CHAPTER

7

The Distributed Services Layer

1.- LOCATION IN THE N-LAYERED ARCHITECTURE

This section describes the architecture area related to this layer, which is logically 'Service Oriented'. In many ways, SOA (*Service Oriented Architecture*) overlaps with 'Service Orientation', but they are not exactly the same concept.

NOTE:
During this chapter, when we use the term 'Service' we are referring to Distributed-Services or Web-Services, by default. We are not referring to internal Domain/Application/Infrastructure Services (DDD patterns).

The following diagram shows how this layer (Distributed Services) typically fits into the '*Domain Oriented N-Layered Architecture*':

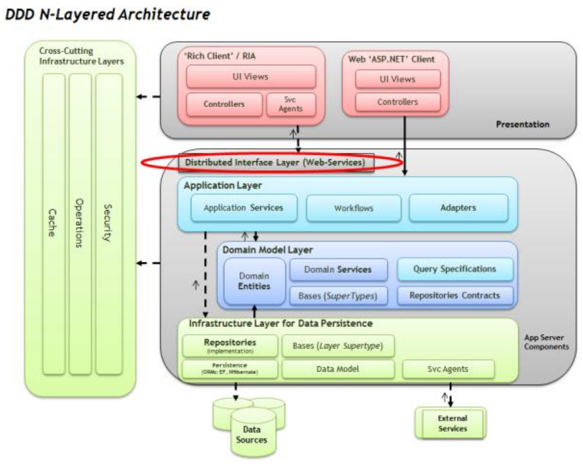


Figure 1.- Location of the Distributed Services Layer

The Service Layer typically exposes the following topics:

- **Services**: Services expose interfaces that receive incoming messages. In short, services are like a façade layer for remote clients. Services expose the application and domain logic to the potential consumers such as Presentation Layer or other remote Services/Applications.

- **Message Types**: In order to exchange data via the Service Layer, data structures are serialized to messages. The service layer will also expose data types and contracts that define the data types used in the messages (like DTOs or *Resource Representations*), although these data classes could be defined within the Application Layer instead of the Distributed Services Layer, so they can be re-used from different channels (direct .NET CLR, etc.).

SOA, however, covers a lot more than the design and implementation of an internal distributed Service Layer for only one N-layer application. The advantage of SOA is that it can share certain Services/Applications and provide access to them in a standard way. It is able to perform integrations in an interoperable manner which, in the past, has always been very expensive.

Before focusing on the design of a Service Layer within an N-layer application, we will provide an introduction to SOA.

2.- SERVICE ORIENTED ARCHITECTURES AND N-LAYER ARCHITECTURES

It is worth noting that **SOA** trends do not contradict ***N-Layered architectures.*** On the contrary, they are complementary architectures. **SOA** is a high level architecture that defines "how" some applications intercommunicate (Services) with others. Simultaneously, each one of the **SOA services/applications** can be internally structured following the design patterns of the ***N-Layer architecture***.

SOA tries to define standard corporate communication buses between the different applications/services of a company, and even between services on the Internet owned by different companies.

The following diagram shows a *standard communication bus* example (following SOA trends) with several corporate applications integrated:

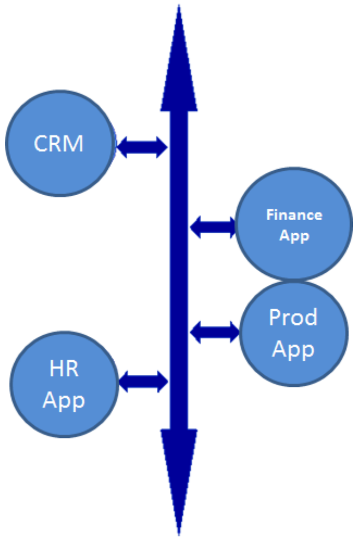


Figure 2.- SOA and Enterprise Service Bus

Take into account that a Service Bus is not mandatory for SOA, but in many cases it can be very useful.

Each SOA Service/Application has to have an internal implementation where the application business logic, data access and entities (states) are implemented. Additionally, the Service input/output communication is based on messages (SOAP or REST messages).

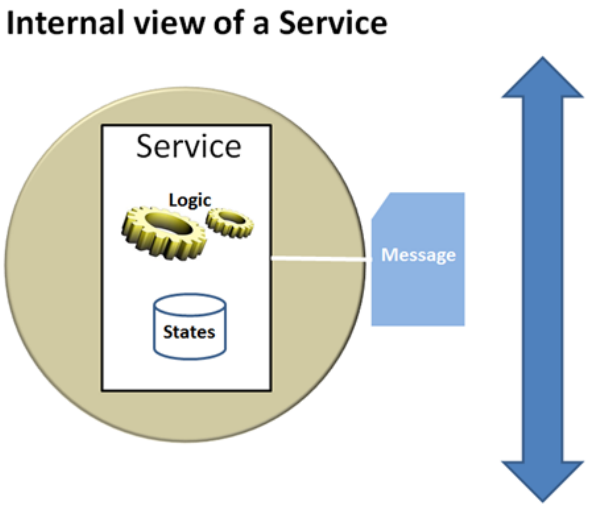


Figure 3.- Internal View of a distributed service

This internal implementation is normally carried out (structurally) following the design patterns of the logical N-layer architectures and physical distribution (deployment in servers) according to N-Tier architecture.

At a deeper level, the specific N-Layered Architecture for that SOA Service could be aligned with the layered architecture we propose in this guide, that is, a DDD NLayered Architecture, following DDD trends. This point will be explained later in more detail.

3.- N-LAYERED ARCHITECTURE RELATIONSHIP WITH ISOLATED APPLICATIONS AND SOA SERVICES

The internal architecture of an SOA service can therefore be similar to that of an isolated application, that is, implementing the internal architecture of both (SOA service and isolated Application) as an ***N-Layer architecture*** (component logical N-layer architecture design).

The main difference between them is that an SOA service is seen from "the outside" (from another external application) as something "not visual." By contrast, an isolated application will also have a **Presentation layer** (that is, the "client" part of the application to be used visually by the end-user).

Keep in mind that **an "independent and visual" application may also be simultaneously a SOA service which could be publishing its components and business logic to other external applications**.

The order we will follow in this guide is: first, an explanation of the basis of SOA Architecture and second, an explanation of the implementation of Distributed Services with **WCF** (*Windows Communication Foundation).*

4.- RELATIONSHIP BETWEEN DDD BOUNDED-CONTEXTS AND SOA SERVICES

It is interesting how similar a SOA Service is to a Bounded-Context. In fact, a SOA Service should be treated as a Bounded-Context. Both concepts require a high degree of isolation, and **both even require most of the old *SOA tenets***:

- **Boundaries** must be **explicit**
- **Services** must be **autonomous** (**Autonomous** development lifecycle).
- Services share schema and contract, not classes/types
- Service compatibility is determined based on policy

5.- WHAT IS SOA?

SOA (*Service Oriented Architecture*) is an evolution of object oriented programming (OOP) and applies aspects learned over time in the development of distributed software.

The reasons for the appearance of SOA are basically the following:

- Integration between applications and platforms is difficult
- Certain systems are heterogeneous (different technologies)
- There are multiple integration solutions, which are independent and unrelated to each other.

A standard approach is necessary, which can provide the following:

- Service oriented architecture
- Based on an interoperable "common messaging" system
- Standard for most platforms

'*Service orientation*' is different from '*Object orientation*', primarily in how it defines the term '*application*'. The "Object oriented development" is focused on applications whose construction is based on libraries of interdependent classes. SOA, however, emphasizes systems that are constructed on the basis of a set of autonomous services. This difference has a profound impact on the assumptions that can be made about development.

A "service" is simply a program used to interact via messages. A set of services installed/deployed would be a "system". Individual services should be constructed consistently (availability and stability are crucial to a service). An aggregated/composite system composed by various services should be constructed to allow change and evolution of these services and the system should be adapted to the presence of new services that appear over time after the services and original clients have been deployed/installed. Furthermore, these changes should not break the functionality of the current system.

Another aspect to note is that a SOA-Service should be, as a general rule, interoperable. Therefore, it should be based on standard specifications at the **protocol levels, serialized data format in communication levels, etc**.

Currently, there are two main trends of Architecture regarding Web Services:

- SOAP (WS-I, WS-* specifications)
- REST (and RESTful services)

SOAP is based on **SOAP messages,** logically. These messages are composed by **XML**, following a specific schema (format). SOAP uses **HTTP** as the communications protocol.

REST is highly oriented to the URI and the Internet. The addressing of resources is based on the HTTP URL and simpler data formats, therefore exchanging messages is simpler and lighter than with SOAP XML messages.

6.- INTERNAL ARCHITECTURE OF THE SOA SERVICES

SOA aims to solve problems of distributed application development. A “Service” can be described as an application that exposes an interface based on messages, encapsulates data and also manages ACID transactions (Atomic, Consistent, Isolated and Durable), with their respective data sources. Typically, SOA is defined as a set of service providers that expose their functionality through public interfaces (which can also be protected/secured). The interfaces exposed by the service providers can be used individually or by adding several services and forming composite service providers.

The SOA services may also provide RPC style interfaces, if required. However, the “synchronized request-response” scenarios should be avoided whenever possible. On the contrary, the asynchronous consumption of Services should be favored.

If using a DDD approach for building a SOA-Service, we can construct it internally by the following layers:

- **Service interface**
- **Application and Domain layers**
- **Data Persistence Infrastructure layer**

In the following diagram we show how the above sample service would be internally structured:

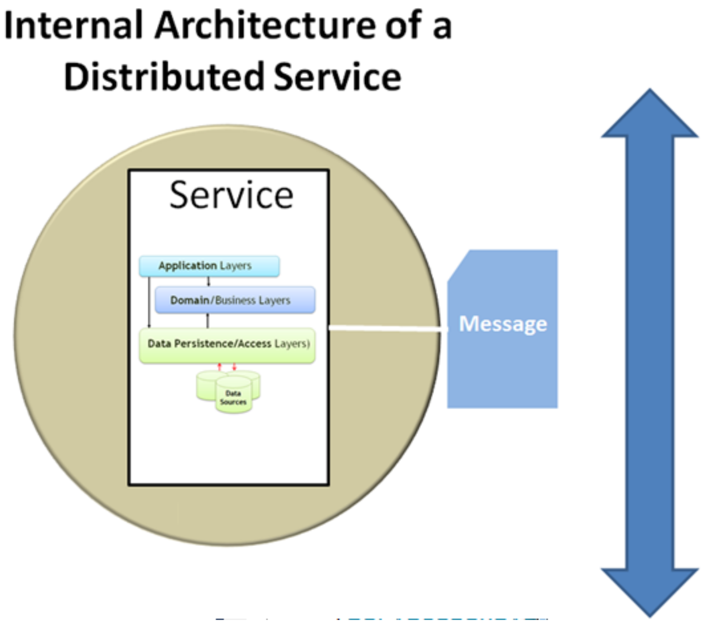

Internal Architecture of a Distributed Service
Service
Application Layers
Domain/Business Layers
Data Persistence/Access Layers)
Data Sources
Message

Figure 4.- Logical layers of a Service

It is very similar when compared to the internal architecture of an N-layer application. The difference is that, logically, a service does not have a presentation layer.

The 'Interface' is placed between the service clients and the facade of service processes. One single service can have several interfaces, such as a *Web-Service* based on HTTP, a message queue system (like MSMQ), a WCF service with binding based on TCP (a TCP port chosen by us), etc.

Normally, a distributed service should provide a coarse grained interface instead of a fine-grained interface. That is, the intention is to perform the highest number of actions within a single method in order to minimize the number of remote calls from the client (round-trips).

In addition, the services are frequently stateless (without state or an internal object life relative to each external call) although they do not always have to be so. A basic *Web Service* (WS-I specifications) is *stateless*, but WCF advanced services (WS-* specifications or Net proprietary) may also have shared objects and states such as the *Singleton*, *Session* types, etc.).

7.- DATA OBJECT TYPES TO BE TRANSFERED

We should determine how we will transfer the entity data through the physical boundaries of our Architecture (Tiers). In most cases, when we want to transfer data from one process to another and especially from one server to another, we must serialize data.

We could also use this serialization when going from one logical layer to another, However, this is generally not a good idea, since we will have penalizations in performance.

In general and from a logical point of view, the data objects to be transferred from one tier to another can be classified as follows:

- **Scalar Values**
- **DTOs (Data Transfer Objects)**
- **Serialized Domain Entities**
- **Sets of records (disconnected artifacts or data sets)**

All of these types of objects must have the capacity to be serialized (to XML, JSON or binary format) and transferred over the network.

Scalar Values

If we are going to transfer a very small amount of data (like a few input arguments) it is quite common to use scalar values (such as int, string, etc.). On the other hand, even when we have a few parameters it is a better practice to create a complex type (a class) merging all those parameters.

Serialized Domain Entities

When we are dealing with volumes of data related to domain entities, a first option (and the most immediate one) is to serialize and transmit their own domain entities to the presentation layer. This may be good or bad, depending on the context. In other words, if the implementation of the entities is strongly linked to a specific data access technology, it is contrary to the DDD Architecture recommendations because we are contaminating the entire architecture with a specific technology. However, we have the option of sending domain entities that are POCO (Plain Old CLR Objects), that is, serialized classes that are 100% custom code and do not depend on any data access technology. In this case, the approach can be good and very productive, because we could have tools that generate code for these entity classes for is. In addition, the work can be streamlined because even these entities can perform concurrency handling tasks for us. This concept (Serializing and transferring Domain Entities to other Tiers / physical levels) will be discussed in the chapter about Web Services implementation.

On the other hand, this approach (Serialization of Domain entities themselves) has the disadvantage of leaving the service consumer directly linked to the domain entities, which could have a different life cycle than the presentation layer data model and even different changing rates. Therefore, this approach is suitable only when we maintain direct control over the whole application (including the client that consumes the web-service), or when the application is a medium size with no several BOUNDED-CONTEXTS, like a typical medium N-Tier application. On the other hand, when implementing SOA services for unknown consumers or when it is a more complex DDD or Composite application it is usually a better option to use DTOs, as explained below.

DTOs (*Data Transfer Objects*)

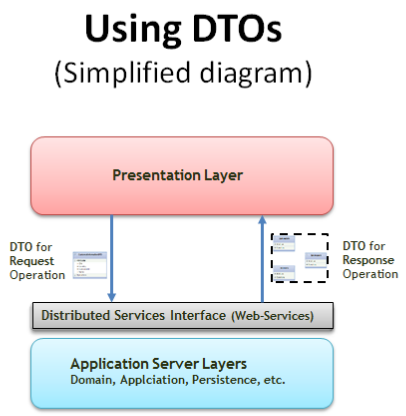
To decouple clients/consumers of Web Services from the Domain Entities, the most common option is to implement DTOs (*Data Transfer Objects*). This is a design pattern that consists in packaging multiple data structures in a single data structure to be transferred usually between “physical boundaries” (remote communication between servers and/or machines).

A DTO is a simple POCO class that exposes properties but no methods. A DTO is useful whenever you need to group values in structures for passing data through your application assets and especially when providing data to external systems.

From a pure design perspective, DTOs are a solution really close to perfection. DTOs help to further decouple presentation from the service layer and the domain model. The downside is that they require manual programming.

A layer of DTOs isolates the domain model from the presentation, resulting in both loose coupling and optimized data transfer.

DTOs are especially useful when the application that uses our services has a data representation or even a model that does not exactly match the Domain Entity Model. This pattern allows us to change the Domain entities (internal implementation in our application Server) as long as the interfaces of Web Services and the DTOs structure do not change. So, in many cases, changes in the server will not affect the consumer application. It also supports a more comfortable version management towards external consumers. This design approach is, therefore, the most suitable when there are external clients/consumers using data from our web-services and when the development team does not have control over the development of these client applications (Consumer client applications could be developed by others).



Using DTOs
(Simplified diagram)
Presentation Layer
DTO for Request Operation
DTO for Response Operation
Distributed Services Interface (Web-Services)
Application Server Layers
Domain, Applciation, Persistence, etc.

Figure 5.- DTOs diagram (Data Transfer Objects)

The typical design of DTOs tries to adapt to the hypothetical needs of the consumer (either presentation layer, or another type of external application). It is also important to design them so that they minimize the number of calls to the web service (minimize round-trips), therefore improving the performance of the distributed application.

Working with DTOs requires having certain adaptation/conversion logic from DTOs to Domain entities and vice versa. In DDD N-layered architecture, these Adapters would be typically placed by us in the Application layer, like we already mentioned within the 'Application Layer' chapter.

Sets of Records/Changes (*disconnected devices*)

The sets of records/changes are usually implementations of disconnected complex data, such as DataSets in .NET. They are mechanisms that are very easy to use. <u>However, they are closely linked to the underlying technology and tightly coupled components regarding the data access technology</u> As such, they are completely contrary to the DDD approach (Domain Layer isolated from the infrastructure layers) and would not be recommended in this type of domain oriented architecture. They are more likely to be recommended in architectures for less complex applications and to be developed in a more RAD manner (*Rapid Application Development*).

Any of these logical concepts (Entity, DTO, etc.) may be serialized to different types of data (XML, binary, different XML formats/schemes, etc.), depending on the specific implementation chosen. However, this implementation is already associated with technology, so we will analyze this later in the section about Distributed Services Implementation in .NET.

References about DTOs

Pros and Cons of Data Transfer Objects (Dino Esposito)
http://msdn.microsoft.com/en-us/magazine/ee236638.aspx

Building N-Tier Apps with EF4 (Danny Simons):
http://msdn.microsoft.com/en-us/magazine/ee335715.aspx

8.- CONSUMPTION OF DISTRIBUTED SERVICES BASED ON AGENTS

The Service Agents basically establish a sub-layer within the client application (Presentation Layer) which centralizes and locates the “consumption” of Web Services in a methodical and homogeneous manner, instead of directly consuming the Services from any area of the client application (form, page, etc.). Ultimately, the use of agents is a way to design and program (pattern) the consumption of Web services.

Definition of Service Agent

“A Service Agent is a component located in the presentation layer, acting as the front-end of communications towards Web Services. It should be solely responsible for actions of direct consumption of Web Services”.

An agent could also be defined as a “*smart-proxy*” class that is an intermediary between a service and its consumers. Consider that the Agent is physically placed on the client side.

From the point of view of the client application (WPF, Silverlight, OBA, etc.), an agent acts “in favor” of a Web-Service. That is, as if it was a local “mirror” offering the same functionality of the service in the server.

