Servicios

Los Servicios (Business Services) representan a cualquier tipo de servicio, administrados por el Dispatcher ya sean transaccionales o de consulta, sincronos o asíncronos.

En el frameworck fwk los servicios se envían desde las aplicaciones atravez del wrapper o service connector bajo el siguiente patron

Se establecen un patrón para ejecución de servicios:

Patron Ejecucion síncrona:

```
[ServiceName]Response = Execute([ServiceName]Request)
```

Patron Ejecucion asíncrona:

```
ExecuteAsync(CallBack,[ServiceName]Request)


CallBack(result)
{
        [ServiceName]Response = result
}
```

Antes de pasar a definir los diferentes tipos de servicios es importante comenzar a explicar técnicamente los componentes que lo conforman:

Request y Response

Los Request y los Response son clases bases del framework que están especialmente diseñadas para que sean recibidas por el despachador de servicios. Todo contrato de servicio que se basen esta arquitectura deberá disponer de una clase que herede de estas clases bases espectivamente dependiendo de si se trata de un contrato request o un contrato response.

Es decir toda estructura de un Request se adapta especialmente a cumplir los requisitos que necesita un determinado servicio para poder ser ejecutado y por otro lado, todo estructura Response representa la forma en que dicho servicio retornara información de respuesta al cliente.

Tanto los Request como los Response implementan una interface IServiceContract a través de su clase base ServiceContractBase como se muestra a continuación

. Esta implementación se hace implícitamente cuando heredan de las clases bases : Request<T> y Response<T> respectivamente.

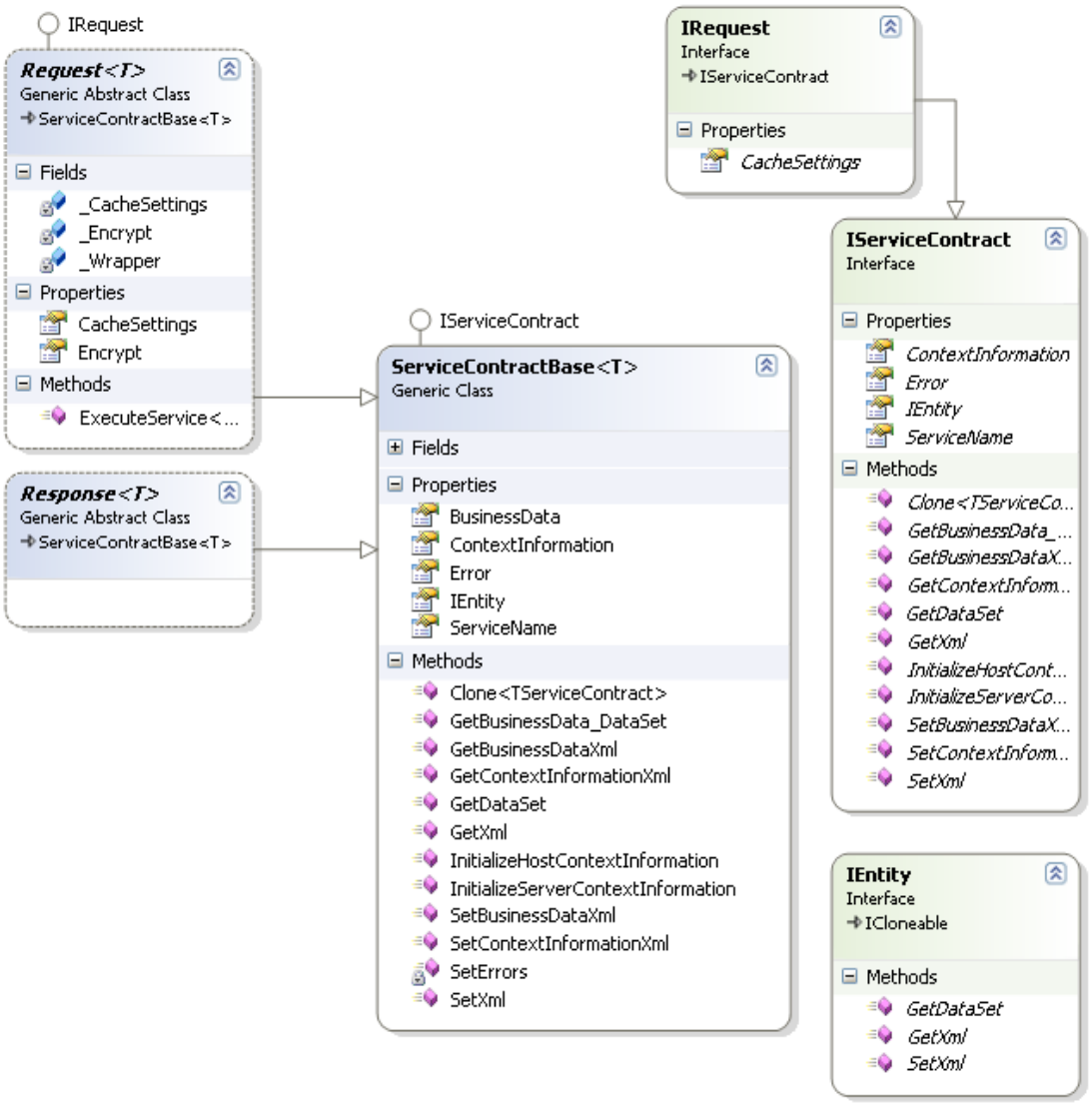
```
ClaseRequest : Action Line.Framework.Bases.BackEnd.Request<T>
ClaseResponse : Action Line.Framework.Bases.BackEnd.Response<T>
Y
Request<T> : ServiceContractBase <T> where T : IEntity
Response<T> : ServiceContractBase <T> where T : IEntity
```

