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. CrearClienteReq) más el nombre del servicio (Ej.: "CrearClienteService"). 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.: CrearClienteRes).

Los servicios síncronos 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 Pelsoft.SitemaContable.Backend.Cliente.BC;
using Pelsoft.SitemaContable.Common.Cliente.ISVC;

namespace Pelsoft.SitemaContable.Backend.Cliente.SVC
{
   public class BuscarClientesService : BusinessService< BuscarClientesReq, BuscarClientesRes>
  {
        BuscarClientesRes  Execute(BuscarClientesReq req )
        {
           BuscarClientesRes  res = new BuscarClientesRes  ();

         ClienteList list =
  ClienteDAC.BuscarClientes(req.BusinessData.Nombre,req.BusinessData.Apellido);

        Res.BusinessData = list;

        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/DACs directamente retornan directamente 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 o DACs 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 Pelsoft.SitemaContable.BackEnd.Cliente.BE;
using Pelsoft.SitemaContable.BackEnd.Cliente.BC;
using Pelsoft.SitemaContable.Common.Cliente.ISVC;

namespace Pelsoft.SitemaContable.BackEnd.Cliente.SVC
{
 public class CrearClienteService : BusinessService<CrearClienteReq, CrearClienteRes>
 {
        CrearClienteRes Execute(CrearClienteReq req)
        {
                CrearClienteRes res = new CrearClienteRes ();

                // Ejecutar lógica del servicio.
                   ClienteDAC.Create(res.);
                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/DACs 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.

Contexto Transaccional de servicios

Todo el manejo transaccional de un servicio es automáticamente gestionado por el Core del Orquestador de servicio (Dispatcher).
Para definir un servicio como transaccional basta con establecer su metadata. Los atributos relacionados a transacciones son:

Cortamiento del ámbito transaccional.
TransactionalBehaviour Support, Required, RequiresNew

Nivel de aislamiento de la transacción.
IsolationLevel ReadCommitted, ReadUncommitted, RepeatableRead etc

Para poder configurar la metadata de un servicio se puede utilizar la herramienta ServiceMetadataConfig del FWK documentada en ServiceManagement.docx

Dicha herramienta permite administración de los servicios SOA-Fwk o SVC.

En los casos donde los servicios marcados como transaccionales necesiten excluir de un contexto transaccional parte de su código: (ejemplo consultas y mapeos que leleven mucho tiempo) podemos crear una transacción de tipo **Suppres** en la parte exacta donde la transacción NO sea requerida.

Con Suppres todas las operaciones se hacen sin estar en un ámbito transaccional.

```
using (TransactionScope innerScope = new TransactionScope(TransactionScopeOption.Suppress))
            {
                // Codigo que desee excluir de la Tx
                innerScope.Complete();
            }
```

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.

Interfaces de servicios Request y Response

Las interfaces Request y Response son los contratos que los aplicativos clientes deberían entender y respetar con la finalidad de consumir los servicios de negocios.

Ejemplo de Implementación

Mostraremos a continuación un ejemplo práctico de como se ven implementados los objetos `Request` y `Response` y como son llamados desde una aplicación cliente.

Construcción de las interfaces o contratos del servicio

Interfaz de petición (Request)

~~Lo primero que debemos crear es un esquema que represente funcionalmente los datos que serán enviados al servicio. Supongamos un servicio que cree una factura en un sistema, lo llamaremos `CrearFacturaService`. Más adelante detallaremos como se implementa un servicio.~~
~~Diseñamos el esquema correspondiente al Request:~~

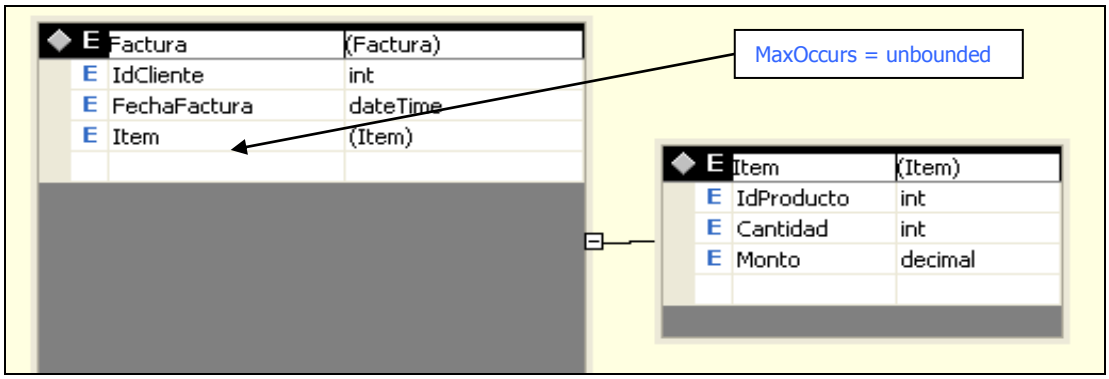

Figura 4. CrearFacturaREQ.xsd

En base a este esquema creamos la clase FacturaBE, ItemBE y ItemBECollection que heredan de Entity y corresponden a los datos funcionales de entrada al servicio.