Below we show a diagram with agents consuming a Distributed Service:

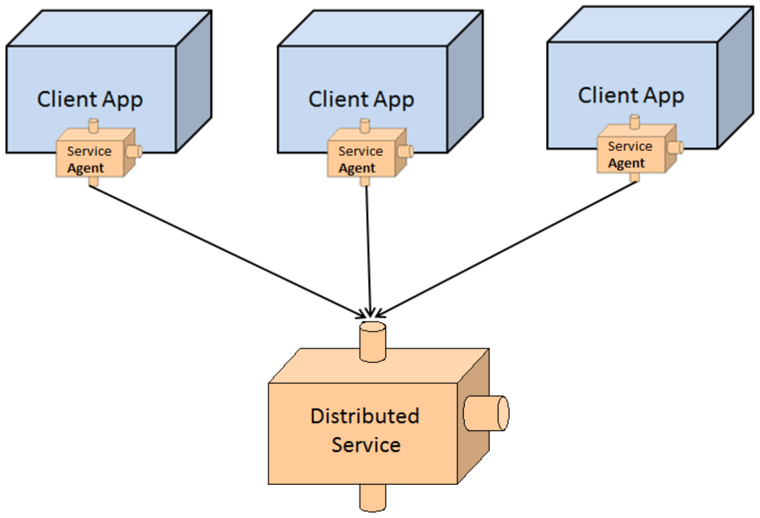


Figure 7.- Diagram of the agents in a Service “Consumption” architecture

The **Agent** should help prepare service requests and interpret the responses from the service.

It is important to consider that an agent is not a part of the service (it must be loosely coupled to the distributed service) and therefore the service must not trust the agent. All the interaction between an agent and a service should be authenticated, authorized and validated by the service in the same way in which a service is accessed directly without an agent.

Some of the advantages of using Agents are:

- **Easy integration**: If a Service has its corresponding Agent, providing this developed agent to whomever is going to consume the service may simplify the development process.

- **Mocking**: Mocking a Web Service is very useful, especially when the Service exposes a system which is not always available during development time, such as a Host, corporate ERP, etc. We should be able to test our client application in an isolated way. The Service Agent would be the place to implement the fake web service.

- **Error Handling**: Reacting correctly to the error conditions is essential and one of the most complex tasks for the developer that uses a Web service from an application. The Agents should be designed to understand mistakes a service can make, greatly simplifying the development of subsequent integrations. One issue when using Web services is the difficulty on raising exceptions that the client can handle. The usual approach is to return a specific value that indicates the occurrence of an exception or failure. Therefore, one useful feature that a Service Agent (which wraps the Web-Service consumption code to add extra functionality) can offer is detecting exceptions by examining the returned value and raising a local exception to the calling code.
  'Retry logic' could also be added to Service-Agents, in order to handle time-outs, intermittent connection fallings, etc.

- **Offline data management and cache**: An agent may be designed to make a "cache" of data for the service correctly and so it can be understood. This can sometimes dramatically improve response times (and therefore performance and scalability) of the requests and even enable applications to work offline.

- **Request validations**: Agents can verify the input data sent to the server components and ensure they are correct before making any remote call (cost in latency to the server). This in no way exempts the server from having to validate data, since the safest way is in the server (the client may have been hacked) but it can normally save time.

- **Intelligent routing**: Some services may use agents to send requests to a specific service server, based on the contents of the request.

In short, the concept is very simple; the agents are classes located in an *assembly* on the client side and they are the only classes on this side that should interact with the proxy classes of the Services. On a practical level, the usual way is to create a class library project in order to implement these Agent classes. After that, we simply have to add a reference to this *assembly* in the client application.

Before reviewing other aspects, it is important to emphasize that the use of Agents does not depend on technology. This pattern can be used for consuming any type of Distributed Service.

9.- INTEROPERABILITY

The main factors affecting interoperability of the applications are the availability of proper communication channels (standard) and formats and protocols that can be understood by the parties involved in different technologies. Consider this guideline:

- In order to communicate with most platforms and devices from different manufacturers, the use of standard protocols and standard data formats are recommended, such as HTTP and XML, respectively. Bear in mind that decisions on protocol may affect availability of consumer clients. For example, target systems can be protected by Firewalls that block some protocols.

- The chosen data format may affect interoperability. For example, the target systems may not understand specific data types related to a technology (for example, ADO.NET Datasets are hardly consumed from JAVA applications) or they may have different ways of managing and serializing the data types.

- The selected communications security can also affect interoperability. For example, some encryption/decryption techniques (like 'message based security') may not be available in many consumer systems.

10.- PERFORMANCE

The design of communication interfaces and data formats to be used will have a considerable impact on the application performance, especially when we cross "boundaries" in communication between different processes and/or different machines. There are techniques we can use to improve performance related to communications between different application tiers.

Consider the following guidelines and best practices:

- Minimize the volume of data transmitted through the network, this reduces overload during objects serialization (For instance, server paging 'is a must').

- **Coarse-grained** Web Services: It is important to bear in mind that we should always **avoid** working with **fine-grained** Web Service interfaces (this is how the internal components are usually designed within the Domain). This is problematic because it forces us to implement the consumption of Web Services in a “chatty” way. This type of design strongly impacts performance because it forces the client application to make many remote calls (many round-trips) for a single global operation and since remote calls have a performance cost (latency because of Web Service activation, data serialization/de-serialization of data, etc.), it is critical to minimize the round-trips. In this regard, the use of DTOs is best when deemed convenient (it allows grouping of different Domain entities into a single data structure to be transferred) although ORM technologies (such as ‘Entity Framework’) also allow serialization of graphs containing several entities.

- Consider using a Web Services Facade which provides a **coarse-grained interface**, encapsulating the Domain components that usually are fine-grained designed.

- If web-service data serialization (XML serialization) impacts on the application performance, consider using binary serialization (although binary serialization is usually not interoperable with other technical platforms).

- Consider using other protocols (such as TCP, Named-Pipes, MSMQ, etc.). In most cases, they substantially improve communication performance. However, we may lose HTTP interoperability.

11.- ASYNCHRONOUS VS. SYNCHRONOUS COMMUNICATION

We should consider the advantages and disadvantages of communicating with Web services in a synchronous vs. asynchronous manner.

Synchronous communication is appropriate for scenarios where we must guarantee certain operations sequence or when the user must wait to see the requested information (although this last point can also be obtained through asynchronous communication).

Asynchronous communication is suitable for scenarios where the response from the application must be immediate or in scenarios where there is no guarantee that the target is available.

Consider these guidelines when deciding on synchronous or asynchronous communications:

- We should consider an **asynchronous communication model** to obtain the highest performance and scalability, a nice loosely-coupled architecture regarding the back-end, and to minimize the system load. . If some clients can only make synchronous calls, a component can be implemented (Service Agent in the client) that is synchronous towards the client but can use web services in an asynchronous manner. This provides the possibility of making different calls at the same time, increasing the overall performance in that area.

- In cases where we must ensure the sequence in the execution of operations or when operations that depend on the outcome of previous operations are used, the most suitable scheme is probably **synchronous communication**. In most cases, a synchronous operation with a certain request can be simulated with coordinated asynchronous operations. However, depending on the particular scenario, the effort put forth in implementing it may or may not be worth the trouble.

- If asynchronous communication is chosen but network connectivity and/or availability of destination cannot always be guaranteed, consider using a system of "saving/sending" messaging **that ensures communication** (**Message queue** system, such as MSMQ), to avoid missing messages. These message queue advanced systems can even extend transactions by sending asynchronous messages to the message queues. If, in addition, you need to interoperate and integrate with other business platforms, consider the use of integration platforms (such as Microsoft BizTalk Server.)

12.- REST VS. SOAP

REST (*Representational State Transfer*) and SOAP (*Simple Object Access Protocol*) **represent two very different styles** to implement a Distributed Service. Technically, REST is a pattern of architecture constructed with simple verbs that fit perfectly with HTTP. Although REST architecture principles could apply to protocols other than HTTP, in practice, REST implementations are fully based on HTTP.

SOAP is a messaging protocol based on XML (SOAP messages with a specific XML format). It can be used with any communications protocol (Transport) including HTTP.

The main difference between these two approaches is the way the service state is maintained. We are referring to a very different state from that of session or application. We are referring to the different states that an application goes through during its lifetime. With SOAP, changing through different states is made by interacting with a single service endpoint which encapsulates many operations and message types.

On the other hand, with REST, we have a limited number of operations and these operations are applied to resources represented and addressed by URIs (HTTP addresses). The messages are composed by current resources states or the required resources state. REST works very well with Web applications where HTTP can be used as protocol for data types other than XML (like JSON). The service consumers interact with the resources through URIs in the same way people can navigate and interact with Web pages through URLs (web addresses).

The following diagram shows what scenarios REST or SOAP fit better:

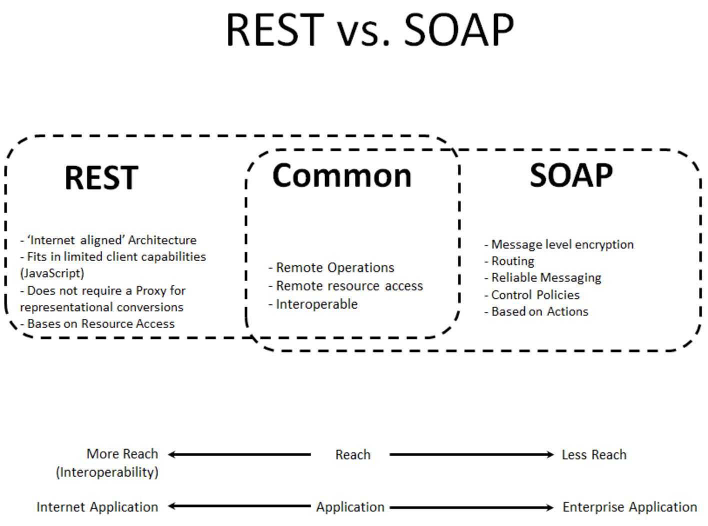

REST vs. SOAP
REST
- 'Internet aligned' Architecture
- Fits in limited client capabilities (JavaScript)
- Does not require a Proxy for representational conversions
- Bases on Resource Access
Common
- Remote Operations
- Remote resource access
- Interoperable
SOAP
- Message level encryption
- Routing
- Reliable Messaging
- Control Policies
- Based on Actions
More Reach (Interoperability)
Reach
Less Reach
Internet Application
Application
Enterprise Application

Figure 8.- REST vs. SOAP scenarios

From a technical point of view, these are some of the advantages and disadvantages from both of them:

SOAP advantages:

- Strongly typed proxies thanks to WSDL
- Works over different communication protocols. The use of protocols other than HTTP (such as TCP, NamedPipes, MSMQ, etc.), can improve performance in certain scenarios.

SOAP disadvantages:

- SOAP messages are not 'cacheable' by CDNs. This is a very serious handicap for high scalable applications in the Internet.
- SOAP messages are not JavaScript friendly (For AJAX, JQuery, etc. REST is the best choice).

REST advantages:

- Governed by HTTP specifications, so the services act as Resources, such as images or HTML documents.
- Data can be maintained strictly or decoupled (not as strict as SOAP)
- REST resources can be easily used from the JavaScript code (AJAX, JQuery, etc.)
- Messages are light, so the performance and scalability offered are high. This is important for many Internet applications.
- REST can use XML or JSON as data format.
- REST messages are 'cacheable' by CDNs thanks to the HTTP GET verb being used like any HTTP-HTML web-site. This is a great advantage for high scalable applications in the Internet.

REST disadvantages:

- Working with strongly typed objects is harder, although this depends on technological implementations and it is improving in the latest versions of most platforms.

- Only works over HTTP, and REST calls are restricted to HTTP verbs (GET, POST, PUT, DELETE, etc.). But, it this really a disadvantage for the Internet and interoperable solutions?

Even though both approaches (REST and SOAP) may be used for similar types of services, the approach based on REST is normally more suitable for Distributed Services that are publicly accessible (Internet) and require high scalability or in cases where a Service can be accessed by unknown consumers. SOAP, on the contrary, is still a good approach for implementing procedural implementation ranges, such as an interface between the different layers of Application architecture or, ultimately, private Business Applications.

SOAP does not limit to HTTP. The standard specification WS-*, which can be used on SOAP, provides a standard and therefore interoperable path to work with advanced aspects such as SECURITY, TRANSACTIONS, ADDRESSING AND RELIABLE-MESSAGING. But, frankly, these advanced WS-* specifications have not been broadly adopted and used by most applications in the Internet and even when they are standard specifications, its degree of compatibility between different technical platforms is not really high, mostly due to its complexity.

REST also offers a great level of interoperability (due to the simplicity of its protocol); however, for advanced aspects, such as those previously mentioned, it would be necessary to implement their own mechanisms, which would be non-standard.

In short, both protocols allow us to interchange data by using verbs. The difference lies in the fact that, with REST, this set of verbs is restricted to coincidence with HTTP verbs (GET, PUT, etc.) and in the case of SOAP, the set of verbs is open and defined in the Service endpoint.

Consider the following guidelines when choosing one approach or the other:

- SOAP is a protocol that provides a messaging framework that a layer abstraction can be built on, and it is mostly used as an RPC calls system (synchronous or asynchronous) where data is transferred as XML messages.

- SOAP manages aspects, such as security and addressing, through its internal implementation of SOAP protocol.

- REST is a technique that uses other protocols, such as JSON (JavaScript ObjectNotation) and Atom as a publication protocol, and simple and light formats of the POX type (Plain Old XML).

- REST enables the use of standard HTTP calls such as GET and PUT to make queries and modify the state of the system. REST is stateless by nature, which means each individual request sent from the client to the server must contain all the necessary information in order to understand the request, since the server will not store data about the state of the session.

12.1.-Design Considerations for SOAP

SOAP is a protocol based on messages that is used to implement the messages layer of a Web Service. The message consists of an “envelope” with a header and a body. The header can be used to provide information external to the operation to be performed by the service (e.g., security aspects, transactional aspects or message routing, included in the header).

The body of the message has contracts, in the form of XML schemes, which are used to implement the Web service. Consider this design guideline which is specific to SOAP Web Services:

- Determine how to manage errors and faults (Normally exceptions generated in internal layers of the server) and how to return the proper information on errors to the Web Service consumer. (See “Exception Handling in Service Oriented Applications” in **http://msdn.microsoft.com/en-us/library/cc304819.aspx**.)

- Define the schemes of operations that can be performed by a service (Service Contract), the structures of data passed when requests are made (Data Contract) and errors and faults that can be returned from a request to the Web service.

- Choose a proper security model. For more information, see “*Improving Web Services Security*: Scenarios and Implementation Guidance for WCF” in **http://msdn.microsoft.com/en-us/library/cc949034.aspx**

- Avoid using complex types with dynamic schemes (such as Datasets). Try to use simple types, DTO classes or entity classes to maximize interoperability with any platform.

12.2.-Design Considerations for REST

REST represents an architecture style for distributed systems and is designed to reduce complexity by dividing the system into resources. The resources and operations supported by a resource are represented and exposed through a set of URIs (HTTP addresses) logically on the HTTP protocol. Consider this guideline specifically for REST:

- Identify and categorize the resources that will be available for Service consumers

- Choose an approach for representation of resources. A good practice would be using names with meaning (Ubiquitous language in DDD?) for REST input points and unique identifiers for specific resource instances. For example, **http://www.mycompany.employee/** represents the input point to access an employee and **http://www.mycompany.employee/smith01** uses an employee ID to indicate a specific employee.

- Decide if multiple representations for different resources should be supported. For example, we can decide if the resource should support an XML format, Atom or JSON and make it part of the resource request. A resource may be exposed by multiple representations.

- Decide if multiple views for different resources will be supported. For example, decide if the resource should support GET and POST operations or simply GET operations. Avoid excessive use of POST operations, if possible.

- Do not implement user session state maintenance within a service and do not try to use HYPERTEXT to manage states. For example, when a user makes a request to add an item to the shopping cart of an e-commerce company, the cart data must be stored in a persistent state storage or a cache system prepared for that purpose, but not in memory as states of the own services (which would also invalidate scalable scenarios of a "Web Farm").

13.- INTRODUCTION TO SOAP AND WS-*

SOAP, originally defined as ‘Simple Object Access Protocol’, is a specification for exchanging information structured in the Web Service implementation. It is specially based on XML as message formats and HTTP as communication protocols (But it can use other communication protocols, as well).

SOAP is the stack base of Web Service protocols, providing a basic frame of messaging on which the Web Services can be built.

This protocol is defined in three parts:

- A message **envelope**, that defines the contents of the body or contents of the message and how to process it
- A set of **serialization** rules to express instances of application data types
- A **conversation** to represent calls and answers to remote methods

In short, it is a system of remote calls based on XML messages at a low level. A SOAP message will be used both for requesting the execution of a method of remote Web Service and for using another SOAP message as an answer (containing the requested information). Due to the fact that the data format is XML (text, with scheme, but text finally), it can be used from any platform or technology. SOAP is interoperable.

The basic standard of SOAP is ‘SOAP WS-I *Basic Profile’*.

14.- WS-* SPECIFICATIONS

Basic web services (such as SOAP WS-I, *Basic Profile*) offer more than communications between the Web service and the client applications that use it. However, the standards of basic web services (*WS-Basic Profile*) were just the beginning of SOAP.

Transactional and complex business applications require many more functionalities and service quality requirements (QoS) than simple communications between client and web services. The following needs are usually required by business applications:

- Message base security or mixed security in communications, including authentication, authorization, encryption, non-tampering, signature, etc.
- Reliable messaging
- Distributed transactions support between different services.
- Routing and addressing mechanisms.

- Metadata to define requirements as policies.
- Support to attach large volumes of binary data when invoking web services (images and/or attachments of any kind).

To define all these "advanced needs", the industry (different companies such as Microsoft, IBM, HP, Fujitsu, BEA, VeriSign, SUN, Oracle, CA, Nokia, CommerceOne, Documentum, TIBCO, etc.) has been and continues to be defining some theoretical specifications that set forth how the extended aspects of the Web services should operate.

All these "theoretical specifications" are known as **WS-* specifications.** (The '*' is given because there are many advanced web services specifications, such as *WS-Security*, *WS-SecureConversation*, *WS-AtomicTransactions*, etc.) To learn more about these specifications, you can review the standards in: **http://www.oasis-open.org**.

In short, these WS-* specifications theoretically define the advanced requirements of the business applications.

The following scheme shows the different functionalities the WS.* tries to solve at a high level.

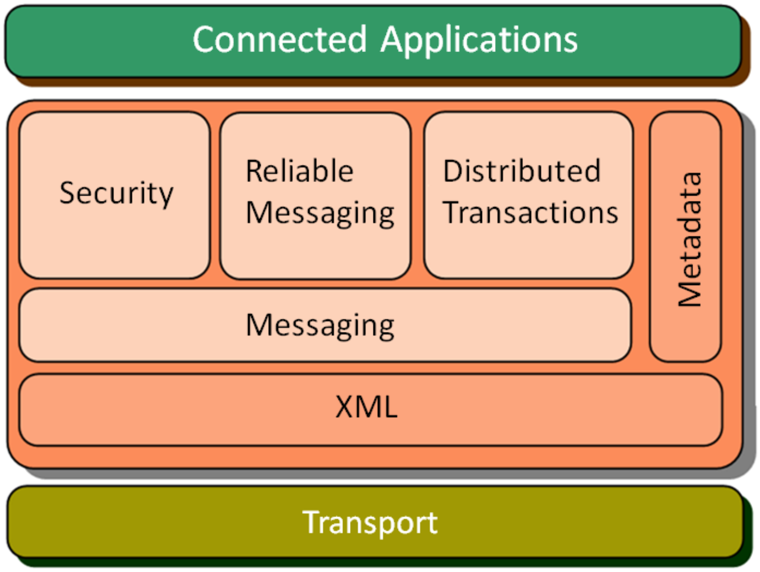


Figure 9.- WS-* functionalities Diagram.

