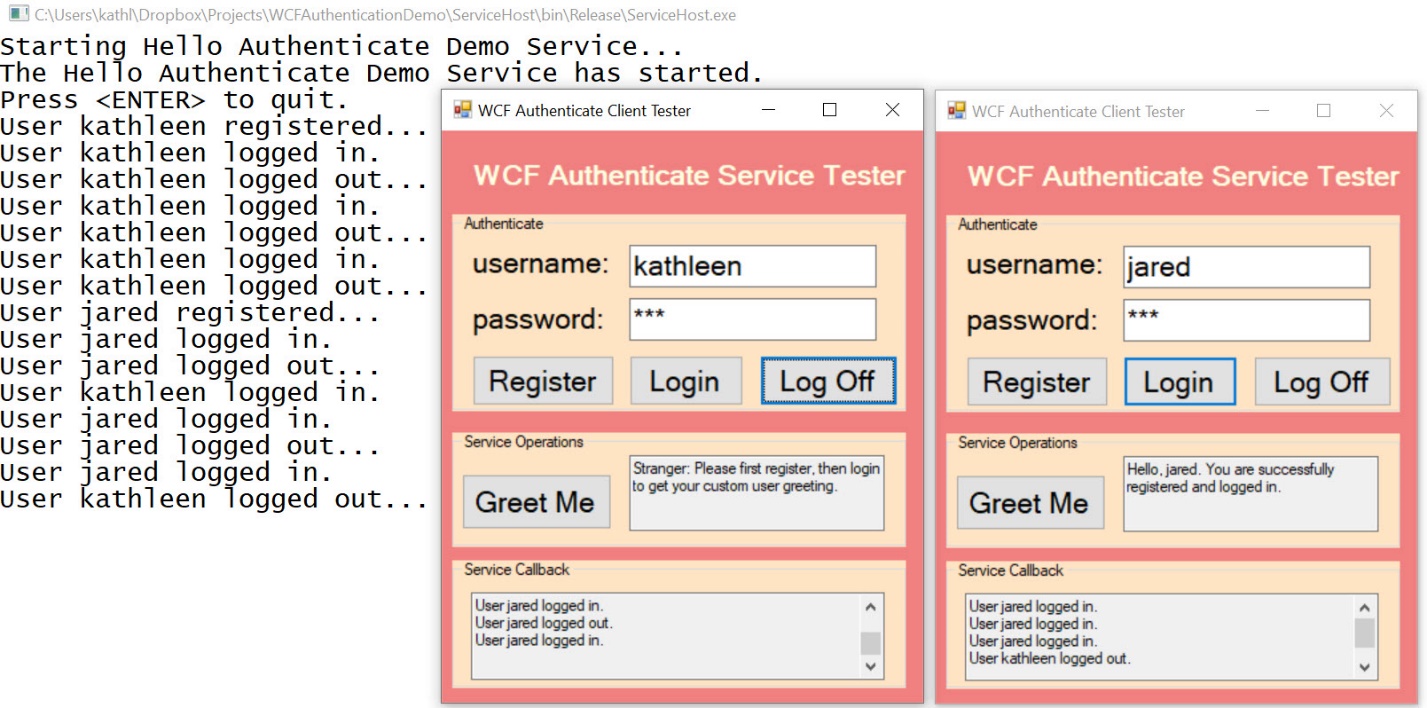
# WCF Authentication Service Application + Custom Validator + Secure Password Storage + Error Handler + Certificates & Client Tester Windows Form Application

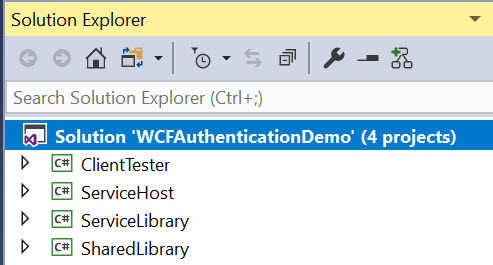
## 

## About

This project presents a Visual Studio solution including a simple demo WCF Authentication Service Application and a “Tester” Client (Windows Form Application) that allows the user to test the user registration, login, logout, and service operations. In addition to demonstrating standard authentication capabilities, the WCF service implements a custom username and password validator pattern. Passwords are stored securely using Password-Based Key Derivation Function PBKD cryptology of which the implementation is discussed. A custom error handler ensures that exceptions are properly wrapped into WCF Faults and communicated to the client caller. Certificates are discussed along with how to implement a server certificate on a client machine for development testing of “integrity” and application trust. The project includes a demo certificate and script for generating self-signed dev certificates, which must be installed into the client certificate store for the client tester application to trust and access the demo service. The client “tester” windows form application is not intended as a UX/UI demo but used to test and verify that the backend authentication service registration, login, logout, service operations, and callbacks are working as expected and sending proper WCF fault messages. Lastly, the project is shown in the demo section with a video and screen captures.



## Architecture



The demo project consists of these simple component topics:

* *SharedLibrary* - Class library shared amongst server and client describing service contract interfaces and common fault models.
* *ServiceLibrary* – Class library that implements the service contracts and backend features
* *ServiceHost* – Console Application that launches and hosts the Service
* *ClientTester* - Client “Tester to Service” Windows Form Application Project

## SharedLibrary Service Contract Interfaces & Common Models

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].

### IHelloAuthenticateService (Interface for Service)

The ServiceContract for the Authentication service contains the following operation contracts. The code is available on GitHub [here].

* bool Register (string username, string password)
  + Register the user with the service. Only distinct usernames are allowed, and the client will receive a *DuplicateUserFault* if the username is already existing on the service. Note: credentials are passed using WCF security.
* bool Login (string username)
  + Logins the user after it passes the custom validation (precursor) and also registers a callback to this user client to receive the service messages. Note: credentials are passed using WCF security.
* bool LogOff (string username)
  + Logs off the user/client from the service and unregisters it to receive callbacks. Note: credentials are passed using WCF security.
* string GreetMe (string username)
  + Receive a customized greeting by the service (if registered and logged in) or a default one.

### Service Error Handling Interface Design



Each of the interfaces are labeled with the attributes of *[FaultContract(typeof(GenericFault))]* so that any exception caught on the service can be packaged as a fault object and sent to the caller with the detailed information regarding that exception. Furthermore, with the Register and Login operational contracts, another fault contract *[FaultContract(typeof(DuplicateUserFault))]* is specified so that it can catch and return issues with duplicate user exceptions in a detailed fault model instead of the generic one.

