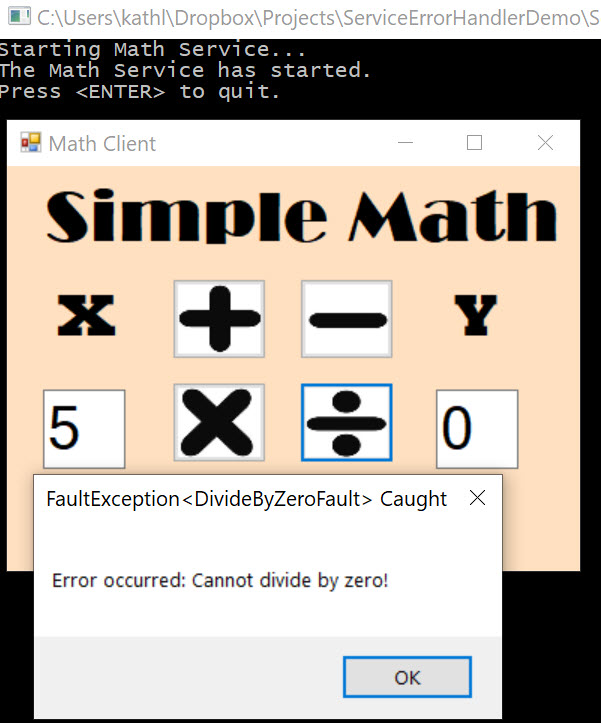
# Concurrency and Instancing in WCF Services – A Demo and Discussion of Service Concurrency and Instancing with Multi-threading Clients

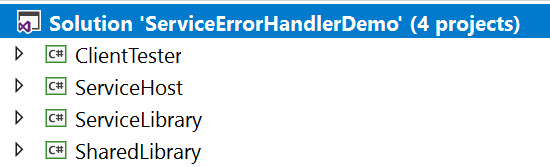
## 

## About

This project presents a simple Math Service and Client Application demonstration that implements error handling on the service and communications of its errors to the client. The Math Service is a self-hosted (service host) WCF application launched and managed with a simple console interface. The client “tester” has a simplified GUI user interface to quickly demo and test the service (Windows Form Application). The GUI has features to add, subtract, multiply, and divide two numbers (integers). The GUI does basic data validation then sends a request to the math service. After it receives the results from the math service, it displays the calculation result or any received error messages (faults) to the user. The objective of this project was not to demo a calculator application but the handling of errors on a service.



## Architecture



The demo project consists of these component topics:

* Shared Class Library Project “SharedLibrary”
  + IMathService (Interface for Service)
  + DivideByZeroFault (Class describing a WCF Exception as a Fault)
  + GenericFault (Class describing a WCF Exception as a Fault)
* Service Class Library Project “ServiceLibrary”
  + MathService (Code that Implements the Service Interface)
  + ServiceErrorHandler (Code that Implements Service Error Handling)
  + App.config (Configuration Reference for Service Host)
  + *Reference to the Shared Class Library*
* WCF Service (Host) Application Project “ServiceHost”
  + Program (Starts, Manages, Stops the Service)
  + App.config (Configuration for Service Host)
  + *Reference to the ServiceLibrary*
* 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 math services (ex: Add) and is referenced by both the client and service host projects. Since exceptions (errors) are really faults in WCF services, two custom error (fault) class models are defined in the shared library so that the client can understand and process the faults (errors).

The *ServiceLibrary* implements the math service and contracts as defined in the *SharedLibrary*. The *ServiceHost* is a simple console application that is responsible for starting the math service, hosting, and managing the service (self-hosted). The service is also implemented as “PerCall” where each service call spins up a separate instance to respond to the client requests.

A client “tester” windows form application tests the service and provides output to the user in a simple GUI. Errors are transmitted by the service to the client and because of the *SharedLibrary* model, it can understand the objects and alert the errors to the user with dialog windows.

## 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].

### IMathService (Interface for Service)

The ServiceContract for the simple Math service contains the following operation contracts.

* int Add(int a, int b);
  + Adds two values, returns the calculation to the caller
* int Subtract(int a, int b);
  + Subtracts two values, returns the calculation to the caller
* int Multiply(int a, int b);
  + Multiplies two values, returns the calculation to the caller
* int Divide(int a, int b);
  + Divides two values, returns the calculation to the caller

#### Discussion

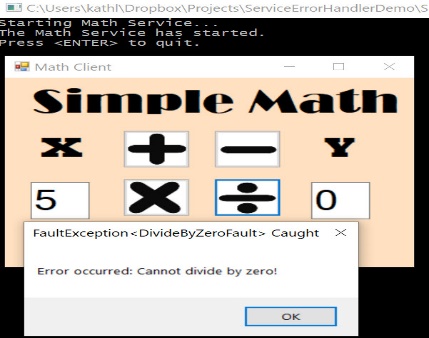
Each of the interfaces are labeled with the attributes of [FaultContract(typeof(GenericFault))] so that any exception caught on the service can be packaged as a fault object and sent to the caller with the detailed information regarding that exception. Furthermore, with the Divide operational contract, another fault contract [FaultContract(typeof(DivideByZeroFault))] is specified so that it can catch and return “Divide By Zero” exceptions in a detailed fault model instead of the generic one.