Done la clase **T** que recibe de forma genérica es cualquier entidad o clase que implemente de `IEntity.`

Esta clase, que implementa `IEntity,` es la que realmente tiene toda la estructura de negocio que necesita el servicio.

Nota: *cuando se definan servicios de alta complejidad la estructuras de las clases* `Entity/Entities` *que reciben los* `Request` *o* `Responses` *deberían ser representados por esquemas XSD. Esto es para disponer de una vista mas rápida y cómoda de estos objetos y además para que puedan ser utilizados para generar clases automáticamente.*

Diagrama de clase los contratos de servicios:



Los `Request` y `Response` que los desarrolladores generen serán clases que no tienen comportamiento ni atributos alguno en la definición de su cuerpo ya que las funcionalidades mas habituales están absorbidas por los métodos heredados de sus clases bases.

Definir una clase `Request` y `Response` es para respetar el patrón de ejecución de un servicio contra un Dispatcher o servidor de aplicaciones. No es solo una convención estándar solamente, el despachador de servicio utilizara la herencia de estas clases para sus propósitos y asegurar la ejecución y correcta respuesta de la petición.

Alguno de los métodos o atributos heredados son:

Método o Atributo	Descripción
BusinessData	Este atributo representa la clase contenida dentro del Request o Response que hereda de Entity y contiene toda información funcional o de negocio.
SetXml(string pXMLService)	Rellena el Request o Response con la información del xml. Contiene tanto información de contexto como la funcional.
GetXml()	Obtiene el xml del Request o response.. Contiene tanto información de contexto como la funcional
ContextInformation	Información de contexto acerca del Request o response del servicio.
SetContextInformationXml()	Inicializa los datos contexto que pertenecen al Request con el contenido del xml.-
GetContextInformationXml()	Retorna el xml del Información de contexto que pertenece al Request o Response.-
InitializeServerContextInformation()	Establece la información de contexto del Request o Response del lado del despachador de servicios.-
InitializeHostContextInformation()	Establece la información de contexto del Request o Response del lado del cliente.-
SetBusinessDataXml(string pXMLData)	Inicializa los datos de negocio que pertenecen al Param o Result con el contenido del xml, dependiendo de si se trata de un Request o Response respectivamente.
GetBusinessDataXml()	Retorna el xml del Param o Result que pertenece al Request o Response respectivamente.-
Errors	Solo valido para los objetos que heredan de Response. Y representa cualquier tipo de error ocurrido desde que el despachador de servicio lanzo la ejecución de algún BusinessService
`Encrypt`	Solo para los Request. Permite que la información viaje encriptada y retorne encriptada entre el cliente y servdor
`CacheSettings`	Informacion de cacheo de los servicios. No es necesario q el programador diseño su propio código de isolation. El framework lo hace automáticamente bajo parámetros standares y configuraciones altamente flexibels.

Context Information

El atributo ContextInformation es muy importante para obtener alguna informacion extra acerca del estado del servicio.

La siguiente tabla lista los atributos que dispone:

Propiedad	Descripción
HostName	Indica el host que inicio la peticion del servicio .
ServerName	Indica el server que atendio el servicio servicio .
UserName	Indica el usuario logueado en el host que inicio el servicio.
ServerTime	Indica fecha y hora de inicio o fin del servicio del lado del Servidor o despachador de servicio. Para un Request : fecha y hora de inicio del lado del server . Para un Response: fecha y hora de finalizacion del lado del server .
HostTime	Indica fecha y hora de inicio o fin del servicio del lado del Cliente. Para un Request: fecha y hora de inicio del lado del cliente (horario de inicio desde el host). Para un Response: fecha y hora de finalizacion del lado del cliente (horario de llegada al host).

Servicios Sincrónicos

Los servicios de negocio sincrónicos (transaccionales o de consulta) representan a *servicios* simples del Framework.