#### 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 registered or logged in to the service. This class had only one property, the Reason for the fault. The property is completed by the service application when it packages its exceptions as faults to send to the client caller. 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]

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

The *GenericFault* class represents a general fault (exception) object in the application. Specifically, it is meant to model any exception that was caught and handled on the service when the client calls an operational contract. This class had only one property, the Reason for the fault. The property is completed by the service application when it packages its exceptions as faults to send to the client caller. 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]

### IServiceCallback (Interface for Service Callback)



The ServiceContract for the Authentication service contains only one callback to the client. Whenever a client is connected to the service, it will receive callbacks from the service with select informational messages happening at that instant in time (example: login and logoff). The client is required to implement the service callback interface in order to receive the instant 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]

## ServiceLibrary Service Implementation and Features

The *ServiceLibrary* project implements and manages the service to client consumers. In addition to the main *HelloAuthenticateService* service class, it contains separate class design and organizational architecture to model users, cryptology requirements for secure password storage, custom username and password validation, and error handling. The subsections will discuss the ServiceLibrary project features. The complete project code is available on GitHub [here].

### User Model



In this service application, a user is modeled with the *User* class. The class code is available on GitHub [here].

#### Fields

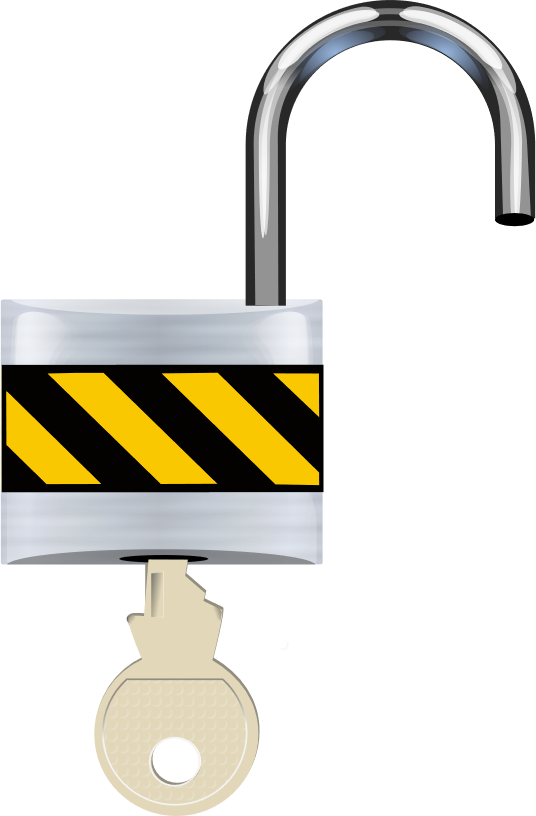
* *String UserName* - Username of the Registered User
* *IServiceCallback Callback* - Callback to Client User for Duplex Communications
* *byte[] Salt* - unique cryptologic salt for user
* *byte[] Password* - derived key password (after PBKD)
* *int WorkFactor* - number of iterations for the PBKD function

#### Constructor

The constructor creates a user using PBDK by first generating a salt, then performing work to derive a secure hash from the user’s password string converted Unicode byte array. PBDK cryptology and the implementation is discussed more in detail in the next section, but the *User* class utilizes the static methods in the *ServiceCryptology* class in this project to obtain PBDK. The PBDK is stored in the *User* model for that is made when a user registers on the service for the first time. Whenever the user attempts to login or logoff, their credentials are converted to PBDK and compared to the in-memory PBDK. First the User is matched/found on UserName and PBDK utilizes the existing/registered User salt, etc.

[insert code here]

### PBDK (Password-Based Key Derivation) Secure Password Cryptology



The *ServiceCryptology* class implements Password-Based Key Derivation Function PBKD for password management. PBKD is a key derivation function with a sliding computational cost, used to reduce vulnerabilities to brute force attacks. PBKDF2 applies a pseudorandom function, such as hash-based message authentication code (HMAC), to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult, and is known as key stretching. [Reference here: <https://en.wikipedia.org/wiki/PBKDF2>]

Building my PBDK I referenced a blog article [here https://www.mking.net/blog/password-security-best-practices-with-examples-in-csharp] that explains the steps of using PBDK in securing a user password on the back-end, etc.

The class code is available on GitHub [here].

#### Generating a Salt

In cryptography, a salt is random data that is used as an additional input to a one-way function that hashes data, a password or passphrase. Salts are used to safeguard passwords in storage. A new salt is randomly generated for each password. In a typical setting, the salt and the password (or its version after key stretching) are concatenated and processed with a cryptographic hash function, and the resulting output (but not the original password) is stored with the salt in a database. Hashing allows for later authentication without keeping and therefore risking exposure of the plaintext password in the event that the authentication data store is compromised. Salts defend against a pre-computed hash attack. Since salts do not have to be memorized by humans, they can make the size of the hash table required for a successful attack prohibitively large without placing a burden on the users. Since salts are different in each case, they also protect commonly used passwords, or those users who use the same password on several sites, by making all salted hash instances for the same password different from each other. [Reference here: <https://en.wikipedia.org/wiki/PBKDF2>]

[insert code]

#### Generating a Hash

This method computes and returns the generated derived password key.

[insert code]

### Custom Username and Password Validator

The *UserNamePasswordAuthenticator* class implements a "Custom User Name and Password Validator" pattern. By default, when a user name and password is used for authentication, Windows Communication Foundation (WCF) uses Windows to validate the user name and password. However, WCF allows for custom user name and password authentication schemes, also known as validators. The application configuration file on the Service Host (discussed in the next section) must include a configuration to point to the project library and class containing the custom validator:

*<serviceCredentials>*

*<userNameAuthentication userNamePasswordValidationMode="Custom" customUserNamePasswordValidatorType="ServiceLibrary.UserNamePasswordAuthenticator, ServiceLibrary"/>*

The class code is available on GitHub [here]. Microsoft has documentation that further explains the custom validator and implementation [here <https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/how-to-use-a-custom-user-name-and-password-validator>].