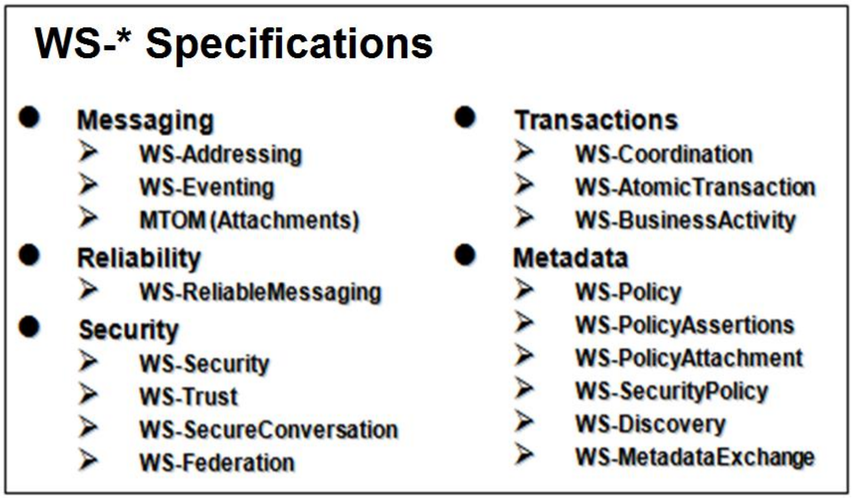
All the central modules (Security, Reliable Messaging, Transactions and Metadata) are precisely the functionalities the basic XML Web services do not have, and what defines WS.*.

The **WS.* specifications** are therefore formed by subsets of specifications:

- WS-Security

- WS-Messaging
- WS-Transaction
- WS-Reliability
- WS-Metadata

They, in turn, are subdivided into other subsets of specifications, which are deeply defined, as shown below:



WS-* Specifications

- Messaging
  - WS-Addressing
  - WS-Eventing
  - MTOM (Attachments)
- Reliability
  - WS-ReliableMessaging
- Security
  - WS-Security
  - WS-Trust
  - WS-SecureConversation
  - WS-Federation
- Transactions
  - WS-Coordination
  - WS-AtomicTransaction
  - WS-BusinessActivity
- Metadata
  - WS-Policy
  - WS-PolicyAssertions
  - WS-PolicyAttachment
  - WS-SecurityPolicy
  - WS-Discovery
  - WS-MetadataExchange

Figure 10.- WS-* specifications

Therefore, WS-* specifications are virtually a whole world, they are not just a small extension to the basic Web Services.

Below we show a table with the needs in SOA business distributed applications and the WS-* standards that define them:

Table 2.- WS-* specifications

Advanced needs in services	WS-* Specifications that define them
• Advanced, including different types of authentication, authorization, encryption, non-	- WS-Security - WS-SecureConversation

tampering, signature, etc.	- WS-Trust
• Stable and reliable messaging	- WS-ReliableMessaging
• Distributed Transactions support between different services	- WS-AtomicTransactions
• Addressing and routing mechanisms	- WS-Addressing
• Metadata to define requirements as policies	- WS-Policy
• Support to attach great volumes of binary data when invoking services (images and/or attachments of any type)	- MTOM

15.- INTRODUCTION TO REST

What is REST?, REST was introduced by *Roy Fielding* in a speech where he described an "architecture style" of interconnected systems. Also, REST is the acronym of "*Representational State Transfer*".

The Web is a set of resources. From an Internet point of view (we're still not focusing on REST), a resource is an element of interest. For example, Microsoft can define a type of resource on a product of its own, which could be Microsoft Visual Studio. In this case, anyone can access this resource using his browser with an URL such as:

http://www.microsoft.com/products/visualstudio

To access this resource, a **representation** of the resource will be returned (e.g., *visualstudio.htm*). This **representation** places the client application in a **state**. The result of the client accessing a link within said HTML page will be another accessed resource. The new representation will place the client application in another state. So the client application changes (**transfers**) the state with each resource representation. In short, there is a "*Representational State Transfer*".

So REST's goal is to show the natural features of the Web that made the Internet a success, but applying those principles to Web Services.

REST is not a standard, it is an architecture style. We probably won't see W3C publishing a REST specification, because REST is only an architecture style. A style cannot be packaged but only understood, and web services can be designed accordingly. It is comparable to an N-tier architecture style, or SOA architecture. We don't have an N-Tier standard or SOA standard.

However, although REST is not itself a standard, it is based on the Internet standards:

- HTTP
- URL
- XML/JSON/HTML/PNG/GIF/JPEG/etc (Resources representations)
- Text/xml, text/html, image/gif, etc. (MIME type)

15.1.-The URI in REST

In conclusion and as an essential concept in REST, one of the most important points in REST is the URI. The purpose of an URI (Uniform Resource Identifier) is to identify a resource on the network and to have a way of manipulating it. It uniquely identifies a web resource and makes it addressable (capable of being used by HTTP). A URI identifies only one Resource, but a Resource can have more than one URI.

URLs and URNs are are special forms of URIs. A URI that identifies the mechanism by which a resource can be accessed is called as URL. HTTP URIs are URLs.

In other words, we can explain it with several examples of WS URIs based on REST. As seen, the definition is self-explanatory, which is one of the objectives of REST; simplicity and self-explanation, so we will not explain such sample URIs as:

http://www.mydomain.com/Providers/GetAll/
http://www.mydomain.com/Providers/GetProvider/2050
http://www.mydomain.com/Providers/GetProviderByName/Smith/Joe

As stated, we will not explain this, considering how easy it is to understand.

15.2.-Resources and Resource Representation

Resources are therefore the fundamental building blocks of REST-Based systems. A resource is anything we expose to the Web.

A **Resource Representation** is a *view* of a resource's state at an instant time, encoded in any format like XML, JSON, etc.

Web consumers exchange representations, they should never access the underlying resource directly. The separation between the representations and their related resources helps on a loose coupling between backend systems and consuming apps. It also helps with scalability, since a representation (when using the GET HTTP verb) can be cached and replicated in the Internet.

There could be different *representations/views* depending on conditions, like depending on the identity accessing a resource, providing different information or even a different format (JSON, XML, etc.).

Regarding **Representation formats**, in REST, each resource representation based on any format, should not have a specific URI per each format, like for instance, the URI should not end with an extension like '.xml' or 'json'. The web should be intelligent enough in order to negotiate the representation format. In other words, a single URI must address multiple resource representations negotiating, like populating the HTTP ACCEPT request header with a list of format types prepared to process.

15.3.-Simplicity

Simplicity is one of the fundamental aspects in REST. Simplicity is pursued in any aspect, from the URI to the XML messages sent or received from the Web service.

This simplicity is a big difference as compared with SOAP, which is quite complex in its headers, etc.

The benefit of this simplicity is the ability to achieve good performance and efficiency because it is light (even when we are working with less than efficient standards, such as HTTP and XML). In the end, however, the data (bits) transmitted are always those of minimum necessity. We have something light, so performance will be quite optimal. On the other hand, if we are based on something quite simple, then complex capabilities (which can be done using SOAP) are almost impossible to achieve using REST. For example, advanced security standards, signature and message level encryption, Distributed transactions between several web services and many other advanced functionalities that are defined in WS-* specifications based on SOAP.

But the goal of REST is not to achieve large or complex functionality, but to achieve a minimum functionality needed by a great percentage of web services on the Internet that are interoperable; that simply transmit the information and are very efficient.

Below we show an example of a REST message returned by a Web Service. The simplicity is in contrast to the SOAP WS-* message that may be quite complex and therefore heavier. REST messages are very light:

```
<?xmlversion="1.0"?>
<p:Clientxmlns:p="http://www.mycompany.com"
   xmlns:xlink="http://www.w3.org/1999/xlink">
<Client-ID>00345</Client-ID>
   <Name>Smith & Brothers</Name>
   <Description>Great company</ Description>
   <Details xlink:href="http://www.mycompany.com
/clients/00345/details"/>
</p:Client>
```

Evidently, it is difficult to design a more simplified XML message than the one above. It is interesting to note the "Details" item of the example, which is that of a link or hyperlink type. That is called Hypermedia, acting as the engine of application state. Next possible transitions are discovered by those links.

15.4.-Logical URLs versus Physical URLs

A resource is a conceptual entity. A representation is a concrete manifestation of this resource. For example:

http://www.mycompany.com/customers/00345

The above URL is a logical URL, not a physical URL. For example, there is no need for an HTML page for each client in this example.

A correct design aspect of URIs in REST is that the technology used in URI/URL should not be disclosed. There should be freedom to change implementation without affecting the client applications that are using it. In fact, this involves a problem for

WCF services hosted in IIS, since these services usually work based on a .svc page. However, in the latest WCF version, we can work using REST with no .svc extension.

15.5.-Core characteristics of REST Web Services

- Client-Server: “pull” interaction style. Complex methods of communications, of the Full-Duplex or Peer-to-Peer type, cannot be implemented with REST. REST is for simple Web services.

- *Stateless*: each request that is made by the client to the server must have all the information necessary to understand and execute the request. No type of server context should be used. This is what the basic Web services are also like in .NET (single-call, stateless); however, in WCF there are more types of instantiation, such as Singleton and shared instantiation (with sessions). This also cannot be implemented with a REST Web service.

- *Internet Cache*: Using the HTTP verb GET, queries can be cached in Internet caches. Therefore, for high scalable public applications it is probably de best choice.

- Uniform interface: all resources are accessed with a generic interface (for example: HTTP GET, POST, PUT, DELETE); however, the most important or predominant interface in REST is GET (like the URLs shown in the above example). GET is considered as “special” for REST.

- The content type is the object model

- Image, XML, JSON, etc.

- Named resources. The system is limited to resources that can be named through an URI/URL.

- Representations of interconnected resources: representations of resources are interconnected through URLs; this enables the client to go from one state to the next.

15.6.-Design Principles of REST Web Services

1. The key to creating web services in a REST network (e.g., the Web on the Internet) is to identify all the conceptual entities to be exposed as services. We saw some examples earlier, such as clients, products, invoices, etc.

2. Create an URI/URL for each resource. Resources should be nouns, not verbs. For example, the following URI would be wrong:

   http://www.mycompany.com/customers/getcustomer?id=00452

   The verb "GetCustomer" would be wrong. Instead, only the name would appear, like this:

   http://www.mycompany.com/customers/customer/00452

3. Categorize resources according to whether the client applications can receive a representation of the resource, or whether client applications can modify (add) to the resource. For the former item, the resource should be made accessible with a HTTP GET, for the latter item, the resources should be made accessible with HTTP POST, PUT and/or DELETE.

4. The representations should not be isolated islands of information. That is why links should be implemented within the resources to allow client applications to search for more detailed or related information.

5. Design to gradually reveal data. Do not reveal everything in a single document response. Provide links to obtain more details.

6. Specify the format of the response using an XML scheme (W3C Schema, etc.)

16.- REST VS. SOAP (PRESENT, FUTURE AND RECOMENDATIONS)

Both approaches can be used, but we (personal opinion of the authors of this guide) really think that the mainstream, especially for the future and everything related to the Internet, is REST.

SOAP and WS-* are great for internal business applications with a high level of complex requirements related to security and many other subjects supported by SOAP and WS-*. But, the mainstream is always going to be affected by what happens in the Internet, Cloud-Computing and high-scalability requirements. For those last subjects, because of the nice use of HTTP Caches by REST services, lightweight messages and even because of its internal simplicity linked to HTTP and how the Web works, we really think that REST is going to be the mainstream trend when talking about Web-Services and SOA, especially when dealing with Internet scalable apps.

REST demands an important change regarding **HTTP**:

'***We must embrace HTTP as an Application protocol, not only as yet another transport protocol***'.

NOTE ABOUT .NET: Using current .NET 4.0 platform, developing Web Services either SOAP or REST approaches, is quite straight forward. On the other hand, API for consumer apps, it is actually easier when consuming SOAP Services (using Web-Service References, proxy classes, complex types, etc.), but this is going to change in the near future with **Microsoft WCF WebAPI** for **REST**: **http://wcf.codeplex.com/**

Additional Resources

- "Enterprise Solution Patterns Using Microsoft .NET" in
  - **http://msdn.microsoft.com/en-us/library/ms998469.aspx**
- "Web Service Security Guidance" in
  - **http://msdn.microsoft.com/en-us/library/aa480545.aspx**
- "Improving Web Services Security: Scenarios and Implementation Guidance for WCF" in
  - **http://www.codeplex.com/WCFSecurityGuide**
- "WS-* Specifications" in **http://www.ws-standards.com/ws-atomictransaction.asp**

17.- ODATA: OPEN DATA PROTOCOL

OData is a higher-level concept than SOAP and REST. It is also the most recent, as it is a proposed standard for high level protocols based on **REST** and **AtomPub**.

Let's start from the beginning. **What exactly is OData?**

Table 3.- OData definition

	Definition
OData Open Data Protocol	**OData** (*Open Data Protocol*) is a web protocol to perform queries and remote updates to access services and data stores. OData emerged based on the **AtomPub** experiences of server and client implementations. OData is used to expose and access information from different resources, including, but not limited to, relational databases. Actually, it can publish any type of resource.

OData is based on certain conventions, especially on **AtomPub** using data oriented **REST** services. These services share resources identified through the use of URIs (*UniformResourceIdentifiers*) and defined as an abstract model of data to be read/queried and edited by clients of such Web services HTTP-REST.

OData consists of a set of specifications such as [OData:URI], [OData:Terms], [OData:Operations], [OData:Atom], [OData:JSON] and [OData:Batch].

So, OData is a **high level protocol designed to share data in the network**, especially in public and interoperable Internet environments. In short, it is a higher level than REST, but following the same trend, using URIs to identify each piece of information in a service, HTTP to transport requests and answers and AtomPub and JSON to manage sets and representation of data.

The main goal of OData is to offer a standard manner of using data via the network and getting consumers of data services to use a series of high level conventions that would be of much interest if widely adopted. Ultimately, using schemes and predefined conventions instead of "reinventing the wheel" during development and birth of each distributed or web service.

Finally, keep in mind that OData is a standard proposed by Microsoft that is born initially from protocols used originally in **ADO.NET Data Services** (currently called ***WCF Data Services***), but the interesting part is that Microsoft has made it evolve and released it through the OSP (*Open Specification Promise*) so that any manufacturer can create implementations of OData.

The benefits of OData as proposed open standard protocol are interoperability and collaboration with other platforms, as well as how it can be implemented by any platform that supports HTTP, XML, AtomPub and JSON. For example, **IBM Web Sphere** is one of the products and manufacturers that support OData (the service called *IBM WebSphereeXtremeScale REST* supports OData), along with many Microsoft products and technologies, primarily the following:

- Base technology/implementation of Microsoft OData
  - ***WCF Data Services***
- Higher level products and technologies:
  - *Windows Azure Storage Tables*
  - *SQL Azure*
  - *SharePoint 2010*
  - *SQL Server Reporting Services*
  - *Excel 2010 (with SQL Server PowerPivot for Excel)*

For the complete list, see **http://www.odata.org/producers**

Due to its nature (REST, Web, AtomPub and interoperability) it is highly oriented to publication and use of data in heterogeneous environments and the Internet and therefore, ‘Data Oriented’ services instead of ‘Domain Oriented’ (DDD). In a complex and private business application, implementation of its internal distributed services is probably more powerful using SOAP and WS-* specifications (Security, transactions, etc.). However, a ‘Domain Oriented’ application may want to publish information to the outside (other applications and/or initially unknown services). That is where OData fits perfectly as an additional access interface to our ‘Domain Oriented’ application/service from the outside world, other services and ultimately "the network".

Currently, in our implementation of the sample application related to the present Architecture (*Domain-oriented N-Layered*) we do not use OData because DDD does not offer a ‘*Data Oriented*’ architecture/application, but one that is ‘*Domain Oriented*’. Moreover, the distributed services are being essentially used from another layer of our application (Presentation layer within our application), so it is more flexible to use SOAP or even REST at a lower level. OData is more oriented to publishing data directly as CRUD services (*Create-Read-Update-Delete*), with pre-set specifications, which is based on *WCF Data Services*.

Finally, because OData is really strategic for Microsoft, in the future OData could evolve towards many more scenarios further than Data-Driven Services/Apps. Keep an eye on **http://odata.org** for more details.

18.- GLOBAL DESIGN RULES FOR SOA SYSTEMS AND SERVICES

Table 4.- Global Design Rules

Rule Nº: D22	**Identify what server components should be SOA services**

- **Rule**
- Not all the components of an application server should be accessed exclusively by Distributed Services.
- **Bear "the end"** in mind, **not "the means"**.
- The components that have **business value** and are **reusable in different applications** and those that should necessarily be accessed remotely (because the Presentation layer is remote, Windows Client Type) should be identified as **SOA Services**.
- If the presentation layer is remote (e.g., Silverlight, OBA, AJAX, etc.), a "Distributed Service Interface" should be published through Services.
- The goal of "transparency" and "interoperability" is achieved.

Table 5.- Global Design Rules

Rule Nº: D23	**The internal Architecture of a service should usually follow the guidelines of N-layer architecture**

- **Rule**
- Each independent service must be internally designed in accordance with the N-layer architecture, similar to the one described in this guide.

Table 6.- Global Design Rules

Rule Nº: D24	**Identify the need to use DTOs vs. serialized Domain Entities, as data structures to communicate between different tiers or physical levels**

○ **Rule**

- This rule means that we have to identify when it is worth the excessive effort of implementing DTOs and DTO adapters versus the direct use of serialized Domain entities.

- In general, if the party that uses our services (client/consumer) is controlled by the same development team as the server components, it will be much more productive to use serialized Domain Entities. However, if the consumers are external, initially unknown and not under our control, or even if the application is quite large and pure DDD oriented, the decoupling offered by DTOs will be crucial and the excessive effort of implementing them will really be worthwhile.

References:

Pros and Cons of Data Transfer Objects (Dino Esposito)
http://msdn.microsoft.com/en-us/magazine/ee236638.aspx

Building N-Tier Apps with EF4 (Danny Simons):
http://msdn.microsoft.com/en-us/magazine/ee335715.aspx

Table 7.- Global Design Rules

Rule Nº: D25	The boundaries of Services must be explicit

- ○ <u>**Rule**</u>

- Whoever develops the **client application** that uses a service should be aware of when and how a service is remotely used in order to consider scenarios of errors, exceptions, low band width on the network, etc. All of this should be implemented in the **Service Agents.**

- Web Services interfaces should be **coarse-grained**, minimizing the number of round-trips from the client application to the Service.

- Maintain <u>**maximum simplicity in the service interfaces.**</u>

Table 8.- Global Design Rules

Rule Nº: D26	Services must be independent in pure SOA architectures

- ○ <u>**Rule**</u>

- <u>In a pure SOA architecture, services should be designed, developed and versioned independently</u>. The services must not depend heavily on their life cycles with respect to applications that use them. In general this requires the use of DTOs (Data contracts) in order to decouple internal DOMAIN MODELS from cosumers.

