Contenido

[Inicio 2](#_Toc42003145)

[Introducción a las buenas prácticas de programación 4](#_Toc42003146)

[Nomenclatura C# Coding Standards and Naming Conventions 4](#_Toc42003147)

[SOLID 10](#_Toc42003148)

[Instalación de Telerik 11](#_Toc42003149)

[Empezando el programa 12](#_Toc42003150)

[Crear nuevo mantenimiento 17](#_Toc42003151)

[JSON 19](#_Toc42003152)

[UserControls 20](#_Toc42003153)

[Crear Botones 20](#_Toc42003154)

[VirtualGrid y GridView 20](#_Toc42003155)

[Operaciones específicas en VirtualGrid 21](#_Toc42003156)

[Operaciones específicas en GridView 22](#_Toc42003157)

[WaitingBar y BackgroundWorkers 23](#_Toc42003158)

[BaseGridControl 24](#_Toc42003159)

[SFDetalles 25](#_Toc42003160)

[DataDialog 25](#_Toc42003161)

[Pedir datos a SQL 25](#_Toc42003162)

[Business 25](#_Toc42003163)

[DataAccess 25](#_Toc42003164)

[ConexionSQL 25](#_Toc42003165)

[El esquema 30](#_Toc42003166)

[Jerarquía Estandard 31](#_Toc42003167)

[Jerarquía On Demand 32](#_Toc42003168)

[RadProgressBar para columnas 34](#_Toc42003169)

[Árbol de nodos 37](#_Toc42003170)

[Lenguaje 39](#_Toc42003171)

[Colores en celdas dependiendo de su valor 40](#_Toc42003172)

[Gestión de excepciones 43](#_Toc42003173)

[Generar Versiones 43](#_Toc42003174)

[Temas 44](#_Toc42003175)

[Summary Rows 45](#_Toc42003176)

[RadContextMenu 46](#_Toc42003177)

[RadScheduler 47](#_Toc42003178)

[RadTextBox 49](#_Toc42003179)

[Documentación Telerik 51](#_Toc42003180)

[Apuntes sobre Visual Studio 52](#_Toc42003181)

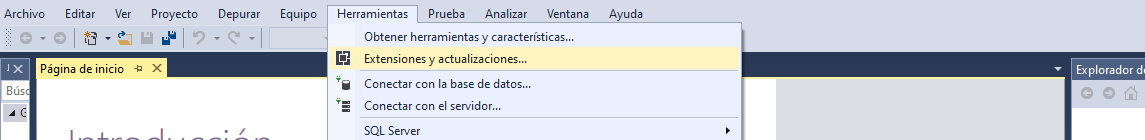
## Inicio

Lo primero será clonarnos el repositorio,lo podemos hacer a través de GitHub en el siguiente enlace:

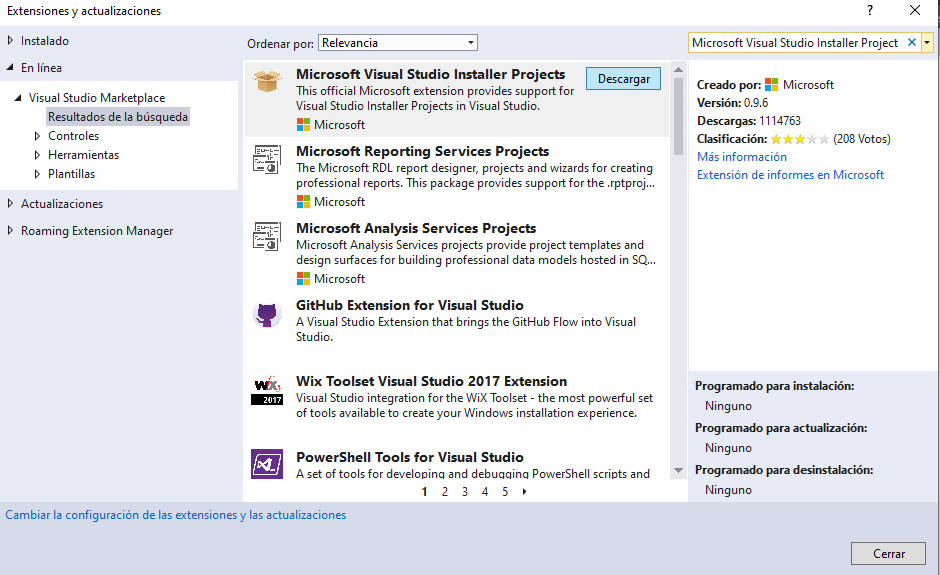
<https://github.com/chidalgoRumbo/Rumbo2020_2.0>

Para poder acceder se necesita autorización.

También necesitaremos instalarnos desde visual studio Herramientas>Extensiones:



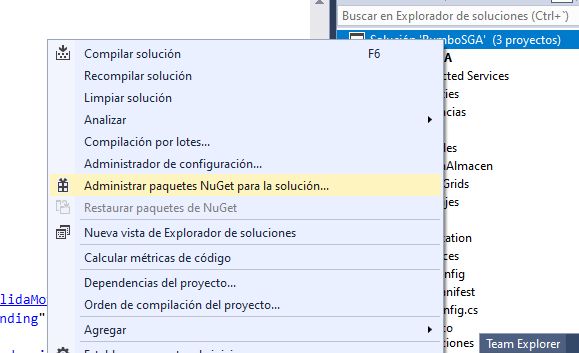
Microsoft Visual Studio Installer Project

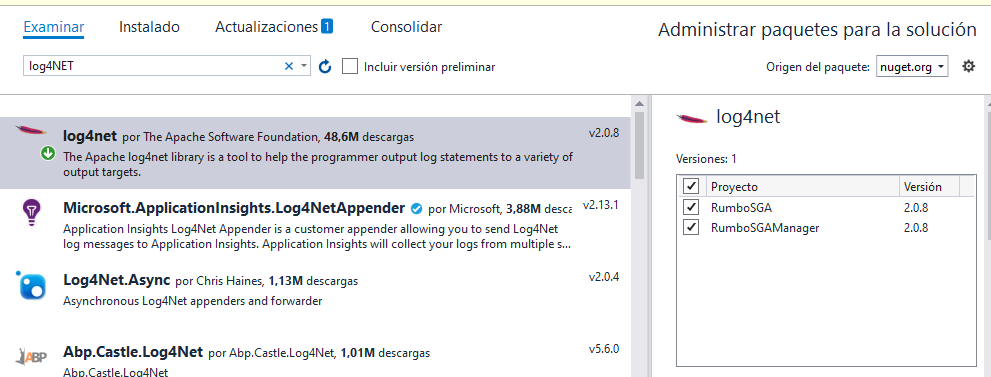


Y desde Herramientas>Administrador de Paquetes NuGet>Administrar paquetes NuGet para la solución

-log4NET

-Newtonsoft.Json





Para gestion de versiones: Automatic Versions 2

<https://marketplace.visualstudio.com/items?itemName=PrecisionInfinity.AutomaticVersions>

Aparte necesitaremos la carpeta Configs en la ruta C:\Rumbo, esta carpeta se sube a P:\UltimasVersiones\Rumbo2020.

(Por el coronavirus y que no siempre está disponible la VPN en este momento los configs se están subiendo al grupo de Teams ImplantaciónSGA en la sección de archivos, junto al instalador)

## Introducción a las buenas prácticas de programación

### Nomenclatura C# Coding Standards and Naming Conventions

Below are our **C# coding standards**, naming conventions, and best practices.  
Use these in your own projects and/or adjust these to your own needs.

douse **PascalCasing** for class names and method names.

1. public class ClientActivity
2. {
3. public void ClearStatistics()
4. {
5. //...
6. }
7. public void CalculateStatistics()
8. {
9. //...
10. }
11. }

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

douse **camelCasing** for method arguments and local variables.

1. public class UserLog
2. {
3. public void Add(LogEvent logEvent)
4. {
5. int itemCount = logEvent.Items.Count;
6. // ...
7. }
8. }

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

do notuse **Hungarian** notation or any other type identification in identifiers

1. // Correct
2. int counter;
3. string name;
5. // Avoid
6. int iCounter;
7. string strName;

**Why**: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

do notuse **Screaming Caps** for constants or readonly variables

1. // Correct
2. public static const string ShippingType = "DropShip";
4. // Avoid
5. public static const string SHIPPINGTYPE = "DropShip";

**Why**: consistent with the Microsoft's .NET Framework. Caps grap too much attention.

avoidusing **Abbreviations**. Exceptions: abbreviations commonly used as names, such as **Id, Xml, Ftp, Uri**

1. // Correct
2. UserGroup userGroup;
3. Assignment employeeAssignment;
5. // Avoid
6. UserGroup usrGrp;
7. Assignment empAssignment;
9. // Exceptions
10. CustomerId customerId;
11. XmlDocument xmlDocument;
12. FtpHelper ftpHelper;
13. UriPart uriPart;

**Why**: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

douse **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

1. HtmlHelper htmlHelper;
2. FtpTransfer ftpTransfer;
3. UIControl uiControl;

**Why**: consistent with the Microsoft's .NET Framework. Caps would grap visually too much attention.

do notuse **Underscores** in identifiers. Exception: you can prefix private static variables   
                    with an underscore.

1. // Correct
2. public DateTime clientAppointment;
3. public TimeSpan timeLeft;
5. // Avoid
6. public DateTime client\_Appointment;
7. public TimeSpan time\_Left;
9. // Exception
10. private DateTime \_registrationDate;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).  
  
douse **predefined type names** instead of system type names like Int16, Single, UInt64, etc

1. // Correct
2. string firstName;
3. int lastIndex;
4. bool isSaved;
6. // Avoid
7. String firstName;
8. Int32 lastIndex;
9. Boolean isSaved;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read.

douse implicit type **var** for local variable declarations. Exception: primitive types (int, string,   
          double, etc) use predefined names.

1. var stream = File.Create(path);
2. var customers = new Dictionary();
4. // Exceptions
5. int index = 100;
6. string timeSheet;
7. bool isCompleted;

**Why**: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

douse noun or noun phrases to name a class.

1. public class Employee
2. {
3. }
4. public class BusinessLocation
5. {
6. }
7. public class DocumentCollection
8. {
9. }

**Why**: consistent with the Microsoft's .NET Framework and easy to remember.  
  
doprefix interfaces with the letter **I**.  Interface names are noun (phrases) or adjectives.

1. public interface IShape
2. {
3. }
4. public interface IShapeCollection
5. {
6. }
7. public interface IGroupable
8. {
9. }

**Why**: consistent with the Microsoft's .NET Framework.

doname source files according to their main classes. Exception: file names with partial classes  
          reflect their source or purpose, e.g. designer, generated, etc.

1. // Located in Task.cs
2. public partial class Task
3. {
4. //...
5. }
6. // Located in Task.generated.cs
7. public partial class Task
8. {
9. //...
10. }

**Why**: consistent with the Microsoft practices. Files are alphabetically sorted and partial classes remain adjacent.

doorganize namespaces with a clearly defined structure

1. // Examples
2. namespace Company.Product.Module.SubModule
3. namespace Product.Module.Component
4. namespace Product.Layer.Module.Group

**Why**: consistent with the Microsoft's .NET Framework. Maintains good organization of your code base.

dovertically align curly brackets.

1. // Correct
2. class Program
3. {
4. static void Main(string[] args)
5. {
6. }
7. }

**Why**: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

dodeclare all member variables at the top of a class, with static variables at the very top.

1. // Correct
2. public class Account
3. {
4. public static string BankName;
5. public static decimal Reserves;
7. public string Number {get; set;}
8. public DateTime DateOpened {get; set;}
9. public DateTime DateClosed {get; set;}
10. public decimal Balance {get; set;}
12. // Constructor
13. public Account()
14. {
15. // ...
16. }
17. }

**Why**: generally accepted practice that prevents the need to hunt for variable declarations.  
  
douse singular names for enums. Exception: bit field enums.

1. // Correct
2. public enum Color
3. {
4. Red,
5. Green,
6. Blue,
7. Yellow,
8. Magenta,
9. Cyan
10. }
12. // Exception
13. [Flags]
14. public enum Dockings
15. {
16. None = 0,
17. Top = 1,
18. Right = 2,
19. Bottom = 4,
20. Left = 8
21. }

**Why**: consistent with the Microsoft's .NET Framework and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise 'OR').

do notexplicitly specify a type of an enum or values of enums (except bit fields)

1. // Don't
2. public enum Direction : long
3. {
4. North = 1,
5. East = 2,
6. South = 3,
7. West = 4
8. }
10. // Correct
11. public enum Direction
12. {
13. North,
14. East,
15. South,
16. West
17. }

**Why**: can create confusion when relying on actual types and values.

do notsuffix enum names with Enum

1. // Don't
2. public enum CoinEnum
3. {
4. Penny,
5. Nickel,
6. Dime,
7. Quarter,
8. Dollar
9. }
11. // Correct
12. public enum Coin
13. {
14. Penny,
15. Nickel,
16. Dime,
17. Quarter,
18. Dollar
19. }

**Why**: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

### SOLID

Seguiremos las recomendaciones S.O.L.I.D. :

**SRP**: Principio de única responsabilidad. Una clase solo hace una cosa, por ejemplo insertar la salida, otra clase genera el packing list, otra clase… Mostrar mensaje de error está en otra clase.