[insert code]

### Service Error Handler (Code that Handles Exceptions as Faults)



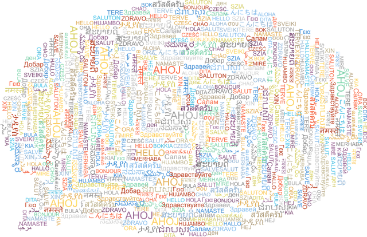
The service-wide error class *ServiceErrorHandler* handles all exceptions that are thrown from the service. The *HelloAuthenticateService* class inherits from this class and throws exceptions directly so that this class so that it can handle the exceptions. Those exceptions will be sent to clients in the Detail property of the enclosing *FaultException<T>* object. On the client side, if the *FaultContract* attribute (example: *[FaultContract(typeof(GenericFault))])* is specified in the interface definition, a *FaultException* for that exception can be differentiated from others in try-catch statements. The *ServiceErrorHandler* class implements two required interfaces: *IErrorHandler* and *IServiceBehavior* that make the service-wide error handling possible.

If you are unfamiliar with a Service Error Handler, I created a demo project and blog article [link <https://portfolio.katiegirl.net/2019/11/11/simple-service-error-handler-a-demo-of-a-wcf-self-hosted-service-client-tester-windows-form-application-exchanging-error-messages/>] that explains the design and implementation of communicating exceptions on a service back to the client with WCF faults.

The class code implementation of the Error Handler is available on GitHub [here].

[insert partial code]

### Hello Authenticate Service



The *HelloAuthenticateService* class implements and manages the registration, login, logoff, client callback messaging, and greeting services to clients. It is a single instance "Shared Service" model for all client consumers and implements the *IHelloAuthenticateService* interface operational contracts that the client consumers expect from the *SharedLibrary* interface modeling. The class code is available on GitHub [here].

#### Properties

Two properties hold lists of registered and logged in users (*User* model). Since this is a demo, the service simulates all service information in-memory that would otherwise be retrieved from a backend datastore.

[insert code]

#### Constructor

This is a simple constructor that initializes the list upon the service loading a single instance.

[insert code]

#### IHelloAuthenticateService Interface Implementation

The service must implement the *IHelloAuthenticateService* interface operational contracts described and referenced in the *SharedLibrary* project. They are described in detail below.

**Register**

The client sends a request to register themselves with a username and password. First the *custom validator* is called before this method. If they are already registered, the custom validator tries to validate their credentials. If unregistered, the validator allows the program to flow to this method without any error.

[insert code]

**Login**

This logs in a registered user and throws any exceptions as faults to the user (custom Error Handler). First the *custom validator* is called before this method. If they are already registered the custom validator tries to validate their credentials. If unsuccessful, the exception is sent to the user immediately and this Login method is never called.

[insert code]

**LogOff**

This log out a registered user and throws any exceptions as faults to the user (custom Error Handler). First the *custom validator* is called before this method. If they are already registered the custom validator tries to validate their credentials. If unsuccessful, the exception is sent to the user immediately and this LogOff method is never called.

[insert code]

**GreetMe**

The *GreetMe* method returns a custom greeting to an authenticated user or a default general message to a stranger. First the *custom validator* is called before this method. If the user is unregistered, then the custom validator will allow the program to flow to this method. If they are already registered the custom validator tries to validate their credentials. If unsuccessful, the exception is sent to the user immediately and this GreetMe method is never called.

[insert code]

#### Callback Methods

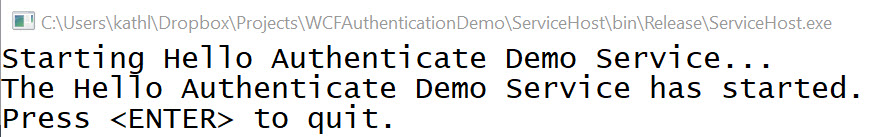
****

**SendMessageToUsers**

The *SendMessageToUsers* method sends a message to all currently logged in users using the *IServiceCallback* callback property of each registered and logged in user (*User* model) to invoke the interface method "SendClientMessage(message)" on the client to receive and process the message. If there is an exception thrown (ex: the client is disconnected), then it automatically logs out the user from the service. The section “Client Tester” in this article discusses the callback service contract implementation on the client end.

[insert code]

## ServiceHost Hosting the Service



The ServiceHost project is a simple console application responsible for starting, managing, and stopping the Authentication service. This demo hosts the service using dual binding *netTcpBinding* protocol for duplex communications so the caller can receive faults and callbacks on a custom port address as described in the application configuration file. The code is available on GitHub [here].

I did first experiment with and utilize the *wsDualHttpBinding* binding protocol but noticed it did not properly communicate faults or exceptions in WCF message security that were thrown in the custom validation class (first step) of a client call to the service. The protocol binding did work for normal duplex communications, but had some unresolved issue in not being able to throw any exception back to the client during the secure message validation phase. I would recommend first try using the *netTcpBinding* protocol and if needed, then creating and implementing an additional error handling interface so it can be shared in both classes. Due to time/effort constraints on my demo project, I decided not to pursue that implementation to try and get it to work 100% with the WS protocol.

### Main Program

The main program is the entry point for the service host application. It creates a *ServiceHost* object of type described in the *ServiceLibrary*, opens the host up for connections, and waits for the user to manually close the service (pressing enter in the console window).

[insert code]

### Application Configuration App.Config

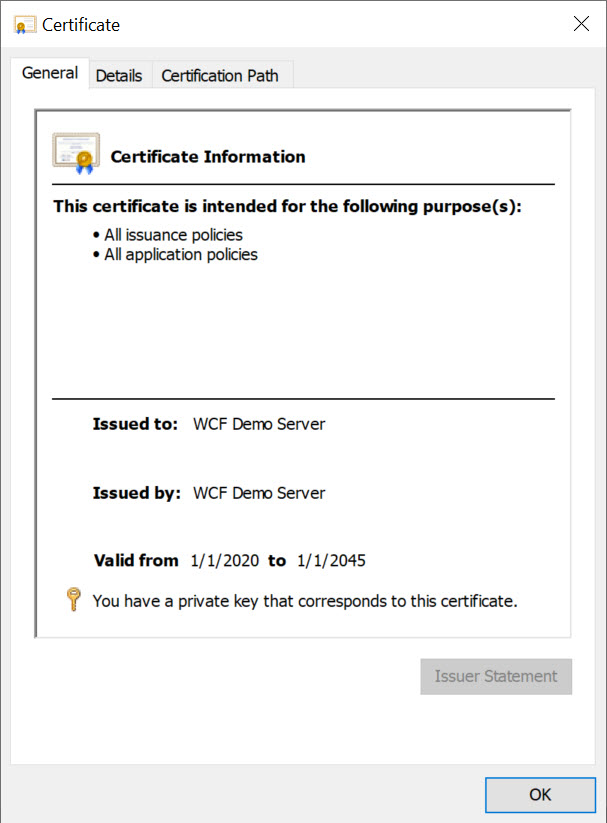
The application configuration simple sets up the service model configuration including the endpoint, interface contract name, protocol, and the base address for this demo. The service host uses this information to correctly host the service. The code is available on GitHub [here].