Table 9.- Global Design Rules

Rule Nº: D28 **Context, composition and state of global SOA services**

- **<u>Rule</u>**

- If the **SOA services** we are treating are GLOBAL SERVICES (to be used by "n" applications), then they should be designed so that **they ignore the context from which they are being "consumed".** This does not mean the Services cannot have a state (stateless), but rather that they should be independent from the context of the consumer, because each consumer context will, in all likelihood, be different.

- '**Loosely coupled**': the SOA Services that are GLOBAL can be reused in "client contexts" which may not be known at the time of design.

- **Value can be created when composing Services** (e.g., booking a holiday with a flight-booking service, with another service to book a car and another to book a hotel).

For more general information on SOA concepts and patterns to be followed, see the following references:

<u>SOA and Service references</u>:

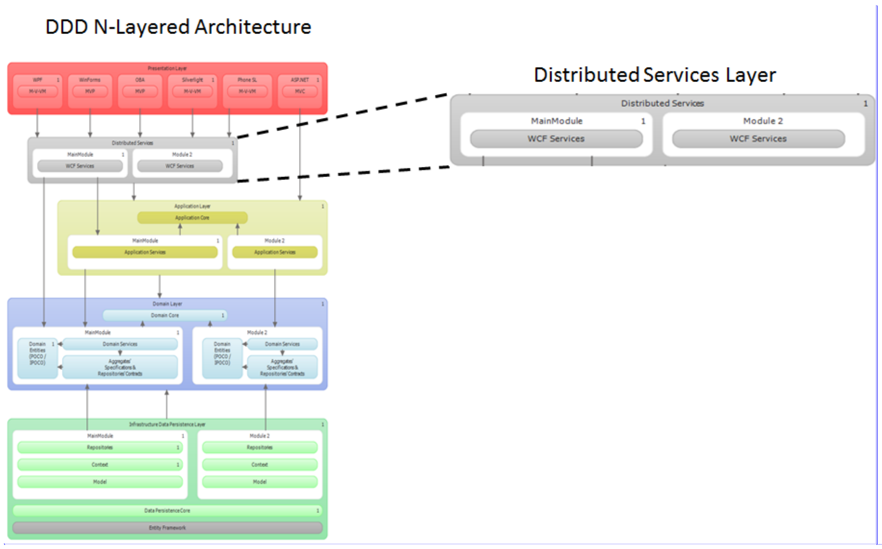
Service pattern
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesServiceInterface.asp

Service-Oriented Integration
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/archserviceorientedintegration.asp

19.- IMPLEMENTING THE DISTRIBUTED SERVICES LAYER WITH WCF 4.0

The purpose of this chapter is to show different options we have at the technology level to implement the Distributed Services layer and of course, to explain the technical options chosen in our .NET 4.0 reference architecture.

We highlight the Location of the Distributed Services layer in the Architecture diagram shown below:



Comment [CESARDL1]: Cha
latest LAYER diagram

Figure 11.- Distributed Services layer in Layer diagram - Visual Studio 2010

There are several possibilities for the implementation of Distributed Services with Microsoft technology, as we will analyze below. However, the most powerful technology is WCF (*Windows Communication Foundation*), so this is how we recommend implementing this layer within our proposed architecture.

20.- TECHNOLOGICAL OPTIONS

In a Microsoft platform, we can currently choose between two message oriented base technologies and Web services:

- *ASP.NET Web Services (ASMX)*

- *Windows Communication Foundation (WCF)*

As well as other derivative technologies of a higher level:

- *Workflow-Services (''WCF+WF'')*

- *RAD (Rapid Application Development)*:

    - *WCF Data.Services (aka. ADO.NET DS)*

        - *Implementation of OData from Microsoft.*

    - *WCF RIA Services*

However, it is not feasible to use RAD technologies for this architecture where we need decoupling between components of the different layers, as RADs are usually tightly-coupled and are also usually more 'data oriented' technologies. That is why the only two options to be initially considered are the core technologies with which we can implement Web services: WCF or ASP.NET ASMX and in some cases *Workflow-Services*.

20.1.-WCF Technology

WCF provides decoupled technology in many ways (protocols, data formats, hosting process, etc.), providing very good configuration control. Consider WCF in the following cases:

- Web services to be created require interoperability with other platforms that also support SOAP and/or REST, such as JEE application servers

- You require either SOAP Web-Services or *REST* based Services.

- A higher performance is required in communications and support for REST, SOAP messages or binary formats.

-

- WS-* specifications support requirement, like the following:
    - WS-Security implementation is required to implement authentication, data integrity, data privacy and message-based encryption.
    - The implementation of WS-MetadataExchange is required in SOAP requests to obtain descriptive information on services, such as its WSDL definitions and policies.
    - The implementation of 'WS-ReliableMessaging' is required to implement end to end reliable communications, even performing a route between the different intermediates of Web services (not just a point to point origin and destination).
    - Consider WS-Coordination and WS-AT (AtomicTransaction) to coordinate *'two-phasecommit'* transactions in the context of Web Services conversations.
      See: http://msdn.microsoft.com/en-us/library/aa751906.aspx
- WCF supports several communication protocols:
    - For public services, those of the **Internet** and those that are interoperable, consider HTTP and specifically **REST** would be the best **scalable** approach because of 'Internet cache' use.
    - For services with higher **performance** and end to end .NET, consider **TCP**
    - For services used within the same machine, consider **named-pipes**
    - For services that must ensure communication, consider **MSMQ**, which ensures communication through messages queues

20.2.-ASMX technology (Web ASP.NET services)

ASMX provides a simpler and older technology for developing Web services, although it is also an older technology and more coupled/linked to certain technologies, protocols and formats.

- ASP.NET web services are exposed through IIS Web server
- It can only be based on HTTP as a communication protocol
- It does not support transactions distributed between different web services

- It does not support advanced standards of SOAP (WS-*), it only supports the SOAP WS-I Basic Profile

- It provides interoperability with other platforms that are not .NET through SOAP WS-I, which is interoperable.

20.3.-Technology Selection

To implement simple web services, ASMX is very easy to use. However, for the context we are addressing (Domain oriented complex business applications), **we strongly recommend the use of WCF** for its greater flexibility regarding technical options (standards, protocols, formats, etc.).Ultimately it is much more powerful than ASP.NET .ASMX web services.

20.4.- Types of WCF Service deployment

The Distributed Services (which is ultimately the whole server application) can be deployed at the same physical tier (same servers) being used for other layers such as a web presentation layer or it can be deployed in a separate tier (other servers) specifically for application/business logic. This last option is often required by security policies or even for scalability reasons (under certain special circumstances).

In most cases, the Service layer will reside in the same level (Servers) as the Domain layers, Application layers, Infrastructure layers, etc. to maximize performance. If we separate the layers into different physical tiers we are adding some latency caused by remote calls. Consider the following guidelines:

- **Deploy the Service Layer in the same physical tier as the Domain, Application, Infrastructure layers, etc**., to improve performance of the application, unless there are security requirements and policies that prevent it. This is the most common case for N-Tier architectures with RIA and Rich Clients.

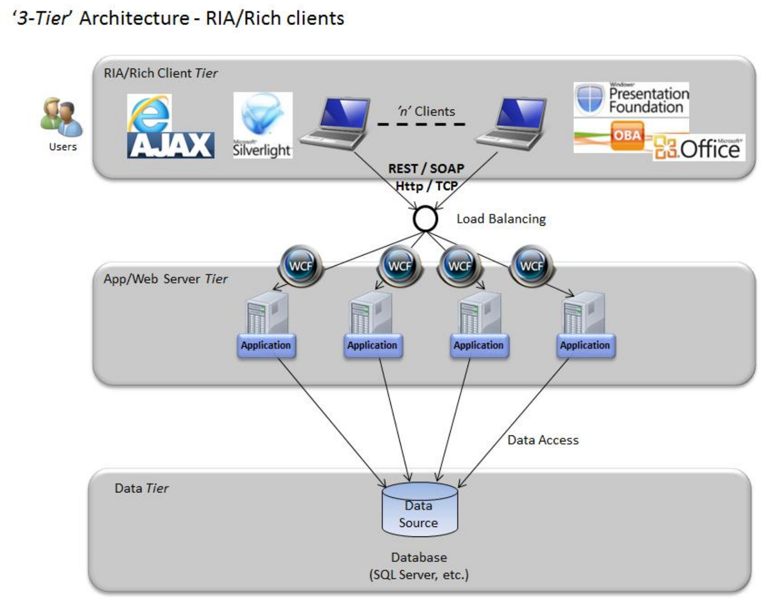


Figure 12.- RIA/Rich Clients remotely accessing to WCF services

- **Deploy the Service layer in the same physical tier as the Presentation layer if this layer is a Web presentation layer (like ASP.NET),** to improve application performance. Separating it from the ASP.NET web tier should only be done for security reasons or because of certain special scalability reasons that are not so common and must be demonstrated. If the Web services are located in the same physical level as the consumer, consider using *named-pipes* as communications protocol. However, another option in this case could be not to use Web Services and using objects directly through the CLR. This is probably the option that offers the best performance. It is pointless to use Distributed Services if we consume them from within the same machine. According to **Martin Fowler**: '*The first law of distributed programming, is "Do not distribute" (unless absolutely necessary).*' However, this approach could be preferable at times for the sake of homogeneity, if we do not want to maintain several versions of the same software, and we prefer to maintain fully SOA software.

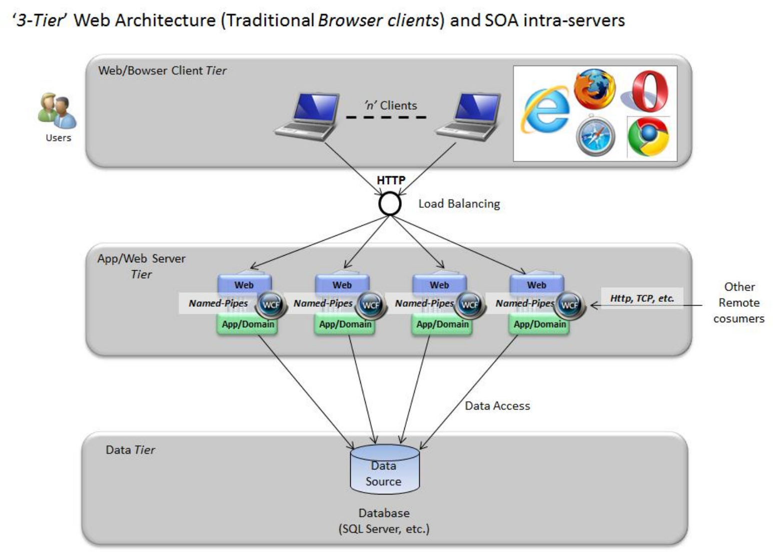


Figure 13.- App-Web with intra-server WCF services, reusable for external calls from other remote consumers

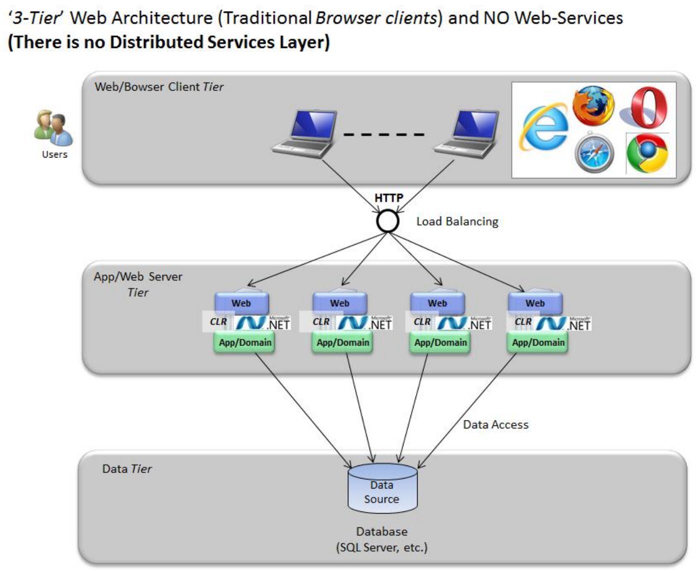
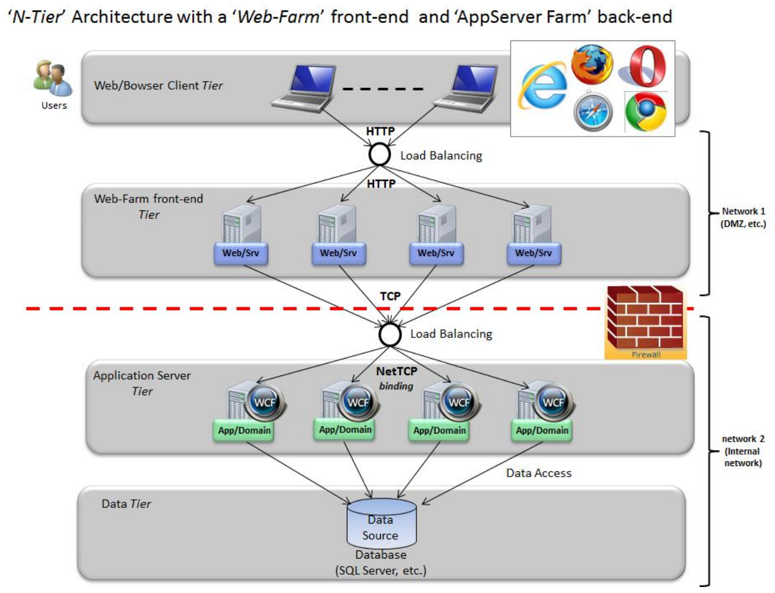

'3-Tier' Web Architecture (Traditional Browser clients) and NO Web-Services
(There is no Distributed Services Layer)
Web/Bowser Client Tier
Users
HTTP
Load Balancing
App/Web Server Tier
Web
CLR
.NET
App/Domain
Data Access
Data Tier
Data Source
Database (SQL Server, etc.)

Figure 14.- App-Web without Distributed Service layer

- **Deploy the Service layer in a different physical layer than the Web presentation layer**. In certain situations separating visual Web front-ends from distributed services back-ends may increase scalability, although this must be proven with a load test. Any introduction of remote communication is, by default, a reason for loss of performance due to latency introduced in communications, so the contrary must be proven if you want to separate layers in different physical servers. Another reason for separating the visual Web front-end (ASP.NET) from the Applications Server (Web services) may be for security reasons and corporate policies, such as having separate public and private networks for internal components. In this case, there is no problem in separating these tiers.



- If consumers are .NET applications within the same internal network and highest performance is desired, consider TCP binding in WCF for communications.

- If the service is public and interoperability is required, use HTTP

20.5.-Service Hosting and configuration (Bindings)

Once we have developed our service (we will see a proposed implementation based on our sample app, later on), we should select a host (process) where our service can be executed (do not forget that, so far, our service is simply a .NET class library, a .DLL that cannot be executed on its own). This hosting process can be almost any type of process; it may be IIS, a console application, a Windows service, etc.

To decouple the host from the service itself, it is convenient to create a new project in the Visual Studio solution that defines our host. Different projects could be created, depending on the hosting type:

- Hosting in IIS/WAS/AppFabric: **Web Application project**
- Hosting in an executable app-console: **ConsoleApplication project**
- Hosting in a 'Windows Service': **Windows Service project**

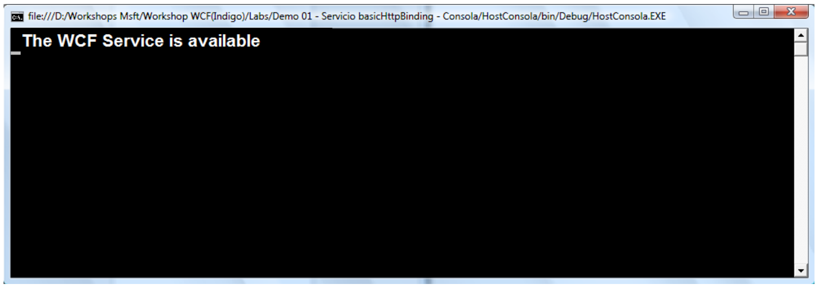
WCF flexibility allows us to have several host projects hosting the same service. This is useful because we can host it in a console application during the development stage of the service. This facilitates debugging and, subsequently, a second project of the web type can be added to prepare the deployment of the service in an IIS or Windows service.

As an example, we will create a console project as a hosting process and we will add a reference to the **System.ServiceModel** library.

Below, in the Main class, the **ServiceHost** object should be instantiated by passing the *Greeting* service type as a parameter.

```
using System.ServiceModel;
static void Main(string[] args)
{
  Type serviceType = typeof(Greeting);
  using (ServiceHost host = new ServiceHost(serviceType))
  {
    host.Open();

    Console.WriteLine("The WCF Service is available.");
    Console.WriteLine("Press a key to close the service");
    Console.ReadKey();

    host.Close();
  }
}
```

In the example above we can see how the service host is defined. If we start it, the process will be running until the user presses a key. At the same time, the WCF Service will be 'listening' and waiting for any request:


file:///D:/Workshops Msft/Workshop WCF(Indigo)/Labs/Demo 01 - Servicio basicHttpBinding - Consola/HostConsola/bin/Debug/HostConsola.EXE

```
The WCF Service is available
_
```

Figure 19.- WCF Service hosted in a Command-Console App (Just for demos/tests)

This execution of the console application by hosting the WCF service actually cannot be performed until we also have the configured service through the XML sections of the .config, as we shall see in a moment. Once we have configured the host with the endpoint information, WCF can then build the necessary runtime to support our configuration. This happens at the moment Open() is called on a particular ServiceHost, as we can see in the code C# above (host.Open();)

After ending the method Open(), the WCF runtime is already constructed and ready to receive messages at the specified address based on each *endpoint*. During this process, WCF generates an *endpoint listener* for each configured endpoint. (An *endpoint*-listener is a piece of code that actually listens and waits for incoming messages by using the specified transport and address).

Information can be obtained on the service at run time through the object model exposed by the **ServiceHost** class. This class allows us to obtain anything we want to know about the initialized service, such as what *endpoints* are exposed and what *endpoints listeners* are currently active.

It is important to emphasize that the use of "console applications" as hosting process for WCF services should be used only for proof of concepts, demos, and testing services and never, of course, for WCF services in production. A deployment of WCF service in a production environment would normally be performed in a hosting process of any of the following types:

Table 12.- Possibilities of WCF Service Hosting

Context	Hosting process	Protocols	Req. Operative System
Client-Service	IIS 6.0	http/https	- Windows Server 2003 - Windows XP - (or later Windows version)
Client-Service	Windows Service	Tcp Named-Pipes MSMQ http/https	- Windows Server 2003 - Windows XP - (or later Windows version)
Client-Service	**IIS7.x-WAS AppFabric**	**Tcp** **Named-Pipes** **MSMQ** **http/https**	- **Windows Server 2008 or later version**
			- Windows Vista
Peer-To-Peer	WinForms or WFP client	Peer-to-Peer	- Windows Server 2003 - Windows XP - (or later Windows version)

IIS 7.x, in Windows Server is usually the preferred option for production environments.

20.6.-WCF Service Configuration

However, in order to run a service we first need to configure it. This configuration may be performed in two ways:

- *Hard-coded*: Configuration specified by C#/VB code.

