**Frornt End**

Las aplicaciones clientes pueden ser aplicaciones web, win 32 o incluso algún otro servicio que requiera hacer peticiones de algún servicio al Dispatcher.

Desde el punto de vista del cliente las piezas más importantes son:

- ClientServiceBase
- Controller
- Wrapper
- Request y Response

**Wrapper**

Esta pieza de software está ubicada del lado del cliente y es la encargada de recibir todas las peticiones que los clientes hacen al servidor o distpatcher.

Este componente tiene la inteligencia para establecer la información de contexto del lado del cliente de los servicios y además se encarga de la serializacion automática de los parámetros `Request` o `Responce` para enviarlos al servidor.

Por otro lado detecta la configuración de canal para poder ubicar al dispatcher y abstrae al cliente de toda complejidad de comunicación ante un dispatcher montado en un WebService o un Servicio de windows remoting.

El Wrapper dispone de un método **`ExecuteService`** que es público para los clientes y permite ejecutar un servicio de negocio.

La interfaz genérica de este método es la siguiente:

```
TResponse ExecuteService<TRequest, TResponse>(string providerName,TRequest pRequest)
            where TRequest : IServiceContract
            where TResponse : IServiceContract, new()
```

Es un método que recibe como parámetros el nombre del servicio más un objeto Request.
Como notaran este objeto debe implementar la interfaz IServiceContract y esto lo hace automáticamente heredando de la clase Request explicada anteriormente.

Por otro lado retorna un objeto Responce que implementa también la interfaz IServiceContract, que de la misma manera al heredar de la clase base Response también implementa esta interfaz.

Tipos de Wrapper:

Wrapper local
Utiliza cuna comunicación directa. No hace ninguna llamada a servicios remotos atreves de protocolos como http o TCP
Es muy útil para realizar pruebas unitarias y/o servicios donde no sea necesario el desacople con el BackEnd

Remoting wrapper
Es el wrapper que permite la comunicación por medio del protocolo TCP con un servicio de windwos. El servicio de Windows es provisto por el framework y se basa en tecnología de remoting para atender todas las peticiones de los clientes.

Esta modalidad realice serializacion binaria de los servicios y es propicia para trabajar en una LAN.

Web service Wrapper:
Es el wrapper que permite la comunicación por medio del protocolo HTTP con un servicio de web service utilizando el protocolo SOAP.

## Implementación de los wrappers en la arquitectura

Las aplicaciones Front-End utilizan el wrapper para hacer llamadas a servicios de negocios que se ejecutan en despachadores de servicios. Dentro del framework existen clases que automáticamente realizan la instanciación de los diferentes wrappers configurados.

Estas clases son:

FrmBase, UserControlBase y la clase base de ambos `ClientServiceBase`

Nosotros como desarrolladores bien podríamos hacer heredar nuestras clases formulario o user controls de `FrmBase` y `UserControlBase` respectivamente o bien, podríamos crear una clase cualquiera que no sea necesariamente visual y hacerla que herede de `ClientServiceBase.`

Lo que `ClientServiceBase` hace es abstraernos de todo tipo de complejidad de factoia de wrapper y demas cuestiones de servicios. Entre ellas:

- ✓ Catching de servicios
- ✓ Seteo inicial de atributos de seguridad
- ✓ Seteo inicial de atributos de auditoria
- ✓ Ejecución de servicios
- ✓ Manejo de errores en el cliente
- ✓ Etc.

```
public static TResponse ExecuteService<TRequest, TResponse>(string providerName,TRequest pRequest)
```