The application configuration specifies additional characteristics of the communication:

* Service Behaviors
  + Fault Exception Details
  + Extension – Error Handling
  + Service Credentials
    - Client Certificate Requirements
    - Service Certificate Details (*discussed in next section*)
    - Custom User Authentication Instructions
* Service Endpoint
  + Name
  + Binding
  + Configuration Names
  + Contract
  + Base Address
* Binding
  + Security Specification “Message”
  + Buffer/Quota Sizes

[insert code]

## Certificates



This project uses and demos a certificate to establish trust from the client connecting to the service. In production scenarios, the service implementation would require the purchase a trusted certificate that inherits from a root third-party trusted certificate provider (example: VeriSign), of which the client already has installed on their computer certificate store. In development scenarios such as this, certificates are self-generated and installed on a server and/or client machine for testing the service. The client can also embed certificates in their application configuration file for trustworthy WCF. The certificate and generation code for this project is available on GitHub [here].

### Good to Know



In Windows platform, the below file types are used for certificate information such as the SSL and Public Key Infrastructure (X.509) certificate types.

#### CER

The CER file is used to store the X.509 certificate, which is used for SSL certification to verify and identify an Internet server trust and security. The CER file contains information about certificate owner and public key. A CER file can be in binary or encoded with Base-64. The certificate generation script (discussed below) exports the certificate into binary format. However, after it is installed on the server, it needs to be converted/exported into base-64 encoded format using a wizard from the certificate store.

#### PVK

Stands for Private Key. Windows uses PVK files to store private keys for code signing in various Microsoft products. PVK is proprietary format.

#### PFX

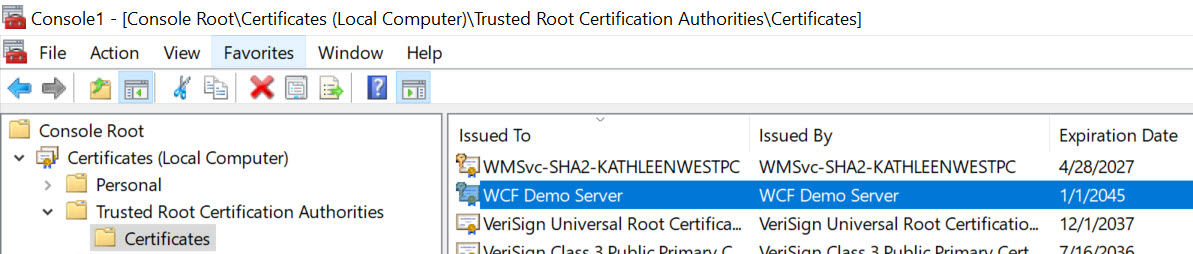
PFX, Personal Exchange Format, is a PKCS12 file. This contains a variety of cryptographic information, such as certificates, root authority certificates, certificate chains and private keys. It’s cryptographically protected with passwords to keep private keys private and preserve the integrity of the root certificates. The PFX file is also used in various Microsoft products, such as IIS. This project created a PFX file in the certificate generation process.

### Certificate Generation

The simple script file CertificateGenerator.bat creates a certificate called “WCF Demo Server” along with additional files. The script file is available on GitHub [here]

[insert code]

### Installation



The specific certificate needs to be installed into the server certificate store in order for this project demo to successful work, otherwise, the certificate configuration can be commented out for untrusted client -> server communications. The blog article [here <https://www.sslsupportdesk.com/how-to-enable-or-disable-all-puposes-of-root-certificates-in-mmc/>] is a good reference on how to install a custom development certificate into the server’s certificate store for testing the project.

### Conversion to Base-64 Encoding

In order to setup the *client* application configuration model of WCF with certificate trust to an Internet server, the option is to either 1.) manually put the server certificate in the local user certificate store and reference this certificate in the *client* application configuration file or 2.) to embed the certificate directly using the base64 encoding string. This project embeds the certificate information directly into the client application configuration file for WCF (*see Client Tester section*). Since this project does not use SVCUTIL to build a proxy, but instead uses the ChannelFactory classes, the base64 encoding string must be obtained from the certificate. The blog article [here <https://bartwullems.blogspot.com/2011/05/wcf-certificate-encodedvalue.html>] discusses how to obtain a Base-64 Encoding of your certificate for insertion into the client’s application configuration file.

### Application Configurations

#### Service

The Service Host uses an application configuration file [insert link] to specify its trusted certificate. Note: This particular demo and project implementation does not require the client be trusted by the service (no client certificate required) and is open to serving anyone.

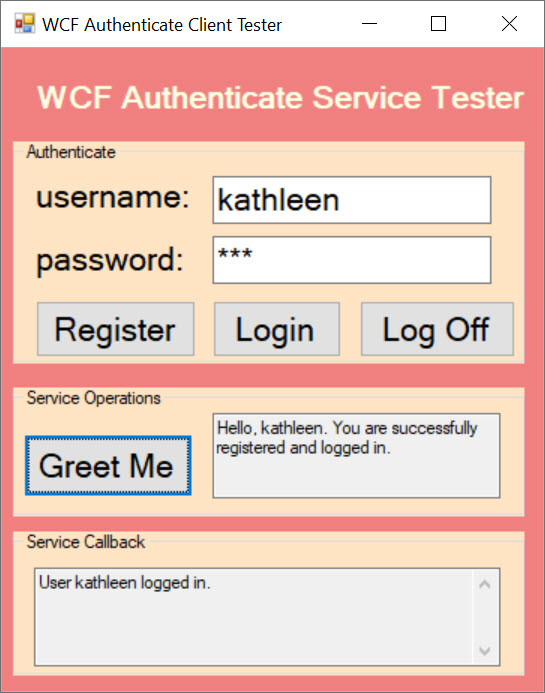
[insert code area here]

#### Client

The ClientTester project uses an application configuration file that embeds a base64 encoded string of the certificate that it will trust for the service configuration. WCF communications verify this is the same certificate stored on the server as specified in the service application configuration file.

[insert code area here]

## Client “Tester to Web API Service” Windows Form Application



The client “tester to service” is a simple windows form application project (*Client*) in the same solution that connects to the “Hello Authenticate 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.