**OCP**: Principio Abierto / Cerrado. Las clases deben estar abiertas para extenderse pero cerradas para no tener que modificarse.

**LSP**: Principio de sustitución de Liskov. Las clases extendidas deben tener las propiedades suficientes para sustituir a la clase principal.

**ISP** Principio de segregación de la interface. Una clase no debe implementar métodos que no necesita. En ese caso es mejor crear dos interfaces.

**DIP**: Principio de inversión de dependencia. Los módulos de alto nivel no deben depender de módulos de bajo nivel sino que ambos deben depender de abstracciones. Spring implementa este principio externalizando la dependencia a ficheros XML

## Instalación de Telerik

Entrar a la página web de Telerik y loggearnos con el siguiente usuario:

usuario: "pboix@rumbosistemas.es",

contraseña: "Es3a2xTelerik"

Este usuario es también el que usaremos para pedir soporte en su página. Una vez loggeados entramos al perfil y buscamos la pestaña “Downloads”. Dentro de esta pestaña buscamos “Progress Telerik UI for WinForms” y lo descargamos. Abrimos el archivo que acabamos de descargarnos,seleccionamos la herramienta a instalar (Telerik UI for Winforms),la seleccionamos y pulsamos “Proceed”. Una vez instalado ya podremos usar los dll de telerik en nuestro proyecto,aparte se nos abrá instalado la aplicación “Demo” en la que podremos ver ejemplos tanto visuales como de código de las distintas herramientas de Telerik.

Desde el panel progress también realizaremos las actualizaciones.

Para pedir soporte hay que especificar que quieres que mande copia a tu correo, ya que por defecto los correos de Telerik se mandan únicamente a Pablo. En todo caso puedes ver tus tickets abiertos dentro del perfil en su pagina web.

**Log**

Para el log usamos la extensión log4NET. Para usarla deberemos instanciarla al inicio de la clase:

private static readonly log4net.ILog log = log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

Para registrar en el log usaremos log.Tipo,el cual puede ser:

log.Debug

log.Info

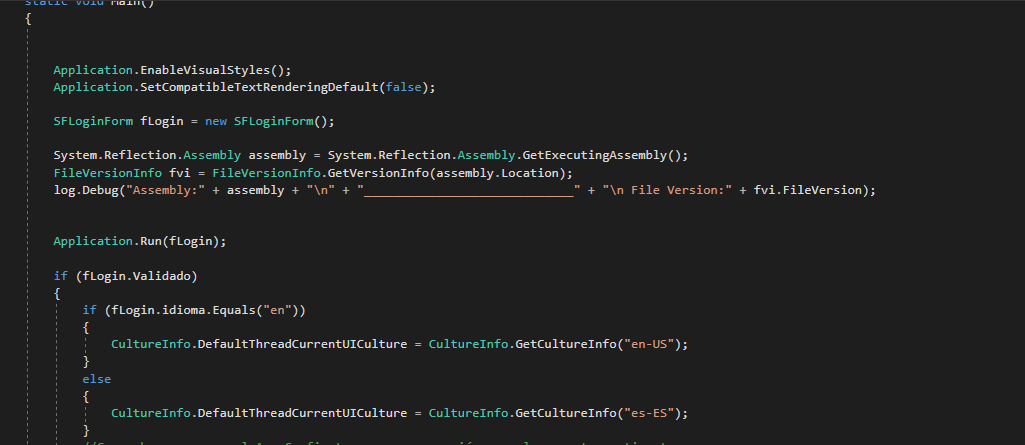
log.Error

log.Fatal

Para trazar los WebService usaremos log.Info antes de llamar al webservice y después, en el log.Info de antes escribimos el nombre del webservice y los parámetros con los que lo estamos llamando,en el de después escribiremos que la llamada a finalizado y en caso de habernos devuelto una respuesta la añadiremos también.

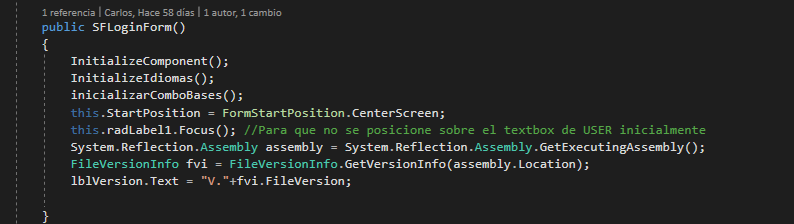
## Empezando el programa

Empieza el programa,entramos en el login

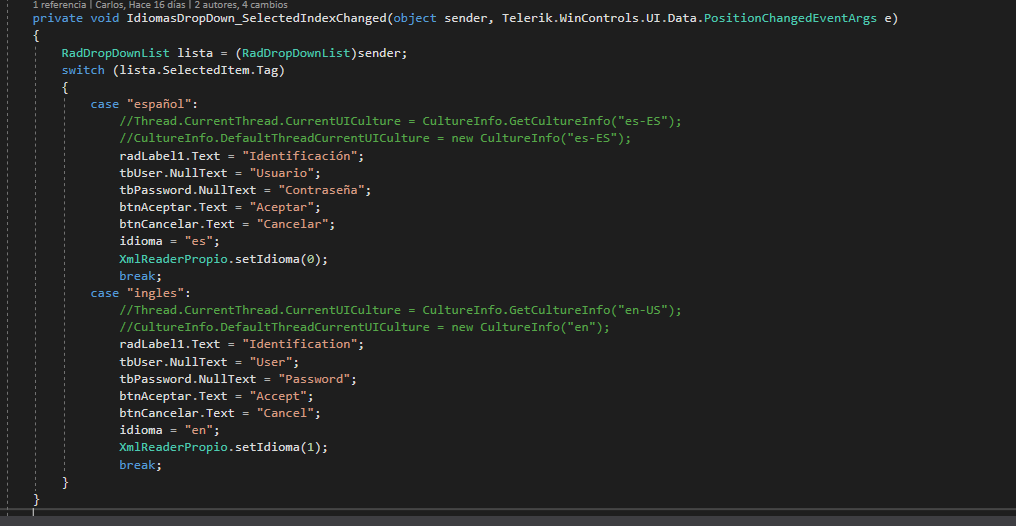


La clase login tiene una propiedad llamada idioma en la cual guardamos “en” para ingles y “es” para español. Una vez realizada la validación comprobamos la propiedad para definir CultureInfo.DefaultThreadCurrentUICulture el cual nos sirve para definir la propiedad CurrentUICulture para todo el programa, de manera que no haya que definirlo en cada clase, esto solo se puede hacer en el program,la primera clase que se ejecuta.

Al entrar en la clase login se inicializan los combos de idiomas y bases de datos, así como la etiqueta con el número de versión.

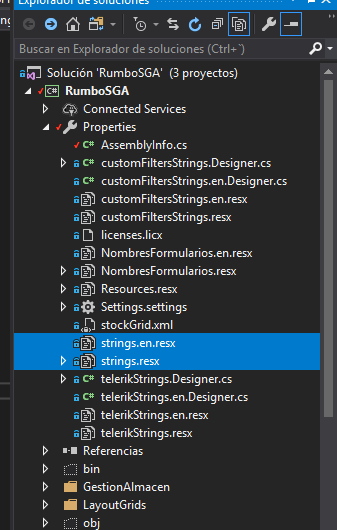


Cuando cambiamos de idioma cambiamos el idioma de los textos,en este caso en concreto hay que hacerlo a mano, ya que el idioma aun no está configurado,y cambiarlo con la ventana abierta no produciría ningún cambio ya que el archivo de recursos no cambiaría.

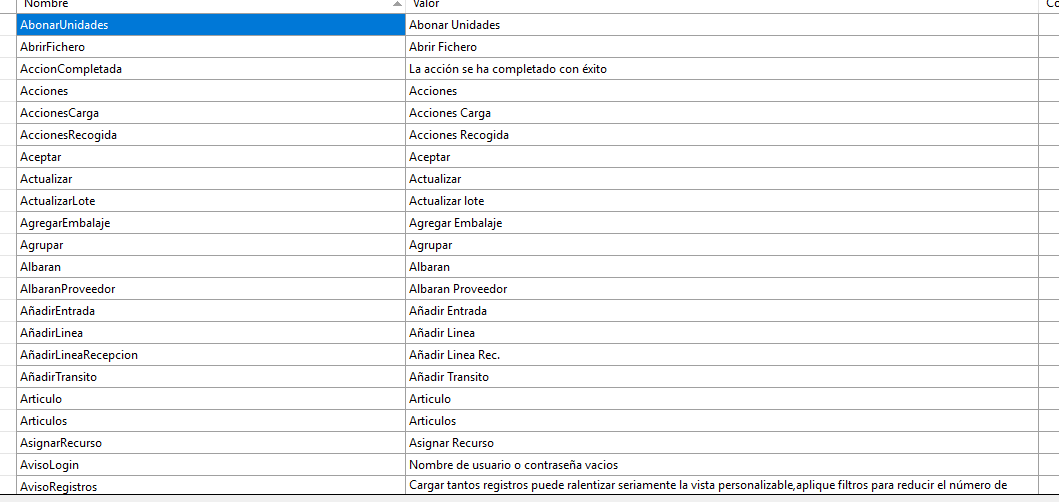


Este es el único caso en el que tenemos que hacerlo así, explicaré el caso habitual cuando ya tenemos un idioma definido.

Los strings los sacamos del fichero de recursos strings.en.resx que se encuentran en Properties. Para utilizarlos escribiremos strings.Clave , siendo clave la clave de la tabla para el valor que buscamos.

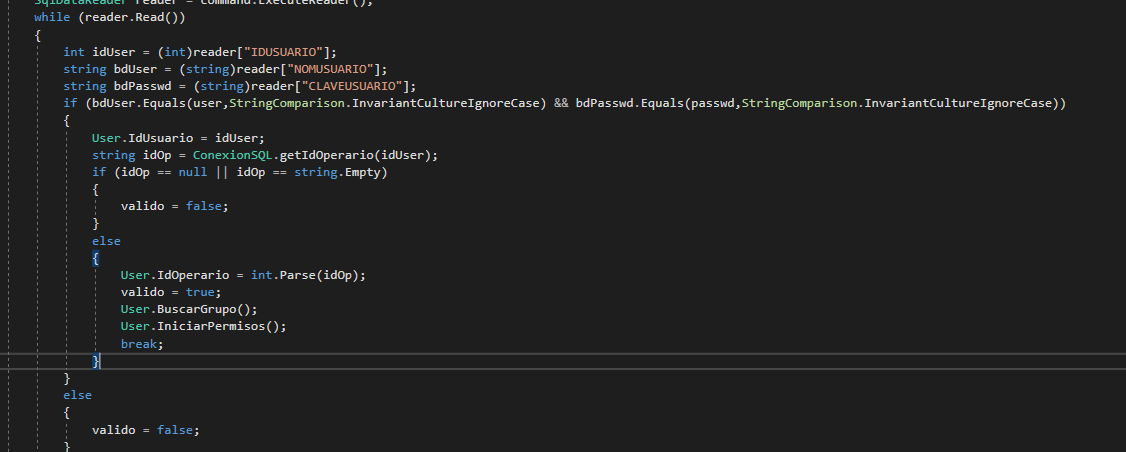


Estos archivos son muy simples de usar, si entramos veremos que se basan en Clave/Valor. A la izquierda escribimos la clave mediante la cual haremos referencia a ese string,a la derecha el valor que tendrá



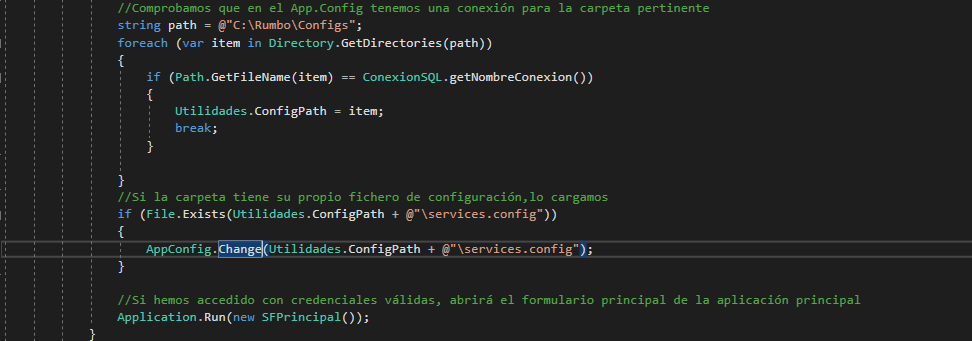
Originalmente habían dos ficheros,uno para ingles y otro para español,en la versión mas reciente se ha cambiado la forma de traducción por lo que el fichero en ingles ya no existe,el fichero en español se sigue utilizando.

