Case Study : Distributed Application Design

Introduction:

The purpose of this section is to provide architecture- and design-level guidance for building distributed applications with the .NET Framework. Designing a distributed application involves making decisions about its logical and physical architecture and the technologies and infrastructure used to implement its functionality. To make these decisions correctly and effectively, one must possess a sound understanding of:

- **Functional requirements** - the underlying business processes that the application will perform.
- **Non-functional requirements** - the required levels of availability, manageability, performance, reliability, scalability, and security.

When you design a distributed application, you must ensure that you design:

- Solves the business problem the application is supposed to solve.
- Addresses security considerations from the start
- Is available and resilient and can be deployed in redundant high-availability data centres.
- Is manageable, allowing operators to deploy, monitor, and troubleshoot the application as appropriate for any given problem?
- Provides high performance and is optimized for common operations.
- Scales to meet the expected demands, and supports many activities and users with minimal resources.
- Works in various scenarios and deployment patterns.

Services:

As organizations seek to integrate their systems across departmental and organizational boundaries, *a service-based approach to building applications has evolved*. Services are very similar to traditional components except that services encapsulate their own business logic and data in such a way that they are not strictly part your system. Rather, they are used by your application to provide a specific service, such as credit card authorization. Services expose their services through a ***service interface*** to which all inbound messages are sent. The definition of the set of the messages that must be exchanged with the service to perform a specific business task constitutes a *contract*. A service interface is similar to a COM interface or to any interface implemented by a C# class.

As an example, an online shopping application may use an external credit-card authorization services to validate the customer's credit card and authorize the sale. After a successful authorization, the application may then use an external service to arrange delivery of goods. The credit-card-authorization and delivery of-goods services each play a role in the overall business process. However, unlike ordinary components, these services exist in their own trust boundary and manage their own data outside the application. This implies two design considerations:

Note that if your application uses a service, you simply need to know the business functionality that the service is attempting to provide, and the details of the contract that you must adhere to in order to successfully interact with the service. The internal implementation of the service is irrelevant to your design.

Internally, services most often contain the same components that other applications do - logic components, business components, and data access components. Additionally, services expose their functionality through a service contract which handles the semantics of exposing the underlying business logic. Your application will often not interact with services directly, but through service agents which communicate with the service on behalf of the application.

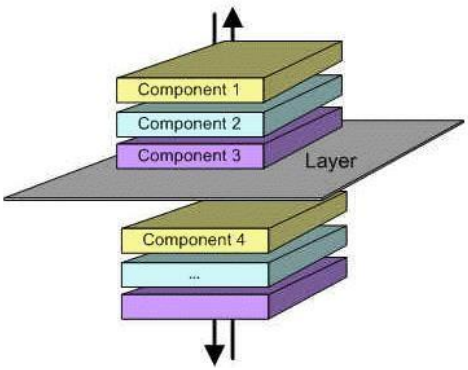
Message-Based Communication:

Finally, note that applications and services that need to be integrated may be built on different platforms by different teams in different organizations, and may most possibly be maintained and updated independently. Therefore, *it is critically important that you implement communication between them with the least coupling possible*. To this effect, ***it is crucial that communication between applications and the services they use be implemented using message-based techniques to provide high levels of robustness and scalability***. And although message-based communication can be designed to be called synchronously, ***it is much more advantageous to call service interfaces asynchronously***. Asynchronously calling service interfaces allows a more loosely coupled approach to distributed application development, and makes it possible to build highly. Scalable, available and long-lasting solutions.

However, asynchronous design does not provide its benefits for free. You will need to take into account issues such as message correlation, optimistic data concurrency management, external service unavailability, and so on.

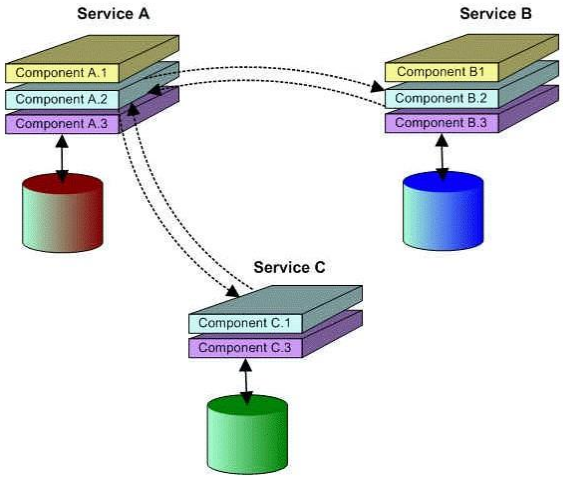
Components and Tiers:

It has become widely accepts that distributed applications should be logically divided into presentation, business and data access tiers. Components that perform similar functionalities should be grouped in the same layer, which in many cases are organized in a stacked fashion as shown below:



Here a component uses the services of the other components in its own layer and in other layers above it and below it.

This partitioned view of components and layers can also be applied to services. From a high-level overview, a service-based application can be seen as composed of many integrated services, each communicating with other services using message-based communication. Conceptually, these services can be seen as other components in the overall solution. However, and this an important point, *each service is made up of a separate set of components just like any other application and these separate set of components that make up a service can be logically grouped into presentation, business and data service*. This concept is illustrated below:



Note the following important points about this figure:

- Services should be designed to communicate with each with the least possible amount of coupling. Message-based

communication helps to decouple availability and scalability of individual services.

- Relying on industry standards such as XML Web Services allows for easy integration with other platforms and technologies.
- Each service can be considered as a separate application with its own data source, business logic, and user interfaces. In other words, a service may have the same internal design as a traditional 3-tier application (services A and B in figure above). On the other hand, a service does not necessarily need to have a user interface (service C in the figure above).
- Each service encapsulates its own data and manages atomic transactions within its own data sources.

An application or service is made up of one or more layers, and a layer is merely a grouping of logically related components. Therefore, a layer helps to differentiate between the different kinds of tasks performed by the components making it easier to see how the overall system works. By identifying the generic kinds of components that exist in most solutions, you can construct a conceptual map of an application or service, and then use this service as the blueprint for your design. The following shows the Microsoft-recommended conceptual-view of applications and services:

A distributed application typically have to span multiple tiers or even organizations in which case, the application must have its own policies regarding application security, operational management, and communications. These policies define rules for the environment in which the application is deployed and how services and applications tiers communicate.

Sample Scenario:

In this sample scenario, customers can order from a retail company either by telephone or through their web-site. In either case, customer's details or order is entered through a Windows-based application. On submitting the order, the customer's credit card details

are authorized using an external credit card authorization service, and delivery is arranged via an external delivery service.

The proposed solution is a component-based design as shown below: