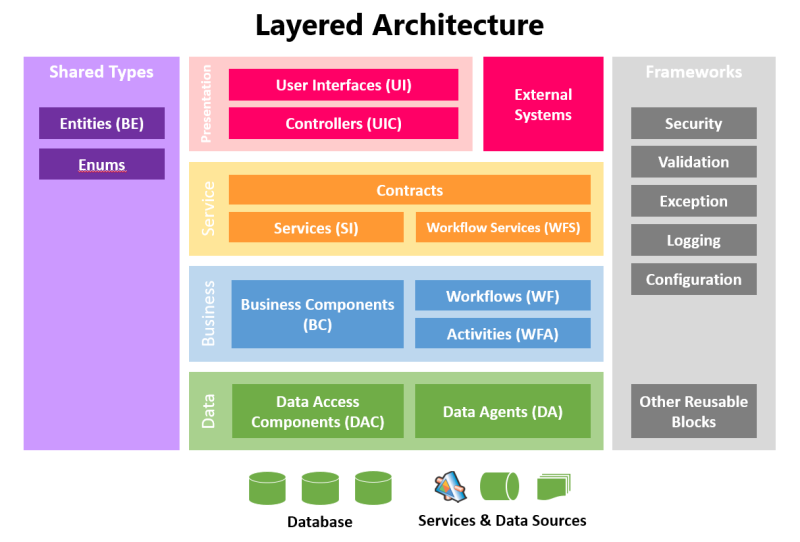
[Layered Architecture: Introduction](http://serena-yeoh.blogspot.com/2013/06/layered-architecture-for-net.html)

I have always been an advocate for the**Layered Architecture pattern**. Since the day I was exposed to it, I have always tried to practice it in my application development.

The Layered Architecture pattern promotes the concept of separation of concerns where code of similar responsibilities are being factored into layers. It is purely a *logical*design but it can be combined with *physical*design patterns such as the *N-tier architecture* to deliver highly scalable and impressive distributed enterprise applications.

Having followed the [Microsoft Application Architecture Guide 2nd Edition](http://msdn.microsoft.com/en-us/library/ff650706.aspx) and its previous edition, I have always tried to materialize the concepts in code with the .NET technologies available at the time. The result of such work can be seen in [Layered Architecture Sample for .NET](http://layersample.codeplex.com/). I actually started testing out the concepts with .NET Remoting but when I published the samples, I have already started learning WCF.

Lately, I realized that many people are starting to adopt the Layered Architecture pattern and I also noticed many newer .NET technologies have emerged. I would like to take this opportunity to provide an article on my thoughts and perhaps an updated version of the Layered Architecture pattern for .NET.

[](http://1.bp.blogspot.com/-TbA8p6DWYJc/Uc0zYwSG_iI/AAAAAAAAAag/cPytbL6oeM4/s800/layered.png)

Conceptually, this is how I visualized the Layered Architecture Pattern to be, in today's modern world. There are of course more sophisticated visualizations but I purposely kept it simple and near to what most of us are familiar with (and closer to the books).

**Data Layer (a.k.a. Data Access Layer or DAL)**

The data layer is where we keep our components that handle the *insertion*(**C**reate),*selection*(**R**ead), *modification*(**U**pdate) and *deletion*(**D**elete) of data - or better known as CRUD operations. While it is simplistic to think that data usually comes from a database, in reality, data can come from or go into various sources as well i.e. Web Services, Flat Files, Message Queues, XML files, SharePoint Lists etc.

Therefore, components that deal with database tables are called**Data Access Components (DAC)** and those that deal with other data sources will be called **Data Agents (DA)** i.e. Service Agents, File Agents, Queue Agents etc.

Data Access technologies that you can use in .NET are **ADO.NET, Enterprise Library Data Access Application Block**and **ADO.NET Entity Framework.**

**Business Layer (a.k.a. Business Logic Layer or BLL)**

The business layer is where the heart of our application resides. It contains all the processing logic to make the application possible. The **Business Component (BC)** is where you put these processing logic where each can be coded into independent*business methods*. Traditionally, we are required to chain-up the business methods manually in code to form the business process but fortunately today, we have workflow technologies.

If you can isolate each business method to function on its own, you can exposed them as a **Workflow Activity (WFA)**. These workflow activities can then be used by a **Workflow (WF)** component to orchestrate the business processes.

**Windows Workflow Foundation** is the workflow technology that can be used in .NET.

**Services Layer (a.k.a. Messaging Layer)**

The services layer plays the most important role in the architecture to enable the functionality of the system to be exposed to client and external applications. It is also the key to achieving multi-platform and interoperable solutions.

Services components expose the functionality of business components or workflows via**Contracts**. In the *Services-Orientation* world, contracts are the *interfaces*where both service providers and service consumers agree on and should be immutable. Contracts are not just limited to describe the service and its operations but can also be used to describe the messages (i.e. Message Contracts) that are to be sent and received.

I use**Services (SI)** to represent components that expose business components directly and**Workflow Services (WFS)** to represent services that expose workflow functionality. The reason for this is because Workflow Services are usually long running and may have special requirements such as correlation.

Services technologies that can be used in .NET are **Windows Communication Foundation (WCF)**, **Workflow Services** and **ASP.NET WEB API**.

