Clicking on this button either connects or disconnects the client from the Chat message service. If the user is “locally” logged in (the field is set to true), the it proceeds to logoff the user and update the GUI accordingly. If the user is logged of, it proceeds to validate the client user name text entry, then try to login to the chat messenger service. If it successfully logs in, it will update the UI and retrieve the chat message history.

[insert code here]

##### Send Message Button Click

Before attempting to send the message, it will first validate the user is logged in before sending a chat message (string) to the chat service. It does not assemble the ChatMessage object locally but rather relies on the service to create the object with the server timestamp and formatting, then return the object through the callback implementation (discussed below).

[insert code here]

##### Chat List Box Formatting Events

Much thanks to the Internet of programmers for finding a solution to word-wrap a formatted string to the Windows Form ListBox control width without having to show the horizontal scrollbars. The code reference is [here].

[insert code here]

#### IChatServiceCallback Implementation

The client must implement the IChatServiceCallback interface contract to be able to receive the instant chat messages from the service from all users, including the ones they sent. This method is called when a message is received from the server. It processes the incoming message from the service from the chat conversation and adds to the UI chat message list box. Also, it checks if UI thread synchronization is required before processing the UI update. If synchronization is required, it invokes a delegate on the UI thread to process the new ChatMessage object in the GUI.

[insert code here]

#### Methods

##### IsUserLoggedIn

Connects to the service to see if the user is logged in.

[insert code here]

##### ValidateGUIUserName

Validates the username text entry to be not empty and unique and not already logged in on the service.

[insert code here]

##### Login

Tries to login this client's username to the service.

[insert code here]

##### LogOff

Logs off the user from the service and updates the GUI locally

[insert code here]

##### GetMessageHistory

Gets a list of ChatMessages from the service to update the UI list box

[insert code here]

##### UpdateChatGUI

Cleans house on the GUI to make it in sync with being logged in, able to send messages, etc...

[insert code here]