Siguiendo con la pantalla de login,al pulsar Aceptar llamamos al método de la clase Security llamado validarLogin() al que le pasamos el usuario y la contraseña. Buscamos en RUMUSUARIOS el nombre de usuario y contraseña y si encajan buscamos el grupo asignado y nos descargamos una tabla con los permisos de ese usuario.



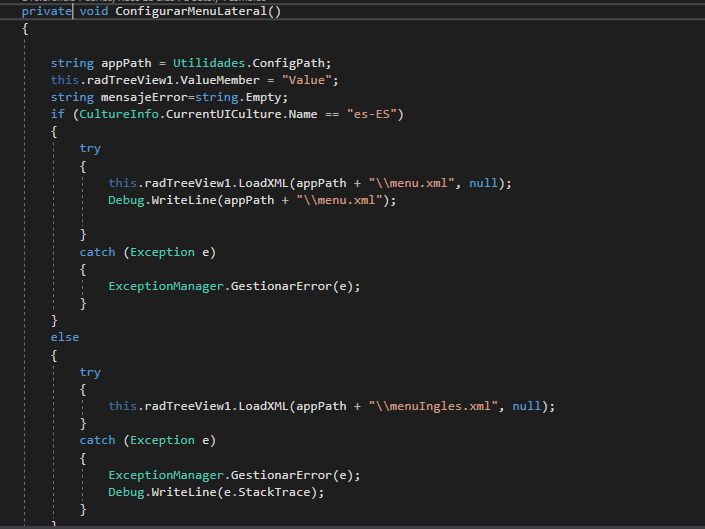
**En la clase User tendremos en todo momento acceso al ID del usuario,a su ID de operario,y los permisos de su grupo.**

Una vez validado el usuario se comprueba que tengamos en el fichero de conexiones una conexión para la base de datos a la que se quiere acceder,para ello compara los nombres de las carpetas con los nombres de las conexiones. Una vez encontrada la conexión configuramos el path de la aplicación. El path base es C:\Rumbo\Configs por lo tanto tendremos que añadir al path la carpeta que se ha seleccionado. Por último comprobamos si en la carpeta del path se encuentra algún archivo llamado “services.config” y en caso de que sí cambiamos el App.Config de la aplicación por el de la carpeta.

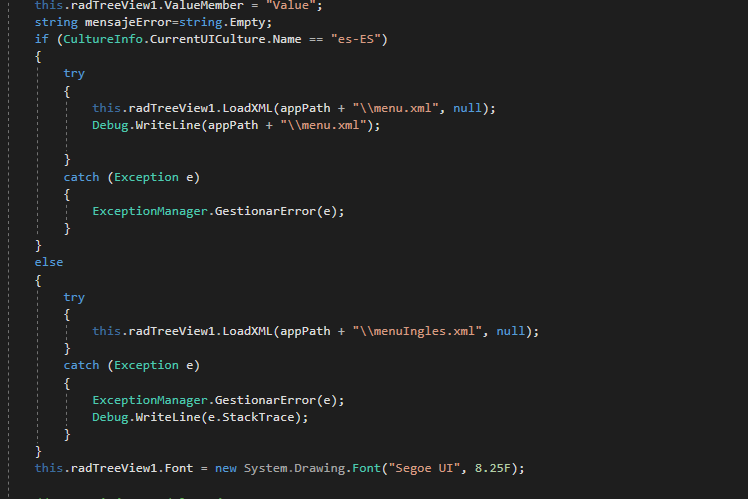


Una vez terminado el proceso se ejecuta la pantalla principal.

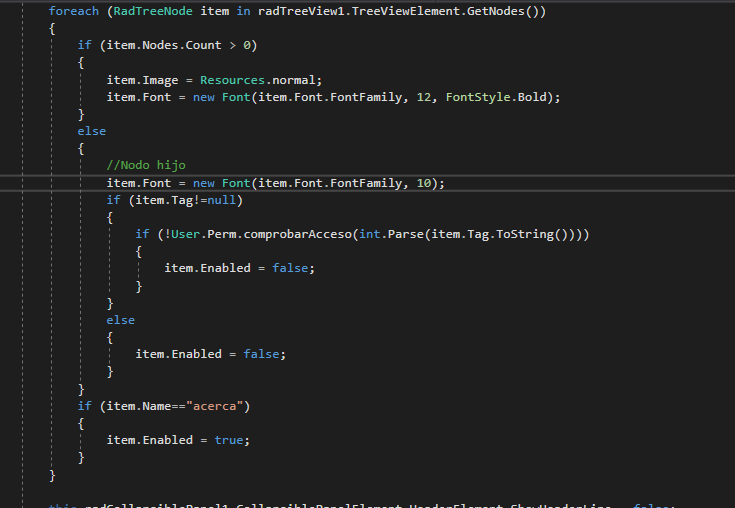
En la pantalla principal lo primero que hacemos es configurar el menú lateral. Para ello debemos mandarle una estructura XML a cargar, que se encontrará en el path de la aplicación



En caso de que el “CurrentUICulture” sea el español,cargamos el menú en español,y si está en ingles cargamos el menú en ingles.



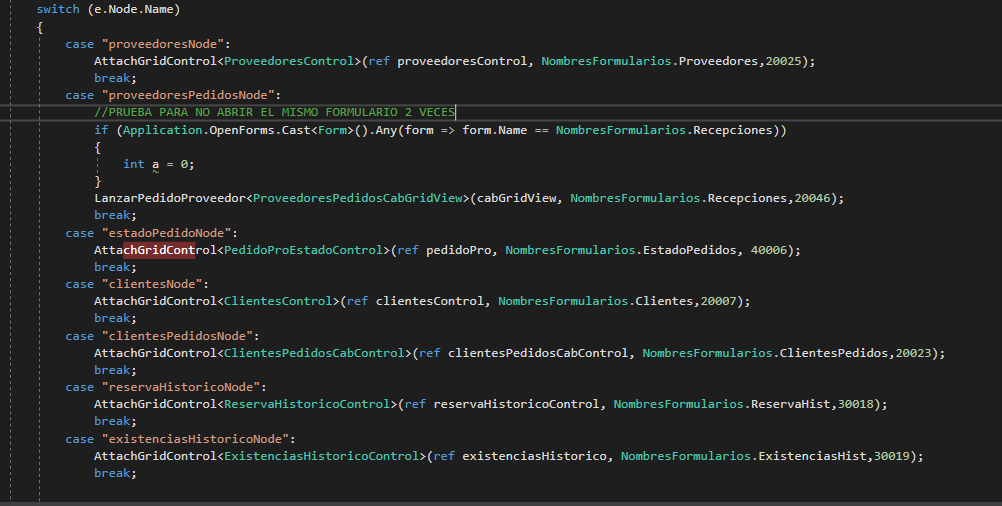
Despues recorremos los nodos ,les añadimos las imágenes (flechitas para mostrar si está desplegado) y comprobamos si el usuario tiene acceso a ese formulario. Basicamente le pasamos el id que hemos definido en el archivo menú.xml para ese formulario al método User.Perm.comprobarAcceso el cual busca en la tabla de permisos que se ha cargado al hacer login el id que le hemos pasado, y activar el nodo si corresponde.



Por último se configuran los eventos del menú al expandirse,al contraerse,al seleccionar un nodo etc.

Cuando pulsamos en un nodo se ejecuta el método cargarGridView el cual abre el formulario seleccionado,para ello le manda al método AttachGridControl() una instancia del formulario a abrir por referencia así como el nombre de ese control y su id.

Los id no son necesarios por el momento ya que está pensado para en un futuro hacer permisos a nivel del control,ya que por ahora los permisos son a nivel de poder abrir o no un formulario,pero en un futuro necesitaremos el id del formulario. El nombre del control es para mostrarlo,en lugar de usar el archivo de recursos “strings” usamos el fichero NombresFormularios para no saturar los ficheros de configuración.



Los formularios que se abren en la misma pantalla principal son del tipo UserControl, y heredan de BaseGridControl.

Los botones superiores forman parte de la pantalla principal,por lo tanto no podemos asignar sus eventos a nivel del UserControl. Para resolver esto creamos 2 metodos,uno en el UserControl y otro en el formulario principal. Cuando se pulsa el botón,el formulario principal llama al evento que haya definido en el control que esté en pantalla, reenviandole el objeto y los argumentos.

## Crear nuevo mantenimiento

Para crear nuevos mantenimientos rápidamente:

1.Crear un archivo JSON con el esquema de datos

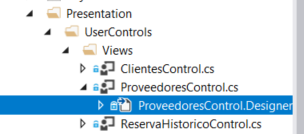
2. En DataAccess.cs, verás funciones separadas por regiones,dentro de la región funciones publicas hay mas regiones(Proveedores,Clientes…). Crea una nueva región (para mantener ordenado el código), copia todos los métodos de proveedores y ajustalos a tu formulario. Tendrás que cambiar los nombres de los métodos, el JSON al que llaman,y editar las querys para que hagan referencia a la tabla deseada.

3. Tendrás que repetir lo anterior en la clase Business.cs. Simplemente habrá que copiar los métodos y cambiar los nombres y las llamadas que hacen a los métodos de DataAccess.

4.Copiar la clase ProveedoresControl y cambiar todas las referencias a la tabla proveedores por referencias a la nueva tabla:

* Cambiar el nombre de la clase
* En la región “Funciones Auxiliares” cambiar ObtencionTotalRegistros para que se llame a tu metodo en Business.cs
* En la región “Eventos sobreescritos de la clase padre”,en el método RefreshData verás “IDPROVEEDOR ASC”. Cambialo por la columna de tu tabla por la que quieras ordenar por defecto. Al final de este mismo método hay un ExecuteQueryAsync que tendría que llamar a tu método de Business.cs.
* Por último,al final de la región anterior hay métodos AckResponse que deberían de llamar a tus métodos de Business.cs

Además, en la parte .Designer.cs de tu control (donde se pone el código autogenerado), tendrás simplemente que cambiar el nombre de la clase por el nuevo nombre.



5. Abrir el archivo menu.xml de la carpeta Resources de la solución RumboSGA. Dentro del tree view crear un nuevo nodo. Puedes simplemente copiar y pegar otro nodo cambiando Name y Tex

6. En SFPrincipal crear variable global de tipo BaseGridControl con el nombre de tu tabla, añadir al switch del método radTreeView1\_SelectedNodeChanged la opción nueva AttachGridControl<TuControl>(ref TuControl); pasando la variable creada como referencia

Estos son todos los pasos que yo estoy siguiendo para crear formularios, aun así cabe la posibilidad de que te de error al intentar abrir el formulario, siempre te dirá que el error está en la línea que parsea el JSON, en realidad el error es que te has dejado algún método que hace referencia a otra tabla y por eso el JSON no se forma correctamente.

En el caso de que tengas un campo que haga referencia a otra tabla y te esté dando error al compilar teniendo bien el codigo, prueba a dejar dicho campo como TextBox(TB) en el JSON ya que esto puede dar problemas por el momento.

## JSON

Los ficheros JSON están en la carpeta C:\Rumbo\Configs. El orden en el que están escritos los datos en el JSON es el orden en el que se van a ver en el grid inicialmente. Algunos parámetros hacen referencia a la forma de los controles en el formulario SFDetalles (Donde el usuario edita, inserta o borra un registro) . En nombre ponemos el nombre de la columna,el prefijo se añadirá dinámicamente, y en el from pondremos únicamente la tabla principal, el código añadirá Outer Joins automáticamente al from en tiempo de ejecución.