### Features

#### Since the focus of the project demo is on the backend service, the client application was meant to only demonstrate consumption of the Hello Authenticate service and receive errors, so this client has limited scope of features. The authentication and operations are not done on the client, but on the service, in order to demo the service username and password validation and error handle features. Some of the client application features are:

#### Simple Design

The user interface is simplified to do basic demo, testing, verification, and showing results in the GUI.

#### Validation

The client validates the entry of the user data in the text boxes, before sending the request to the Hello Authenticate service. The username and password boxes must not be empty.

#### Error Handling

The client catches specific (DuplicateUserFault) and generic faults (GenericFault) sent from the service before it catches its own exceptions. It displays the fault (error) information to the user as a message box dialog window.

#### Results in the GUI

Successful and unsuccessful operations are communicated to the user with quick message boxes. The results of the GreetMe service request will be displayed in a text box in the same group. Likewise, the entire list of service callbacks (while the client was connected) will be displayed in a list box control in the callbacks group.

### Program Code

The code behind file is for the client tester and manages the application in one file. The code is available on GitHub [here].

#### Fields

Private fields help the client with the following:

* Store their username and password (insecurely)
* Storing their registration and login states
* Storing all service callbacks and greeting messages received

#### Methods

##### Main

This main method is the entry point of the client tester application and does no special setup in this demo other than initialize the form components.

##### Events

Multiple button-click events handle the user requests to register, login, logoff, and get a greeting message. Some events may first validate user text input from the GUI.

**Client to Service Communications Channel Setup**

All of user-driven events communicate with the service by assembling a proxy using a *DuplexChannelFactory* class of type *IHelloAuthenticateService* specifying a configuration name as detailed in the client application configuration file. Client credentials (username and password) must be setup on the channel and added after the default configuration is removed. An example of configuration is shown below.

[insert code]

##### Methods

The only non-event driven method receives a callback message from the service (implements the IServiceCallback interface) and adds/updates the list of callbacks on the main UI thread.

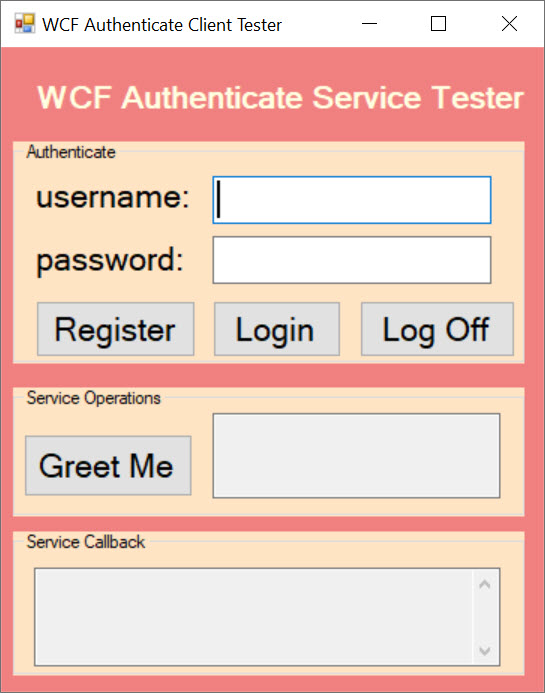
## Demo

A video demonstration of the project is available on YouTube here:

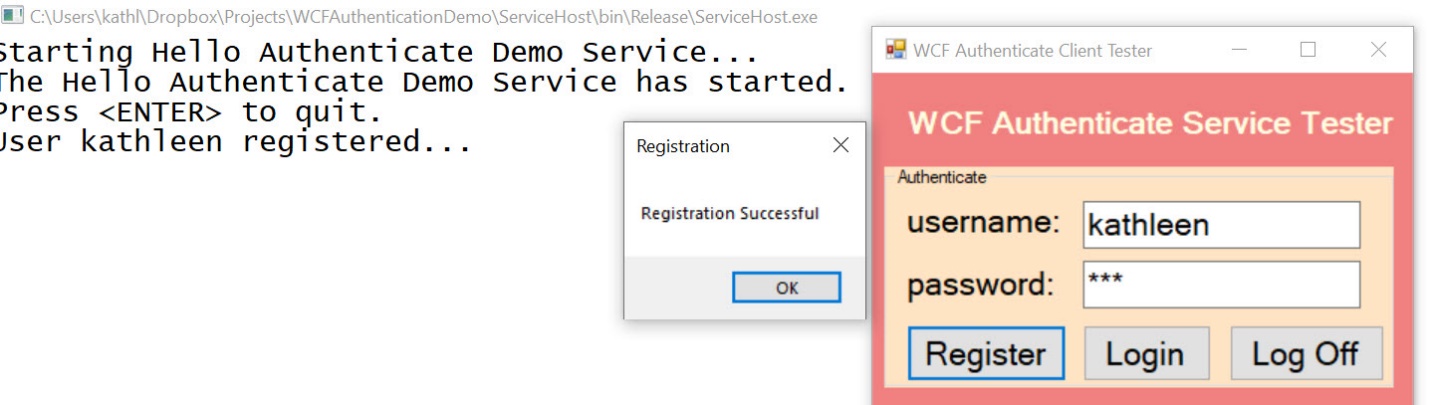
[insert YouTube video]

Screen captures are shown below for various user activities:

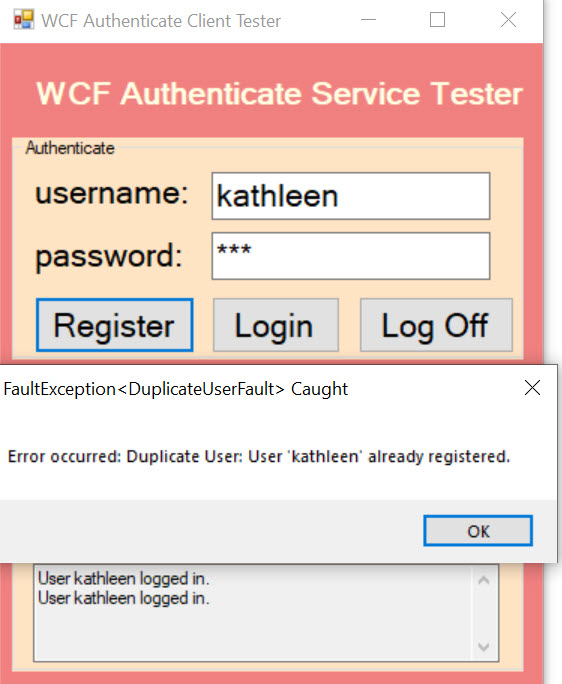
### Startup



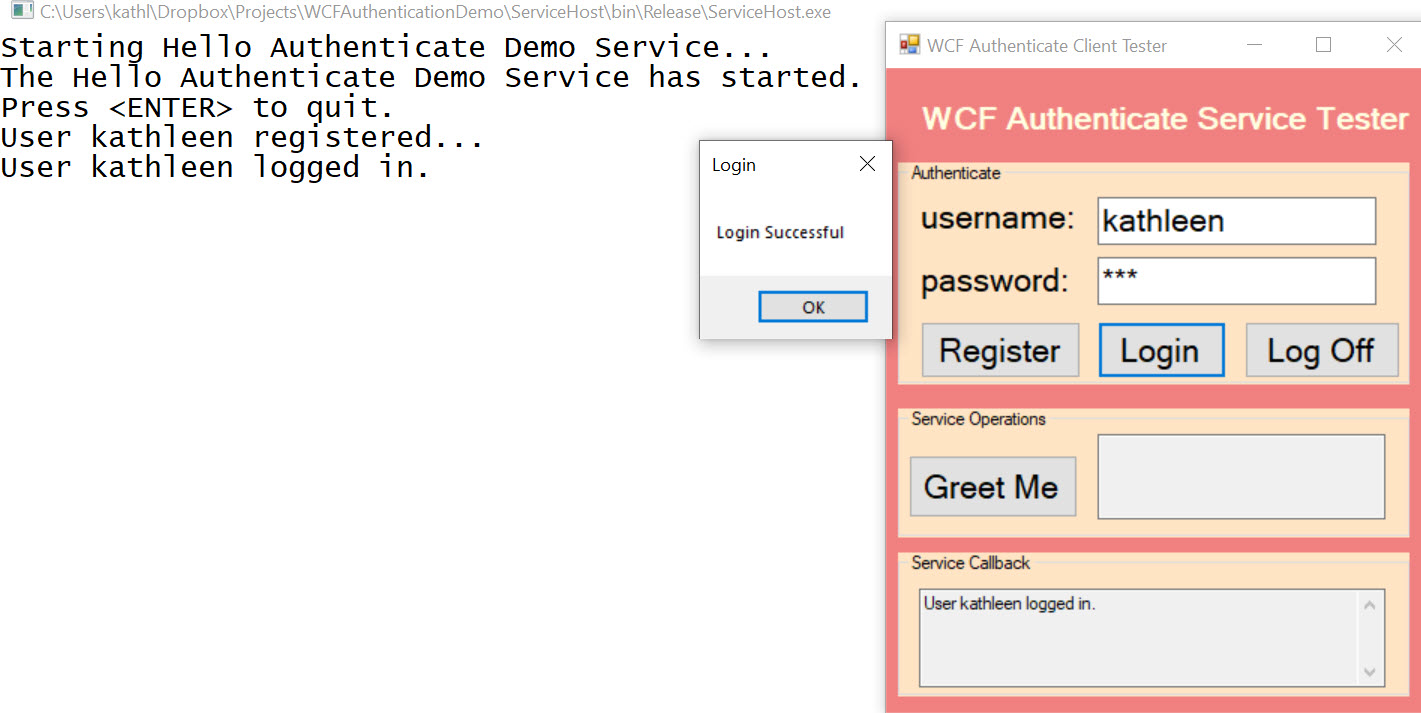
### Registration



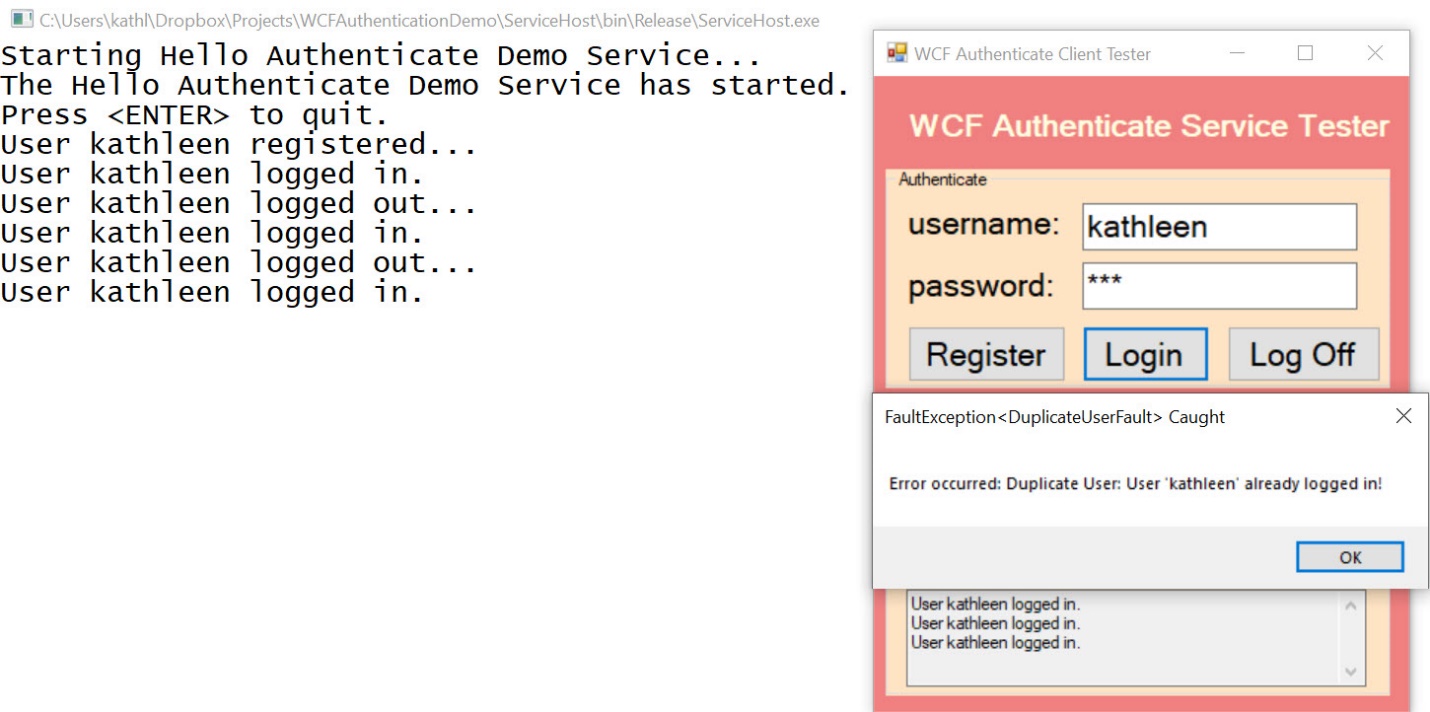
### Registration Error



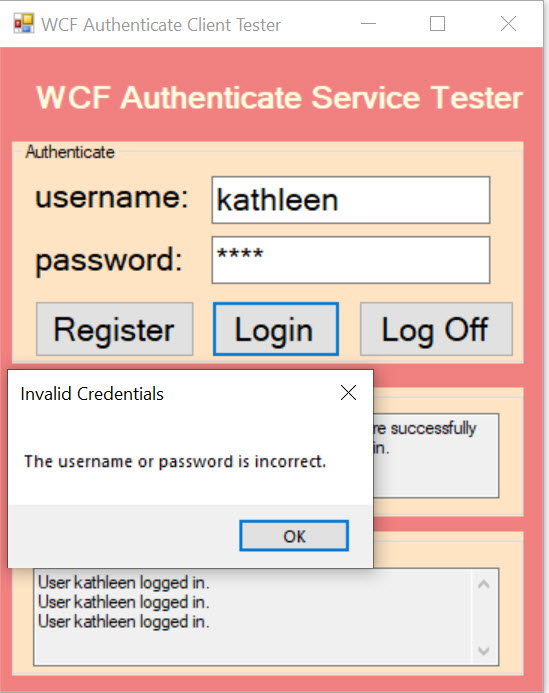