Configuración del wrapper

Ver documento:
Arquitectura Tecnológica Configuración wrapper.pdf

## 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:

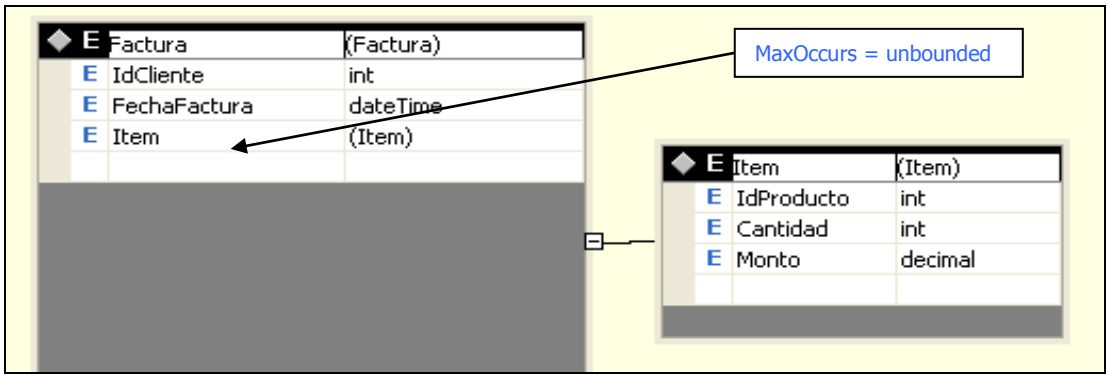



**Figura 4. CrearFacturaREQ.xsd**

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. Notemos aquí que la primera clase FacturaBE es la principal y la que encapsula todos los datos de negocio y es la que tiene la posibilidad de hacer llamadas a las demás subclasess clases que aparecen en el esquema (en este caso solo la colección de Items).

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

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

A contunuacion mostramos como nos queda el codigo del Request completo:

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

namespace Allus.SistemaContable.Facturas.ICVC
{
   public class CrearFacturaRequest : Request<FacturaBE >
   {
   }

   [XmlInclude(typeof(FacturaBE)), Serializable]
   public class FacturaBE : Entity
   {
      private int? _IdCliente;
      private DateTime? _FechaFactura;
      private ItemBECollection _ItemBECollection = new ItemBECollection();
```

```
        public int? IdCliente
        {
            get { return _IdCliente; }
            set { _IdCliente = value; }
        }
        public DateTime? FechaFactura
        {
            get { return _FechaFactura; }
            set { _FechaFactura = value; }
        }
        public ItemBECollection ItemBECollection
        {
            get { return _ItemBECollection; }
            set { _ItemBECollection = value; }
        }
    }

    public class ItemBECollection : Entities<ItemBE>{}

    [XmlInclude(typeof(ItemBE)), Serializable]
    public class ItemBE : Entity
    {
        private int? _IdProducto;
        private int? _Cantidad;

        public int? IdProducto
        {
            get { return _IdProducto; }
            set { _IdProducto = value; }
        }

        public int? Cantidad
        {
            get { return _Cantidad; }
            set { _Cantidad = value; }
        }

    }
}
```

Esta clase CrearFacturaRequest 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:

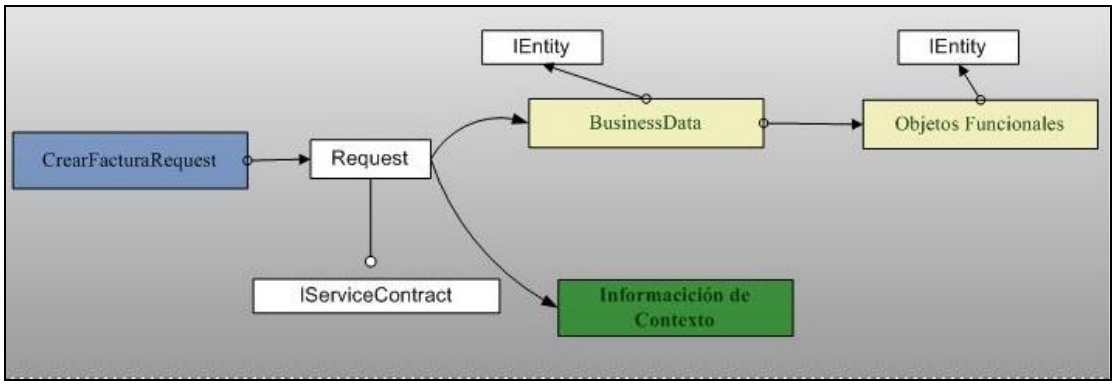



**Figura 5. Modelo Request de un Servicio**

### 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:

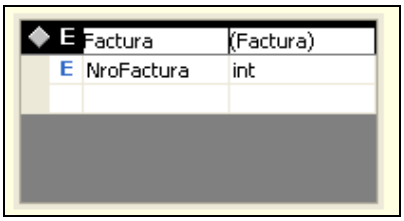



**Figura 6: CrearFacturaRES.xsd**

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 Allus.SitemaContable.Facturas.ICVC
{
    /// <summary>
    /// Request CrearFacturaResponse .-
    /// </summary>
    public class CrearFacturaResponse : Response< FacturaBE >
    {
    }

    public class FacturaBE : Entity
    {
        private int? _NroFactura;

        public int? NroFactura
        {
            get { return _NroFactura; }
            set { _NroFactura = value; }
        }
```

```
    }
```