**Presentation Layer (a.k.a. User Interface Layer)**

The presentation layer should not need much explanation. It is basically the part of the system where the user interacts with. Your screens, forms, web pages and reports are all **User Interfaces (UI)** which are part of the presentation layer. User Interfaces can make use of **User Process Components** or **Controllers (UIC)**to communicate with the back-end and to navigate or process the UI.

A carefully designed layered application should be able to support any form (or platform) of UI. If you are able to encapsulate all your processing logic behind the service layer, you can have whatever UI (Web, Desktop or Mobile) that you desire to connect to it - even UI-less external systems.

Presentation technologies that can be used in .NET are **Windows Presentation Foundation (WPF), Silverlight, ASP.NET Web Forms, ASP.NET MVC, Windows Phone, Windows Store Applications**and **Windows Forms**.

**Shared Types**

So far we have covered the components in all the layers but we have not yet discussed about how data is being passed between them. Traditionally, .NET developers use DataSets and DataTables but these are heavy-weight objects. **Entities**are *Plain-Old-CLR-Object*s (POCO), which means they are just classes with properties that ferry data across your layers. Sometimes, they are also called *Data Transfer Objects* (DTO).

It is recommended that you do not put any processing logic inside the Entities. If there are any processing logic, they should be placed in the business components. The reason is because when entities are being serialized to non .NET platforms, your processing logic may not carry over.

Some property values in entities can be represented using **Enumerations (Enums)**for easier readability. Example, it is much better to strongly-type your Status property with an Enum to show meaningful statuses such as Pending, Cancelled or Approved, instead of 0, 1 or 2.

**Frameworks (a.k.a. Cross-cutting Framework)**

In every system, there are bound to be code that can be shared across all the layers i.e. logging, auditing, validation and etc. You can treat these components as **Framework**components that can be shared by any of the layers. Framework components can be from 3rd-party (i.e. Microsoft Enterprise Library) or any custom in-house built components i.e. string manipulation functions, custom validation functions, extension methods etc.

**Conclusion**

I hope the explanation in this article can be a useful foundation for adopting the Layered Architecture pattern. If you wish to see code samples on how it can be implemented with .NET technologies, please feel free to visit [Layered Architecture Sample for .NET](http://layersample.codeplex.com/).

[Layered Architecture: Deployment Strategies](http://serena-yeoh.blogspot.com/2013/07/layered-applications-3-tier.html)

In my previous [post](http://serena-yeoh.blogspot.com/2013/06/layered-architecture-for-net.html), I have briefly explained the functions of each logical components in the **Layered Architecture Pattern** and in this post, I'll be covering the deployment architecture (or physical architecture) of layered applications.

To understand the deployment architecture of layered applications, we will first need to understand the difference between *layering* and *tiering*. **Layering**is the concept of partitioning code into components that form *logical* layers and **Tiering** is the term used to describe the distribution of code units to *physical* boundaries. In simpler terms, layering is *"how you organize your code into assemblies"* (.dll, .exe) and tiering is *"where you deploy those assemblies"*.

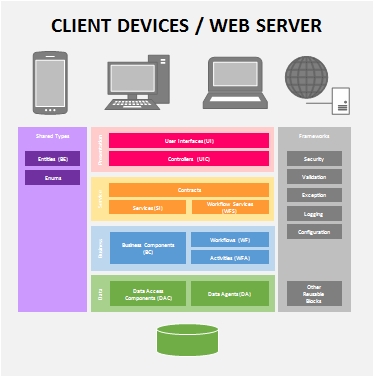
A layered application can generally support flexible tiering through the introduction of a service layer that employs any distributed communication technologies such as**DCOM, Remoting, Sockets, Message Queues, Web Services** and etc. How many tiers are required will most likely depend on the security, scalability and infrastructure requirements (or constraints) of your application. You must know that performance degrades with each tier being introduced as the application will need to perform cross-process, cross boundary calls to each tier.

Generally, basic enterprise applications opt for*3-tiers* (Web, App and Database) with some sophisticated ones spanning to 4 or more tiers (*N-tier*). The layer diagram which I presented in my previous [post](http://serena-yeoh.blogspot.com/2013/06/layered-architecture-for-net.html) depicts a standard 4-layer-3-tier architecture application and it will be used as the basis for discussion in this post.

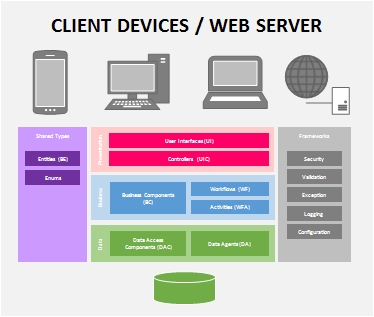
Take note that if you find the text in the diagrams too small, please refer to my previous[post](http://serena-yeoh.blogspot.com/2013/06/layered-architecture-for-net.html). I had intentionally shrunk the diagram to illustrate how the components would fit in a deployment and would expect that you are already familiarize with the color-coded boxes.

**Single-Tier Architecture**

The simplest way to deploy a layered application is to deploy everything to a single server. This is called the **single-tier** or *monolithic*approach. This approach is common for desktop and some types of mobile applications. It is also common when there is a server budget constraint for testing new web applications.

[](http://2.bp.blogspot.com/-JJ46Fcymu6E/UfSXesyHBeI/AAAAAAAAAdI/PMjDJIjYMqE/s1600/monolithic.jpg)

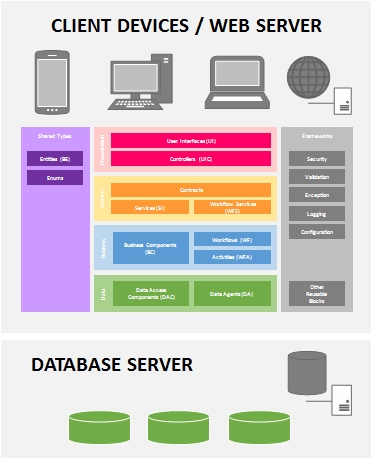
If you are certain that your application will not grow (which is rarely the case), you can improve the performance by completely removing the service layer.

[](http://4.bp.blogspot.com/-BUHYOxjWP5g/UfSXzUIKfoI/AAAAAAAAAdQ/4h7FhSCtJ0w/s1600/mono-nosvc.jpg)

It is generally safe to omit the service layer for monolithic client applications but I recommend to keep it there for web applications because the tendency for web applications to grow is higher. To reduce the impact on performance for the service layer in a monolithic web application, you can use the [netNamedPipeBinding](http://msdn.microsoft.com/en-us/library/ms731291.aspx) for the WCF/WF services.

**Two-Tier Architecture**

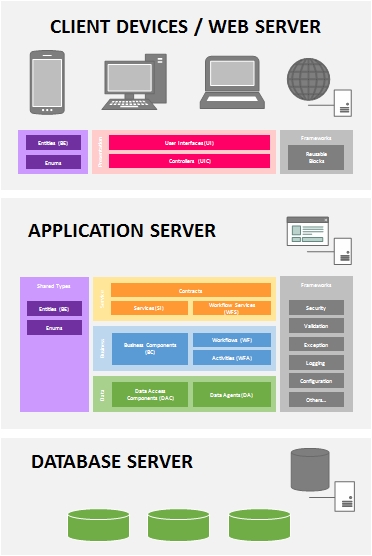
It is usually very rare to find the single-tier approach in an enterprise environment. You may find them in development servers but most of the time, enterprise applications will employ the **two-tier** or *client-server* architecture due to the need for data centralization. (Like the single-tier architecture, you can remove the service layer for performance).

[](http://3.bp.blogspot.com/-6CHNOAcQpcQ/UfSW6w47wKI/AAAAAAAAAdA/74fYkWZYTJA/s1600/2-tier.jpg)

The client devices or web server will host all the presentation and processing logic while accessing data from a centralized database server or a data service.

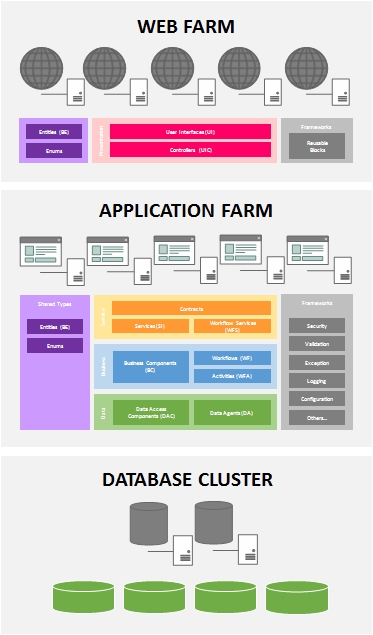
**3-Tier Architecture**

Due to scalability and security concerns, typical enterprise web applications usually employ the **3-tier** deployment architecture. The web servers are usually placed in a public-facing *perimeter network*and the application servers are placed behind a firewall within a *secured network*.

[](http://3.bp.blogspot.com/-n6Nv689k4iA/UfNx9WDj-jI/AAAAAAAAAbg/lufcDdONUOQ/s1600/3-tier.jpg)

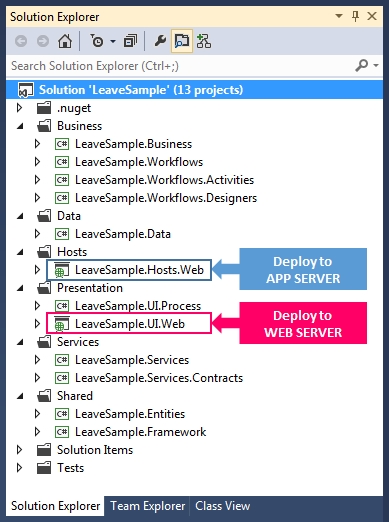
A properly designed service layer will enable the back-end to be accessible by external systems and client-devices. Using technologies such as **Windows Communication Foundation (WCF)** and **ASP.NET WEB API**, the service layer can provide support for a variety of client platforms including non-.NET platforms such as iOS, Android, Java and etc.

In a highly-scalable and available environment, the web servers are load-balanced into a**Web Farm**and the application servers are load-balanced into an **Application Farm**. The database servers are *clustered* for high-availability.