{

"Nombre": "IDFAMILIA",

"Etiqueta": "Familia",

"EtiqIngles": "Family",

"Tipo": "VARCHAR(10)",

"Control": "TB",

"EsPK": true,

"EsModificable": false,

"Columna": 0,

"Fila": 0,

"Ancho": 50,

"Alto": 0

},

Si algún campo es clave ajena hay que añadir esto al final de su definición en el JSON:

"EsFK": true,

"CmbObject": {

"TablaRelacionada": "TBLRELACIONADA",

"CampoRelacionado": "CAMPORELACIONADO",

"CampoMostrado": "CAMPOAMOSTRAR",

"ValorDefecto": "VALOR"

}

Estos JSON los podemos formar de distinta manera según las necesidades,el esquema básico creado por Miguel Angel para los mantenimientos no permite mas de una columna añadida de otra tabla,para solventar esto usamos la misma estructura pero usándola distinto. Ejemplo:

{

"Nombre": "lin.IDPEDIDOPRO",

"Etiqueta": "ID",

"EtiqIngles": "ID",

"EsPK": true,

"EsModificable": false,

"EsVisible": true

}

En este caso tenemos que especificar al prefijo de la tabla en la sección del nombre. Si la tabla no tiene operaciones de inserción/borrado/clonado entonces no son necesarias las etiquetas Tipo,Control,Columna,Fila,Ancho y Alto,ya que se utilizan para formar la pantalla de edición. Como estamos poniendo el prefijo explícitamente, en el from tendremos que declarar nosotros los alias y añadir todos los join necesarios.

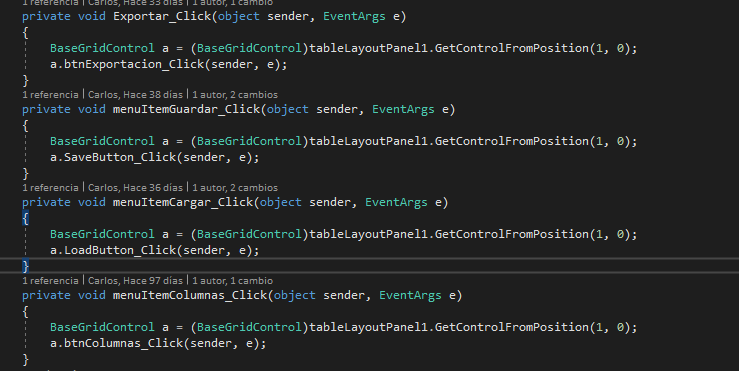
From del ejemplo anterior:

"From": "TBLPEDIDOSPROLIN LIN INNER JOIN TBLARTICULOS art on lin.IDARTICULO=art.IDARTICULO",

## UserControls

Los grids de la parte derecha de la pantalla son UserControls,esto es así porque originalmente los botones estaban en ese nivel,el UserControl era un grid junto con su fila de botones,posteriormente los botones se pusieron en la barra superior de la pantalla principal,por lo que ahora los UserControls son simplemente grids,pero se ha mantenido la estructura anterior. Para utilizar los botones de la pantalla principal sobre operaciones con el grid lo que hacemos es reenviar los eventos,es decir si salta el evento de que se quieren borrar los filtros del grid,en el método reenviamos ese evento al método que declaramos en el control.

Ejemplo:



## Crear Botones

Pulsamos click derecho sobre el proyecto y vamos a propiedades,esto nos llevará a la pantalla de recursos. Añadimos un nuevo recurso con la imagen del botón,luego desde el diseñador arrastramos un RadButton al formulario y lo posicionamos. Si pulsamos click derecho podremos acceder a las propiedades del control y cambiar su “Image” por la que hemos subido,así como cambiar su ImageAlignment y TextAlignment para que estos no se solapen.

## VirtualGrid y GridView

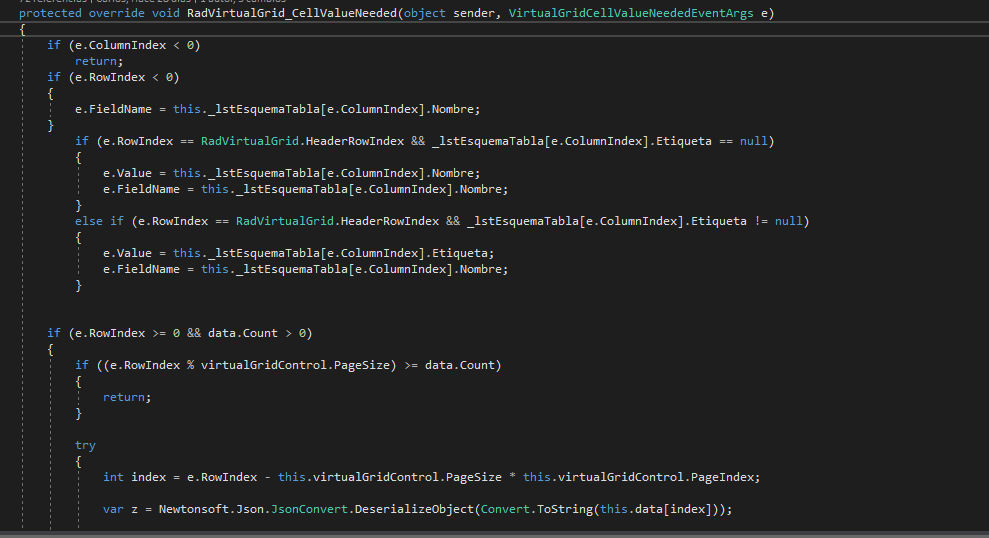
En el programa se utilizan 2 tipos de grid,el virtual grid y el grid view,la principal diferencia es que el VirtualGrid está pensado para mostrar una gran cantidad de datos de manera rápida,este grid no tiene guardados todos los datos que muestra,si no que los pide tras cada operación de cambio de pagina,cambio de filtro o de ordenación.

El virtual grid necesita el evento CellValueNeeded,que le dice como rellenar las celdas. Para rellenar estas celdas almacenaremos los datos JSON y los asignaremos celda por celda en este evento. Tambien es en este evento donde pondremos los nombres de las columnas.

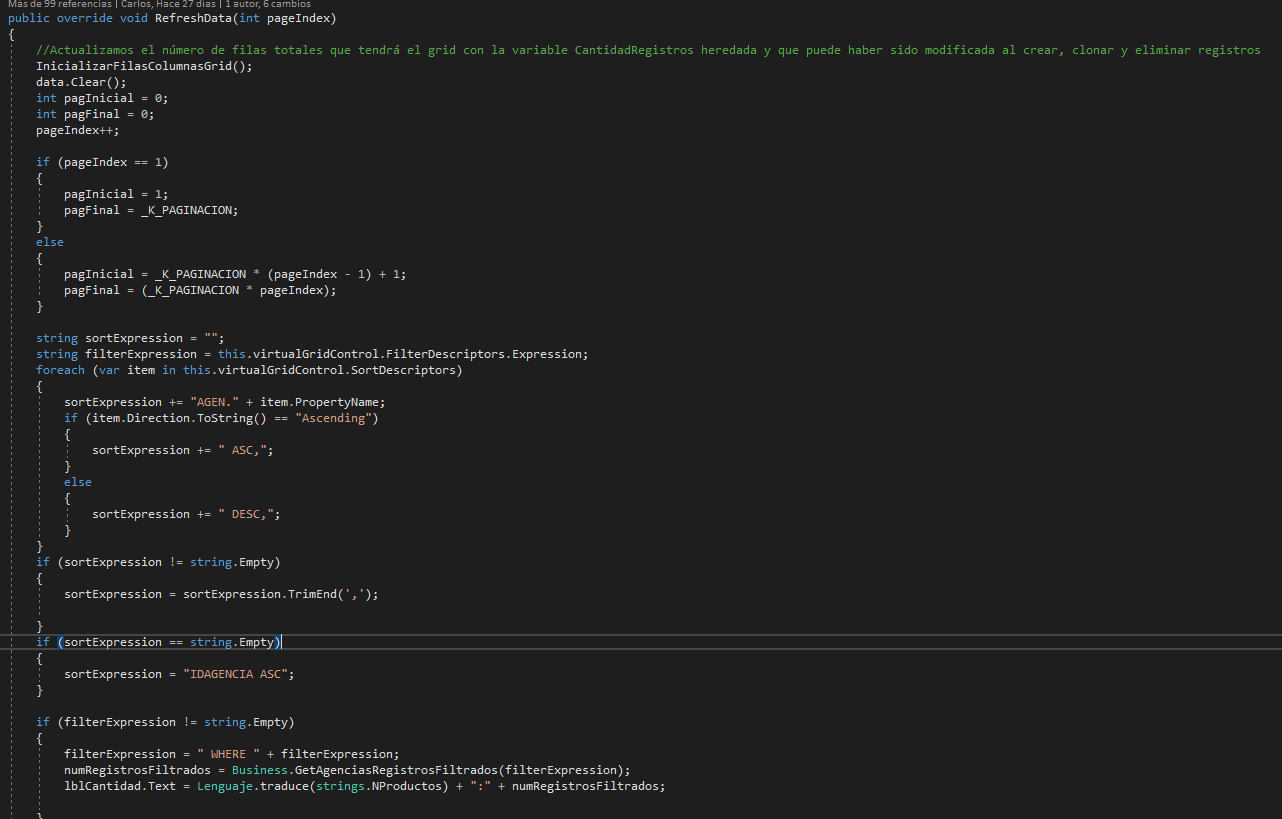
El grid view por su lado necesita que le den directamente todos los datos que vaya a mostrar,para ello llenamos una estructura de datos,por ejemplo un DataTable y se lo asignamos a su propiedad “DataSource”. El grid view realiza de manera automática la paginación,filtrado y ordenación,lo único que debemos decirle es que los active. El grid view nos permite entre otras cosas añadir una barra de búsqueda y crear agrupaciones de registros.

## Operaciones específicas en VirtualGrid

El grid se carga en su CellValueNeeded,en este evento es donde se especifica el contenido de cada celda del VirtualGrid,debemos de haber recogido los datos con anterioridad. Comprobamos en que celda nos encontramos,en caso de encontrarnos en la cabecera del grid,le pasaremos el nombre de la columna mediante el campo “Etiqueta” del esquema. En caso de que sea una celda le pasamos la información correspondiente a esa celda.

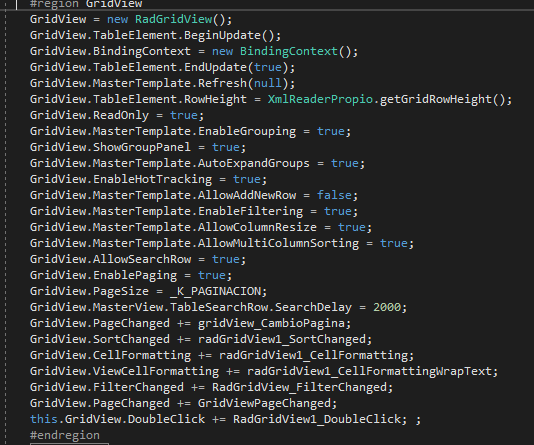


Tambien necesitaremos métodos para la ordenación,fltrado y paginación,en estos métodos simplemente llamaremos a la función RefreshData,en la cual volveremos a pedir los datos,pero esta vez con la paginación/filtro/ordenación nuevos.