Son clases .NET que heredan de una clase base abstracta llamada BusinessService, provista por el Framework de Action Line, dicha clase base tiene un método que se debe implementar denominado Execute y exige que se pase un Request y retorne un Response.

```
public abstract TResponse Execute(TRequest pServiceRequest);
```

La clase base BusinessService Es la clase de la que deben heredar todas aquellas clases que sean implementaciones de servicios de negocio.

El Dispatcher recibe un objeto de tipo Request especializado (Ej. CrearClienteRequest) mas el nombre del servicio (Ej.: "CrearCliente"). Tal cual como se muestra con el ejemplo citado en "Utilización de las interfaces del lado del Cliente" cuando se dispara el servicio de *CrearFactura*.

El Request contiene la información necesaria para dar de alta un cliente más información de contexto del sistema que requiere el Dispatcher para identificar el servicio a ejecutar. El Dispatcher localiza la información necesaria para identificar el servicio a trabes de una meta data que puede ser un XML de configuración o una base de datos de SQL..
Una vez que el Dispatcher identifica el servicio a ejecutar, instancia la clase, visualizada anteriormente, y ejecuta el método *Execute*, pasando como parámetro un objeto de tipo Request especializado.

Una vez que el servicio ejecuta su propia lógica retorna un objeto de tipo Response especializado (Ej.: CrearClienteResponse).

Los servicios sincronos simples pueden ser transaccionales o de consulta, a continuación describiremos más detalladamente ambos y se mostraran ejemplos con pseudo código para su mayor comprensión.

Servicios de consulta (SC)

Los servicios de consulta solo recuperan datos del sistema de información y/o de sistemas externos. Estos interactuarán con uno o más BCs que accederán a los datos a través de los componentes de acceso a datos.

Los servicios de consulta retornarán DataSets no tipificados, dichos DataSets podrán ser retornados por BCs. Los datos contenidos en el DataSets podrán sufrir transformaciones, si fueran necesarias, realizadas en los o DACs, BCs.

A continuación se visualiza un pseudo ejemplo de un servicio sincrónico de consulta que utiliza directamente un BC para obtener los datos.

```
using System;
using Allus.SitemaContable.Backend Cliente.BC;
using Allus.SitemaContable.Common.Cliente.ISVC;

namespace Allus.SitemaContable.Backend.Cliente.SVC
{
   public class ConsultarClientesPorFiltroService : BusinessService< ConsultarClientesPorFiltroRec
  ConsultarClientesPorFiltroResponse>
  {
        ConsultarClientesPorFiltroResponse  Execute(ConsultarClientesPorFiltroRequest req )
        {
           ConsultarClientesPorFiltroResponse  res = new ConsultarClientesPorFiltroResponse  ();

          ClienteBC wClienteBC = new ClienteBC();
          DataSet  wDts = wClienteBC.ConsultarClientesPorFiltro(req.Param.Nombre,req.
  Param.Apellido);

                this.MappingResult (wDts, out res )

                return res;
        }

  }
}
```

Conclusiones:

- Son el punto de entrada para cualquier consulta que se quiera realizar sobre el sistema de información.
- Los servicios de consulta utilizarán BCs directamente retornan directamente DataSets, DataTables o entidades de negocios que heredan de las clases bases de framework.
- Informar, a través del sistema de excepciones del framework (excepciones técnicas o funcionales), cualquier error que pudo haber surgido. El detalle de cómo implementar este mecanismo de excepciones se explica en el documento: Arquitectura Tecnológica manejo de excepciones V2.0.doc

Servicios transaccionales

Los servicios transaccionales son aquellos que actualizan o mantienen el Entorno. También interactúan con uno o más BCs para ejecutar la transacción.

A continuación se visualiza la estructura básica de un servicio sincrónico que crea un cliente en el sistema.

```
using System;
using System.Data;
using Action Line.Framework.Bases.BackEnd;
using Allus.SitemaContable.BackEnd.Cliente.BE;
using Allus.SitemaContable.BackEnd.Cliente.BC;
using Allus.SitemaContable.Common.Cliente.ISVC;

namespace Allus.SitemaContable.BackEnd.Cliente.SVC
{
 public class CrearClienteService : BusinessService<CrearClienteRequest, CrearClienteResponse>
 {
       CrearClienteServiceResponse Execute(CrearClienteServiceRequest req)
       {
              CrearClienteResponse res = new CrearClienteResponse ();

              // Ejecutar lógica del servicio.

              return res;
       }
  }
}
```

Conclusiones:

- Los servicios transaccionales son utilizados cuando se requiere actualizar / mantener el sistema de información / sistemas externos involucrados.
- Los servicios transaccionales utilizan uno o más BCs para resolver su lógica.
- Son encargados de transformar / relacionar los datos del Request a los datos requeridos por los BCs, y de transformar / relacionar el resultado retornado por los BCs al Response.
- Informar, a través del sistema de excepciones del framework (técnicas o funcionales), cualquier error que puede haber surgido.
- La interfaz de entrada y salida de los STs son request y response especializados.
- La interfaz de entrada para interactuar con BCs son id's y/o BEs simples o colecciones.