### Login



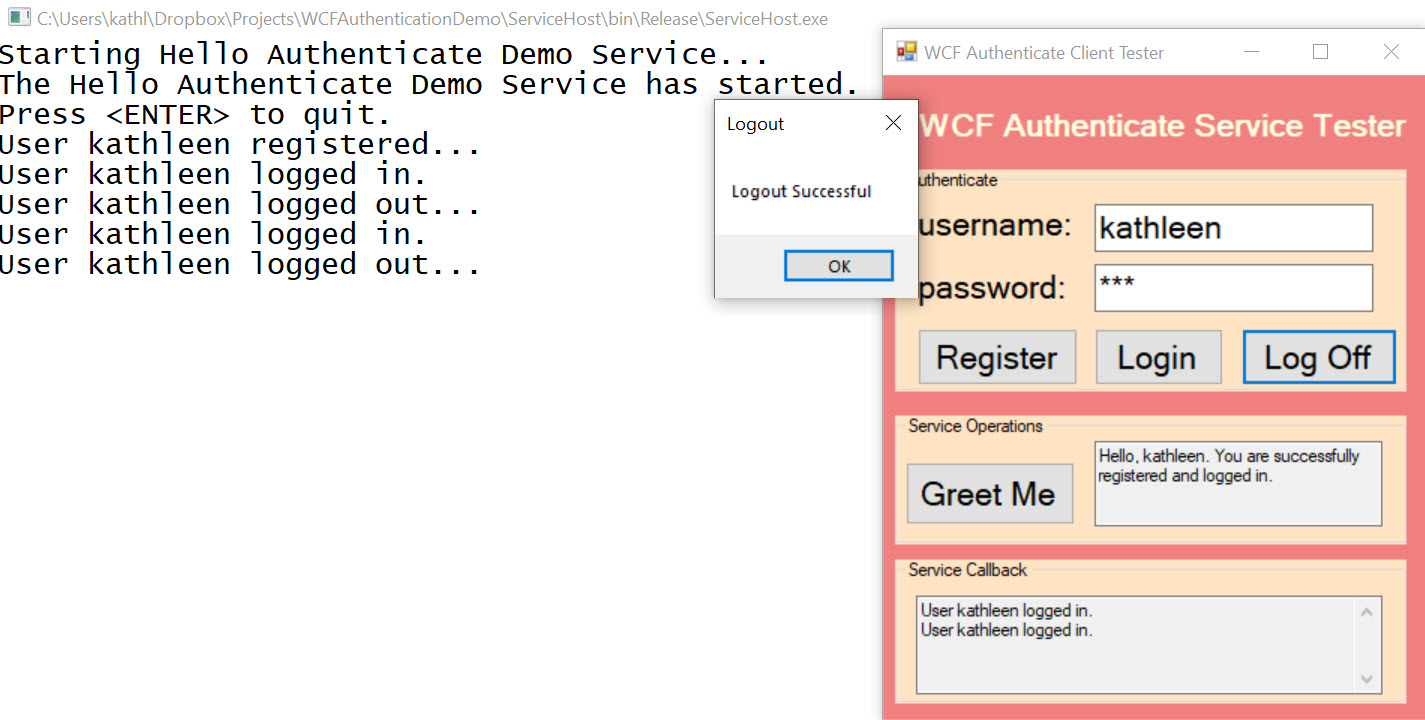
### Duplicate Login



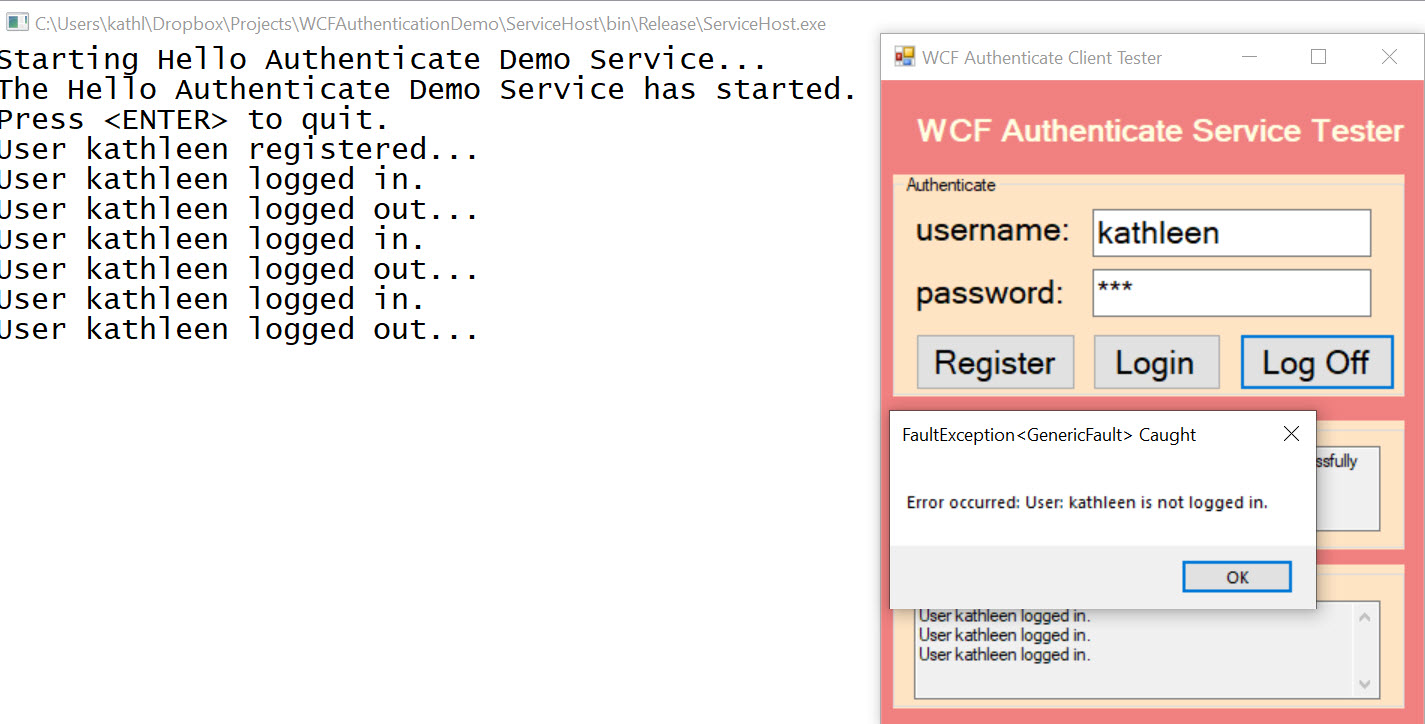
### Invalid Credentials



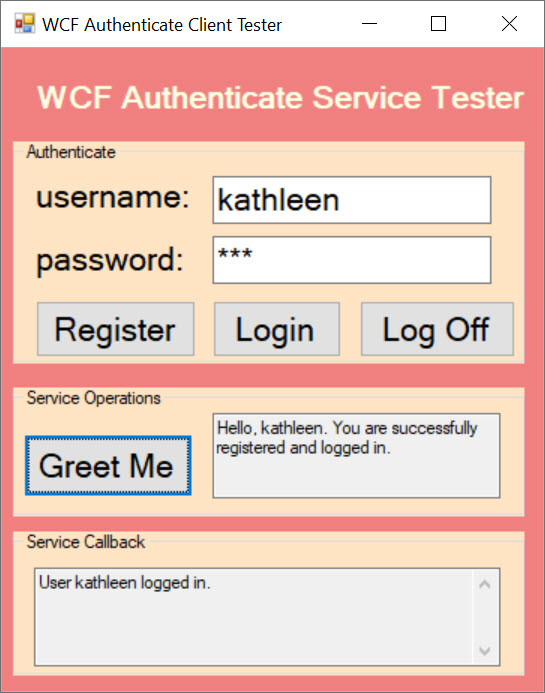