Por último solo nos resta construir la clase CrearFacturaResponse 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:

```
CrearFacturaResponse wResponse = null;
lblFactura.Text = "Factura N°: " + wResponse.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:

CrearFacturaRequest : 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 CrearFacturaRequest para que reciba tal colección de modo que se vea de la siguiente manera:

```
[Serializable]
public class CrearFacturaRequest : 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
```

**Utilización de las interfaces del lado del Cliente:**

Tanto los Requests como los Responses creados en la sección anterior son utilizados en el Back-End por los servicios y en el Front-End por los mismos clientes para llamar los servicios o bien para utilizar sus resultados.

La utilización de estas clases es muy sencilla, simplemente basta con crear una instancia de la clase Request y rellenarla con la información necesaria de acuerdo las reglas de negocios exigidas y luego pasársela al método del ***ExecuteService (..) .***

*Nota: El método anteriormente nombrado es de la clase ClientServiceBase. De modo que si queremos ejecutar un servicio tendriamos que heredar de esta directamente o atravez de FrmBase o UserControlBase o utra clase personalizada que anteriormente haya heredado de ClientServiceBase*

La mejor forma de entender esto es a través de un ejemplo:

Supongamos un método dentro de nuestro formulario de Windows (que hereda de `ClientServiceBase`) que rellene el objeto CrearFacturaRequest a partir un control DataGridView. Entonces para encapsular la complejidad dejamos que toda esta tarea sea llevada a cabo en un método separado, lo llamaremos BuildRequest.

Nota: Con respecto a al información de contexto del request, no es necesario que el desarrollador se esfuerce en establecer sus valores, se debe dejar este trabajo para que sea realizado por la lógica del wrapper.

```
private CrearFacturaRequest BuildRequest()
{
        CrearFacturaRequest wRequest = new CrearFacturaRequest();


        wRequest.BusinessData.IdCliente = (int) txtNumeroCliente.Text;
        wRequest.BusinessData.FechaFactura = cmbFecha.Value;
        foreach (DataGridViewRow wRow in grdDetalle.Rows)
            {
              ItemBE wItemBE = new ItemBE();

              wItemBE.Cantidad = Convert.ToInt32(wRow.Cells["Cantidad"].Value);
              wItemBE.IdProducto = Convert.ToInt32(wRow.Cells["IdArticulo"].Value);

              wRequest.BusinessData.ItemBEList.Add(wItemBE);
            }

            return wRequest;
}
```

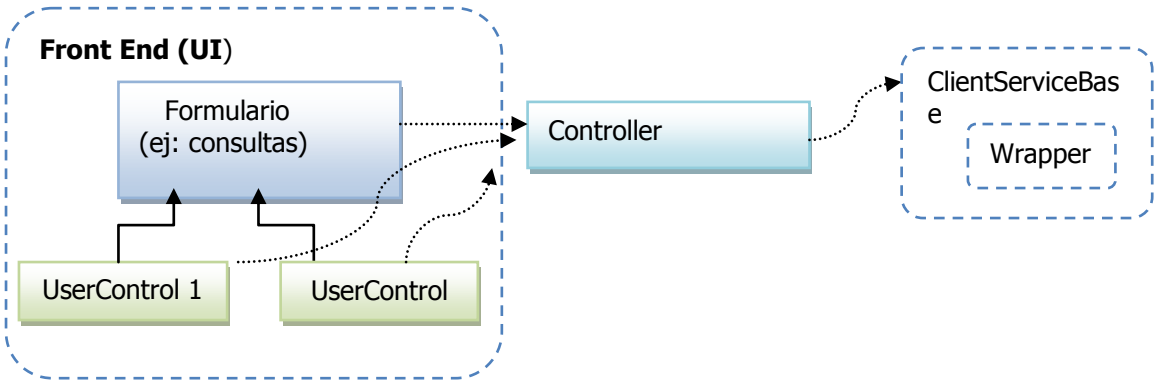
Luego, una vez que llenamos el Request con la información necesaria, lo que nos resta es llamar al servicio pasándole este y esperar el resultado.
Para esto solo tenemos que enviar esta información al dispatcher y esto lo hacemos a través del ExecuteService que esta embebido en la clase base.