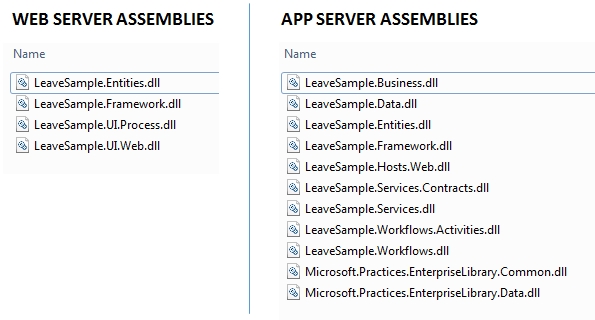
[](http://4.bp.blogspot.com/-KsgEE9KXDGU/UfN1N2bOZhI/AAAAAAAAAcM/QQpRgHKTUwM/s1600/farm.jpg)

**Deploying the Layered Application**

If you are following closely to the project structure as illustrated in [Layered Architecture Sample for .NET](http://layersample.codeplex.com/), you may be wondering which projects should you be deploying. For the sample Leave application, you only need to publish/deploy the **Web** project to the**web server**and the **Hosts**project to the **application server**.

[](http://3.bp.blogspot.com/--t-VdyWEq0I/UfN5MGKwJMI/AAAAAAAAAcc/3mnNMbeoL00/s1600/se.jpg)

You will noticed that the application server will contain the service, business and data layer assemblies whereas, the web server will only contain the presentation assemblies. This keeps business rules safe in the application server and reusable for any type of client applications.

[](http://3.bp.blogspot.com/-CBhzYAOx990/UfN7bGNNLYI/AAAAAAAAAcs/4t8wFB1zBSI/s1600/dll.jpg)

[Layered Architecture: Deploying to Windows Azure](http://serena-yeoh.blogspot.com/2014/01/layered-applications-and-windows-azure.html)

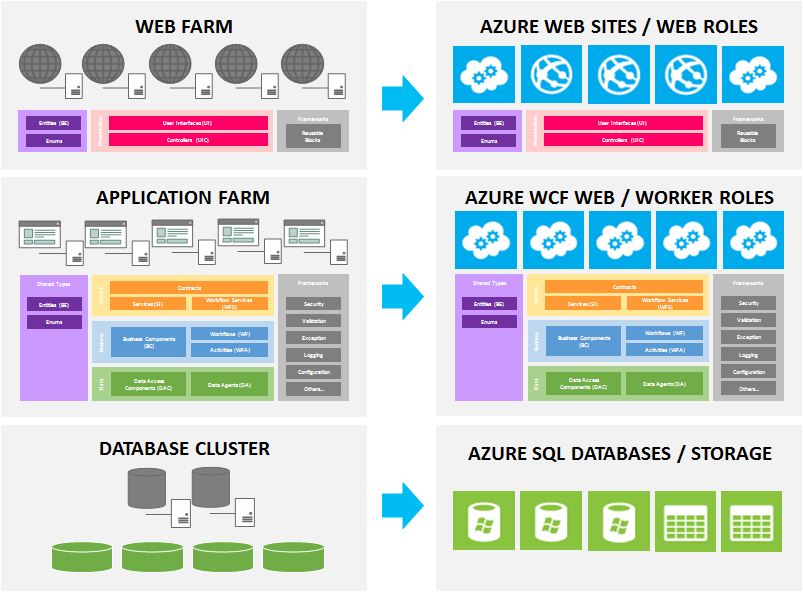
With the arrival of cloud computing, we may wonder whether the **Layered Architecture Pattern**would still be relevant. In theory, everything should work as-is if we are leveraging on **Infrastructure-as-a-Service (IaaS)** because that only involves moving our servers to the cloud, but what about **Platform-as-a-Service (PaaS)**?

Well, I'm glad to learn that Windows Azure provides a variety of deployment options for enterprise layered applications (provided if the applications were properly layered on-premise).

[Note:] If you find the text in the diagrams too small, please refer to my previous [post](http://serena-yeoh.blogspot.com/2013/06/layered-architecture-for-net.html) for a larger illustration.

**PaaS - Windows Azure Web Sites and Cloud Services**

For new or enterprise applications that can be migrated completely to use the PaaS model, we can leverage on [**Windows Azure Cloud Services**](http://www.windowsazure.com/en-us/services/cloud-services/) which offer us the options of deploying **Web Roles** and **Worker Roles**. Web applications containing the presentation layer can be deployed to Web Roles and the back-end stack containing the service, business and data layers, can be deployed to either Web or Worker roles.

[](http://3.bp.blogspot.com/-rD8Rd3IJhb8/Us4Pph48pDI/AAAAAAAAAto/fMGnAC5wV9E/s1600/paas.png)