The code is available on GitHub [here].

[insert code here]

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

The *DivideByZero* class represents a fault (exception) object in the application. Specifically, it is meant to model a DivideByZero Exception that was caught and handled on the service when the math caller, client application, sends a request to divide a number by zero. 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.

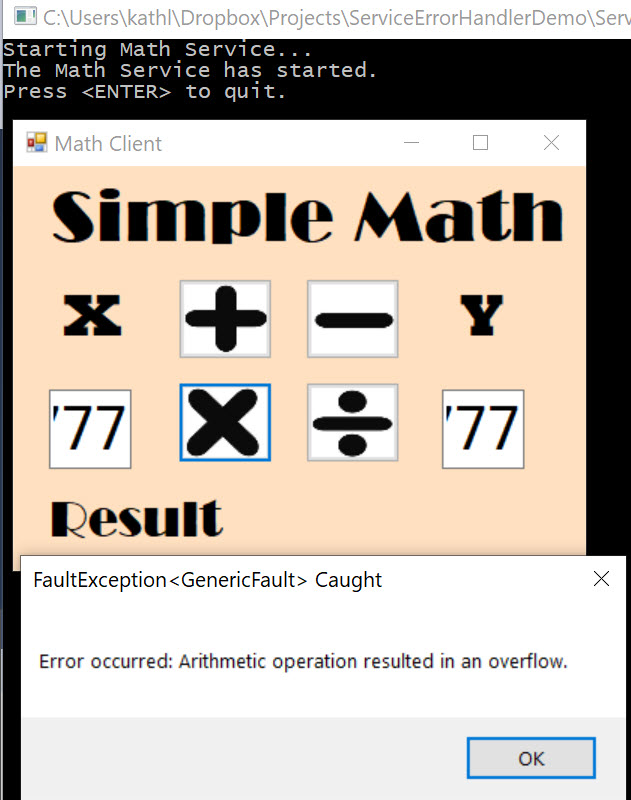


The code is available on GitHub [here].

[insert code here]

### 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 caller requested a math calculation. Per design, the more specific exception “DivideByZero” is caught by the service and handled as a “DivideByZero” fault object rather than the generic fault. 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 code is available on GitHub [here].

[insert code here]

## 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].

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

The service implementation code has service behaviors configured to allow for multiple instances were each service call connects to a new service instance. The service library references a shared class library called “*SharedLibrary*” that contains the interface contract definition for the service. This code implements the contracts for the IMathService interface. The MathService class also inherits from the *ServiceErrorHandler* class (discussed in the next section) to allow generic exceptions on the service to be handled and communicated as faults to the client callers. The code is available on GitHub [here].

#### Service Interface Implementation Methods

The service implementations are described below.

##### Add

Performs the add mathematical calculation and returns the result to the user. Catches all exceptions and throws them so the *ServiceErrorHandler* can package them as faults and return to the caller.

[insert code here]

##### Subtract

Performs the subtract mathematical calculation and returns the result to the user. Catches all exceptions and throws them so the *ServiceErrorHandler* can package them as faults and return to the caller.

[insert code here]

##### Multiply

Performs the multiple mathematical calculation and returns the result to the user. Catches all exceptions and throws them so the *ServiceErrorHandler* can package them as faults and return to the caller.

[insert code here]

##### Divide

Performs the divide mathematical calculation and returns the result to the user. First catches “Divide by Zero” exceptions and throws them as new fault objects (*DivideByZeroFault*). Then catches all other exceptions and throws them so the *ServiceErrorHandler* can package them as faults and return to the caller.

[insert code here]

### ServiceErrorHandler (Code that Handles Exceptions as Faults)

The service-wide error class *ServiceErrorHandler* handles all exceptions that are thrown from the service. The MathService 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 code is available on GitHub [here].

The ServiceErrorHandler class implements two required interfaces: *IErrorHandler* and *IServiceBehavior* that make the service-wide error handling possible.

#### IErrorHandler Implementation

##### Handle Error

This interface method enables error-related processing and returns a value that indicates whether the dispatcher aborts the session and the instance context in certain cases. This can and will shut down the service session and communications to a client under certain circumstances of errors.

public bool HandleError(Exception error)

This method functionality was not implemented in this simple demo project.

##### Provide Fault

This interface method enables the creation of a custom FaultException<GenericFault> that is returned to the client after a service method throws an exception. Communications protocol have to be duplex for the client to receive the fault. If the service already created and threw a FaultException object (example: DivideByZero), this method will not do any additional processing. However, all exceptions that were not converted to FaultExceptions and thrown by the service, will now be packaged into FaultException<GenericFault> objects.