- Service Configuration based on configuration files (**.config*). This is the most highly recommended option, since it offers flexibility to change service parameters such as addresses, protocols, etc. without recompilation. Therefore, it facilitates deployment of the service and ultimately provides greater simplicity because it allows us to use the **Service Configuration Editor** utility provided by Visual Studio.

So far, we had already defined the contract (interface), implementation (class) and even the hosting process within the '**ABC' model** But now it is necessary to associate this contract with a specific binding, address and protocol. This will normally be done within the .config file (app.config if this is our own process, or web.config if IIS is the selected *hosting* environment).

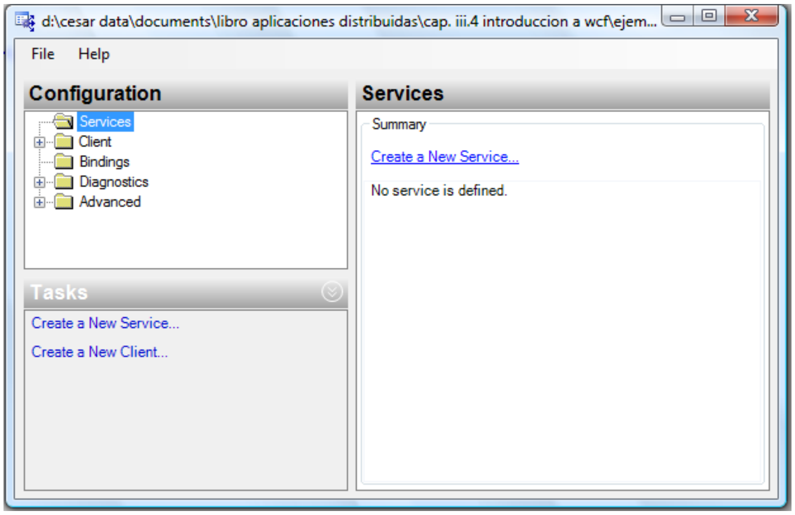
So, a very simple example of an app.config XML file is the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
…
<system.serviceModel>
<services>
<service name="MyCompany.MyApplication.MyWcfService.Greeting">
<endpoint

address="http://localhost:8000/ServiceGreeting/http/"
                   binding="basicHttpBinding"

contract="MyCompany.MyApplication.MyWcfService.IGreeting"
                   bindingConfiguration="" />
</service>
</services>
</system.serviceModel>
…
</configuration>
```

Of course, app.config or web.config may have additional XML sections. In the app.config above, we can clearly see the '**ABC of WCF'**: **Address, Binding and Contract.**

Keep in mind that in this example our service is "listening" and offering service via HTTP (specifically through **http://localhost:8000**). However we are not using a web server such as IIS; This is simply a console application or it could also be a Windows service that also offers service through HTTP. This is because WCF provides internal integration with **HTTPSYS**, which allows any application to become an ***http-listener***.

The ".config" file configuration can be done manually by writing the XML directly into the .config (we recommend doing it like this, because this is how you really learn to use bindings and their configurations).Or, if we are starting with WCF or even if there are things we do not remember, such as XML configuration, we can do it through the Visual Studio wizard. This wizard is available in Visual Studio.



No service is defined.

Figure 20.- WCF Configuration Wizard of Visual Studio

To do this in our example console project once we have added an *app.config* file, right-click on this .config file and select *"Edit WCF Configuration…"*. The **Service Configuration Editor** window will be displayed.

21.- IMPLEMENTATION OF WCF SERVICE LAYER IN N-LAYER ARCHITECTURE

First of all, we would like to highlight that the protocols and bindings that our WCF implementation in the SampleApp, is actually temporal, as it is currently based on basicHttp binding (basic SOAP specifications). Our intention is to migrate it to REST, which is more interoperable with other lighter clients like HTLM5/Jscript/JQuery. But, because probably the best approach for most consumers will be **WCF WebAPI** (**http://wcf.codeplex.com/**) and it is still in beta state and even its beta still does not support our current client (Silverlight), we will change that in the future.

On the other hand, the internal part of the service will be quite similar whether using SOAP or REST. In any case, our WCF services are the entry point to our Application Server, the point where all Dependency Injection starts and the area where we are exposing our Application Layer Services as a Façade for remote clients. This is almost the same concept, no matter if we use REST or SOAP. So we are going to focus on that.

In our sample application there is a solution tree similar to the following, where we highlight the location of the WCF services:

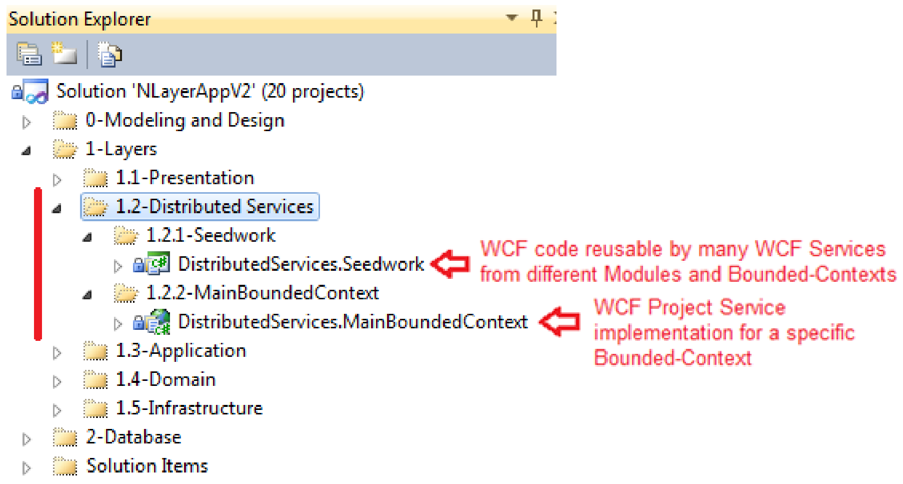
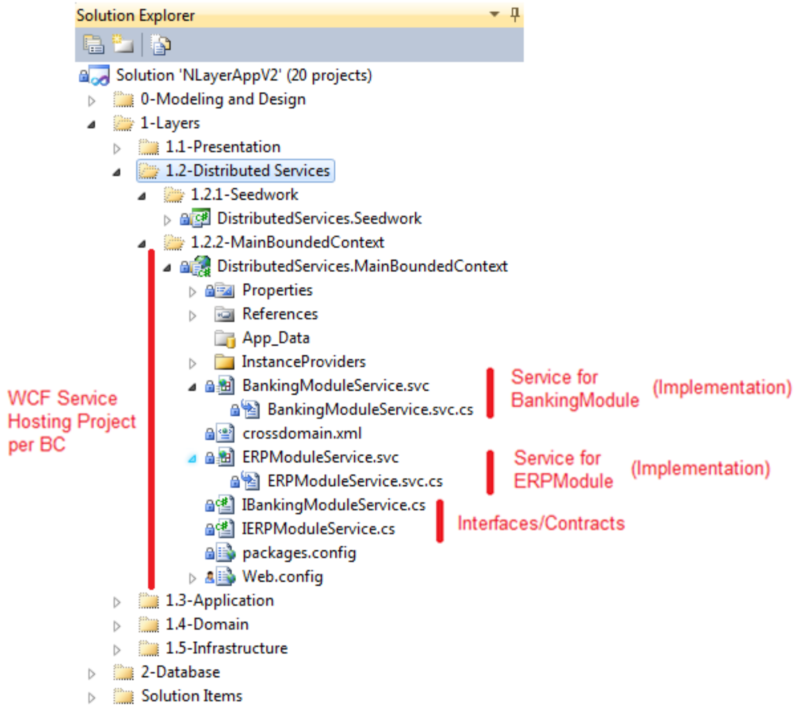

Solution Explorer
Solution 'NLayerAppV2' (20 projects)
0-Modeling and Design
1-Layers
1.1-Presentation
1.2-Distributed Services
1.2.1-Seedwork
DistributedServices.Seedwork
WCF code reusable by many WCF Services from different Modules and Bounded-Contexts
1.2.2-MainBoundedContext
DistributedServices.MainBoundedContext
WCF Project Service implementation for a specific Bounded-Context
1.3-Application
1.4-Domain
1.5-Infrastructure
2-Database
Solution Items

Figure 21.- WCF service projects location tree

There may be a class library with reusable code for many different WCF services from different Modules and Bounded-Contexts. In this example, that library is called "*DistributedServices.Seedwork*" and contains certain reusable code for WCF *Faults* and WCF Exceptions handling.

As shown above, for each application the Distributed Service Layer will be comprised by a single hosting project called '*DistributedServices.MainBoundedContext*' (one by default, but there could be several types of hosting, depending on the needs). In this case, we chose a Web project that will be hosted in IIS server (or *Cassini*, the VS Web Server, during development environment).

For each vertical module within our BOUNDED-CONTEXT, we could have a different WCF service (When using SOAP, we could have one .svc and related class per MODULE in order to have a better organization). In this case, we have two modules, therefore we have two WCF Service classes and .svc (.svc files shouldn't be used for REST services, as it is a not so pure REST design). These files are shown below.


Solution Explorer
Solution 'NLayerAppV2' (20 projects)
0-Modeling and Design
1-Layers
1.1-Presentation
1.2-Distributed Services
1.2.1-Seedwork
DistributedServices.Seedwork
1.2.2-MainBoundedContext
DistributedServices.MainBoundedContext
Properties
References
App_Data
InstanceProviders
BankingModuleService.svc
BankingModuleService.svc.cs
crossdomain.xml
ERPModuleService.svc
ERPModuleService.svc.cs
IBankingModuleService.cs
IERPModuleService.cs
packages.config
Web.config
1.3-Application
1.4-Domain
1.5-Infrastructure
2-Database
Solution Items

A WCF Service can be all defined within a single project, both for hosting and for Service implementation (like the one we show, which is coupled to a Web-Service project), or it could be implemented in a Class library (WCF Service classes and Interfaces) and then another project only for hosting purposes. Because we will be using IIS and mostly HTTP, especially if implemented based on REST, we don't need to split it in two projects (hosting project plus implementation project). But when having several different hosting processes, then that could be a good idea.

22.- TYPES OF DATA OBJECTS TO COMMUNICATE WHEN USING WCF SERVICES

As we explained in the chapter on Distributed Services (at the logical level), and in order to unify options, the usual types of data objects to communicate when passing from one tier to another remote tier within an N-Tier architecture are:

- Scalar values
- DTOs (*Data Transfer Objects*)
- Serialized Domain Entities
- Sets of records (disconnected artifacts)

With these options, at the technology level we would see the following mapping:

Table 13.- Mapping Options

Logical object type	.NET technology
Scalar values	- String, int, etc.
DTOs (Data Transfer Objects)	- Custom plain .NET data classes gathering data related to one or more entities
Serialized Domain Entities coupled to a data access infrastructure technology	- Simple/native entities of Entity Framework (this was the only direct possibility in EF 1.0)
Serialized Domain Entities NOT coupled to the data access infrastructure technology	- **POCO entities mapped to Entity Framework entity model** (available from EF 4.0) - ***SELF-TRACKING* entities mapped to Entity Framework entity model** (although STEs are not directly deriving from EF classes and all the STEs are part of our code, they require .NET/Silverlight on the client

	and their interfaces depends on a specific way of working) - Other POCO entities
Sets of records depending on the data access infrastructure technology (disconnected devices) Concept 1	- **DataSets, DataTables** of ADO.NET

Although simple/native EF Entities are normally not the best pattern for N-Tier applications (they have direct dependence on EF technology), it was the most viable option in the first version of EF (EF 1.0). However, EF4 significantly chenged the options for N-Tier programming. Some of the new features are:

1.- New methods that support offline operations, such as *ChangeObjectState* and *ChangeRelationshipState*, which change an entity or relation to a new state (for example, added or modified); *ApplyOriginalValues*, which allows us to establish original values of an entity and the new event *ObjectMaterialized* that is activated each time the *framework* creates an entity.

2.- Compatibility with POCO entities and foreign key values in the entities. These features allow us to create entity classes that can be shared between implementation of Server components (Domain Model) and other remote levels that may not even have the same version of *Entity Framework* (.NET 2.0 or Silverlight, for example). The POCO entities with foreign keys have a simple serialization format that simplifies the interoperability with other platforms, such as JAVA. The use of external keys also allows a much simpler concurrency model for relationships.

3.- T4 templates to customize generation of code of the POCO or STE classes (*Self-Tracking Entities*).

The Entity Framework team has also used these features to implement the STE entity pattern (Self-Tracking Entities) in a T4 template. Therefore, this pattern is much easier to use because we will have a code generated without implementing it from scratch.

In general, when making decisions on design and technology about data types to be handled by the Web Services with these new capabilities in EF 4.0, we should evaluate the pros and cons of the aspects from **Purist Architecture approaches** (decoupling between Domain entities of data structures managed at Presentation Layer using **DTOs**, Separation of Concerns, Service Data Contracts different from Domain Entities) **versus Productivity and *Time-To-Market*** (optimistic concurrency management already implemented without having to develop conversions between DTOs and Entities, etc.)

If we place the different types of patterns we can use for implementing data objects to communicate in N-Tier applications (data traveling via the network thanks to the Web Services), we would have something like this:

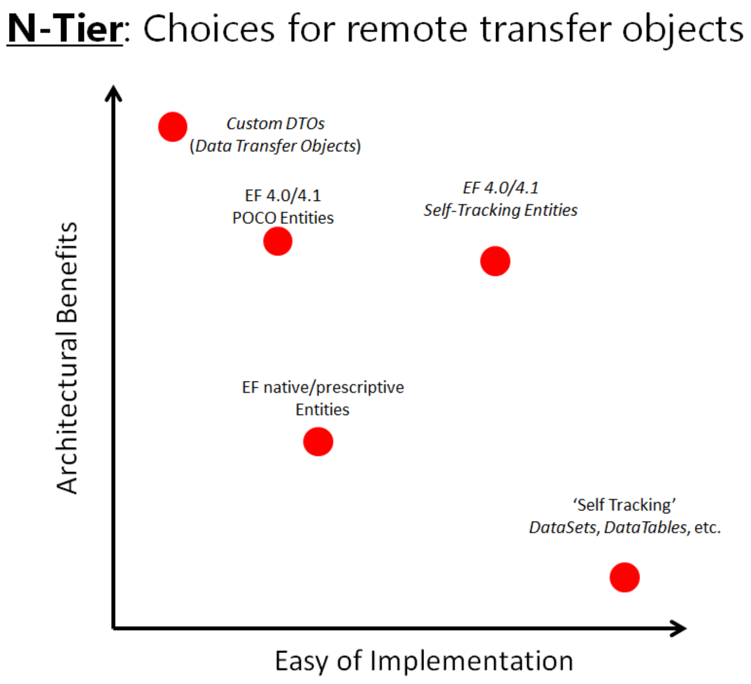


Figure 22.- Balance between Ease of Implementation and Architectural Benefits

The right pattern for each particular situation depends on many factors. In general, DTOs (as introduced in a logical way in the initial section of current chapter), provide many architectural advantages, but have a high implementation cost. The "Record change sets" (DataSets, DataTables, etc.) do not have many advantages at the Architecture level, because they do not meet the PI principle (*Persistence Ignorance*) or provide alignment with logical Entities of a model, etc. However they are very easy to implement (e.g., the ADO.NET DataSet is ideal for applications that are not as complex at business logic level and very data oriented, that is, Architectures that are not Domain Oriented or "non DDD").

In short, we recommend a pragmatic and streamlined balance between these concerns (Productivity vs. Purist Architecture). **One convenient choice (using EF 4.0) could be the STE (*Self-Tracking Entities*)** if there is end to end control of the application (Client consumer and Server). The other option would be **steering towards the DTO** pattern as the situation requires it, like if we need to decouple the domain entity model from the data model used by the presentation layer or external consumers.

The STE will provide initial high productivity while obtaining very significant data tracking benefits. In addition, they represent a much better balance than the *DataSets* or prescriptive/native entities of EF. On the other hand, **DTOs are definitely the best choice as an application gets bigger and more complex, and the most decoupled option regarding the internal Domain MODEL**. This is also true if we have requirements that cannot be met by the STE, such as different development changes speed in the Application/Domain Layers regarding the Presentation layer, so that makes it desirable to decouple the domain entities from the presentation layer data model.

These two types of serialized objects traveling between physical levels (**STE** or **DTO**) are currently probably the most important ones to consider in **N-Tier** architectures that are at the same time **DDD** Architectures (for complex *Domain Driven Design* applications with a lot of domain and application logic coordinated by Services).

OData (WCF Data Services) and DDD?

On the other hand, if we are building a long term *Data Driven Service*, where we might not know who is going to consume this Service (maybe published in the Internet so it needs the best interoperability available), then the best approach would be **OData (WCF Data Services)**, but in this case, it should be a ***Data Driven Service*** (Not DDD).

In a DDD scenario, regarding Reads/Queries, OData would be ok in many situations (We could even create a special OData endpoint for external publishing). But regarding updates and business logic, it is a different matter.

OData is oriented to update directly against data entities, from the client/consumer (using http verbs like UPDATE, INSERT, DELETE directly against data entities).

But, when dealing with a DDD app/service, we are dealing with Application/Domain Services (commands, tasks, transactions, etc.). In DDD we do not directly expose entities for data updates, we work against business/application/domain services. For doing so, OData is not the best choice, actually, WCF Data Services is oriented to publish data entities, directly.

Also, OData bases its engine on internal entites that are anemic-entities (no own domain logic within entities themselves). This is also contrary to DDD where we do not want to have anemic domain models (See chapter 4).

In short, OData is great for N-Tier Data-Driven Apps (and a very strategic technology for Microsoft), but it is not ideal for N-Tier DDD Apps, at least for the transactional front-end (Business/Domain Services Façade). Of course, internally we can use OData for accessing/updating 'only data' stuff, engaging to any database/storage. But that will be a 'data only matter' related to the Data Persistence Layer.

Finally, and as we already said, because OData is really strategic for Microsoft, in the future OData could evolve towards many more scenarios further than Data-Driven Services/Apps. Keep an eye on **http://odata.org** for more details.

CQRS Architecture and OData

For a short intro about CQRS, please read the last pages of chapter number 10.

Regarding CQRS, OData (WCF Data Services) fits perfectly with the **CQRS Read-Model** (where we only want to query a data model), but it does not fit well with the WCF Services axposing the Write-Model, because in this case we are using Commands and executing Business/Domain Services. Commands and Services make the domain entities transparent when executing Business/Domain tasks.

23.- EXPOSING APPLICATION AND DOMAIN LOGIC

The publication of Application and Domain logic in general should not be directly done through the Distributed Services layer. In other words, we will generally not provide direct visibility of all the classes and methods from the Domain or Application Layer. On the contrary, we should implement a Web Service interface that shows what we want to be used by remote Clients (Presentation Layer or other external applications). Therefore, it is common to create a coarse-grained interface, trying to minimize the number of remote calls from the Presentation Layer.

23.1.-Decoupling the Architecture internal layers objects using Dependency Injection and UNITY container

Using UNITY DI starting at the Web Services Layer is crucial because it is here, in the WCF Web Services, where we have the entry point to the Application Server components. Therefore, this is where we have to start with the initial explicit use of the UNITY container (explicitly determining the objects to be instantiated, using Resolve<>). UNITY is used as well in the rest of the Layers (Application, Domain, Persistence), but automatically or implicitly through dependency injection based on constructors. However, the recommendation for a correct and consistent "Dependency injection" is: '**we should only use the initial "Resolve()" at the entry point of our Server (Web Services in an N-Tier application, or ASP.NET MVC for a Web application).**'

Use the UNITY container explicitly only at the entry point to the Server: We should only use "**Resolve<>**" at the entry point to our Server (Web Services in an N-Tier application or ASP.NET MVC for a Web application), starting objects graph thanks to the Dependency Injection, in a single point which must be the '*WCF Service class Inception*'or first instantiation of any class within our WCF Service. Other than that, we would be using the IoC container only as a 'ServiceLocator pattern' which in many cases is an anti-pattern.

23.1.1.- DI should start from the WCF Service class Inception: IInstanceProvider

The Dependency Injection should start with the WCF-Service class and constructor themselves. Doing so, we can have **a single line of code calling the Unity container for the whole solution**. '*Dependency Injection should start from the WCF Service class Inception*'.

This approach requires a bit of extra development, but it is only a few code and reusable for all the WebServices we may have in our project. Therefore it is worth to do it and ultimately, our application code and WCF Services will be even thinner (a single line of code for each WCF Service method implementation).

The approach we have chosen is based on a **WCF Extensibility point** that WCF has which is the **IInstanceProvider interface**. Creating a custom InstanceProvider that implements that interface is probably the best way to really do DI and integrate our IoC container instances creation with WCF Services.

This is our custom UnityInstanceProvider class which will be invoked automatically by WCF:

```
C#
/// WCF Service – Custom UnityInstancePRovider.
```

Implements WCF IInstanceProvider as extensibility point

```
public class UnityInstanceProvider : IInstanceProvider
{
    Type _serviceType;
    IUnityContainer _container;

    public UnityInstanceProvider(Type serviceType)
    {
        if (serviceType == null)
            throw new ArgumentNullException("serviceType");

        this._serviceType = serviceType;
        _container = Container.Current;
    }
```

Method who gets the initial instance of our WCF Services class, but based on our IoC Container (Unity)

```
    public object GetInstance(InstanceContext instanceContext,
                              System.ServiceModel.Channels.Message message)
    {
        //This is the only call to UNITY container in the whole solution
        return _container.Resolve(_serviceType);
    }
```

Root resolution with UNITY: Use of Resolve<Interface>() of UNITY only at the entry points to the Server.