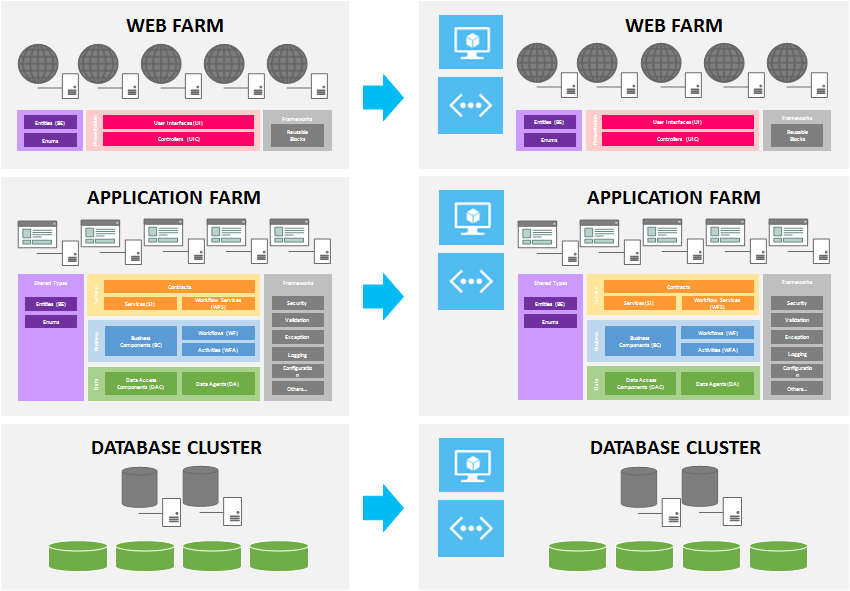
If the service layer was developed in WCF or WEB API, the back-end stack can be deployed to a Web Role. It is not necessary for the back-end stack to be using Worker Roles unless necessary.

Web applications can also be deployed to [**Azure Web Sites**](http://www.windowsazure.com/en-us/services/web-sites/) if they are simple pages but I would recommend using Web Roles instead because they are more suited for application environment (i.e. network isolation, setting up start-up tasks, support for virtual networks, multi-deployment environments and etc).

As for the database portion, some rethinking is required. The notion of database clusters is somewhat non existence in a PaaS model and instead, PaaS uses the concept of replicas for sustaining high-availability. Also, large databases may need to be *shredded*(horizontal partitioned) in to smaller databases and later use **Federation**to query them**.**

**Iaas - Windows Azure Virtual Network and Virtual Machines**

For existing layered applications that could not be migrated to the PaaS model, Windows Azure also provides IaaS options through [**Virtual Network**](http://www.windowsazure.com/en-us/services/virtual-network/) and [**Virtual Machines**](http://www.windowsazure.com/en-us/services/virtual-machines/). Similar to an on-premise environment; web, application and database servers can be virtualized into Virtual Machines and then configured within a Virtual Network on Windows Azure.

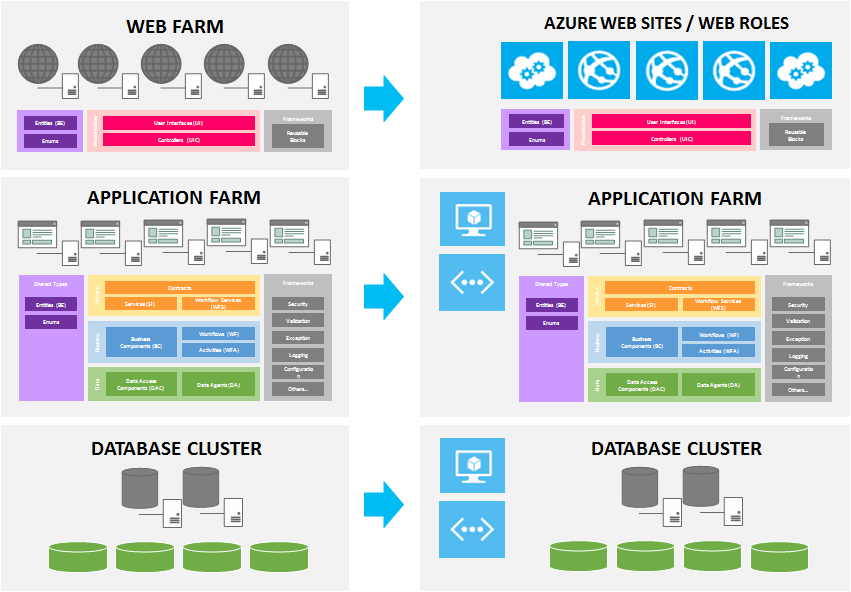
[](http://1.bp.blogspot.com/-T6secPwjkC0/Us4TVOIUCcI/AAAAAAAAAt0/eLbTWNKe2jk/s1600/iaas.png)

Notice that we are able to setup database clusters for high-availability in IaaS but unlike PaaS, the tasks and responsibilities of setting up all the servers (Web, App and DB) are on us.

Organizations may consider the IaaS model to reduce the risk of migrating applications to the cloud as it closely resembles the architecture of existing on-premise infrastructures. It is also a good option for quickly provisioning servers for testing out prototype solutions and applications. Organizations who want to have more control on their servers will also find IaaS more suitable for their liking.

**Paas and Iaas Working Together**

Windows Azure do not limit us to an all-or-nothing option when it comes to deploying applications. Through layering we can leverage on one of its benefits whereby each layer can be developed, migrated and upgraded separately from the others. In this case, we can have scenarios where our web application are migrated to an Azure Web Site or Web Role, while the back-end stack can be hosted on Virtual Machines in a Virtual Network.