## Operaciones específicas en GridView

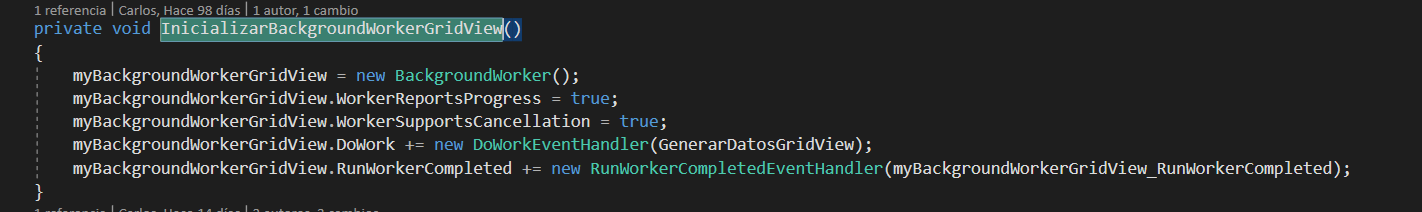
El GridView es un control mas sencillo a la hora de programar,por un lado le tenemos que dar sus datos en su propiedad DataSource. También deberemos de activar las funcionalidades que queramos las cuales se ejecutaran automáticamente sin necesidad de capturar los eventos. Algunas de estas características activar son:



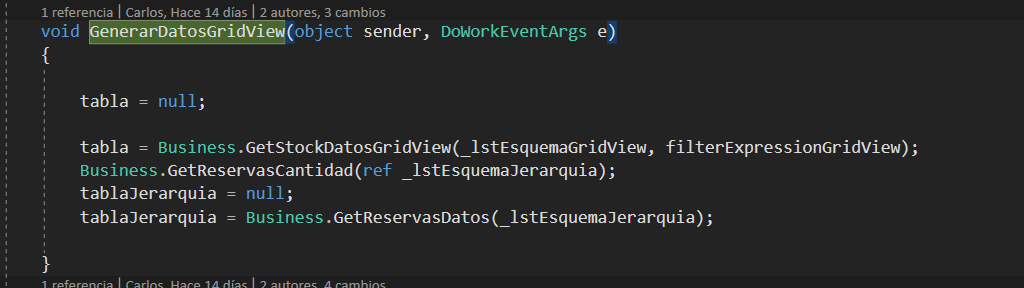
## WaitingBar y BackgroundWorkers

En algunos formularios debemos mostrar una barra de progreso mientras se espera ya que tardan en recoger los datos. Para esto Telerik dispone del control RadWaitingBar,que se puede asociar a cualquier control de Telerik. El problema es que esta barra no se puede utilizar en el mismo hilo en el que estamos cargando los datos,ya que al empezar la ejecución la barra se queda paralizada, para esto utilizamos los BackgroundWorker,lo cual es simplemente una manera de gestionar hilos en C#. El background worker se encargará de ejecutar lo que esté en su método DoWork() en otro hilo,cuando termine volverá al hilo principal y ejecutará lo que haya en su método BackgroundWorkerCompleted. Lo que hacemos es crear el background worker y en su método DoWork hacemos las llamadas a SQL para recoger los datos. Luego en el método BackgroundWorkerCompleted le añadimos estos datos al grid (grid.DataSource=datos) y asignamos cualquier propiedad que sea necesaria,cuando ya haya acabado todo lo que tenga que hacer pararemos la barra de espera.

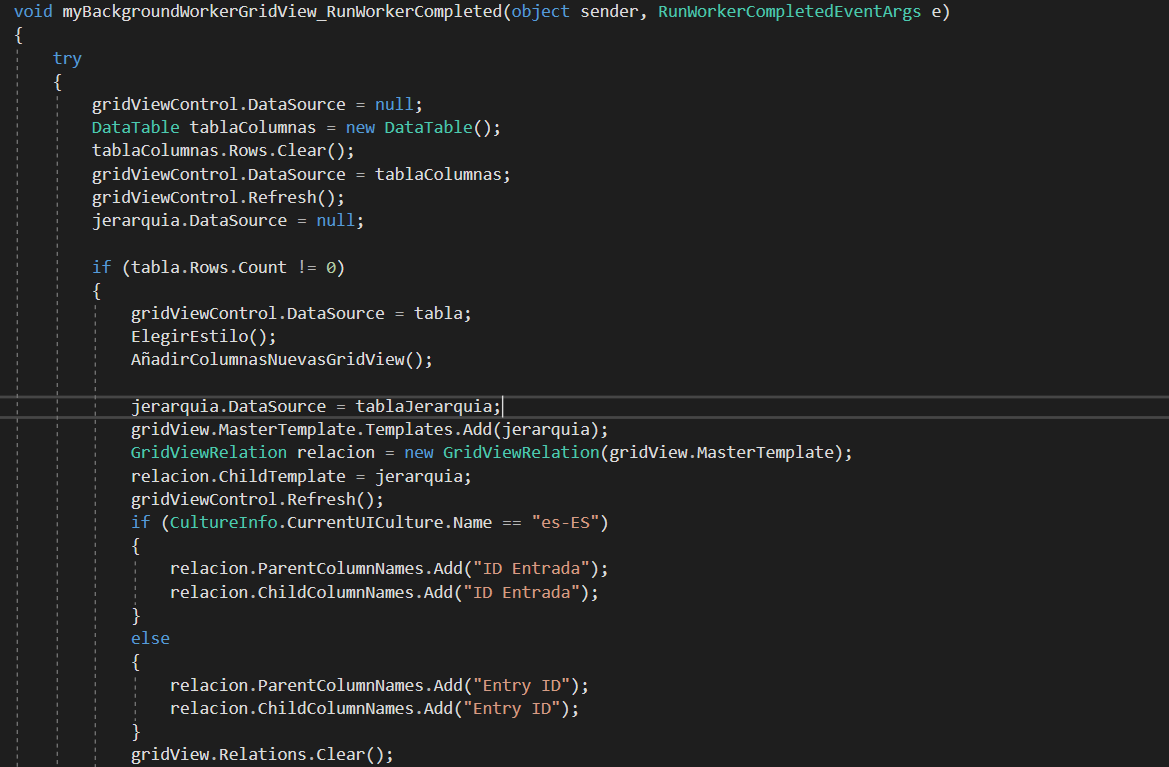
Inicializando workers:



Ejemplo Metodo DoWork:



Ejemplo Metodo WorkerCompleted:



El virtual grid ya realiza sus operaciones en otro hilo por defecto y tiene integrada ya una barra de progreso a la que simplemente tendremos que llamar con su método IsWaiting: this.virtualGridControl.MasterViewInfo.IsWaiting = true;

this.virtualGridControl.MasterViewInfo.IsWaiting = false;

## BaseGridControl

La clase **BaseGridControl** hace de base para todas las tablas de los formularios simples y define los métodos: **RadGridView1\_FilterChanged** (cada vez que hay un filtro recarga la primera página), **RadGridView1\_SortChanged** (cada vez que se ordena carga la primera página), **VirtualGridElement\_PageIndexChanging** (refresca los datos de la página seleccionada del Grid) así como los eventos al clickar los botones.

## SFDetalles

**SFDetalles** es el formulario que se abre al seleccionar una celda para visualizar o editar datos. La disposición de los elementos en este formulario depende de las propiedades del JSON: ancho, alto, columna y fila. Se compone de los controles TitleBarControl y DataDialog. También usa la clase NameValue en el que se crean los controles a través del elemento “Control” del json,ahora mismo trabaja con los valores: TB(textBox),CMB(combo) y DATE (Date Picker),si se quiere ampliar esto se tendría que hacer en la clase NameValue. La clase NameValue se utiliza después en la clase SFDetalles,en el método **CargarDatos** en el cual se crean los controles.

## DataDialog

DataDialog es la parte central del formulario SFDetalles, con los botones y el espacio a rellenar con el esquema del json.

## Pedir datos a SQL

### Business

En esta clase recibimos el esquema json por referencia y lo devolvemos modificado. Aquí llamamos al método GetFieldSQL en el cual extraemos la información pertinente del archivo json,recorriéndolo y guardando la información (los campos de la select,el from,el where) para mandarlos después a la clase DataAccess en la que ensamblaremos la query.

### DataAccess

**DataAccess** tiene métodos para: obtener el esquema de la tabla, obtener los datos de la tabla y crear, editar y borrar. En esta clase es donde juntamos lo extraido del json (los nombres del select,el from..) para crear la query final,la cual será mandada a ConexionSQL.

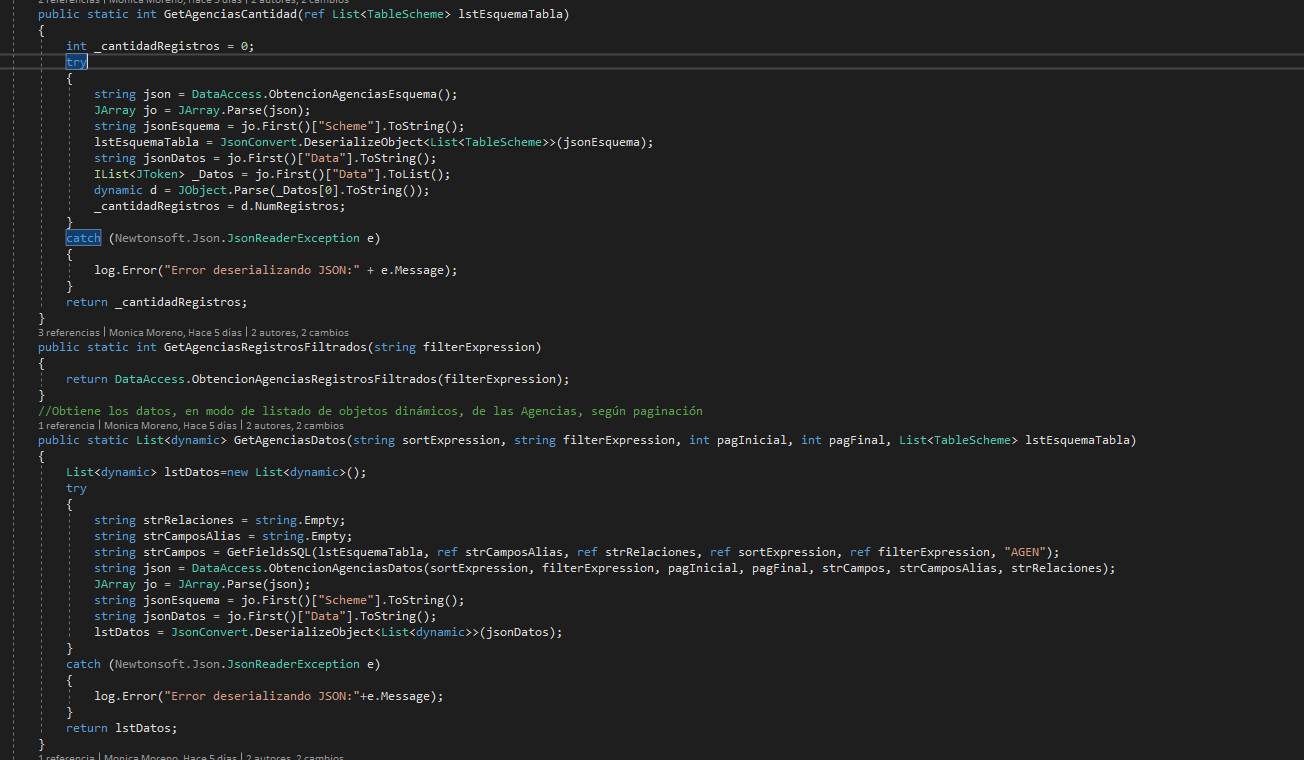
Tiene también un método **GenerarComandoSQL** que recibe el tipo de comando(INSERT,UPDATE..), la tabla, la información de la clausula WHERE y un Hashtable de datos,así como también el método LoadJSON() el cual recibe el nombre del json y lo devuelve en texto.

## ConexionSQL

En esta clase se ejecutan las consultas SQL,el método mas básico es **GetDataTable** el cual recibe por parámetro una query,la ejecuta y devuelve una estructura DataTable con el resultado. Tambien dispone del método **SQLClienteExec** para las opciones de insertar,borrar y clonar. Esta clase tiene dos propiedades estáticas que se inicializan con la información del login,estas son “connectionString”,con la cadena de conexión que usaremos para ejecutar la consult y NombreBD con el nombre de la base de datos.

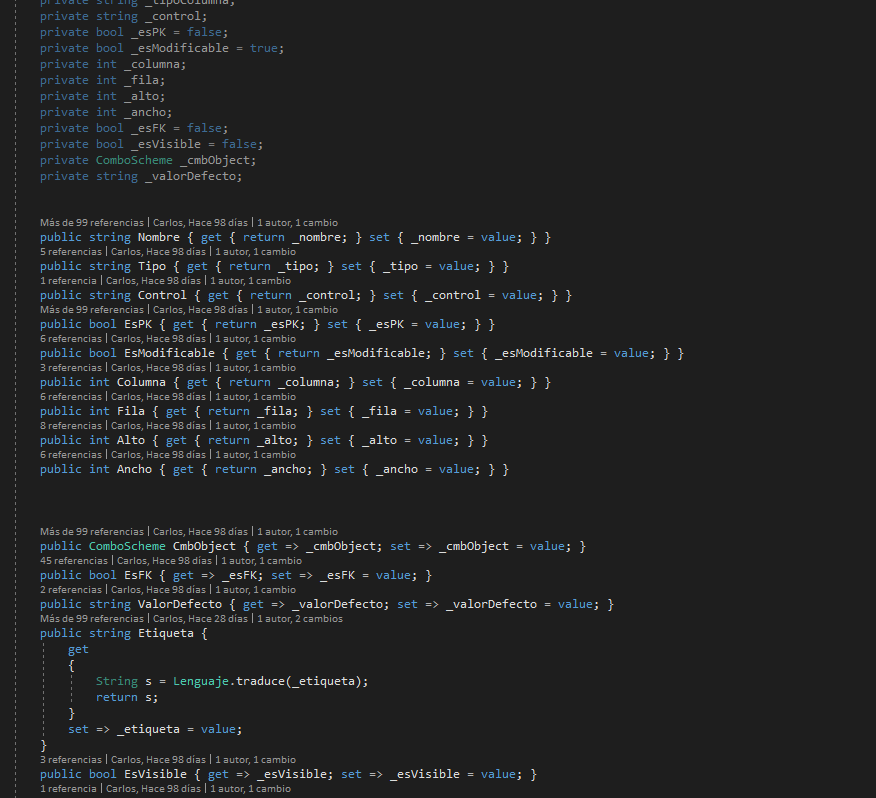
En general en Business recogemos los datos del json,en DataAccess ensamblamos las querys y en ConexionSQL hacemos las llamadas a base de datos.

