# Building an SOA application with WCF

## Introduction

The solutions are intended to demonstrate how a fully functioning application (in this case a fictitious bailiff application) could be implemented using an SOA (Service-Oriented Architecture) / WCF (Windows Communication Foundation) architecture.

## Assumptions

It is assumed that the reader is familiar with the concepts underpinning SOA and WCF and therefore is familiar with the concepts of interfaces and service contracts that are used to define the functionality that they expose. It is also assumed that the reader is familiar with .NET and C#.

## Top level architecture

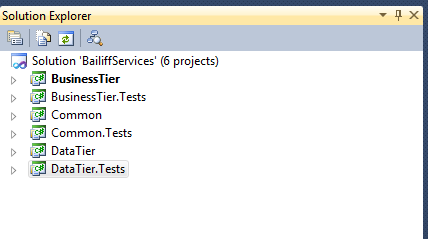
The entire architecture consists of three separate .NET solutions.

* **BailiffServices** – This solution defines and implements the WCF services that will be consumed by the client applications and in essence is the backbone of the architecture. It provides the business rules and logic that will be invoked by the client applications. It provides WCF services that can be consumed by the client application.
* **BailiffHost** – This solution provides a simple hosting environment for the **BailiffServices**. It is a self-hosting console application that is capable of hosting WCF services. In a production environment I would expect this to be replaced with IIS (Internet Information Services).
* **BailiffClient** – This solution provides a simple WCF client application that invokes the services provided by **BailiffServices** and therefore demonstrates how to invoke a WCF service. This solution is only intended to provide a demonstration. The clients of the **BailiffServices** in a production environment will probably be Windows forms, web pages etc.

## BailiffServices

As already mentioned, this application provides the key business services (which in this case are bailiff services) that will be consumed by the client applications. It does this by exposing interfaces which implement clearly defined service contracts. The solution consists of 6 projects.

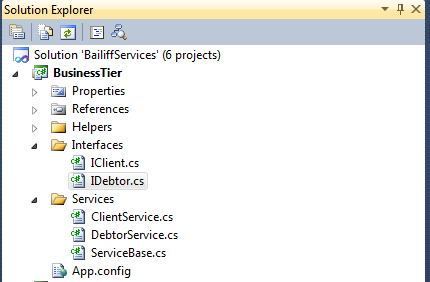
* **BusinessTier** – This is where the interfaces and service contracts are defined.
* **BusinessTierTests** – This is a set of unit tests that exercise the code defined in **BusinessTier**.
* **Common** – Provides functionality that can be used by all other parts of the solution such as Helpers, Attributes, Enums etc. N.B. This project must NOT contain any references to other projects or else you will end up with a circular reference.
* **Common.Tests** – This is a set of unit tests that exercise the code defined in **Common**.
* **DataTier** – This provides the data access functions that retrieve the data from the SQL database. At the time of writing this was only a small subset of the entire database, but was sufficient for the purposes of this prototype. All database interactions are made via stored procedures. There are NO SQL queries anywhere in the application.
* **DataTier.Tests** - This is a set of unit tests that exercise the code defined in **DataTier**.



## How it works

The best way to explain how the application works is to take one of the example methods and explain it from start to finish. I will use the *GetDebtorInfo*() function on the *IDebtor* interface for the purposes of explanation.

Select the **BusinessTier** project and expand the **Interfaces** and **Services** folders.



## Interfaces

Each interface is defined in its own separate file. The *IDebtor* interface is defined within the *IDebtor.cs* file.

[ServiceContract]

public interface IDebtor

{

[OperationContract, FaultContract(typeof(UnexpectedServiceFault))]

DebtorEntity GetDebtorInfo(int debtorId);

}

Notice how the interface definition is decorated with the *[ServiceContract]* attribute. This indicates that the definition forms part of a service contract i.e. a WCF service contract. There is only one method defined so we will use the *GetDebtorInfo()* function for the purposes of our explanation.

## Services

The implementations of all the interfaces are contained within the **Services** folder. The implementation for *IDebtor.GetDebtorInfo()* can be found in *DebtorServices.cs*

All service implementations are inherited from the *ServiceBase* class and implement the associated interface which in this case is *IDebtor*.

public class DebtorService : ServiceBase, IDebtor{}

All debtor related functionality will eventually be implemented within this class.

The service function uses the *DatabaseManager* class to retrieve the required data from the database.

DatabaseManager dm = new DatabaseManager();

return dm.GetDebtorInfo(debtorId);

## Service exception handling

WCF services must NOT use .NET exception types in their exception handling. As the WCF client can be any platform or technology, the exception information that is passed back to the client must be a **SoapException** or similar. This guarantees that any client can consume the exceptions that are thrown by the WCF service.