[](http://2.bp.blogspot.com/-eTvIm7vhgR8/Us4WFocU-5I/AAAAAAAAAuA/w7dEv8UmdVQ/s1600/paas-and-iaas.png)

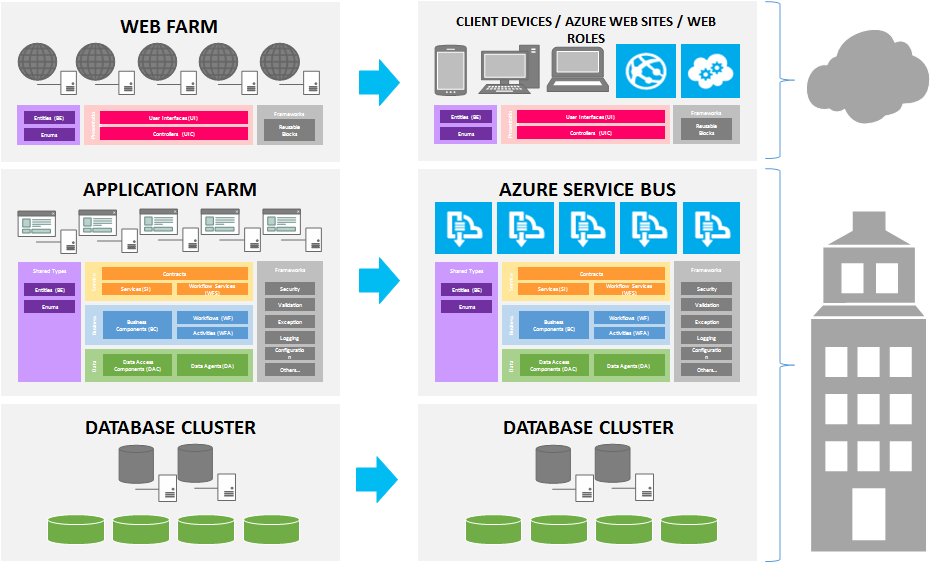
The use of Azure Web Site or Web Roles will surface some differences here. With Web Roles, your web application will be able to join the Virtual Network of the VMs. The Web Sites can only call the App server from external endpoints that we need to configure.

**Hybrid On-Premise and Cloud**

In all enterprises, there will be applications that cannot be migrated to the cloud. This can be due to governing policies, readiness or even the need to support for legacy systems. In such situations, all hopes are not lost as Windows Azure also provides the ability to connect on-premise applications to the cloud.

A common method may be to deploy certain servers to Virtual Machines in an Azure Virtual Network and then configure VPN to connect back to the on-premise environment. Windows Azure provides [Site-to-Site](http://www.windowsazure.com/en-us/manage/services/networking/cross-premises-connectivity/) and [Point-to-Site](http://msdn.microsoft.com/en-us/library/windowsazure/dn133792.aspx) VPN connectivity for this purpose.

However, if the applications are properly layered, we can actually leverage on [**Azure Service Bus**](http://www.windowsazure.com/en-us/services/messaging/) to expose any on-premise service stack to the cloud.

[](http://4.bp.blogspot.com/-UKSEvaDkt60/Us4broPxZeI/AAAAAAAAAuQ/9S7bQgVGLyI/s1600/hybrid.png)

Service Bus can be used to expose any on-premise services to other external systems (i.e. partner extranets) that are either hosted on other premises or on the cloud. It can also be used to expose services to mobile applications. You can secure your Service Bus endpoints using [Access Control Service](http://msdn.microsoft.com/en-us/library/hh147631.aspx).

**Summary**

As we can see, the Layered Architecture Pattern stays relevant despite the emergence of cloud computing. In fact, having a carefully layered design may assist in easing the migration to the cloud. Even, if you are not developing applications for the cloud today, I would still strongly encourage you to consider layering your applications.

Arguably, we can still deploy monolithic web applications (everything in a server) to the cloud and ~~abuse~~ use the elastic scaling capabilities of the cloud by throwing in more instances but that will not provide us with the option to provide isolation and perform granular tuning or scaling i.e. 4 Web server instances serving content and 2 App server instances processing logic.

You can check-out samples of layered applications developed for Windows Azure in[Layered Architecture Sample for Azure](http://cloudsample.codeplex.com/).

# [Layered Architecture: Applied Technologies](http://serena-yeoh.blogspot.com/2014/01/applied-technologies-in-layered.html)