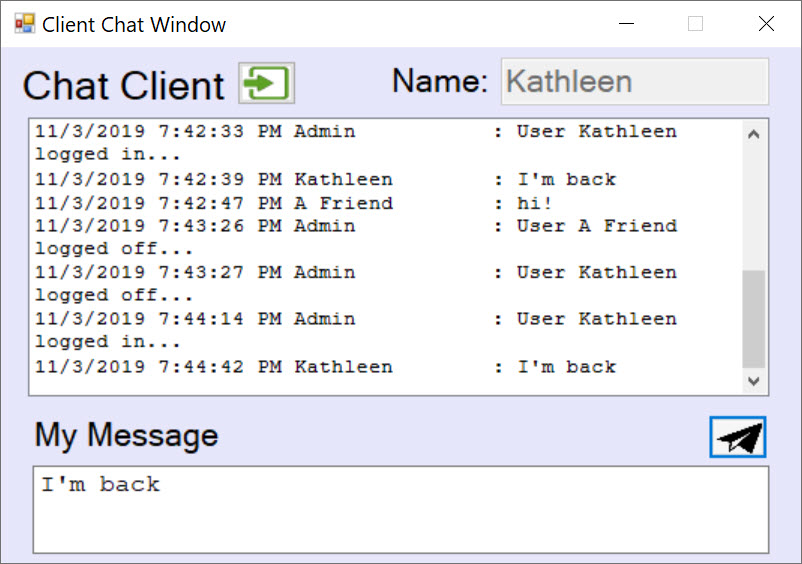
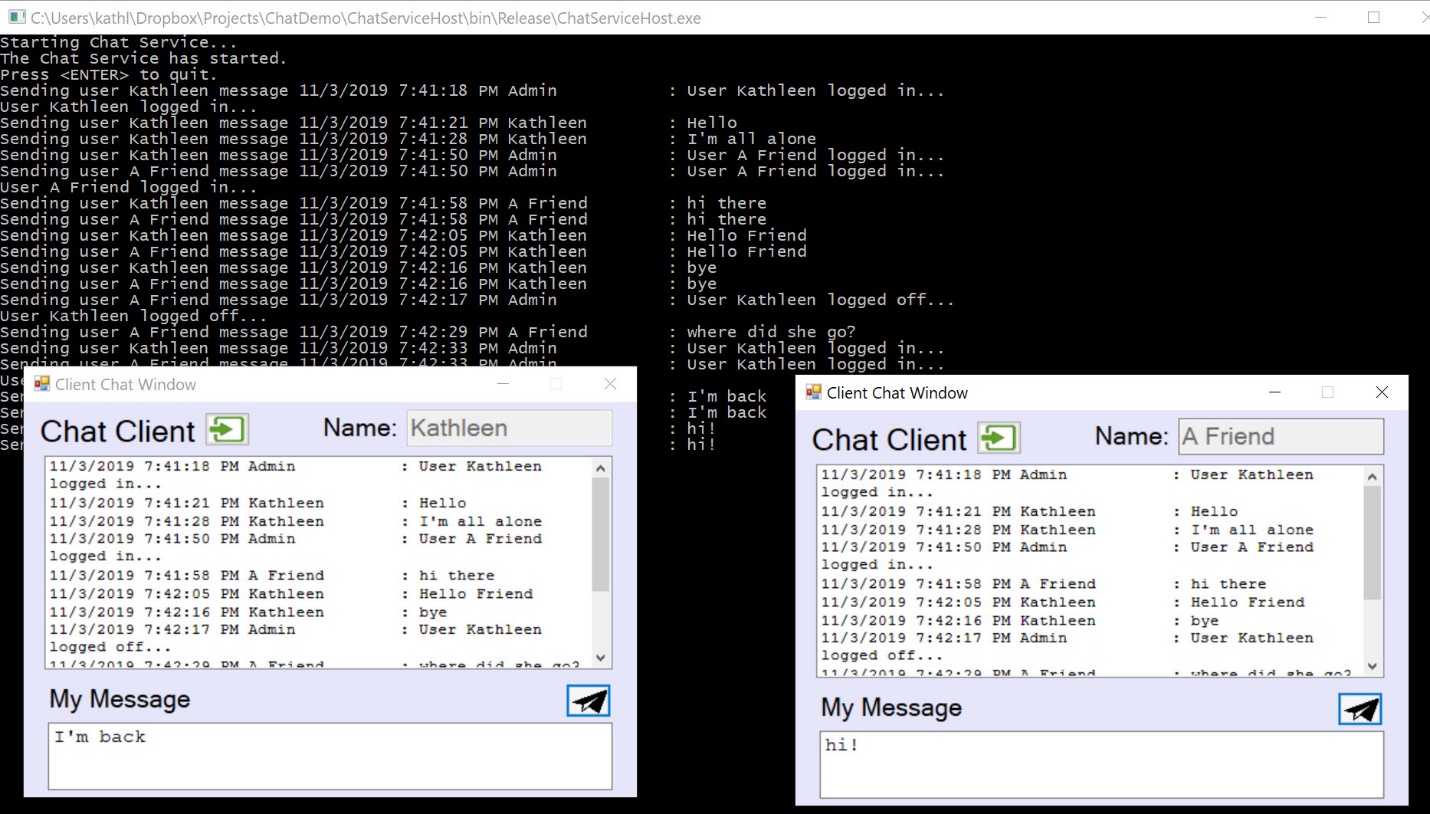
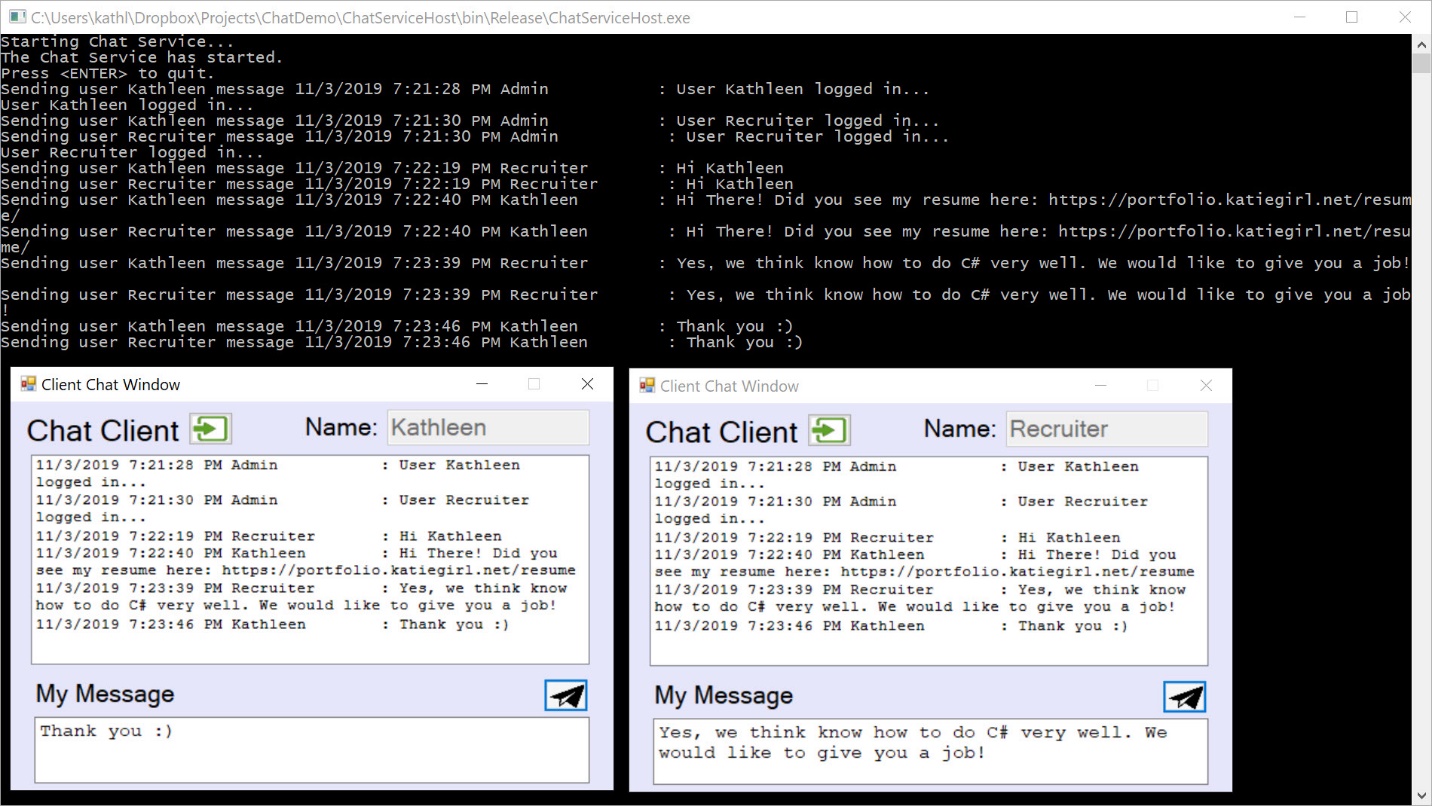
### ChannelFactory Proxy and Application Configuration

The application configuration file (App.config) specifies the service endpoint address, binding, and contract information for the client to create a proxy and connect. The service contract information is in the Shared Class Library that has the service interface. The code is available on GitHub [here].

[insert code here]

## Demo

[insert video link]



## Code

The entire project code repository is available on GitHub [here].