**Ejemplo Business:**

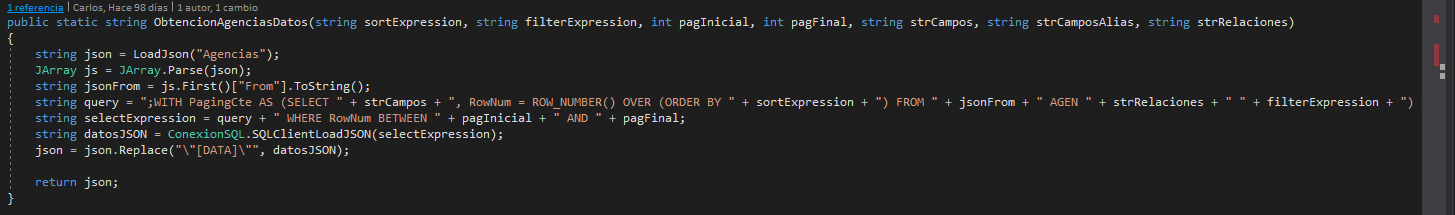


Como se puede observar lo que hace principalmente esta clase en sus métodos es obtener los campos SQL que se van a tener que llamar en la select,una vez tenemos estos campos llamamos a la clase DataAccess pasándoselos para que forme la query y ejecute la consulta,la clase DataAccess nos devolverá los datos en formato JSON,los cuales serán deserealizados en una lista del tipo TableScheme,este tipo es simplemente una clase que tiene declaradas las propiedades del archivo json para recogerlas al serializar.

**TableScheme:**



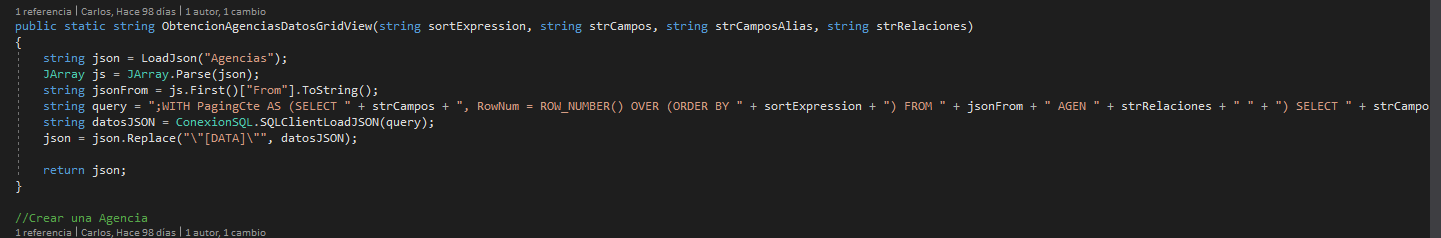
**Ejemplo ensamblar query en Data Access:**



La imagen no se visualiza bien pero creo que el concepto se entiende. En este método recibimos por parámetro el criterio de ordenación, el filtro, la página, el alias de los campos y su relación. Con todo esto formamos la query y la ejecutamos.

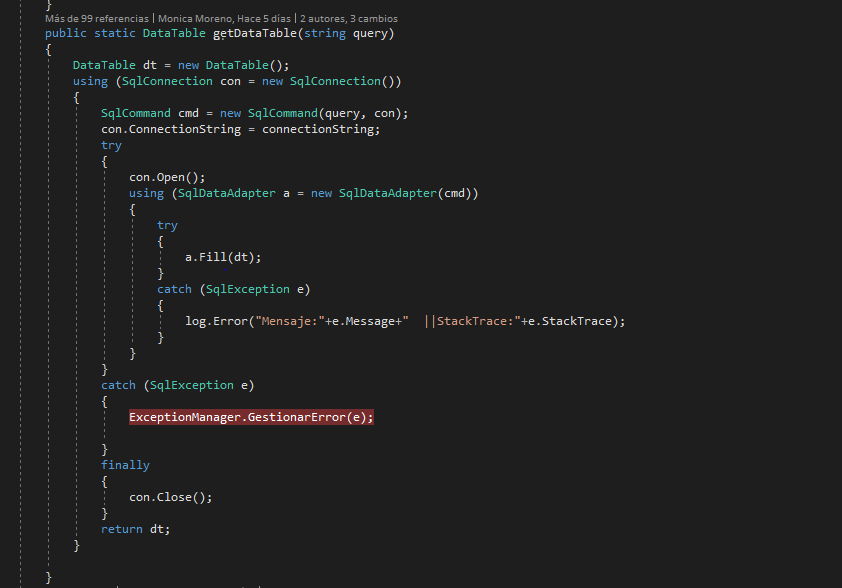
Este ejemplo está sacando los datos para un VirtualGrid,por lo tanto añadimos la parte final (Where RowNum Between…) para indicarle la página que queremos, en el caso de un grid view esto no es necesario. Con estas SELECT nos viene también una columna RowNum con el número de la columna que posteriormente se quita ya que no se debe visualizar en el grid.

**Ejemplo datos GridView:**



**Aviso:Las clases de Business y DataAccess se pretende que sean cambiadas en un futuro, de manera que se use la encapsulación de forma eficiente para reducir el tamaño de las clases y los métodos sean mas genéricos, en lugar de tener métodos específicos para cada formulario. Otro apunte es que debido a que el programa ha pasado por varias manos hay algunos métodos que siendo similares hacen cosas distintas, principalmente esto se verá en métodos relacionados con las clases ControlTareas,OrdenesRecogida y AcopiosProduccion que fueron realizadas por otro compañero que ya no está. Por ejemplo para realizar querys mas complejas que las que se estaban usando en el momento creo mas clases como GetSQLField en Business para poder formar la query como el necesitaba, esto habría que revisarlo ya que es muy posible que se pueda reducir código también en esa sección.**

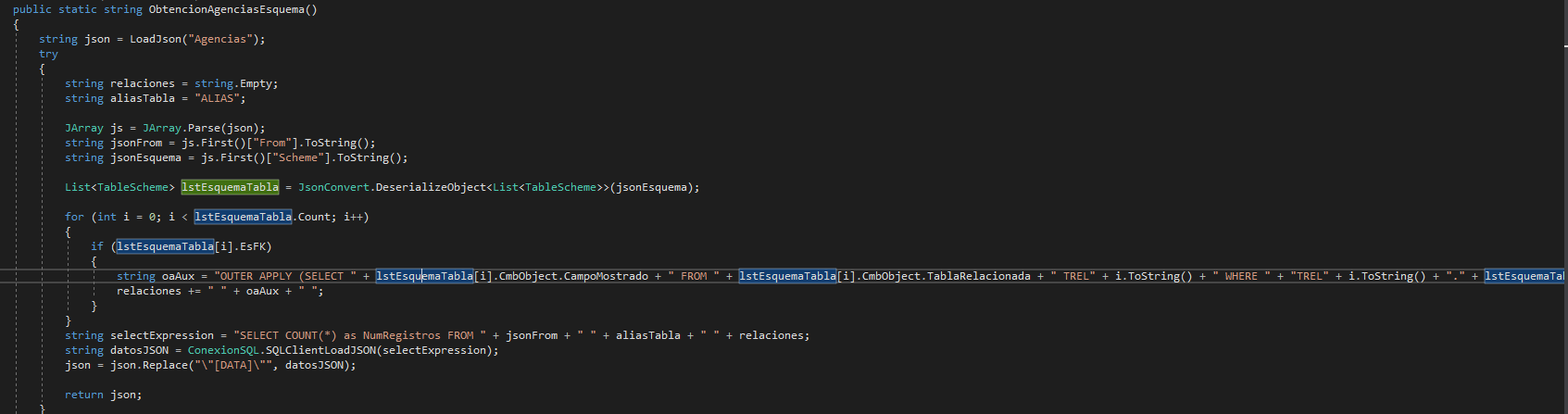
**Ejemplo ejecutar SQL en ConexionSQL:**



Abrimos una SQLConnection y un SQLCommand,al command le pasamos la query y la SQLConnecion,a la SQLConnection hay que asignarle el connection string. Instanciamos una estructura para guardar los datos,en este caso una DataTable,y usando un SQLDataAdapter,el cual ejecuta el comando,usamos el método Fill() pasándole la data table que tiene que llenar,por último se devuelve.

## El esquema

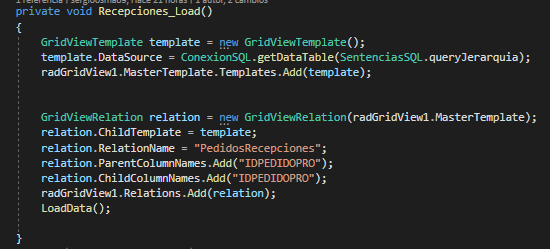
Para recoger los datos json usamos una List del tipo TableScheme que ya se ha mencionado anteriormente. En un principio se declara vacia y cuando vamos a recoger el esquema de Business la enviamos como referencia,el método de Business llamará al método de obtención de esquema de DataAcess. En este método cargamos el archivo json correspondiente y lo deserializamos sobre el esquema que hemos enviado desde nuestro UserControl. Con el esquema ya rellenado lo recorremos para formar la query que nos dará la cantidad de registros y hacemos un replace sobre la sección “DATA” del JSON.



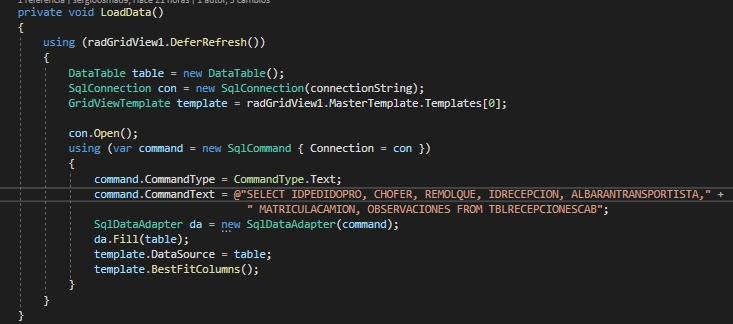
Siempre llamamos primero al método del esquema para tener ya la estructura del json en nuestra List. Una vez ya tenemos el json cargado,para recoger los datos para llenar las tablas seguiríamos un procedimiento similar, llamamos al método de Business que nos da los datos, en este caso también pasamos el esquema json como parámetro sin embargo no lo pasamos como referencia ya que no es necesario. De nuevo,los datos serán puestos en la sección DATA del json.

## Jerarquía Estandard

Para conseguir que el Grid sea jerárquico debemos crear un GridViewTemplate y un GridViewRelation. Al GridViewTemplate le proporcionaremos un DataSource y lo añadiremos a nuestro grid, y en el GridViewRelation añadiremos los campos relacionados de ambas tablas.



Este método a su vez llamará a otro con el cual cargaremos los datos jerárquicos en forma de tabla. Para ello llenaremos un DataTable a partir de una query(command).



Para evitar que los registros que no tienen nada en su tabla hija muestren líneas vacías y ocultar el símbolo + ,añadimos el siguiente código(esto no funciona en el siguiente tipo de jerarquía):

**private bool IsExpandable(GridViewRowInfo rowInfo)**

{

if (rowInfo.ChildRows != null && rowInfo.ChildRows.Count > 0)

{

return true;

}

return false;

}

**private void radGridView1\_ViewCellFormatting(object sender, CellFormattingEventArgs e)**

{

GridGroupExpanderCellElement cell = e.CellElement as GridGroupExpanderCellElement;

if (cell != null && e.CellElement.RowElement is GridDataRowElement)

{

if (!IsExpandable(cell.RowInfo))

{

cell.Expander.Visibility = Telerik.WinControls.ElementVisibility.Hidden;

}

else

{

cell.Expander.Visibility = Telerik.WinControls.ElementVisibility.Visible;

}

}

}

**private void radGridView1\_ChildViewExpanding(object sender, ChildViewExpandingEventArgs e)**

{

e.Cancel = !IsExpandable(e.ParentRow);

}

Deberemos que suscribir los eventos al grid:

gridView.ViewCellFormatting += radGridView1\_ViewCellFormatting;

gridView.ChildViewExpanding += radGridView1\_ChildViewExpanding;

## Jerarquía On Demand

On demand o bajo demanda quiere decir que los datos no están cargados hasta que los visualizas,es decir, hasta que no expandes la fila para ver sus jerárquicos los datos de estos no se solicitan.