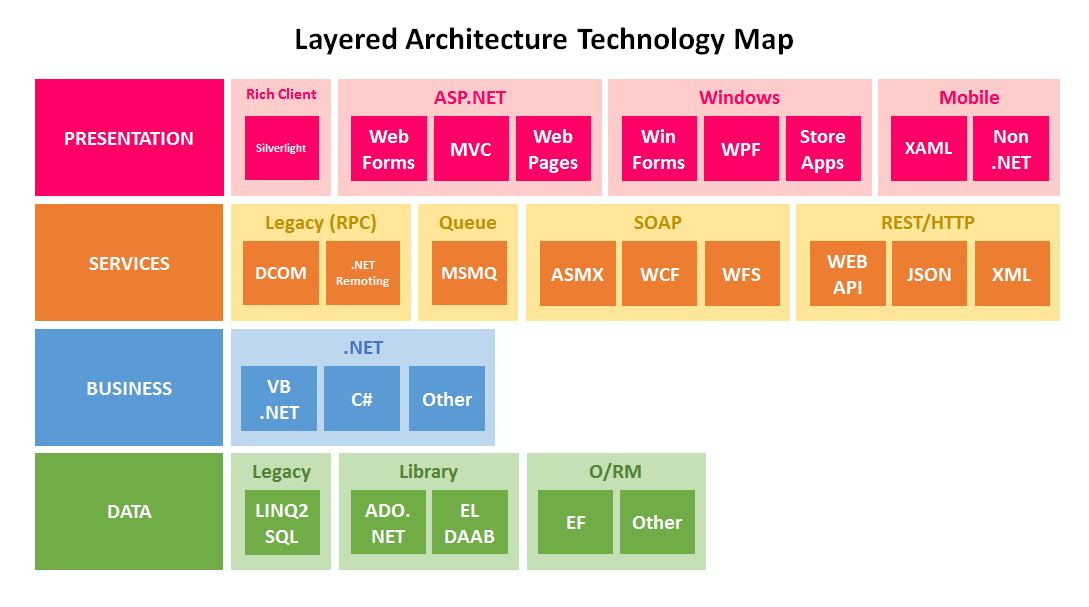
The **Layered Architecture Pattern** promotes isolation as one of its benefits and with layer isolation, we are given the idea of being able to upgrade and migrate each layer to newer technologies without impacting the business logic. Having gone through few iterations of experimenting the pattern with various .NET Technologies, it has proven the idea to be somewhat accurate, and indeed the pattern proves to be a versatile architectural style that can evolve across time and sustain technology evolutions.

I used 'somewhat accurate' in my description due to certain caveats which I will disclose at the end of this post.

Through the exercise of testing the pattern, it is observed that the layers can easily be *'swapped-in*' and '*swapped-out*' when their interfaces are being abstracted carefully. You can download the samples from [Layered Architecture Sample for .NET](http://layersample.codeplex.com/releases/view/117278) to see the implementations.

The [January 2014 release](https://layersample.codeplex.com/releases/view/117278) of the samples demonstrates a variety of technologies assembled to form different flavours of layered applications. While the technology in other layers can be different, the business logic does remain intact.

To provide an easier visualization of what technology can be applied in the layers, I have came out with the following technology map. Take note that this is intended only as a basic guide and you are free to use whatever technologies you deemed fit in your environment.

[](http://4.bp.blogspot.com/-gqnaUzvD6bU/UtxvMtjhb7I/AAAAAAAAAvg/UVWZfq6VHCw/s1600/layer-techmap.png)

For the **presentation layer**, we have a choice of:

* ASP.NET Web Forms
* ASP.NET MVC
* ASP.NET Web Pages
* Windows Forms
* Windows Presentation Foundation (WPF)
* Windows Store Apps
* Windows Phone Apps

It is quite common to leverage on ASP.NET for web applications. ASP.NET has many variants today to suit the skill-sets of different developers. Windows Forms are still widely used in enterprises where fast and responsive client-server desktop applications are required. For exciting UI applications, WPF is there to fill the gap and layering can be interestingly integrated to it even with the MVVM pattern.

Windows Store Apps are new in Windows 8, but both Store Apps and Windows Phone Apps are more suited to be connected to a cloud-based back-end (that can also be layered). Silverlight is being included as a rich-client technology but do take note that it is no longer being developed.

For the **services layer**, we have:

* Windows Communication Foundation (WCF)
* Windows Workflow Foundation (WF) hosted as a service (Workflow Services)
* ASP.NET WEB API exposing JSON or XML (POX)
* Microsoft Message Queue (MSMQ)

The most common distributed communications technology today are WEB API and WCF. For most resource-based and web-based services, WEB API is the preferred choice. It is also very suitable for providing back-ends for mobile applications. WCF still exist in large enterprises to facilitate interoperability between legacy and service-oriented systems and MSMQ is there to provide queue-based solutions.

Deploying old-style ASMX Web Services for new applications is not recommended, even-though you may encounter them in legacy systems. Traditionally, the services layer were fulfilled by RPC technologies such as Distributed Component Object Model (DCOM) and .NET Remoting. DCOM is still supported in the latest versions of Windows but .NET Remoting has been superseded by WCF. These legacy technologies should not be used in newer applications.

For the **business layer**, it is all based on our code and processing logic. At the most basic level, it will just be the .NET programming languages that we use to build our components, C# or VB.NET - F# anyone?

For the **data layer**, we have:

* ADO.NET
* ADO.NET Entity Framework
* Enterprise Library Data Access Application Block (DAAB)

Nothing beats native ADO.NET when it comes to data access performance but some may prefer a lightweight wrapper over it such as the DAAB. If you are still using LINQ2SQL, I will suggest migrating to ADO.NET Entity Framework. Other 3rd-party **Object-Relational Mappers (O/RM)** can also be used here as well.

