WcfNQueueSMEx2 Visual Studio Solution Structure

George Stevens

|  |  |  |
| --- | --- | --- |
| **version** | **date** | **author** |
| 0.1 | 5/18/2015 | George Stevens |
| 0.5 | 6/6/15 | George Stevens |
| 0.6 | 6/12/15 | George Stevens |
| 0.7 | 6/26/15 | George Stevens |

# Introduction

The structure of the code in WcfNQueueSMEx2 is my design. I have used portions of the IDesign architecture in it as follows: The component roles (client, managers, engines, resource accessors (aka data accessors)) and their relationships, the layers and closed architecture, and the concept of a subsystem. This approach works well in practice to produce systems that are both cost effective and high quality. It also helps achieve the overall goals of this project, stated here: [Introducing the “SO Apps” Series of Blog Articles and Code Examples](https://dotnetsilverlightprism.wordpress.com/2015/05/12/introducing-the-so-apps-series-of-blog-articles-and-code-examples/). The blog concerning WcfNQueueSMEx2 is [SO Apps 2, WcfNQueueSMEx2 – A System of Collaborating Microservices](https://dotnetsilverlightprism.wordpress.com/2015/06/12/so-apps-2-wcfnqueuesmex2-a-system-of-collaborating-microservices/). It contains a link to the example code.

For more information about [IDesign](http://www.idesign.net) please click the link. For information about the IDesign Method™ and its architecture please see the sections called “What is the IDesign Method™” and “Design Method™ Notations Example” in the document “[The IDesign Method: Capturing Critical Design Aspects](http://www.idesign.net/Downloads/GetDownload/1902)”. You can learn how to do such architectural designs yourself at these classes: [IDesign Architects Master Class](http://www.idesign.net/Training/Architect-Master-Class) and [IDesign Architecture Clinic](http://www.idesign.net/Clinics/Architecture-Clinic). By taking a class you are also able to participate in the valuable IDesign Alumni private Forum, from which the Visual Studio Solution structure presented below was obtained.

The structure of a Visual Studio solution can act to greatly enhance the effectiveness of working with a service oriented code structure. Thus, I am using a solution structure of folders, namespaces, and projects described by Monty Montgomery in the IDesign Alumni thread “Namespace convention and namespace best practices”. Monty is a Master Architect at IDesign. This structure fulfills the following goals that enable Service Oriented Apps:

* Use the IDesign practice of having a single developer implement a single service in a single assembly. This speeds implementation time by not having to take time required for multiple developers to coordinate their work on a common service.
* Facilitating the composition of larger services from smaller services, and scalability. This is required for typical SOA scenarios as follows:
  + Service Oriented Apps are typically composed of a number of services, each of which may be hosted wherever -- In the data center, in the cloud, on several different virtual machines, with at times multiple instances of the same service alive at once and possibly running in different hosts when scaled out. Services or components of a relatively smaller granularity (i.e. a smallish number of service operations) make such scenarios work better than services having a quite large granularity.
* Promote reuse by presenting a broad view of the service oriented solution in source control and the Visual Studio Solution Explorer. This minimizes the depth of solution folder hierarchies allowing developers to easily explore the available components by scrolling through them.
* Partition assemblies containing service and data contracts so as to produce the smallest possible client side foot print.
* Allow easy finding of components in the Visual Studio Solution when looking at IDesign architecture diagrams. Examples of these diagrams are shown in “[The IDesign Method: Capturing Critical Design Aspects](http://www.idesign.net/Downloads/GetDownload/1902)”.

The above goals are realized in the structure of the WcfNQueueSMEx2 Visual Studio Solution presented later.

Here is Monty’s basic Namespace convention, with Product being optional:

**<Company>.<Concept>.[<Product>].<Subsystem>**

For example, in the WcfNQueueSMEx2 solution there is a namespace and project named **GS.Manager.DataFeed**. This is where the code for DataFeedManager component (a WCF service) resides.

* Company is **GS** (abbreviation for my name for lack of anything better in this code example).
* Concept is **Manager**. Please see the following definitions for more about Managers.
* Product is omitted since there is only one product in this company at present. However it could be something like Mobile if there were a mobile version. Or NextGen for the next major release. Etc.
* Subsystem is **DataFeed**.

Before I list the folders, namespaces, and projects in the WcfNQueueSMEx2 solution some explanation is needed of the key terms used in this document. **Such knowledge will also be invaluable when inspecting and understanding the code of WcfNQueueSMEx2 since it often provides insights into the intent or “reasons why” things are done as they are**.

# Definition of Terms and the Component Roles

**Component Roles Overview** – This architecture style decomposes a system into components which have several standard roles. Using a few standard component roles makes the architecture simple in that the same roles are repeated over and over, but have unique specific purposes and contexts each time a role is used. This aids in quickly gaining an understanding of the big picture view of the software system’s structure and responsibilities of its components. And, each component serves to encapsulate specific kinds of volatility by design, based both on the role and also on the work a specific component is designed to do in performing the intended role. Decoupling the structure of the code from the volatities (changeability) of the functional requirements and domain model greatly decreases the amount of refactoring of existing code that is necessary to fit in new functionality. This can significantly reduce Time-To-Market and Total-Cost-of-Ownership, thus more closely aligning IT/Product Development with business goals.