```
private void btnCrear_Click(object sender, EventArgs e)
{
 CrearFacturaResponse res= null;

req = BuildRequest();
//no espesificamos provedor de wrapper
res = base.ExecuteService<CrearFacturaRequest,CrearFacturaResponse>( wRequest);

MessageBox.Show("Se creó la Factura N°: " + wResponse.Result.NroFactura);

if (res.Error != null)
        {
           throw Fwk.Exceptions.ProcessException(res.Error);
        }
}
```

*Nótese que en este caso no se especifica el nombre del proveedor de wrapper en la ejecución del servicio. Por lo tanto se utilizara el proveedor por defecto.*

**Arquitectura de front end**

En esta sesión se intentara establecer una modalidad estándar de diseño de los componentes fron end para la comunicación a través de servicios.




Aquí tenemos un formulario cualquiera que utiliza sus user controls todos estos componentes interactúan con un componente común el controller.

Por lo tanto, los componentes de UI utilizaran una instancia singlenton del controller o un controller estático. (El diseño del controller estático o instanciado bajo patrón singlenton se deja a criterio del grupo de desarrollo)

La idea de crear un controller es solo para definir un único lugar donde se alojen todas las llamadas a los servicios. Trabajando de esta manera logramos un estándar y/o convención de trabajo, de modo que cuando ocurran cambios en llamadas a servicios o siertos mudulos del sistema dejen de utilizar una arquitectura SOA. Simplemente se actualizaran una capa/componente de la arquitectura → El controller.

Responsabilidades del controller:

- El controller es el único componente que tendrá la responsabilidad de hacer llamadas a los servicios.

El controller es un componente que hereda de ClientServiceBase

Ejemplo: `public class SurveyController : Fwk.Bases.FrontEnd.ClientServiceBase`

La clase base ClientServiceBase la provee el framework y es la que tiene toda la inteligencia para resolver las siguientes cuestiones:

- Identificar el o los wrapper/s de comunicación con el despachador de servicios.
- Aplica lógica de catching de acuerdo los parámetros establecidos en los componentes de chache en el Request.
- Seteo inicial de atributos de seguridad
- Seteo inicial de atributos de auditoria
- Ejecución de servicios
- Manejo de errores en el cliente
- Etc.

**Patrón de ejecución de servicios en el controller:**

A continuación se muestran unos ejemplos de métodos que realizan llamadas a servicios del lado del cliente por medio de un controller.

Dado el siguiente servicio:

```
SearchParamsByTypeService:BussinesService<SearchParamsByTypeRequest,SearchParamsB
yTypeResponse>

SearchParamsByTypeRequest: Request<Param>
SearchParamsByTypeResponse:Response<ParamList>
```

Se trata de un servicio de busqueda de tipos de parametros por tipo:
Ejemplo: Tipos de Colores, tipos de empleado, tipos de salas, etc.

***Nota:*** *en otra sección se explicara el patrón de Parameters con mayores detalles.*

```
/// <summary>
/// Busca params por
/// </summary>
/// <param name=" pType">tipo de la enumeracion TypesEnum </param>
/// <returns>lista de valores tipo params</returns>
public ParamList SearchParams(ParamsEnum.TypesEnum pType)
      {

       /Creo Request
       SearchParamsByTypeRequest req = new SearchParamsByTypeRequest();
       //Seteo parametros
       req.BusinessData.ParamTypeId = pType;
       req.CacheSettings.CacheOnClientSide = true;

       ///Llamo al servicio y obtengo el response
       SearchParamsByTypeResponse res =
                base.ExecuteService< SearchParamsByTypeRequest,
                                     SearchParamsByTypeResponse>
                                     ("NOMBRE_PROV_WRAPPER",req);

        //Si veinen errores proceso la excepcion
       if (res.Error != null)
        {
                throw Allus.Common.Helpers.Exceptions.ProcessException(res.Error);
        }
```

```
        //retorno el resultado (colección de parametros)
        return res.BusinessData;
}
```

Note que en el controller no hay nada relacionado a coneccions a web services o servicios remotos, tampoco hay reglas de negocio ni accesos a bases de datos. Toda esa complejidad es absorvida por las capas correspondientes como catching, blocking, loging del framework y Dispatcher,SVC, BC, DAC etc diseñados bajo los patrones de la arquitectura.

Entonces desde cualquier componente como un formulario o user control podríamos llenar una grilla haciendo una llamada al controller:

```
private void btnfillGrid_Click(object sender, EventArgs e)
{
   grgParamsGrid.DataSource =
        _Controller.SearchParams(ParamsEnum.TypesEnum.TipoEmpleado);
}
```