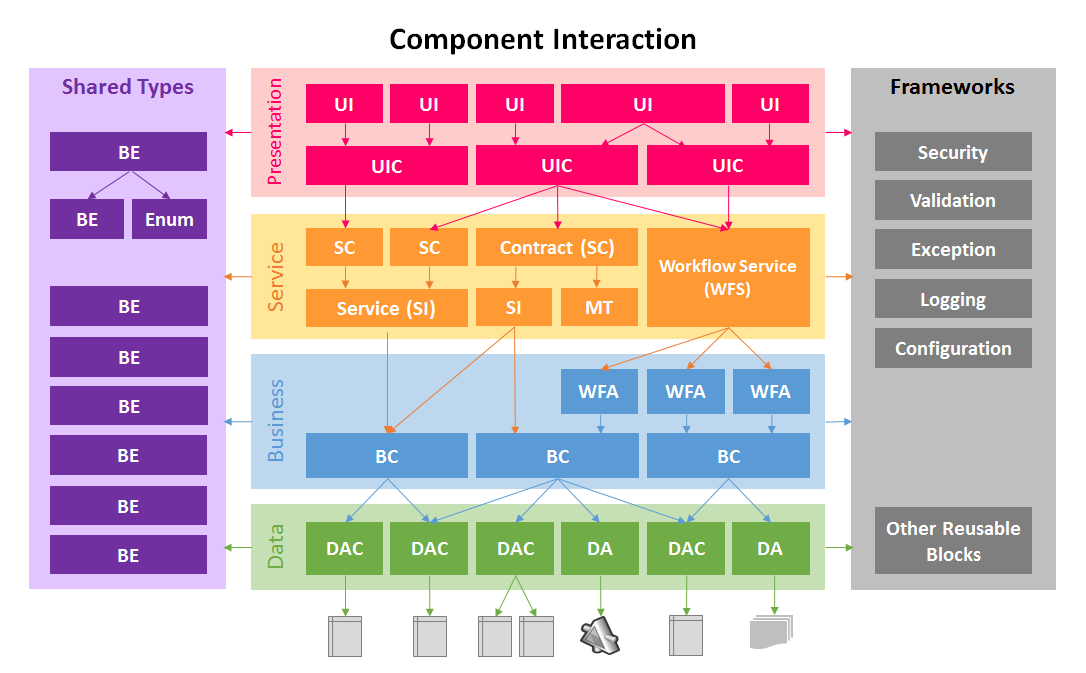
Hopefully with the above technology map, you are able to get an idea of the technologies that can be used for building layered applications. Take note that in some scenarios, there can be more than one technology in a layer (i.e. WCF and Message Queue) and in some, the technologies may not be easily compatible.

The caveat which I mentioned earlier is that while minor technology upgrades can be isolated to a single layer (i.e. replacing data layer with a newer technology), major technology upgrades (i.e. migrating from WCF to WEB API) may affect more than one layers. But nevertheless, the business logic is still preserved. As a conclusion, it is always best to plan any technology upgrades for the layers carefully to minimize the impact.

[Layered Architecture: Component Interactions](http://serena-yeoh.blogspot.com/2014/01/layered-architecture-components.html)

The **Layered Architecture** principle states that components in one layer should only know and interact with components that are in the layer directly below it, and that components in each layer, should only serve components that are in the layer directly above it. This means that in a *strict-layering* practice, layers will communicate in a top-down fashion from **Presentation** -> **Services** -> **Business**-> **Data**.

It is often easier said in theory but spells a lot of confusion to developers, especially to beginner practitioners, when it comes to implementation. To help visualize the components' relationships and interactions better, I have developed the following diagram and provided some basic guidelines.

[](http://4.bp.blogspot.com/-_rIdNouv2Zg/UtyekFwyKrI/AAAAAAAAAvw/H0yO2VmXgFs/s1600/layer-component.png)

**Legend**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BE | = | Business Entity | BC | = | Business Component | SI | = | Service Implementation |
| Enum | = | Enumerations | WFA | = | Workflow Activity | MT | = | Message Type |
| DAC | = | Data Access Component | WFS | = | Workflow Service | UIC | = | User Interface Controller |
| DA | = | Data Agent | SC | = | Service Contract | UI | = | User Interface |

When designing component interactions, use the following guidelines:

**Shared**

* An entity may contain other entities. i.e. Order with a List<OrderItem>
* An entity may use one or more enumerations for its properties.
* All components in the layer can reference entities and enumerations.

**Data**

* A data access component should refer to a Table or View in the database.
* A data access component may manage more than one related tables i.e. Orders and OrderItems.
* A data agent should be used to manage the access to external services (known as Service Agent)
* A data agent should be used to manage access to files (known as File Agent)

**Business**

* A business component should call more than one data access components. One-to-one mapping of business component to data access component is an early indication of something is amiss.
* A business component may call a mixed of data access components and data agents which may also be called by other business components.
* A business component may have some or all of its methods exposed as workflow activities.
* A workflow activity should map to one business component method (although mapping to more than one is OK but not recommended).

**Services**

* A service may call one or more business components.
* A workflow service usually contains more than one workflow activities to construct workflows.
* A workflow service may contain workflow activities that are exposing methods from different business components.
* A contract exposes a service (if using WCF).
* A service may have more than one contracts (if using WCF).
* A contract may use message types to consolidate data into request or response messages. In this case, they can be data contracts or message contracts. Message Types can also be used for WEB API.

**Presentation**

* A controller may call one or more services through contracts (If using WCF).
* A controller may call into a service (when WCF is not used).
* A controller may call a mixed of contracts and workflow services. (WFS are actually WCF).
* A controller may be called by more than one user interfaces.
* A user interface may call one or more controllers which may be called by other user interfaces.

**Framework**

* All framework components can be called by components in one or more layers.

Take note that these are just the guidelines I tried to practice in strict layering. In a relax layering practice, the guidelines should be less rigid.