```
    public object GetInstance(InstanceContext instanceContext)
    {
        return GetInstance(instanceContext, null);
    }

    public void ReleaseInstance(InstanceContext instanceContext, object
instance)
    {
        if (instance is IDisposable)
            ((IDisposable)instance).Dispose();
    }
}
```

After having implemented our custom InstanceProvider we need a way to tell WCF that we want him to invoke our InstanceProvider. The way to do that in WCF is using **WCF-Behaviors**.

So we have created a **WCF-Behavior** called ‘UnityInstanceProviderServiceBehavior’. We also created it as an **Attribute**, so we can easily apply it to our WCF Services:

```
C#
/// WCF ServiceBehavior – Custom UnityInstancePpoviderServiceBehavior.
```

Atribute behavior for InstanceProvider system

```
public class UnityInstanceProviderServiceBehavior : Attribute,
                                                     IServiceBehavior

{
    public void ApplyDispatchBehavior(
                                     ServiceDescription serviceDescription,
                                     ServiceHostBase serviceHostBase)
    {
        foreach (var item in serviceHostBase.ChannelDispatchers)
        {
            var dispatcher = item as ChannelDispatcher;
            if (dispatcher != null)
            {
                //Add new instance provider for each end point dispatcher
                dispatcher.Endpoints.ToList().ForEach(endpoint =>
                {
                    endpoint.DispatchRuntime.InstanceProvider =
                                             new UnityInstanceProvider(
                                         serviceDescription.ServiceType
                                                 );
                });
            }
        }
    }

    //Other ommitted methods…

}
```

Then, we simple decorate our WCF Service class with that custom behavior, in a similar way we are applying our exception handling attribute, like the following code:

```
C#
/// The WCF Service Class
```

Attribute to create initial WCF Service instance while injecting the graph dependencies using the Unity container

```
//
[UnityInstanceProviderServiceBehavior()]
[ApplicationErrorHandlerAttribute()] // manage all unhandled exceptions
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
public class ERPModuleService : IERPModuleService
{
    //Dependencies aggregating several Application Layer Services of this MODULE
    ICustomerAppService _customerAppService;
    ISalesAppService _salesAppService;

    //
```

WCF Class Constructor where we specify our DEPENDENCIES

```
    public ERPModuleService(ICustomerAppService customerAppService,
                            ISalesAppService salesAppService)
    {
            //… Validations ommitted…

           // Keep dependencies' objects graphs created by the Unity Container
           _customerAppService = customerAppService;
           _salesAppService = salesAppService;
        }

        //
        //
        public List<CustomerListDTO> FindCustomersByFilter(string filter)
        {
          return _customerAppService.FindCustomers(filter);
        }
```

Errors handling and instances creation are based on WCF behaviours attributes

A single line of code per WCF Service Method (Usually calling Application Services)

```
       //Other WCF-Service methods
       //…
    }
```

As it can be shown, our WCF-Services are the simplest, 'one line of code per method'.

If we were using REST instead SOAP, we would have to add how to address resource URIs and how to map URIs to specific methods. Also we'd need to change WCF bindings configuration, etc. But the essence of what we do within each Web-Service method, and how we implement Dependency Injection, will be the same.
We also might have to check what 'resources' we are going to expose. In many cases REST-resources will be mapped to AGGREGATES, but when dealing with domain/business logic, we'd need to create resources (named with a noun) exposing that domain logic or commands (like 'Transfer').

23.2.- Handling Exceptions in WCF Services

By default, the internal exceptions that occur are transformed by WCF into FAULTS (to be serialized so that they reach the consumer client application). However, unless we change it, the information included about the Exception within the FAULT is generic (A message such as "*The server was unable to process the request due to an internal error.*"). That is, for security reasons, no information about the Exception is included, since we could be sending sensitive information relating to a problem that has occurred.

However, when we are *Debugging* and we see the errors that have occurred, we need to be able to send to the client specific information about the internal error/exceptions of the server (for example, specific error of a Database access problem, etc.). To do so, we should include this configuration in the Web.config file, indicating that we want to include the information on all exceptions in the WCF FAULTS when the type of compilation is 'DEBUG':

```
CONFIG XML

<behavior name="CommonServiceBehavior">
                    ...
<serviceDebug includeExceptionDetailInFaults="true" />

                    ...
</behavior>
```

Include in the FAULTS info on exceptions, only in DEBUGGING mode

In this case, as long as the compilation is in "*debug*" mode, the exception details will be included in the FAULTS.

```
CONFIG XML

<system.web>
    <compilation debug="true" targetFramework="4.0" />
</system.web>
```

We set the "debug" compilation → Exceptions Details will be sent

23.2.1.- Using WCF Attributes for handling exceptions in WCF Services

When implementing exception handling in WCF Services' methods, we might find ourselves writing a lot of redundant code related to 'exceptions catching' and logging actions, etc. for evry web-service method. In order to minimize that redundant code, the approach we decided to implement in V2.0 is to locate the Exception Handling code in a single point, a WCF behavior attribute which calls to a global Error-Handler used by every web service method. Doing this way, the web methods are cleaner, showing code related to the business component they are remotely publishing but with no 'noise' regarding exception handling.

Therefore, our WCF Service looks like the following:

```
C#
/// The WCF Service Class
```

Attribute to handle exception handling from a single point

```
//
[ApplicationErrorHandlerAttribute()] //manage all unhandled exceptions
[UnityInstanceProviderServiceBehavior()]
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
public class ERPModuleService : IERPModuleService
{
    //Dependencies aggregating several Application Layer Services of this MODULE
    ICustomerAppService _customerAppService;
    ISalesAppService _salesAppService;

    //
```

WCF Class Constructor where we specify our DEPENDENCIES

```
    public ERPModuleService(ICustomerAppService customerAppService,
                            ISalesAppService salesAppService)
    {
        //… Validations ommitted…

       // Keep dependencies' objects graphs created by the Unity Container
       _customerAppService = customerAppService;
       _salesAppService = salesAppService;
    }

    //
   //
    public List<CustomerListDTO> FindCustomersByFilter(string filter)
    {
      return _customerAppService.FindCustomers(filter);
    }
```

Errors handling are based on WCF behaviours attributes, so there's no code here about it.

A single line of code per WCF Service Method (Usually calling Application Services)

```
   //Other WCF-Service methods
   //…
}
```

The reader can observe that we are implementing the [ApplicationErrorHandlerAttribute()] in order to handle the exceptions. This is the implementation of this attribute:

```
C#
/// The ApplicationErrorHandlerAttribute
    Implementation of the Attribute to handle exception handling from a single point
//
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public sealed class ApplicationErrorHandlerAttribute
                         : Attribute, IServiceBehavior
{
    public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
                                      ServiceHostBase serviceHostBase)
    {
        if (serviceHostBase != null
            &&
            serviceHostBase.ChannelDispatchers.Any())
        {
            //add default error handler for all dispatchers in WCF services
            foreach (ChannelDispatcher dispatcher in
                                                  serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(new ApplicationErrorHandler());
            }
        }
    }
    // OTHER METHODS OMMITTED
    }
}
```

Then, we are using the **ApplicationErrorHandler** class (**IErrorHandler**) in order to really manage or handle every exception:

```
C#
/// The IErrorHandler implementation
    Implementation of the IErrorHandle where we really take actions regarding exceptions, like logging.
//
public sealed class ApplicationErrorHandler
                                   : IErrorHandler
{
    public bool HandleError(Exception error)
    {
        if (error != null)
            LoggerFactory.CreateLog().
                            LogError(Messages.error_unmanagederror, error);

        //set  error as handled
        return true;
    }

    public void ProvideFault(Exception error,
                             MessageVersion version,
                             ref Message fault)
    {
        if (error is FaultException<ApplicationServiceError>)
        {
            MessageFault messageFault =
                              ((FaultException<ApplicationServiceError>)error).
                                                          CreateMessageFault();

                //propagate FaultException
                fault = Message.CreateMessage(version, messageFault, (
                                (FaultException<ApplicationServiceError>)error)
                                                                        .Action
                                                                     );
            }
            else
            {
                //create service error
                ApplicationServiceError defaultError =
                                                  new ApplicationServiceError()
                {
                    ErrorMessage =
                                Resources.Messages.message_DefaultErrorMessage
                };

                //Create fault exception and message fault
                FaultException<ApplicationServiceError> defaultFaultException =
                     new FaultException<ApplicationServiceError>(defaultError);

                MessageFault defaultMessageFault =
                                    defaultFaultException.CreateMessageFault();

                //propagate FaultException
                fault = Message.CreateMessage(version, defaultMessageFault,
                                                defaultFaultException.Action);
            }
        }
    }
```

23.3.- Hosting WCF Services

We can host a WCF service in any process (.exe), although there are two main types:

- ***Self-Hosting***
    - The Service will be hosted in a custom process (.exe). The hosting process is performed by our own code, either a console application, as we have seen, a Windows service or a form application, etc. In this scenario, we are responsible for programming this WCF hosting process. This, in WCF terms, is known as 'Self-Hosting.'
- ***Hosting (IIS/WAS and Windows Server AppFabric)***
    - The process (.exe) where our service will run is based on IIS/WAS. IIS 6.0/5.5 if we are in Windows Server 2003 or Windows XP, or IIS 7.x and WAS only for IIS versions as from version 7.0. Therefore, that process (.exe) will be any IIS pool process.

The following table shows the different characteristics of each major type:

Table 14.-Self-Hosting vs. IIS and AppFabric

	Self-Hosting	**IIS/WAS/AppFabric**
Hosting process	Our own process, our source code	Process of IIS/WAS. This is configured with a .svc file/page
Configuration file	App.config	Web.config
Addressing	The one specified in the EndPoints	This depends on the configuration of the virtual directories
Recycling and automatic monitoring	NO	YES

We will review the most common specific types of *hosting* below.

Hosting in Console Application

This type of hosting is very similar to the one shown above, so we will not explain it again. Keep in mind that it is useful to perform demos, ***debugging***, etc., but it should not be used to deploy WCF services in a production environment.

Hosting a Windows Service

This is the type of hosting we would use in a production environment if we do not want to/cannot rely on IIS. For example, if the server operating system is Windows Server 2003 (we do not have WAS within IIS), and we also want to use a protocol other than HTTP (for example, TCP, Named-Pipes or MSMQ), then the option we should use for hosting our Service has to be a Windows service developed by us (Service managed by the **Windows Service Control Manager** to start/stop the service, etc.).

In short, the configuration is similar to console application hosting (like the one we have seen before, app.config, etc.), but it varies in the place where we should program the start/creation code of our service, which will be similar to the following (code in a project of the "Service-Windows" type):

```
namespace HostServicioWindows
{
  public partial class HostServicioWin : ServiceBase
  {
     //Host WCF
     ServiceHost _Host;

     public HostServiceWin()
     {
       InitializeComponent();
     }

     protected override void OnStart(string[] args)
     {
       Type serviceType = typeof(Greeting);

       _Host = new ServiceHost(serviceType);

       _Host.Open();

       EventLog.WriteEntry("Host-WCF  Service",  "The  WCF  Service  is
available.");
     }

     protected override void OnStop()
     {
        _Host.Close();
        EventLog.WriteEntry("Host-WCF Service", "WCF service stopped.");
     }
   }
}
```

We have to instantiate the WCF Service when the Windows-service starts (OnStart() method), and close the WCF service in the OnStop() method. Other than that, the process is a typical Windows Service developed in .NET, and we will not review this topic any further.

Hosting in IIS (Internet Information Server)

WCF services can be activated using IIS with hosting techniques similar to the previous traditional Web-Services (ASMX). This is implemented by using the .svc extension files (comparable to .asmx files), within which the service to be hosted is specified in just one line:

.svc page content

```
  <%@Service class="MyCompany.MyApplication.MyWcfService.Greeting" %>
```

This file is placed in a virtual directory and the service assembly is deployed (.DLL) within its \bin directory or in the GAC (Global Assembly Cache). When this technique is used, we should specify the *endpoint* of the service in the web.config. However, there is no need to specify the address, since it is implicit in the location of the .svc file:

```
<configuration>
<system.serviceModel>
<services>
<service type=" MyCompany.MyApplication.MyWcfService.Greeting ">
<endpoint address=" binding="basicHttpBinding"
contract=" MyCompany.MyApplication.MyWcfService.IGreeting"/>
</service>
      ...
```

If you navigate with a *browser* to the .svc file, you can see the help page for this service, showing that a ServiceHost has been automatically created within the application domain of the ASP.NET. This is performed by a 'HTTP-Module' that filters incoming requests of the .svc type, and automatically builds and opens the appropriate ServiceHost when necessary. When a website is created based on this project template, the .svc file is generated automatically, along with the corresponding implementation class (located within the *App_Code* folder). It also brings us a configuration of an endpoint by default, in the *web.config* file.

NOTE about REST Services: When developing a *RESTfull* Service using WCF, we usually only rely on URIs and don't need (and it is better not to) the .svc files.

Hosting in IIS 7.x – WAS

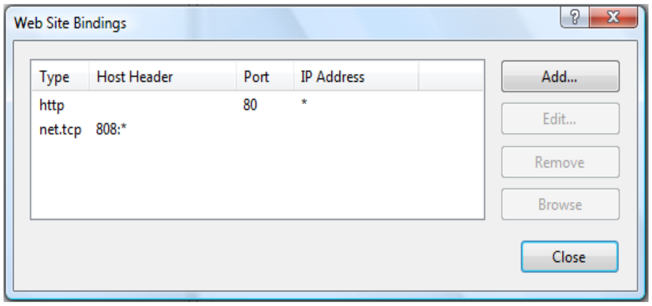
In 5.1 and 6.0 versions of IIS, the WCF activation model is linked to the '*pipeline of ASP.NET*', and therefore, to the HTTP protocol. However, starting with **IIS 7.0** or higher, (starting on **Windows Vista**, **Windows Server 2008 or higher**), an activation mechanism is introduced that is independent from the transport protocol. This mechanism is known as '*Windows Activation Service*' (WAS). Using WAS, WCF services can be published on any transport protocol, such as TCP, Named-Pipes, MSMQ, etc. (as we do when using a '*Self-Hosting*' processes).

Use of a different protocol (like TCP) based on WAS, required prior configuration of IIS, both at the Web-Site level and at the virtual directory level.

Adding a binding to a site of IIS 7.x/WAS

Adding a new *binding* to an IIS/WAS site can be done with the IIS 7.x GUI or from the command line by invoking IIS commands.

The example below shows the configured bindings in a Web Site of IIS 7.x:



Type	Host Header	Port	IP Address
http		80	*
net.tcp	808:*		

Figure 23.- Bindings in an IIS 7.x Web Site

It can also be added with a command *script*:

```
    %windir%\system32\inetsrv\appcmd.exe  set  site  "Default  Web  Site"  -
+bindings.[protocol='net.tcp',bindingInformation='808:*']
```

We should also enable *bindings* later, which we want to be able to use in each virtual directory where our WCF service resides:

```
    %windir%\system32\inetsrv\appcmd.exe set app "Default Web
Site/MyVirtualDirectory" /enabledProtocols:http,net.tcp
```

Hosting in 'Windows Server AppFabric'

This is similar to the *hosting* in IIS 7.x (WAS), but AppFabric brings extended capabilities for Monitoring WCF and Workflow Services. Nonetheless, the deployment over AppFabric is exactly the same, since AppFabric is based on IIS/WAS infrastructure.

Hosting in WPF or Windows Form Application

It may seem strange to want to host a service (which apparently is a server aspect), in a Windows form application or even in a WPF form application (Windows Presentation Foundation). However, this is not so rare since in WCF there is a type of binding that is '*Peer-to-Peer*', where client applications are WCF services at the same time. In other words, all the applications talk to each other because they are WCF services. There are many examples of 'Peer-to-peer' communications on the Internet, such as file sharing software, etc. This is 'yet another type' of *Self-Hosting*.

24.- WCF SERVICE DEPLOYMENT AND MONITORING IN WINDOWS SERVER APPFABRIC (AKA. DUBLIN)

Windows Server AppFabric has two very different main areas:

- **Service Hosting and Monitoring** (Web services and Workflow Services), called '*Dublin*' during beta version.
- **Distributed Cache** (called '*Velocity*' during its beta version)

The AppFabric Cache is explained in the chapter on '*Cross-cutting Layers*'.

The '**Service Hosting and monitoring**' capabilities are related to the WCF 4.0 Services and Workflow-Services deployment.

This part brings us the added value of a series of functionalities for easy **deployment** and mainly for **monitoring**. Now, with AppFabric, an IT professional (no need to be a developer) may localize issues in a WCF service application (or validate correct executions) simply by using the IIS 7.x Management Console. Before the appearance of *Windows Server AppFabric*, monitoring WCF services had to be done exclusively by developers and at a much lower level (tracing, *debugging*, etc.). For an ITPro, the WCF Services of any application were "black boxes" with unknown behaviors.

Actually, *Hosting* a Service on AppFabric does not create a completely new hosting environment. Instead, it is based on IIS 7.x and WAS (*Windows Process Activation Service*) but it adds certain execution and WCF service management capabilities including *Workflow-Services*. The following diagram shows *Windows Server AppFabric* architecture.

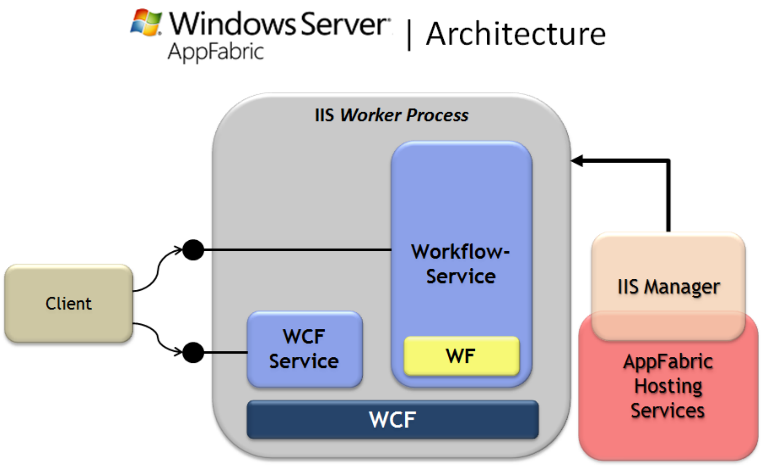
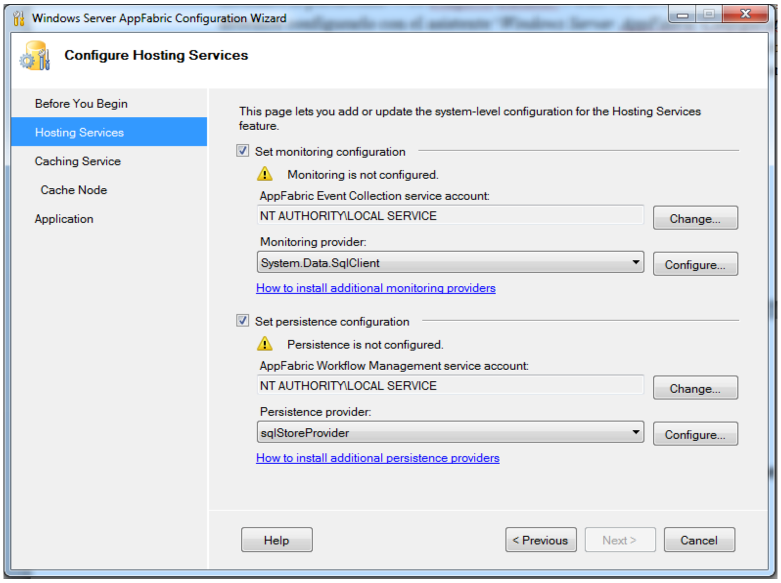


Figure 24.- 'Windows Server AppFabric Architecture'

24.1.- Windows Server AppFabric Installation and configuration

Logically, the first thing we should do is install *Windows Server AppFabric*, whether from the public *download* or using the '*Web Platform Installer*'. After installing 'the bits', we should configure it with the '*Windows Server AppFabric Configuration Wizard*', where the aspects related to Cache and WCF monitoring are configured. In this case we only need to review the required points in order to configure WCF Services management and monitoring. We now ignore the cache configuration.

First, we should configure the monitoring database to be used, as well as the credentials for such access.


Windows Server AppFabric Configuration Wizard

Configure Hosting Services

This page lets you add or update the system-level configuration for the Hosting Services feature.

[✓] Set monitoring configuration

Monitoring is not configured.

AppFabric Event Collection service account: NT AUTHORITY\LOCAL SERVICE

Monitoring provider: System.Data.SqlClient

[✓] Set persistence configuration

Persistence is not configured.

AppFabric Workflow Management service account: NT AUTHORITY\LOCAL SERVICE

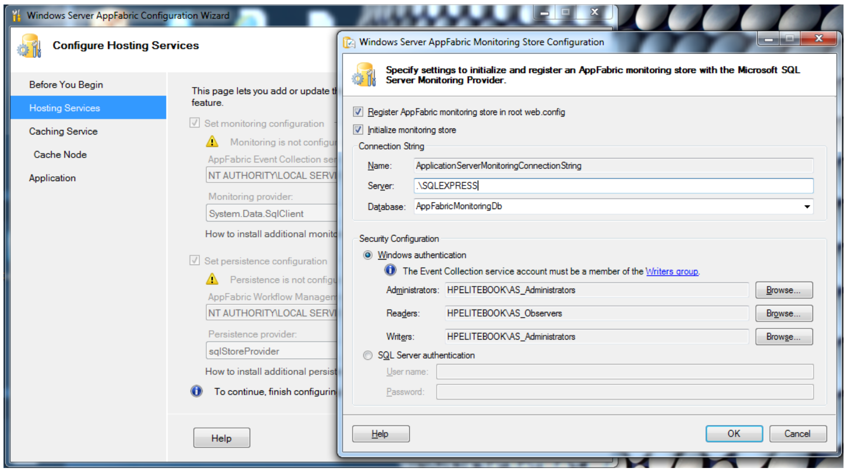
Persistence provider: sqlStoreProvider

Figure 25.- Hosting Services configuration

At the 'hosting services' level we need two SQL Server databases; one to monitor any WCF service and Workflow-Services (called '*Monitoring Store*') and a second database for the persistence of workflow services execution.

In the example, we used *SQL Server Management Studio* to create an empty database (called *AppFabricMonitoringDb*) for the monitoring database ('*Monitoring Store*') in a local SQL Server Express. However, a remote Standard

SQL Server could also have been used instead. Once the empty database is available, we specify it in the AppFabric configuration, as shown in the diagram below.


Windows Server AppFabric Configuration Wizard
Configure Hosting Services
Before You Begin
Hosting Services
Caching Service
Cache Node
Application
Help
Windows Server AppFabric Monitoring Store Configuration
Browse...
Browse...
Browse...
Help
OK
Cancel

This page lets you add or update t
feature.

☑ Set monitoring configuration

Monitoring is not configu

AppFabric Event Collection se
NT AUTHORITY\LOCAL SERV

Monitoring provider:
System.Data.SqlClient

How to install additional monit

☑ Set persistence configuration

Persistence is not config

AppFabric Workflow Managem
NT AUTHORITY\LOCAL SERV

Persistence provider:
sqlStoreProvider

How to install additional persist

To continue, finish configurin

Specify settings to initialize and register an AppFabric monitoring store with the Microsoft SQL Server Monitoring Provider.

☑ Register AppFabric monitoring store in root web.config
☑ Initialize monitoring store

Connection String

Name: ApplicationServerMonitoringConnectionString
Server: .\SQLEXPRESS
Database: AppFabricMonitoringDb

Security Configuration

● Windows authentication

The Event Collection service account must be a member of the Writers group.

Administrators: HPELITEBOOK\AS_Administrators
Readers: HPELITEBOOK\AS_Observers
Writers: HPELITEBOOK\AS_Administrators

○ SQL Server authentication

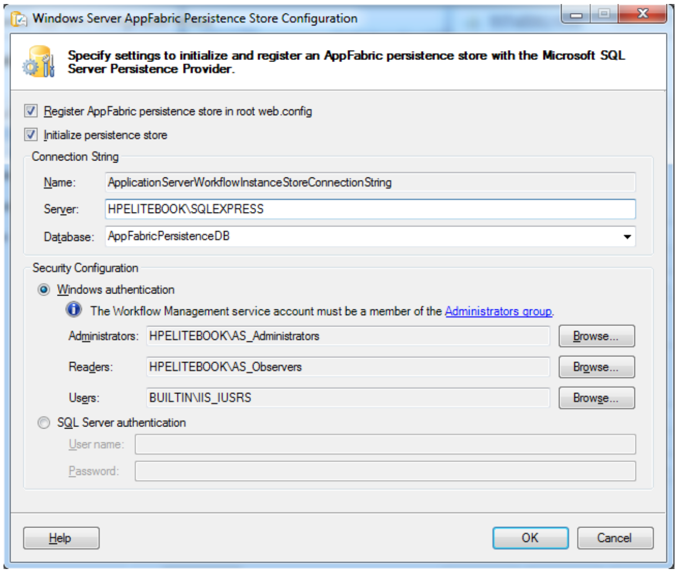
User name:
Password:

Figure 26.- Hosting Services - Monitoring Database config

We can also opt to use Standard SQL authentication with SQL Server own users.

For the long workflows executions we also need a persistence database where WF automatically dehydrates the workflow execution. That requires creating a second database called, for example, '*AppFabricPersistenceDB*' and also specified in the AppFabric configuration.

NOTE: In NLayerApp Sample we are not using WF or Workflow Services, so all AppFabric Workflow Services configuration could be omitted.


Windows Server AppFabric Persistence Store Configuration

Specify settings to initialize and register an AppFabric persistence store with the Microsoft SQL Server Persistence Provider.

☑ Register AppFabric persistence store in root web.config

☑ Initialize persistence store

Connection String

Name:	ApplicationServerWorkflowInstanceStoreConnectionString
Server:	HPELITEBOOK\SQLEXPRESS
Database:	AppFabricPersistenceDB

Security Configuration

● Windows authentication

The Workflow Management service account must be a member of the Administrators group.

Administrators:	HPELITEBOOK\AS_Administrators
Readers:	HPELITEBOOK\AS_Observers
Users:	BUILTIN\IIS_IUSRS

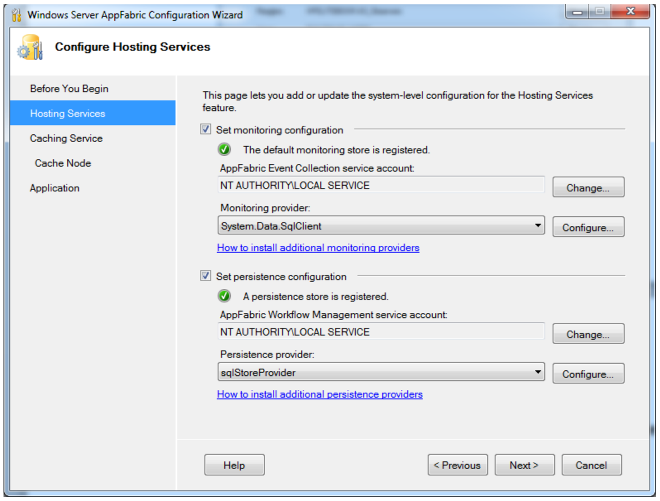
○ SQL Server authentication

User name:

Password:

Figure 27.- Hosting Services – WF Persistence Database config

This is how the AppFabric monitoring section is finally configured.


Windows Server AppFabric Configuration Wizard

Configure Hosting Services

This page lets you add or update the system-level configuration for the Hosting Services feature.

☑ Set monitoring configuration

The default monitoring store is registered.

AppFabric Event Collection service account:

NT AUTHORITY\LOCAL SERVICE

Monitoring provider:

System.Data.SqlClient

How to install additional monitoring providers

☑ Set persistence configuration

A persistence store is registered.

AppFabric Workflow Management service account:

NT AUTHORITY\LOCAL SERVICE

Persistence provider:

sqlStoreProvider

How to install additional persistence providers

Figure 28.- Final Hosting Services configuration

The rest of the wizard configuration will not be addressed in this guide because it is related to Cache services.

If we start IIS Manager, we can see some new icons for AppFabric management:

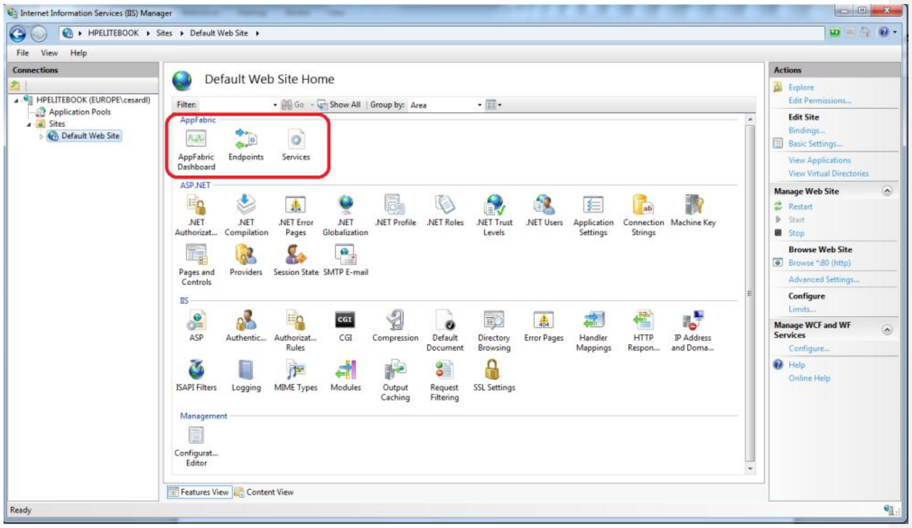
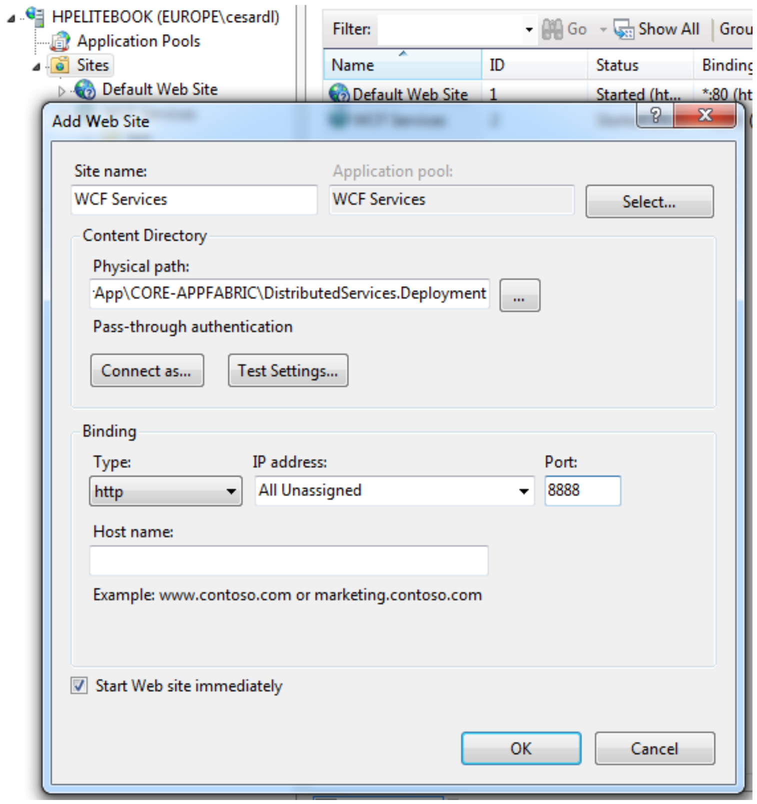


Figure 29.- AppFabric snap-in – IIS Manager integrated

24.2.- WCF Service deployment in Windows Server AppFabric.

After installing and configuring *AppFabric*, we can deploy our WCF services and application/domain components on *AppFabric*, so WCF services will be now monitored.

In the development environment (our PC), we will directly map our WCF service project to an IIS website. To do so, we should first create an IIS website with the TCP port chosen for our WCF services, such as the 8888. We will then specify our Distributed Services project directory as the root path for this Website:



HPELITEBOOK (EUROPE\cesardl)
Application Pools
Sites
Default Web Site

Filter: Go Show All Grou

Name	ID	Status	Binding
Default Web Site	1	Started (ht...	*:80 (ht

Add Web Site

Site name: WCF Services
Application pool: WCF Services
Select...

Content Directory
Physical path: App\CORE-APPFABRIC\DistributedServices.Deployment
Pass-through authentication
Connect as... Test Settings...

Binding
Type: http
IP address: All Unassigned
Port: 8888
Host name:
Example: www.contoso.com or marketing.contoso.com

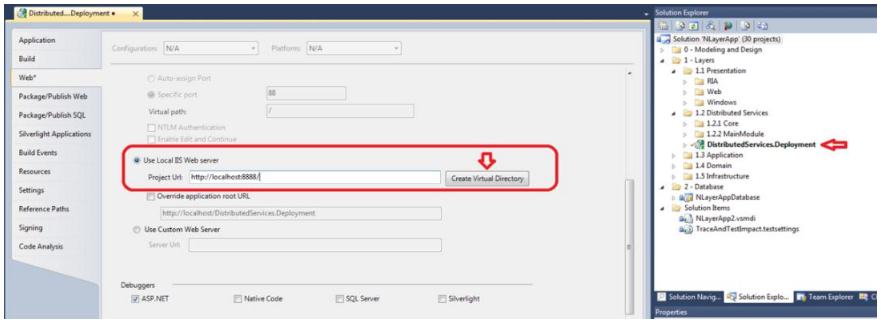
☑ Start Web site immediately

OK Cancel

Figure 30.- WCF WebSite configuration

We need to change the properties of our WCF project so that, when we *debug*, it will be done over IIS and not over VS.2010 'Cassini'. In order to do that, we have to visualize the properties of the hosting project in VS.2010, and specify in the "Web" tab that we want to use the local IIS Web server. We will specify 'http://localhost:8888/' as the project URL. Finally, we must specify in which physical directory we want to map the Virtual Directory of IIS.

When we are done, we can press the '*Create Virtual Directory*' button to verify if it is properly connected to IIS (although in this case no virtual directory is created because there is already a root Website).



Use Local IIS Web server

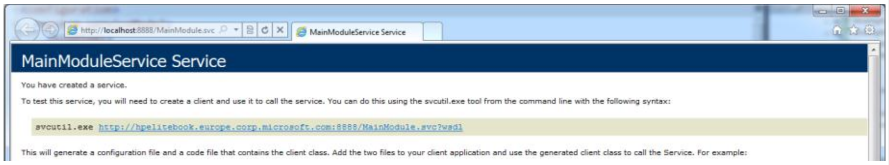
Project Url: http://localhost:8888/

Override application root URL: http://localhost/DistributedServices.Deployment

Debuggers: [✓ ASP.NET] Native Code SQL Server Silverlight

Figure 31.- WCF project deployment from VS.2010

Now, we can test our WCF service with the testing page, running on the IIS website. This can be done by launching it from the VS.2010 debugging environment or directly from the browser or IIS Manager.



MainModuleService Service

You have created a service.

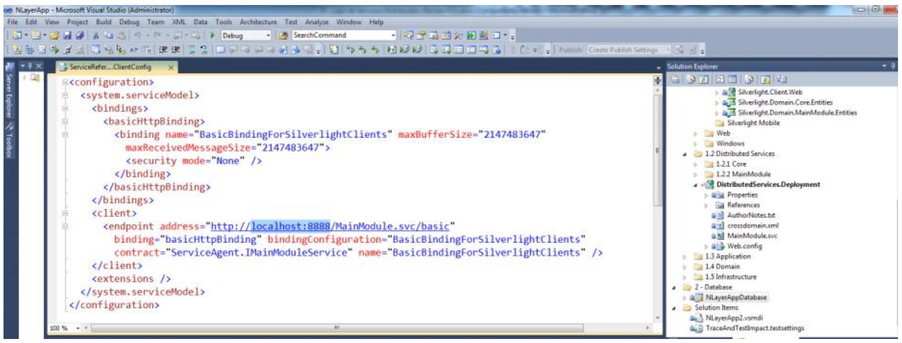
To test this service, you will need to create a client and use it to call the service. You can do this using the svcutil.exe tool from the command line with the following syntax:

```
svcutil.exe http://hpelitebook.europe.corp.microsoft.com:8888/MainModule.svc?wsdl
```

This will generate a configuration file and a code file that contains the client class. Add the two files to your client application and use the generated client class to call the Service. For example:

Figure 32.- Testing our WCF Service already deployed on IIS

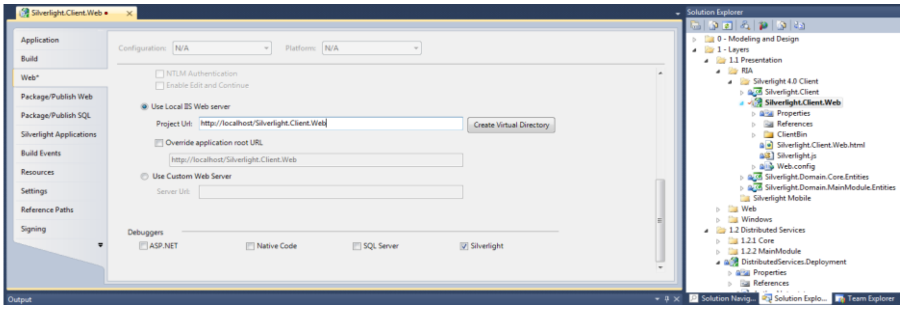
Finally, in order for the application clients to work correctly, we should change the WCF endpoint configuration URL in all the Client Applications who are consuming our Service (Silverlight, WPF, etc.), specifying that the TCP port is now 8888 or any other change we have made for the URL.