### Logout



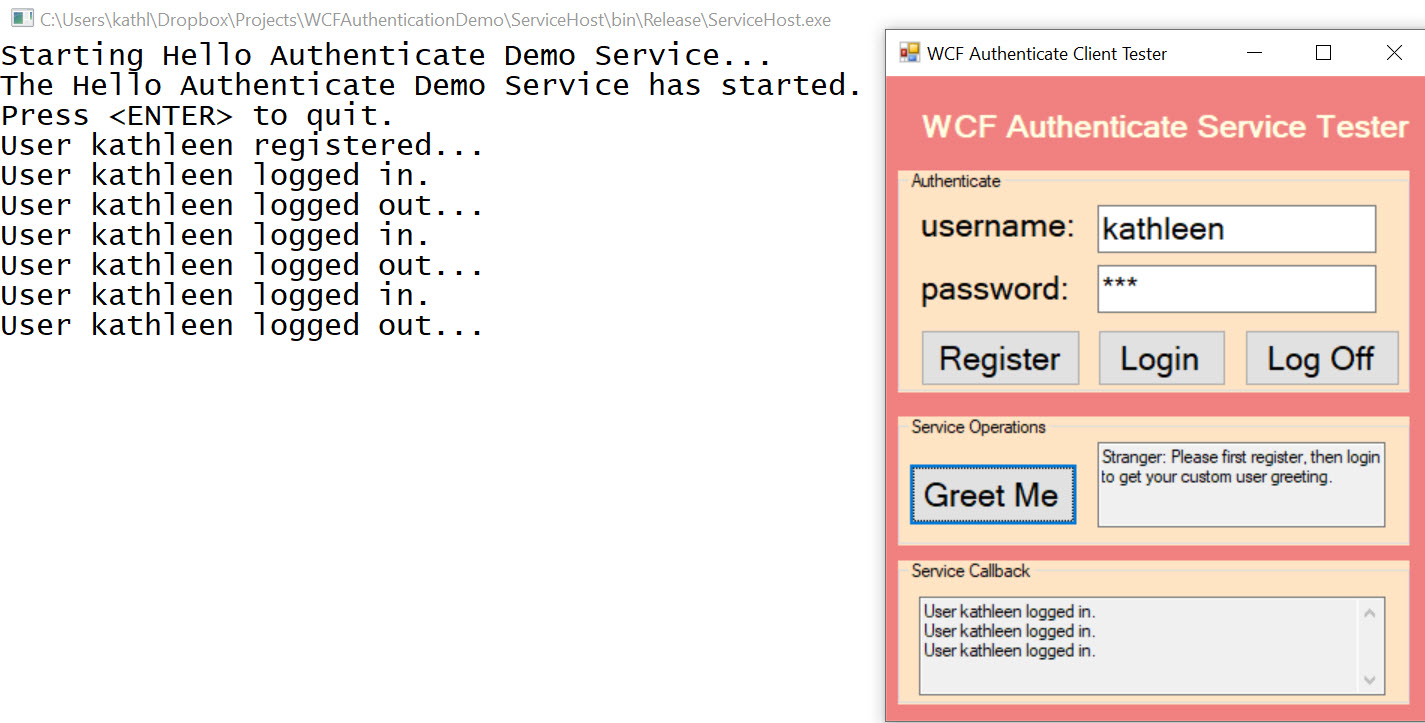
### Logout Error



### Greet Me Logged In



### Greet Me Unregistered



## Code

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