Para usar jerarquía bajo demanda primero tendremos que crear el template por un lado y luego declarer un metodo que se encargue de decirle lo que necesita en cada línea

Ejemplo crendo el template:

GridViewTemplate templateRecepcionesPreavisos = new GridViewTemplate();

templateRecepcionesPreavisos.Caption = strings.Preavisos;

string jsonPreavisos = DataAccess.LoadJson("RecepcionesJerarquicoPreavisos");

JArray js2 = JArray.Parse(jsonPreavisos);

string jsonEsquemaPreav = js2.First()["Scheme"].ToString();

List<TableScheme> esquema2 = JsonConvert.DeserializeObject<List<TableScheme>>(jsonEsquemaPreav);

for (int i = 0; i < esquema2.Count; i++)

{

if (esquema2[i].Etiqueta != string.Empty && esquema2[i].Etiqueta != null)

{

templateRecepcionesPreavisos.Columns.Add(esquema2[i].Etiqueta);

}

}

}

templateRecepcionesPreavisos.BestFitColumns();

gridRecepciones.Templates.Add(templateRecepcionesPreavisos);

templateRecepcionesPreavisos.HierarchyDataProvider = new GridViewEventDataProvider(templateRecepcionesPreavisos);

Para llenar el template tendremos que crear el método RowSourceNeeded y asignarlo

gridRecepciones.RowSourceNeeded +=Recepciones\_RowSourceNeeded;

Ejemplo de RowSourceNeeded:

private void Recepciones\_RowSourceNeeded(object sender, GridViewRowSourceNeededEventArgs e)

{

try

{

string valor = string.Empty;

DataRowView rowView = e.ParentRow.DataBoundItem as DataRowView;

if (e.Template.Caption == strings.Lineas)

{

DataRow[] rows = rowView.Row.GetChildRows("ID");

valor = e.ParentRow.Cells["ID"].Value.ToString();

DataTable lineas = Business.GetRecepcionesJerarquicoLineas(valor);

for (int a=0;a< lineas.Rows.Count;a++)

{

GridViewRowInfo row = e.Template.Rows.NewRow();

for (int i = 0; i < lineas.Columns.Count; i++)

{

row.Cells[i].Value = lineas.Rows[i][row.Cells[i].ColumnInfo.HeaderText].ToString();

}

e.SourceCollection.Add(row);

}

templateEntradas.Columns.Add(checkBoxColumnLineas);

templateEntradas.Columns.Move(checkBoxColumnLineas.Index, 0);

}

if (e.Template.Caption == strings.Entradas)

{

DataRow[] rows = rowView.Row.GetChildRows("ID");

valor = e.ParentRow.Cells["ID"].Value.ToString();

DataTable lineas = Business.GetRecepcionesJerarquicoEntradas(valor);

for(int i=0;i<lineas.Rows.Count;i++)

{

GridViewRowInfo row = e.Template.Rows.NewRow();

for (int a = 0; a < lineas.Columns.Count; a++)

{

row.Cells[a].Value = lineas.Rows[a][row.Cells[i].ColumnInfo.HeaderText].ToString();

}

e.SourceCollection.Add(row);

}

}

## RadProgressBar para columnas

Los pasos para poder tener una barra de progreso en una de nuestras columnas son los siguientes:

**Restricciones**:

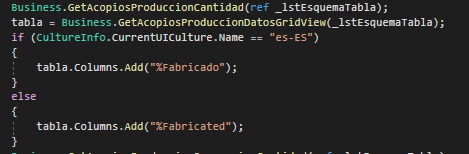
No se pueden añadir directamente a columnas dadas por el DataSource, es decir,por un lado se cargan los datos y luego por otro se añade la columna.

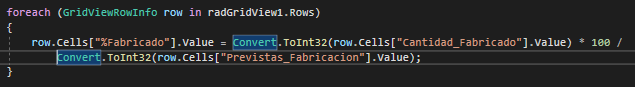
El valor de la nueva columna con barra de progreso será el del porcentaje que nosotros queramos darle.

Este porcentaje no podrá ser mayor a 100 ni menor que 0.

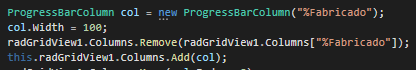
Pasos a seguir:

En este caso añadiremos una nueva tabla al DataTable que contiene el DataSource con el nombre de “%Fabricado”. A esta columna le tendremos que dar el valor del porcentaje que queramos (en nuestro caso “Cantidad\_Fabricado \* 100 / ”Previstas\_Fabricacion”).





Al ya tener los valores de la columna, se creará una nueva de tipo ProgressBarColumn con el mismo nombre y quitaremos la columna anterior antes de añadir la nueva.



Al crear la columna se llamará a una clase que tendremos creada llamada ProgressBarColumn que cogerá la columna que hemos creado anteriormente y que a su vez llamará a otra clase llamada ProgressBarCellElement que se encargará de customizar nuestra columna.

En esta clase se añadirán las barras de progreso con el método CreateChildElements, se añadirán los valores máximos, mínimos y de texto, un método para añadir los temas a la celdas y otro para la compatibilidad entre las columnas.

public class ProgressBarColumn : GridViewDataColumn

{

public ProgressBarColumn(string fieldName) : base(fieldName)

{

}

public override Type GetCellType(GridViewRowInfo row)

{

if (row is GridViewDataRowInfo)

{

return typeof(ProgressBarCellElement);

}

return base.GetCellType(row);

}

}

public class ProgressBarCellElement : GridDataCellElement

{

public ProgressBarCellElement(GridViewColumn column, GridRowElement row) : base(column, row)

{

}

private RadProgressBarElement radProgressBarElement;

protected override void CreateChildElements()

{

try

{

base.CreateChildElements();

radProgressBarElement = new RadProgressBarElement();

radProgressBarElement.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;

this.Children.Add(radProgressBarElement);

}

catch (Exception ex)

{

ExceptionManager.GestionarError(ex);

log.Error("Mensaje:" + ex.Message + "\n StackTrace:" + ex.StackTrace);

}

}

protected override void SetContentCore(object value)

{

try

{

if (this.Value != null && this.Value != DBNull.Value)

{

int valor = 0;

this.radProgressBarElement.Maximum = 100;

radProgressBarElement.Minimum = 0;

if (Convert.ToInt32(this.Value) >= 100)

{

valor = 100;

radProgressBarElement.Value1 = valor;

this.radProgressBarElement.Text = valor + "%";

radProgressBarElement.ProgressOrientation = ProgressOrientation.Left;

}

else if (Convert.ToInt32(this.Value) <= 0)

{

valor = 0;

radProgressBarElement.Value1 = valor;

this.radProgressBarElement.Text = valor + "%";

radProgressBarElement.ProgressOrientation = ProgressOrientation.Left;

}

else

{

this.radProgressBarElement.Value1 = Convert.ToInt16(this.Value);

this.radProgressBarElement.Text = this.radProgressBarElement.Value1 + "%";

radProgressBarElement.ProgressOrientation = ProgressOrientation.Left;

}

}

}

catch (Exception ex)

{

ExceptionManager.GestionarError(ex);

log.Error("Mensaje:" + ex.Message + "\n StackTrace:" + ex.StackTrace);

}

}

protected override Type ThemeEffectiveType

{

get

{

return typeof(GridDataCellElement);

}

}

public override bool IsCompatible(GridViewColumn data, object context)

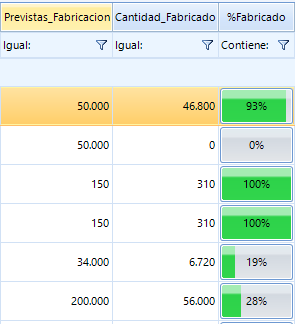
{

return data is ProgressBarColumn && context is GridDataRowElement;

}

}

**Vista final:**



## Árbol de nodos

La herramienta de OrdenesRecogida en su vista Ordenes de Carga contiene un árbol de nodos el cual debe de llenarse de manera genérica(no sabemos cuántos niveles hay). Esto se hace mediante recursividad, es decir, un método que se llama a sí mismo.



Este método(cagaNodosArbol) se encuentra en la clase RumTreeView que se utiliza para construir el árbol. Esta clase al iniciarse recibe por parámetro el número del árbol a cargar y el prefijo de la imagen del nodo (las imágenes se llaman nivel1,nivel2,nivel3.. por lo que el prefijo es “nivel” y para cada nodo le añadirá la imagen “nivelX” siendo X el nivel del nodo.

Como el método se va a llamar a si mismo constantemente, esto puede dar lugar a un bucle

infinito, por lo que es importante tener una condición de salida, en este caso la condición de

salida es el NivelMaximo de nodos.

Este árbol carga sus nodos de RUMARBOLLIN,concretamente del número 7. Para saber el nivel

máximo del árbol lo busca y hace un count, ya que cada fila representa un nodo.

Las consultas SQL necesitan saber cual es el nodo padre, por ello estas consultas llevan entre

corchetes el árbol,el nodo y la columna, ej:WHERE IDRUTA=[7.1.IDRUTA] Del árbol 7,nivel de

nodo 1,su IDRUTA.

Esto se sustituye en la ejecución mediante el método sustituirValores. La primera vez que se ejecute el método no pasará por ese trozo del código ya que el primer nivel no tiene padre, a partir de ahí cada vez que se ejecute sustituirá la parte de los corchetes por su valor real, en el caso del ejemplo el IDRUTA del nodo anterior.

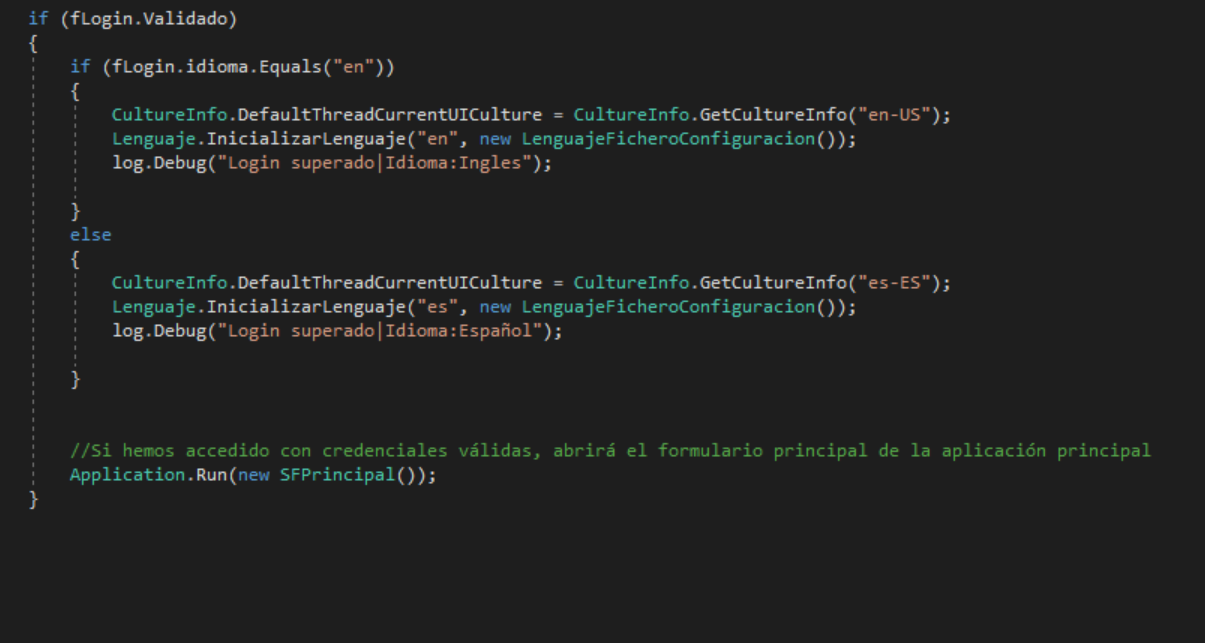
Esto se va ejecutando constantemente hasta que llegue al nivel máximo de nodos, tras lo cual no pasará por la condición y saldrá del bucle.