```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicBindingForSilverlightClients" maxBufferSize="2147483647"
          maxReceivedMessageSize="2147483647">
          <security mode="None" />
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:8888/MainModule.svc/basic"
        binding="basicHttpBinding" bindingConfiguration="BasicBindingForSilverlightClients"
        contract="ServiceAgent.IMainModuleService" name="BasicBindingForSilverlightClients" />
    </client>
    <extensions />
  </system.serviceModel>
</configuration>
```

Figure 33.- Changing Client WCF endpoint configs

Finally, we also need to deploy the *Silverlight* web/hosting project in IIS. We simply need to specify it in the project properties and create it as a virtual ISS directory with the project name. The virtual directory in IIS will be created by pressing the '*Create Virtual Directory*' button.



Configuration: N/A   Platform: N/A

☐ NTLM Authentication
☐ Enable Edit and Continue

● Use Local IIS Web server
Project Url: http://localhost/Silverlight.Client.Web
☐ Override application root URL
http://localhost/Silverlight.Client.Web
○ Use Custom Web Server
Server Url:

Debuggers
☐ ASP.NET   ☐ Native Code   ☐ SQL Server   ☑ Silverlight

Figure 34.- Silverlight project deployment in IIS

The same operation may be performed with any *ASP.NET MVC* project so that it also runs on *IIS*.

24.2.1.- SQL Server DB Access Identity and WCF Impersonation

The following should be considered: When we are running our application in *'debugging'* mode from Visual Studio and with WCF services based on the '*Dev Web Server*' of Visual Studio (*Cassini*), and the connection string is using "Integrated Security", the identity of the user who accesses SQL Server through ADO.NET EF connections is the identity of the user we logged onto the machine with, who will probably not have any problem accessing our local SQL Server.

On the other hand, if our WCF service is running on IIS/AppFabric, the identity it will try to access the SQL Server with will be the identity of the user-account that our Website process pool is being run with; this is normally 'ApplicationPoolIdentity' or 'NetworkService', which probably do not have access to our SQL Server, by default.

For our WCF services to have access to SQL Server, we recommend following the "Trusted Subsystem":

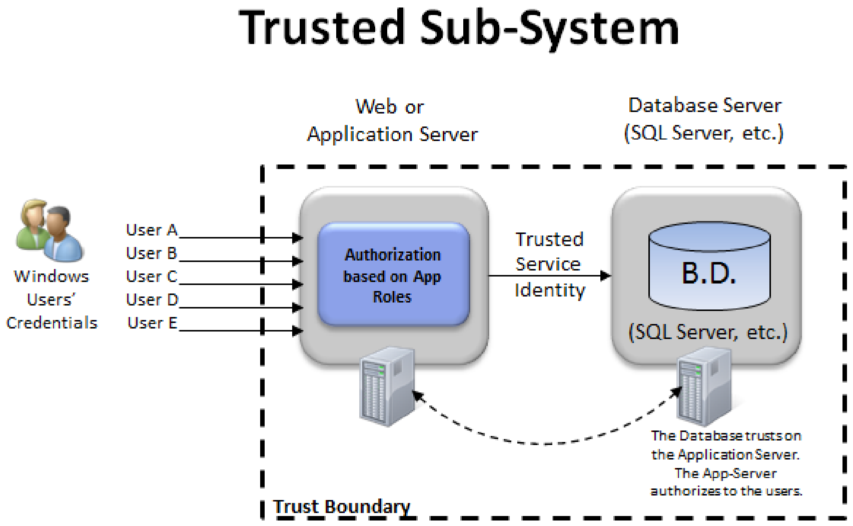
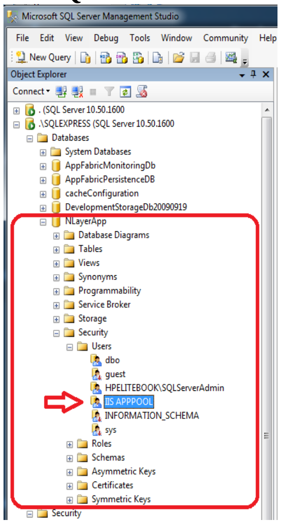

Trusted Sub-System
Web or Application Server
Database Server (SQL Server, etc.)
Windows Users' Credentials
User A
User B
User C
User D
User E
Authorization based on App Roles
Trusted Service Identity
B.D.
(SQL Server, etc.)
The Database trusts on the Application Server. The App-Server authorizes to the users.
Trust Boundary

Figure 35.- Trusted Subsystem

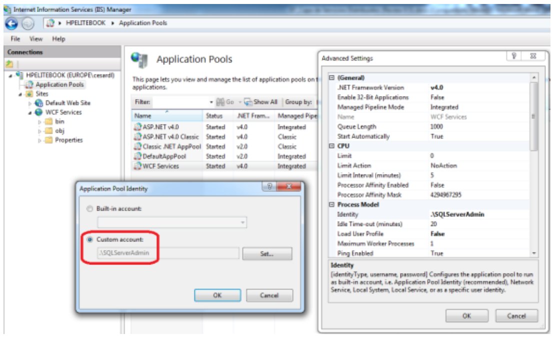
Having an EF/ADO.NET connection string with integrated security model means we need the IIS pool we are using to be able to access our database in SQL Server. This is either because we provide DB access to the user the IIS pool is run with by default or because we change the identity the IIS pool is run with. The first option is probably more recommendable. This provides the necessary DB access to a default user of the IIS pool, so we do not have to create specific new Active Directory or Windows users. This, of course, will depend on our application, if we need to make other remote accesses using that same identity, etc.

Granting Access to the IIS pool user-account to the SQL Server database

Changing the IIS pool identity (Optional)


Microsoft SQL Server Management Studio
File Edit View Debug Tools Window Community Help
New Query
Object Explorer
Connect

```
. (SQL Server 10.50.1600
.\SQLEXPRESS (SQL Server 10.50.1600
  Databases
    System Databases
    AppFabricMonitoringDb
    AppFabricPersistenceDB
    cacheConfiguration
    DevelopmentStorageDb20090919
    NLayerApp
      Database Diagrams
      Tables
      Views
      Synonyms
      Programmability
      Service Broker
      Storage
      Security
        Users
          dbo
          guest
          HPELITEBOOK\SQLServerAdmin
          IIS APPPOOL
          INFORMATION_SCHEMA
          sys
        Roles
        Schemas
        Asymmetric Keys
        Certificates
        Symmetric Keys
  Security
```



```
HPELITEBOOK (EUROPE\cesardl)
  Application Pools
  Sites
    Default Web Site
    WCF Services
      bin
      obj
      Properties
```

This page lets you view and manage the list of application pools on t applications.

Name	Status	.NET Fram...	Managed Pipe
ASP.NET v4.0	Started	v4.0	Integrated
ASP.NET v4.0 Classic	Started	v4.0	Classic
Classic .NET AppPool	Started	v2.0	Classic
DefaultAppPool	Started	v2.0	Integrated
WCF Services	Started	v4.0	Integrated

Custom account: .\SQLServerAdmin

(General)	
.NET Framework Version	**v4.0**
Enable 32-Bit Applications	False
Managed Pipeline Mode	Integrated
Name	WCF Services
Queue Length	1000
Start Automatically	True
CPU	
Limit	0
Limit Action	NoAction
Limit Interval (minutes)	5
Processor Affinity Enabled	False
Processor Affinity Mask	4294967295
Process Model	
Identity	**.\SQLServerAdmin**
Idle Time-out (minutes)	20
Load User Profile	**False**
Maximum Worker Processes	1
Ping Enabled	True

Identity
[identityType, username, password] Configures the application pool to run as built-in account, i.e. Application Pool Identity (recommended), Network Service, Local System, Local Service, or as a specific user identity.

Figure 36.- DB access identity options

Another viable option is to use a connection string with SQL Server standard security, where the access credentials to SQL Server will be within the connection string. This has the disadvantage of being less secure & recommended because we have to specify (within this connection string) the user name and the password and these data remain visible in our web.config file. Normally, the web.config file may only be accessed by administrative staff and developers. In any case, it is less secure than specifying some administrative level credentials in the IIS console and being encrypted and protected by the IIS system (within the *applicationHost.config* file, the credentials to be used by the IIS pool remain encrypted).

24.3.- Monitoring WCF services from the Windows Server AppFabric Console in the IIS Manager.

After consuming/accessing any WCF Service hosted in AppFabric (from any client app, such as WPF or Silverlight in the *NLayerApp* sample), several times, we will be able to check those accesses and their performance from the AppFabric console.

The following is a global view from the Dashboard.

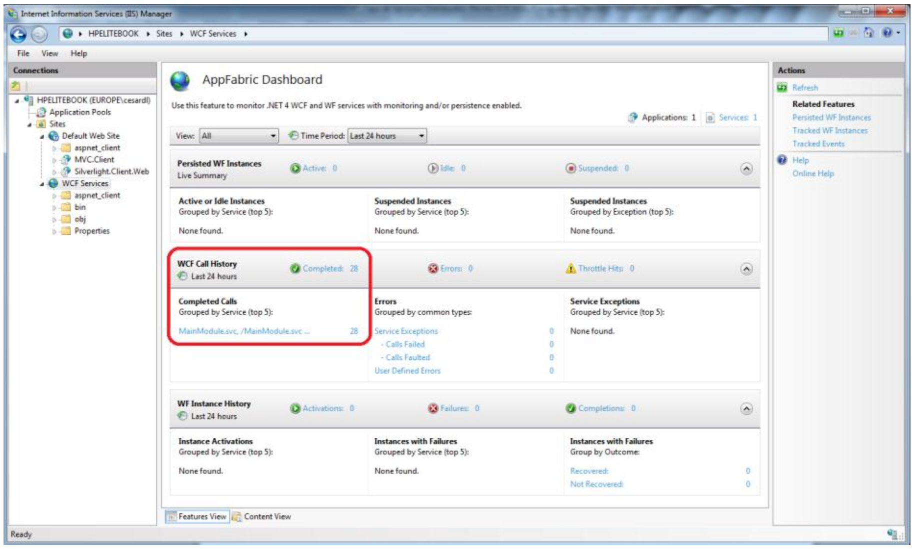
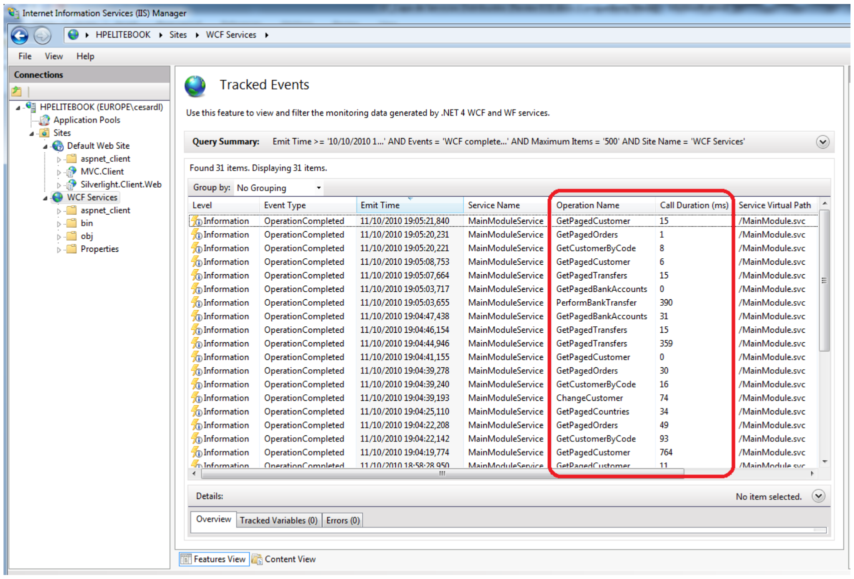


Figure 37.- Monitoring WCF Services through AppFabric

We can then review the entire list of calls to WCF services in detail. **Therefore, an IT Pro (no need to be a developer), can examine the behavior of an WCF Web service application, and even analyze the performance of every call to Web Services**, in order to detect performance issues, without prior knowledge of the details of the development.



Tracked Events

Use this feature to view and filter the monitoring data generated by .NET 4 WCF and WF services.

Query Summary: Emit Time >= '10/10/2010 1...' AND Events = 'WCF complete...' AND Maximum Items = '500' AND Site Name = 'WCF Services'

Found 31 items. Displaying 31 items.

Group by: No Grouping

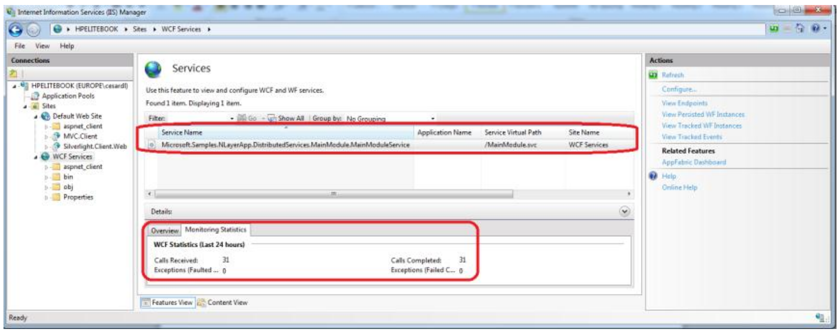
Level	Event Type	Emit Time	Service Name	Operation Name	Call Duration (ms)	Service Virtual Path
Information	OperationCompleted	11/10/2010 19:05:21,840	MainModuleService	GetPagedCustomer	15	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:05:20,231	MainModuleService	GetPagedOrders	1	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:05:20,221	MainModuleService	GetCustomerByCode	8	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:05:08,753	MainModuleService	GetPagedCustomer	6	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:05:07,664	MainModuleService	GetPagedTransfers	15	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:05:03,717	MainModuleService	GetPagedBankAccounts	0	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:05:03,655	MainModuleService	PerformBankTransfer	390	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:47,438	MainModuleService	GetPagedBankAccounts	31	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:46,154	MainModuleService	GetPagedTransfers	15	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:44,946	MainModuleService	GetPagedTransfers	359	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:41,155	MainModuleService	GetPagedCustomer	0	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:39,278	MainModuleService	GetPagedOrders	30	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:39,240	MainModuleService	GetCustomerByCode	16	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:39,193	MainModuleService	ChangeCustomer	74	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:25,110	MainModuleService	GetPagedCountries	34	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:22,208	MainModuleService	GetPagedOrders	49	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:22,142	MainModuleService	GetCustomerByCode	93	/MainModule.svc
Information	OperationCompleted	11/10/2010 19:04:19,774	MainModuleService	GetPagedCustomer	764	/MainModule.svc
Information	OperationCompleted	11/10/2010 18:58:28,950	MainModuleService	GetPagedCustomer	11	/MainModule.svc

Details: No item selected.

Overview | Tracked Variables (0) | Errors (0)

Figure 38.- AppFabric Monitoring details

We can also analyze the global statistics for a specific Web service.



Services

Use this feature to view and configure WCF and WF services.

Found 1 item. Displaying 1 item.

Filter: Go Show All | Group by: No Grouping

Service Name	Application Name	Service Virtual Path	Site Name
Microsoft.Samples.NLayerApp.DistributedServices.MainModule.MainModuleService		/MainModule.svc	WCF Services

Details:

Overview | Monitoring Statistics

WCF Statistics (Last 24 hours)

Calls Received:	31	Calls Completed:	31
Exceptions (Faulted ...	0	Exceptions (Failed C...	0

Figure 39.- AppFabric Monitoring – Global statistics

25.- SERVICES AND WCF GLOBAL REFERENCES

Exception Handling in Service Oriented Applications
http://msdn.microsoft.com/en-us/library/cc304819.aspx

"Data Transfer and Serialization":
http://msdn.microsoft.com/en-us/library/ms730035.aspx

"Endpoints: Addresses, Bindings, and Contracts":
http://msdn.microsoft.com/en-us/library/ms733107.aspx

"Messaging Patterns in Service-Oriented Architecture":
http://msdn.microsoft.com/en-us/library/aa480027.aspx

"Principles of Service Design: Service Versioning":
http://msdn.microsoft.com/en-us/library/ms954726.aspx

"Web Service Messaging with Web Services Enhancements 2.0":
http://msdn.microsoft.com/en-us/library/ms996948.aspx

"Web Services Protocols Interoperability Guide":
http://msdn.microsoft.com/en-us/library/ms734776.aspx

"Windows Communication Foundation Security":
http://msdn.microsoft.com/en-us/library/ms732362.aspx

"XML Web Services Using ASP.NET":
http://msdn.microsoft.com/en-us/library/ba0z6a33.aspx

"Enterprise Solution Patterns Using Microsoft .NET":
http://msdn.microsoft.com/en-us/library/ms998469.aspx

"Web Service Security Guidance":
http://msdn.microsoft.com/en-us/library/aa480545.aspx

*"Improving Web Services Security: Scenarios and Implementation Guidance for WCF":***http://www.codeplex.com/WCFSecurityGuide**

"WS- Specifications":***http://www.ws-standards.com/ws-atomictransaction.asp**