Here are the component roles, listed in the order they appear in the layered architecture that is used:

* Client
* Manager
* Engine
* Data Accessor/Resource Accessor
* Database or other such physical Resource

The roles of Manager, Engine, Data Accessor/Resource Accessor are each implemented as a service when using WCF on .NET platforms. These components can be considered to be microservices at the lowest level of granularity.

**Layered Closed Architecture** – This is a layered, closed architecture modeled after the IDesign architecture style. The layers are, from the top down:

* Presentation Layer -- Clients
* Business Logic Layer – Managers and Engines
* Data Access/Resource Access Layer – Data Accessors and Resource Accessors
* Data/Resource Layer – Databases, files, ORMs, caches, images, external APIs, etc.

Succinctly, calls between components can only be done downward, never upward. Engines cannot call Managers. Data Accessors can neither call Engines nor Managers. This acts to simplify the interaction between the roles and prevents a tangled crisscross of call chain sequences that are time consuming to work with and add little or no value. Please see this link for more, plus the links it contains – [Stack Overflow -- What is meant by open and closed layered architectures?](http://stackoverflow.com/questions/904257/what-is-meant-by-open-and-closed-layered-architectures).

For more information about these roles and this architecture please see the document noted above, “[The IDesign Method: Capturing Critical Design Aspects](http://www.idesign.net/Downloads/GetDownload/1902)”.

Finally, both Engines and Data Accessors/Resource Accessors are WCF services in .NET implementations that run “In Process” to the Manager that uses them via the NetPipeBinding. Please see pages 71 – 74 of Programming WCF Services, 3rd Edition, by Juval Lowy for examples of how to do this using ServiceModelEx’s InProcFactory<T> class. Also see my blog article associated with the WcfNQueueSMEx2 code example for the advantages of running Engines and Data Accessors/Resource Accessors in process to Managers.

For a great conceptual diagram of how all the above generally fits together please see “Figure 1 – Modern Software Architecture” in Monty Montgomery’s article “[From the Field: Escaping Appland](http://iasaglobal.org/iasa/from-the-field-escaping-appland/)”. **Please do not pass up this diagram**! Plus, this article provides a strong basis for understanding service orientation and its benefits in our current technology environment.

# WcfNQueueSMEx2 Solution Structure

Below are listed the folders, namespaces, and projects in the WcfNQueueSMEx2 solution, along with brief comments on their general purpose where appropriate. For the most part the list focuses on those that are actually in use in the solution. However for the area of infrastructure (iFX) standard projects are listed that are not currently in the solution in order to give you a sense of what things this area contains. For a much broader view of possible namespaces please see the IDesign Alumni thread “Namespace convention and namespace best practices”.

“Contract” folders and projects contain service and data contracts shared by clients and services they consume. “Common” folders and projects contain things used across multiple subsystems and/or by multiple components.

**Contract** folder – All in this folder are contracts shared by client and server code. Each project contains the data and service contracts required by a manager and client code that calls it

GS.Contract.DataFeed – Project

GS.Contract.Admin -- Project

GS.Contract.SomeOtherManager – Project

**DataAccess** folder – All in this folder are concerned with providing Data Access or Resource Access

GS.DataAccess.Common – Project

Contains Data Access service and data contracts, their implementations and supporting classes that are **used by multiple Engines or Managers**, i.e. components that can call Resource or Data Accessors

GS.DataAccess.FeedAdmin – Project

Contains Data Access service and data contracts, their implementations and supporting classes relating to the Administration of Data Feeds (not to the ingested data) used only by the IFeedAdmin service contract. If this usage changes by it being used by Managers or Engines in multiple subsystems, then the files need to be moved to GS.DataAccess.Common. Note the AdminDA microservice that implements IFeedAdmin resides in GS.DataAccess.Common since it is used by multiple subsystems.

**Engine** folder – Contains things strictly pertaining to Engines

GS.Engine.Common – Project

Contains Data Access service and data contracts, their implementations and supporting classes that are **used by multiple Managers**, i.e. components that can call Engines

GS.Engine.Something – Project

Contains the SomeDataAnalysisEngine service and data contracts, their implementations, and supporting classes, etc. relating to this Engine that are used only within the “Some” subsystem. If this usage changes, i.e. it becomes used by Managers in multiple subsystems, then the files need to be moved to GS.Engines.Common.