Servicios Asíncronos

Los servicios asincrónicos están preparados para ejecutar varios pasos en secuencia sin que se consuman los recursos que administra el del framework del Dispatcher. Cuando un servicio se está ejecutando y realiza una operación que puede tomar un largo tiempo, se consumen recursos innecesariamente ya que el servicio está mucho tiempo esperando que otros componentes externos terminen de realizar sus tareas para retornar el valor final al Front-End.

Utilizando el sistema de conectores, el Dispatcher libera los recursos consumidos por un servicio, mientras el conector se encarga de ejecutar la transacción con un sistema externo y de esperar que la transacción finalice. Cuando esto ocurre, el conector comunica al *Dispatcher* para que continúe con la ejecución del servicio con los resultados obtenidos a través del conector. De esta forma mientras el conector realiza su tarea el Dispatcher puede atender otros servicios ya que tiene recursos libres para ello. De esta forma logramos mejorar el nivel de escalabilidad del sistema.

Los servicios asincrónicos interactúan con el sistema de conectores y a diferencia de los servicios sincrónicos, están constituidos por dos métodos, Execute y ExecuteStep. Los servicios asincrónicos tienen un método llamado Execute donde comienza la ejecución y el servicio prepara la llamada al sistema externo.

Cuando el conector finaliza la ejecución, devuelve los resultados al *Dispatcher*, que procede a ejecutar el paso siguiente llamando al método ExecuteStep. De esta forma el servicio conoce que se trata de la continuación de la ejecución a uno o varios conectores.
Los servicios asincrónicos heredan de la clase base BusinessAsyncService, a diferencia de los sincrónicos que heredan de BusinessService.

Conclusiones:

- Los servicios asincrónicos se utilizan solo en escenarios en los cuales se requiere ejecutar consultas o transacciones en-línea contra sistemas externos o para ejecutar servicios de consulta o transaccionales cuyo tiempo de respuesta es elevado. En estos casos existe mucho tiempo de I/O dado que el procesamiento de la transacción no consume recursos del Dispatcher, sino del sistema en el cual se ejecuta dicha transacción o consulta.

- Dado que un servicio asincrónico puede involucrar uno o más pasos, es posible que se ejecuten consultas o transacciones contra sistemas externos y, en el mismo servicio, consultar o ejecutar una actualización al sistema propio.

- Para utilizar servicios asincrónicos es necesario desarrollar previamente los conectores a los sistemas externos necesarios.

Conectors

Uno de los objetivos del sistema de conectores externos es unificar la interfaz de programación para los servicios que acceden a sistemas externos. A efectos de cumplimentar dicho objetivo, se ha definido un mecanismo por el cual los desarrolladores de servicios puedan interactuar con otros sistemas sin la necesidad de aprender una nueva interfaz de programación para interactuar con cada uno de ellos.

De esta forma los servicios pueden solicitar la ejecución de una transacción o consulta a un sistema externo utilizando siempre una misma interfaz. Este soporte de conectores es también extensible de manera tal que los usuarios pueden crear nuevos conectores que interactúen con sistemas externos propios.

El sistema de conectores está pensado para que se puedan crear nuevos conectores cuando sean necesarios. Los conectores son clases .NET que implementan la interfaz IServiceConnector que se define en el Framework.

Conclusiones:

- Los conectores son necesarios para ejecutar transacciones *en-línea* o consultas a través de servicios asincrónicos.

- Los conectores deben ser piezas de código eficientes que conocen con mucho nivel de detalle los sistemas externos con los cuales se encuentran interactuando.

- Dado que los conectores son utilizados en el sistema *en-línea*, deben estar orientados a minimizar el tiempo de respuesta. Ejemplo, si sabemos que la conexión a SAP toma bastante tiempo, es responsabilidad del conector de manejar un pool de conexiones.

- La transaccionabilidad cuando se interactúa con sistemas externos a través de conectores está dada por mecanismos de compensación / reversas.

- El subsistema de conectores define una interfaz programática común, para ser utilizada en los servicios de negocio en los escenarios en los cuales se requiera acceso a sistemas externos o tecnologías / rutinas con tiempos altos de respuesta.

- La comunicación con el sistema de conectores internos es a través de las DACs. Es decir que si los componentes BCs necesitan hacer una llamada a un servicio de un sistema externo utilizarán algún componente DAC que exponga la interfaz que necesite el conector a dicho sistema.