Regular .NET exception types can be used everywhere else throughout the application, just not by the services that are exposed to the client. The application throws all exceptions up the call stack then hands a *FaultException* to the client.

throw new FaultException<UnexpectedServiceFault>(

new UnexpectedServiceFault { ErrorMessage = ex.Message }, new FaultReason(FaultReasons.GetDebtorInfo));

The **FaultReason** text messages are contained within the *FaultReasons* class within *Common.Enums*.

public class FaultReasons

{

public const string GetDebtorInfo = "Fault occurred determining debtor info in GetDebtorInfo()";

public const string GetClientInfo = "Fault occurred determining client info in GetClientInfo()";

}

## 

## Retrieving data from the database

As mentioned previously the service implementation for *GetDebtorInfo()* invokes the *DatabaseManager* to retrieve the required data from the database.

DatabaseManager dm = new DatabaseManager();

return dm.GetDebtorInfo(debtorId);

As with all Helper classes the *DatabaseManager* class is derived from the *ManagerBase* class.

The *DatabaseManager* class is a wrapper class to the data classes that are contained within the **DataTier** assembly. Therefore to retrieve debtor information the *DataManager* class invokes functionality defined within the *DebtorData* class that is contained within the **DataTier** assembly.

The *DebtorData* class as with all data classes is derived from the *DataTierBase* class.

public class DebtorData : DataTierBase{}

All database functions invoke stored procedures. They do NOT pass SQL statements for execution.

All database functions return a regular .NET entity object rather than a SQL specific object such as a *SqlDataReader*. This is to prevent the **BusinessTier** assembly from having to have any knowledge of the underlying data. Therefore the data that is returned from the database is mapped to a .NET class which is then returned.

**All database functions should return business objects.**

This mapping between the *SqlDataReader* and the debtor entity class is achieved using a combination of .NET **Reflection** and by decorating the class with the *DataField* attribute which indicates what the underlying column name is.

Here is the code that maps the *SqlDataReader* to the .NET debtor entity class.

result = ReflectPropertyInfo.ReflectType<DebtorEntity>(reader);

This is a custom function that implements **Reflection** to achieve the mapping.

And here is how the *DebtorEntity* class must be decorated to achieve this mapping.

[DataContract]

public class DebtorEntity

{

[DataMember, DataField("DF\_REFNO")]

public string Refno { get; set; }

[DataMember, DataField("DF\_CLIENT")]

public string Client { get; set; }

[DataMember, DataField("DF\_CREF")]

public string CRef { get; set; }

[DataMember, DataField("DF\_LIAB\_ORDERID")]

public string LiabilityOrderId { get; set; }

[DataMember, DataField("DF\_COMPANY")]

public string Company { get; set; }

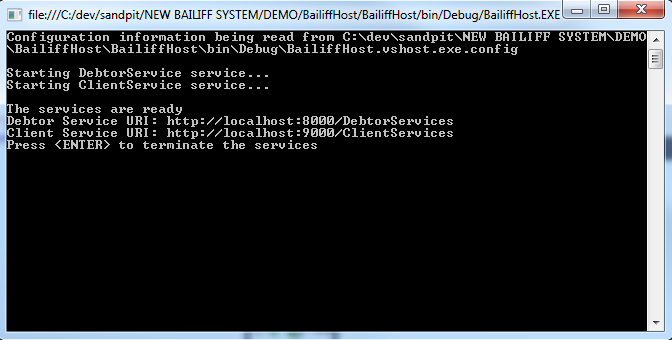
}

The entity classes that are returned by the **DataTier** are defined within *Common.Entities*.

## Running the application

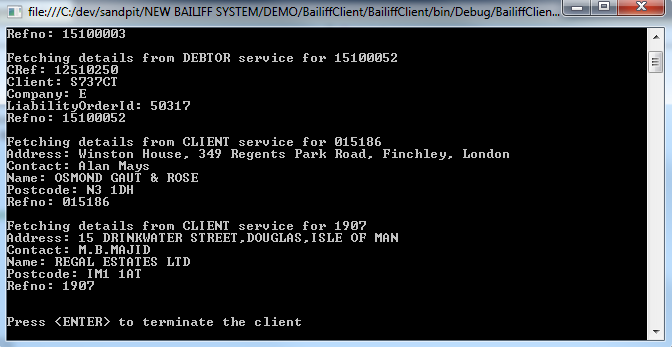
In order to run the application do the following.

* Make sure you are running the latest version of the service.
  + Open the **BailiffService** solution in Visual Studio and perform a *Rebuild*
  + Copy the files **BusinessTier.dll, Common.dll and DataTier.dll** from the BusinessTier\bin\Debug folder and copy them into the BailiffHost\Refences folder
* Start the service host.
  + Open the **BailiffHost** solution in Visual Studio
  + Press F5 to run the application. You should see a console screen as in the following screenshot.



The WCF services are now running and ready to be consumed by the client application.

* Run the client application.
  + Open the **BailiffClient** solution in Visual Studio
  + Ensure you have the latest service references.
    - Expand the **Service References** folder
    - Right click on each of the references and select **Update Service Reference**
  + Press F5 to run the application. You should see a console screen as in the following screenshot.



As you can see in the screenshot the content from the bailiff tables is output to the screen. This was achieved by invoking the WCF service.