**Explore** folder – Contains exploratory coding projects and learning projects

GS.Explore.AzTableStorageApp – Project

Contains all files associated with this exploratory coding app

**iFX** folder – iFX means Infrastructure. Things in the infrastructure do not know about nor contain business logic. Generally the infrastructure is a collection of small to medium- sized rather independent productivity aids, rather than being a large integrated framework. Below are **first listed the standard infrastructure projects**, followed by those unique to the WcfNQueueSMEx2 solution.

GS.iFX.Common project

Contains non-business logic, non-contract infrastructure items shared by both the client and server

GS.iFX.Contract project

Contains non-business logic infrastructure data and service contracts shared by both the client and server

GS.iFX.Core project

Contains non-business logic infrastructure items shared on the server only -- like caches, configuration helpers, mappers, security items, and non-service-oriented helpers

GS.iFX.Host project

Not used. See the folder iFX.Host for hosting. A separate folder is used since hosting is done on-prem and in the cloud, each requiring separate projects

GS.iFX.Proxy project

Contains non-business logic infrastructure items that are proxy related

GS.iFX.Service project

Contains non-business logic infrastructure items that are service-oriented and shared on the server

GS.iFX.Test project

Contains testing code for unit and integration tests

GS.iFX.Azure project

This contains Azure helpers having no business logic. There could also be separate projects for other clouds for composability

GS.iFX.TestUI project

Contains UI helper classes for test clients and also for trace and console writelines used in server side code

**iFX.Host** folder

GS.iFx.Host.AdminNSomeServiceHost – Project

Contains WCF service host and its supporting items for the Admin and “Some” subsystems combined, running in a console app or a Windows Service

GS.iFX.Host.Azure.DataFeedSvc – Project

Contains items required to host a WorkerRole in Azure for the DataFeed subsystem

GS.iFX.Host.Azure.DataFeedWorker – Project

Contains WorkerRole and its supporting items for the DataFeed subsystem running in Azure

GS.iFx.Host.DataFeedServiceHost – Project

Contains WCF service host and its supporting items for the DataFeed subsystem running in a

console app or a Windows Service

**Manager** folder

No common project is required here since the Manager’s service and data contracts are in the above Contracts folder so they can be shared by clients and the service herein.

GS.Manager.AdminManager – Project

Contains the AdminManager implementation and any supporting classes that are used only within the Admin subsystem

GS.Manager.DataFeedManager – Project

Contains the DataFeedManager implementation and any supporting classes that are used only within the DataFeed subsystem

GS.Manager.SomeManager – Project

Contains the SomeManager implementation and any supporting classes that are used only within the DataFeed subsystem

**Proxy** folder

GS.Proxy.Admin– Project

Contains the proxy implementation and supporting classes for proxies that are used by clients accessing the AdminManager

GS.Proxy.DataFeed – Project

Contains the proxy implementation and supporting classes for proxies that are used by clients accessing the DataFeedManager

GS.Proxy.Some – Project

Contains the proxy implementation and supporting classes for proxies that are used by clients for accessing the SomeManager

**Test** folder – Note that none of the below test projects have yet to be implemented.

GS.Test.Common – Project

Contains unit and integration test service and data contracts, their implementations and supporting classes that are shared by both throughout the server

GS.Test.Admin – Project

Contains unit and integration test service and data contracts, their implementations and supporting classes for the Admin subsystem

GS.Test.DataFeed – Project

Contains unit and integration test service and data contracts, their implementations and supporting classes for the DataFeed subsystem

GS.Test.Some – Project

Contains unit and integration test service and data contracts, their implementations and supporting classes for the DataFeed subsystem

**Test.Client** folder -- Contains Test Clients used for user interaction and implemented as standalone programs, plus items unique to each Test Client.

GS.Test.Client.AdminNSomeTester – Project

GS.Test.Client.DataSourceSimulator – Project

**UI** folder -- No UI projects have yet to be implemented. However, projects in this folder would be those required to implement an MVC app, or a WPF app, or a Windows Phone App, etc. for this system.

**Utilities** folder – Note that no utility projects have yet to be truely implemented.

GS.Utilities.Timer – Project

Contains service and data contracts, their implementations and supporting classes for the Timer utility

Revision History

V 0.1, 5-18-15, George Stevens. Initial document using Monty’s posts in the above mentioned thread.

V 0.2, 5-19-15, George Stevens. Refined concepts and namespace/solution structure based on feedback and deeper understanding of the basic organizing principles involved.

V 0.3, 5-29-15, George Stevens. First try at the “Solution Structure… “ section.

V 0.4, 6-4-15, George Stevens. Major edit. Added most of the “Definition of Terms..” section. Revised the “Solution Structure..” section to sync with the code.

V 0.5, 6-6-15, George Stevens. Minor edits to the “Introduction” and “Definition of Terms..” sections. Major edits to “Solution Structure…” section.

V 0.6, 6-12-15, George Stevens. Major edits to incorporate feedback, plus finalization of the “Solution Structure…” section to clean it up and sync with the code.

V 0.7, 6-26-15, George Stevens. Fixed minor errors in folder and project definitions and included a link to the blog article.