FacturaBE como es la cabecera de todas las demás es la que va a ser llamada desde el objeto CrearFacturaReq principal a través del atributo heredado "BusinessData"
Ej: CrearFacturaReq.BusinessData

Por lo tanto, una vez que tenemos diseñadas las clases de negocio, solo nos falta construir una clase que sea la que va a heredar de la clase Request, es decir CrearFacturaReq:

```
public class CrearFacturaReq : Request< FacturaBE >
{
}
```

A continuación mostramos como nos queda el código del Request completo:

```
using System;
using System.Collections.Generic;
using Fwk.Bases;
using System.Xml.Serialization;

namespace Pelsoft.SistemaContable.Facturas.ICVC
{
   public class CrearFacturaReq : Request<FacturaBE >
   {

   }
```

```
[XmlInclude(typeof(FacturaBE)), Serializable]
public class FacturaBE : Entity
{
    public int? IdCliente{ get ; set; }
    public DateTime? FechaFactura   { get ; set; }
    public ItemBECollection ItemBECollection { get ; set; }
}

public class ItemBECollection : Entities<ItemBE>{}

[XmlInclude(typeof(ItemBE)), Serializable]
public class ItemBE : Entity
{
    public int? IdProducto{ get ; set; }
    public int? Cantidad{ get ; set; }
}
}
```

Esta clase CrearFacturaReq es la clase que realmente recibe el Dispatcher y contiene tanto información técnica o de contexto como información funcional o de negocio.
Gráficamente a un objeto Request lo podemos ver de la siguiente manera:

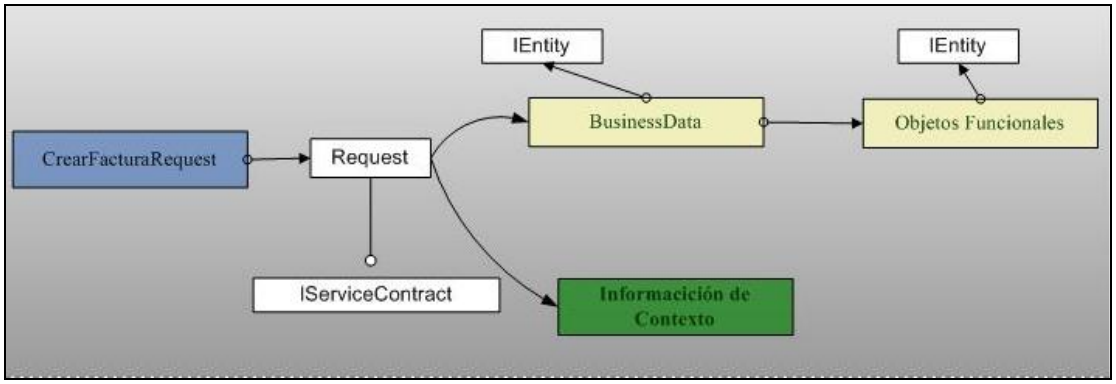

Figura 5. Modelo Request de un Servicio

Interfaz de respuesta

De manera similar necesitamos construir una interfaz que represente el resultado del servicio, es decir, el Response.
~~Entonces construimos el esquema XSD que lo represente, como se muestra a continuación:~~

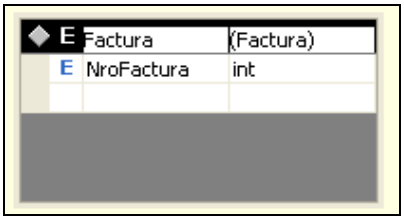

Figura 6: CrearFacturaRES.xsd

Luego desarrollamos el objeto de negocio, `FacturaBE`, que representa al esquema de arriba viéndose como se muestra en sniped debajo.

```
using System;
using System.Collections.Generic;
using Fwk.Bases;
using System.Xml.Serialization;

namespace Pelsoft.SitemaContable.Facturas.ICVC
{
    /// <summary>
    /// Request CrearFacturaRes .-
```

```
    /// </summary>
    public class CrearFacturaRes : Response< FacturaBE >
    {
    }
    public class FacturaBE : Entity
    {
        public int? NroFactura { get ; set; }
    }
```

Por último solo nos resta construir la clase CrearFacturaRes que la agregamos en el mismo namespace donde agrego la clase FacturaBE. Cuando creamos esta clase lo que estamos haciendo vemos que le pasamos como parámetro genérico, a la clase base Response, un tipo de dato definido por nosotros que es FacturaBE y esta misma clase va a ser la será el contenido del **BusinessData** del Response.

Entonces cuando se quiera acceder al valor de negocio retornado por el servicio se lo haga de la siguiente manera:

```
CrearFacturaRes res = null;
lblFactura.Text = "Factura N°: " + res.BusinessData.NroFactura;
```

Tipo genérico funcional recibido por un Request o Response (IEntity)

Como habrán notado las dos clases tanto Request como Response recientemente construidas son definidas como una clase que hereda de una clase base que recibe como parámetro genérico cualquier clase que implemente IEntity (Entity o Entities). Estas clases bases están definidas en el framework.

Por lo tanto si en nuestra clase del ejemplo:

CrearFacturaReq : Request<FacturaBE>

Es un request que recibe un objeto de negocio de tipo factura. Si quisiéramos que, en un futuro, este request pueda recibir una colección de facturas (lista), simplemente bastaría con definir una clase de tipo Entities<Entity> llamémosla FacturaList que reciba una clase FacturaBE. Es decir crearíamos una lista contenedora de objetos FacturaBE:

```
public class FacturaList : Entities<FacturaBE>{}
```

Y luego adaptar nuestra clase CrearFacturaReq para que reciba tal colección de modo que se vea de la siguiente manera:

```
[Serializable]
public class CrearFacturaReq : Request<FacturaList>
{
}
```

Esto es posible dado que la firma de las clases Requests o Responses base es:

```
:Request<T> where T : IEntity
:Response<T> where T : IEntity

Y

Entity : IEntity
Entities : IEntity
```