[insert code here]

#### IServiceBehavior Implementation

##### Add Binding Parameters

Provides the ability to pass custom data to binding elements to support the contract implementation.

public void AddBindingParameters(ServiceDescription description, ServiceHostBase serviceHostBase,

Collection<ServiceEndpoint> endpoints, BindingParameterCollection parameters)

This method functionality was not implemented in this simple demo project.

##### Apply Dispatch Behavior

This interface method provides the ability to change run-time property values or insert custom extension objects such as error handlers, message or parameter interceptors, security extensions, and other custom extension objects. This project uses this method to apply a custom error handler (*IErrorHandler*) object to every open Channel used by the client to connect to service. Basically, when the client calls the service, it spins up a new instance and attaches the error handler object to the communications channel that it is using.

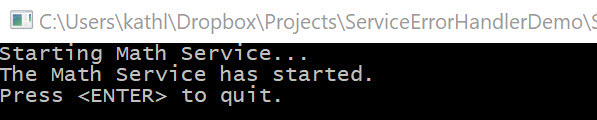
This behavior can also be placed in a configuration file instead of programmatically attached as shown below. See the *Application Configuration* section for the code example (which is currently commented out) of using the configuration approach to attaching an error handler to the service behavior. However, it is a good practice to keep the error handler implemented programmatically versus in the application configuration file.

[insert code here]

### Application Configuration App.Config

The application configuration simply sets up the service model configuration including the endpoint, interface contract name, 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].

## Math Service Host (ServiceHost)



The *ServiceHost* project is a simple console application responsible for starting, managing, and stopping the Math service. This demo hosts the service using dual binding “wsDualHttpBinding” protocol for duplex binding so the caller can receive faults 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 *ServiceLibrary*, opens the host up for connections, and waits for the user to manually close the service (pressing enter in the console window).

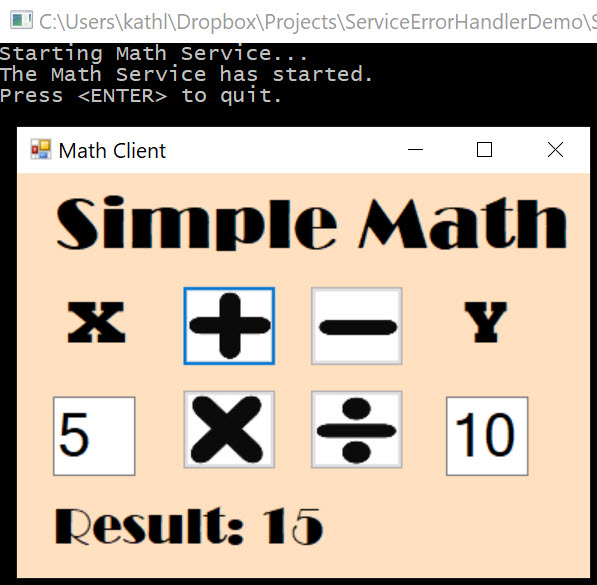
[insert code here]

### 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 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 “Math 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 Service 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 Math Message service and receive errors, so this client has limited scope of features. The math calculations are not done on the client, but on the service, in order to demo the error handle features. Some of the client application features are:

#### Simple UI Design

The user interface is simplified to do basic math calculations with the click of a button representing the operation.

#### Validation

The client validates the parsing of the user data entry in the text boxes, before sending the request to the Math service to process the calculation. The input is parsed to two integer data types.

#### Error Handling

The client catches specific (*DivideByZeroFault*) and generic faults (*GenericFaults*) 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.

### 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:

* Numbers X & Y to perform the math calculation
* The result of the math calculation delivered by the service

[insert code here]

#### Constructors

No customization was implemented for this demo project.

[insert code here]

#### Events

##### Add Button Click

This processes the user request to add two numbers. First it calls a helper method to parse and validate the user data entry in the two text boxes before calling the service. The service is called and the method catches the faults from the service, its own exceptions, and displays those errors to the user. If there are no faults, the results are displayed on the main UI.

[insert code here]

##### Subtract Button Click

This processes the user request to subtract two numbers. First it calls a helper method to parse and validate the user data entry in the two text boxes before calling the service. The service is called and the method catches the faults from the service, its own exceptions, and displays those errors to the user. If there are no faults, the results are displayed on the main UI.

[insert code here]

##### Multiply Button Click

This processes the user request to multiply two numbers. First it calls a helper method to parse and validate the user data entry in the two text boxes before calling the service. The service is called and the method catches the faults from the service, its own exceptions, and displays those errors to the user. If there are no faults, the results are displayed on the main UI.

[insert code here]

##### Divide Button Click

This processes the user request to divide two numbers. First it calls a helper method to parse and validate the user data entry in the two text boxes before calling the service. The service is called and the method catches the faults from the service, its own exceptions, and displays those errors to the user. If there are no faults, the results are displayed on the main UI.

[insert code here]

#### Methods

##### ValidateParseDataEntry

Validates the user input and parses the data to the integer type.

[insert code here]

##### UpdateResult

Updates the Math calculation result on the UI.

[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. Note: The protocol needs to be dual binding so that the client can receive the faults from the service. 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].