## Lenguaje

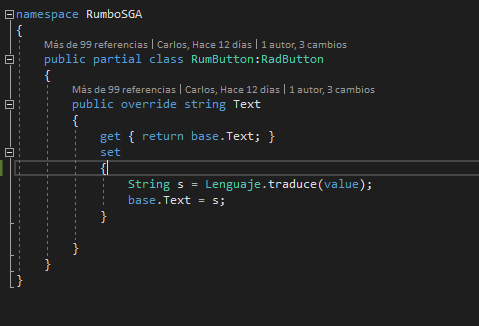
Para realizar la traducción se usa la dll Rumbo.Core.Lenguajes (este nombre podría cambiar) proporcionada por nuestro compañero Javi.Con la dll incluida en el proyecto lo primero será decir en la clase program,es decir la primera que se ejecuta,el path donde se encuentra el fichero de traducciones. Este fichero se encuentra en X:\Lenguajes. Actualmente el fichero se está copiando de manera local al proyecto y el path está apuntando a la misma aplicación,por lo tanto debemos copiar manualmente el fichero para crear las versiones. No obstante si le instalamos la aplicación a nuestro compañero Mark,le pondremos el Path real en X para que pueda ver reflejados los cambios que hace en el fichero original.

Con la dll añadida y el path configurado,usaremos Lenguaje.traduce() pasándole el string a traducir cuando queramos traducir un control,en caso de que no encuentre traducción para esa string ,te devuelve el texto original.

Para no tener que llenar el proyecto de llamadas a la dll lo que hacemos es crear controles heredados de los controles de Telerik,sobreescribiendo el evento “Text” de manera que cada vez que le pongamos el texto a ese control llamará a la función de traducción de manera automática.



Ejemplo con RumButton que hereda de RadButton:



De esta forma cuando queramos añadir un RadButton lo que haremos será añadir un RumButton que es lo mismo pero con la función de traducción.

**Aviso: La clase de MessageBox aun no ha sido creada como control propio ya que está pendiente hacerla con mas funcionalidades que la traducción(por ejemplo botón de copiar texto) por lo que queda pendiente. En este momento se llama a la función de traducción siempre que mostramos un MessageBox.**

## Colores en celdas dependiendo de su valor

Para este ejemplo usaremos el estado de las máquinas en AcopiosProduccion.

Para hacer este proceso necesitamos un nuevos formulario que muestre los distintos tipos de estado de máquinas, dónde elegiremos el color del estado con un RadColorDialog.

El evento para poder seleccionar el color de la celda, en la columna adecuada es el siguiente:

private void radGridView1\_CellDoubleClick(object sender, GridViewCellEventArgs e)

{

if (e.Column.HeaderText == "Estado\_Maquina")

{

string color = SetCellBackground();

e.Row.Cells[e.ColumnIndex].Style.CustomizeFill = true;

e.Row.Cells[e.ColumnIndex].Style.BackColor = dialog.SelectedColor;

string estado = e.Row.Cells["Estado\_Maquina"].Value.ToString();

XmlReader.setColorMaquina(color, estado);

}

}

Con esto al clicar dos veces en la celda obtendremos el menú del RadColorDialog.

Ahora añadiremos un método que devolverá el valor del color seleccionado del menú:

private string SetCellBackground()

{

int row = (int)this.radGridView1.CurrentCell.RowInfo.Index;

int column = (int)this.radGridView1.CurrentCell.ColumnInfo.Index;

string valorColor = "";

((RadForm)dialog.ColorDialogForm).ThemeName = "CAMBIO COLOR";

dialog.SelectedColor = this.radGridView1.Rows[row].Cells[column].Style.BackColor;

if (dialog.ShowDialog() == DialogResult.OK)

{

radGridView1.Rows[row].Cells[column].Style.BackColor = dialog.SelectedColor;

radGridView1.CurrentCell.BackColor = dialog.SelectedColor;

valorColor = dialog.SelectedColor.ToArgb().ToString();

}

return valorColor;

}

Al tener ya esto simplemente queda añadir en un método los diferentes estados que nos podemos llegar a encontrar con un ConditionalFormattingObject y recogerlos en la clase XMLReader:

private void ColorEstadoMaquina()

{

ConditionalFormattingObject obj1 = new ConditionalFormattingObject("Mi Condición1", ConditionTypes.Equal, "AC", "", false);

obj1.CellBackColor = XmlReader.getColorMaquina("AC");

ConditionalFormattingObject obj2 = new ConditionalFormattingObject("Mi Condición2", ConditionTypes.Equal, "IN", "", false);

obj2.CellBackColor = XmlReader.getColorMaquina("IN");

this.radGridView1.MasterTemplate.Columns["Estado\_Maquina"].ConditionalFormattingObjectList.Add(obj1);

this.radGridView1.MasterTemplate.Columns["Estado\_Maquina"].ConditionalFormattingObjectList.Add(obj2);

}

En la clase XMLReader tendremos un Set y un Get que cogerá los valores de los colores de cada estado que tendremos dentro de un xml.

SET:

public static void setColorMaquina(string color, string estado)

{

string path = @"XML\coloresMaquina.xml";

//string path = @"\\P\Carlos\stockColoresPedidos.xml";

XDocument X = XDocument.Load(path);

switch (estado)

{

case "AC":

X.Element("Colores").Element("ColorAC").SetValue(color);

X.Save(path);

break;

case "IN":

X.Element("Colores").Element("ColorIN").SetValue(color);

X.Save(path);

break;

}

}

GET:

public static Color getColorMaquina(string estado)

{

string path = @"XML\coloresMaquina.xml";

XDocument X = XDocument.Load(path);

string color = "";

switch (estado)

{

case "AC":

color = X.Element("Colores").Element("ColorAC").Value;

break;

case "IN":

color = X.Element("Colores").Element("ColorIN").Value;

break;

default:

break;

}

Color colorcito = new Color();

try

{

colorcito = Color.FromArgb(Convert.ToInt32(color));

}

catch (Exception e)

{

Debug.WriteLine(e.Message);

}

return colorcito;

}

Ahora ya podemos cambiar los colores de los estado con el nuevo formulario, pero si queremos que se vean los colores en las celdas de otros formularios que también tengan estados de máquinas deberemos añadir este método también en dichos formularios:

private void ColorEstadoMaquina()

{

try

{

ConditionalFormattingObject obj1 = new ConditionalFormattingObject("Mi Condición1", ConditionTypes.Equal, "AC", "", false);

obj1.CellBackColor = XmlReader.getColorMaquina("AC");

ConditionalFormattingObject obj2 = new ConditionalFormattingObject("Mi Condición2", ConditionTypes.Equal, "IN", "", false);

obj2.CellBackColor = XmlReader.getColorMaquina("IN");

this.radGridView1.MasterTemplate.Columns["Estado\_Maquina"].ConditionalFormattingObjectList.Add(obj1);

this.radGridView1.MasterTemplate.Columns["Estado\_Maquina"].ConditionalFormattingObjectList.Add(obj2);

}

catch (Exception ex)

{

ExceptionManager.GestionarError(ex);

log.Error("Mensaje:" + ex.Message + "\n StackTrace:" + ex.StackTrace);

}

}

## Gestión de excepciones

Por el momento usamos la clase ExceptionManager que por ahora tiene un único método que recibe una excepción y muestra en pantalla el mensaje y el Stack Trace. En un futuro gestionaremos las excepciones de otra manera, pero por ahora llamamos a este método para tenerlo centralizado.

## Generar Versiones

Para generar versiones usamos la extensión Microsoft Visual Studio Installer Projects. Con esta extensión se nos añadirá un nuevo proyecto a la solución llamada Setup. En este proyecto podemos pulsar click derecho y entrar en View>Sistema de archivos, donde podremos ver todo lo que se instalará. Para empezar añadimos los Resultados del proyecto, esto es lo que se genera en las carpetas de Debug/Release al compilar las aplicaciones y que son necesarias para su funcionamiento.. En el instalador también se incluyen los archivos XML que utiliza el proyecto así como la carpeta de recursos que se añade automáticamente al añadir los resultados del proyecto. Una vez están añadidos los resultados del proyecto no será necesario volver a hacerlo, si por ejemplo añades una imagen nueva a los recursos del proyecto para un nuevo botón, no será necesario que añadas esa imagen a los archivos del instalador, simplemente deberás recompilarlo

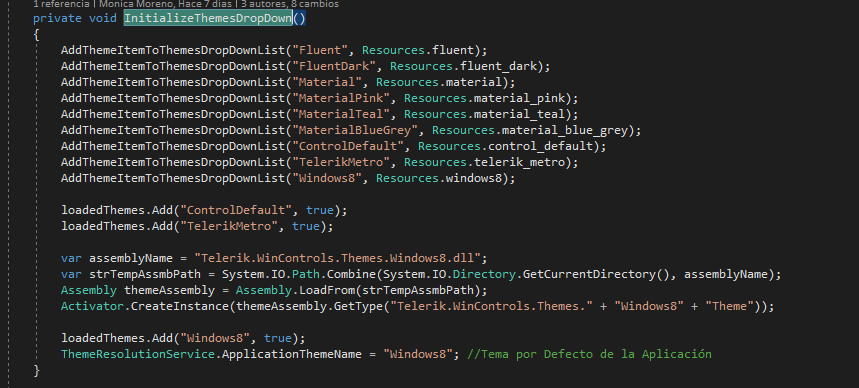
Es importante que cada vez que generemos una versión pulsemos encima del setup y cambiemos a la pestaña de propiedades. Aquí deberemos de subirle la versión al proyecto, cuando nos pregunte si queremos cambiar también el ProductCode le decimos que si. Esto lo hacemos para que al instalar esa versión en un ordenador que ya tiene instalada la versión anterior,sepa que es una versión nueva y sobreescriba la versión anterior. Si no hacemos esto al tratar de instalar nos dirá que el producto ya está instalado y que debemos de desinstalarlo primero.

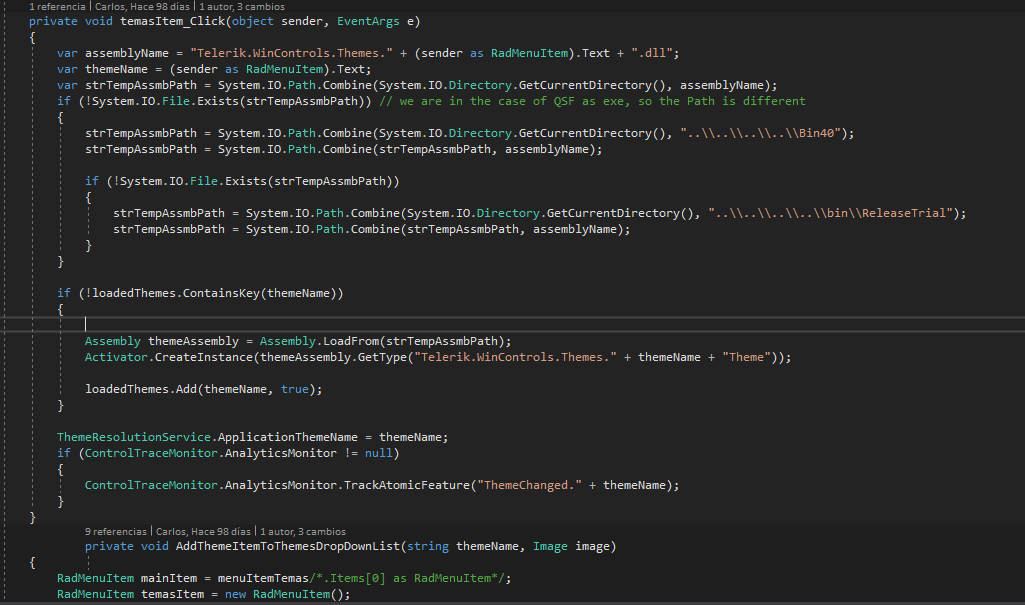
**Muy importante recordar que cada archivo XML nuevo que se incluya en el proyecto deberá incluirse también en el setup, también en el caso de que añadamos nuevos recursos tendremos que recompilar el proyecto de instalación para que recoja las cosas nuevas del proyecto.**

**Extras**

## Temas

Al principio de SFPrincipal, en el constructor, se llama al método InitializeThemesDropDown() donde inicializamos el combo de temas y se los añadimos. Tambien cargamos el tema por defecto, en nuestro caso usamos el de Windows8.



En el método temasItem\_Click cargamos el nuevo tema que se seleccione y lo configuramos como tema de la aplicación.

Por último,si abrimos una ventana nueva y queremos aplicarle el tema de la aplicación,en su consctructor haríamos:

this.ThemeName = ThemeResolutionService.ApplicationThemeName;

## Summary Rows

Para añadir filas con funciones agregadas en las agrupaciones:

GridViewSummaryItem summaryItem = new GridViewSummaryItem();

summaryItem.Name = NOMBRECOLUMNA;

summaryItem.Aggregate = FUNCIÓN;

GridViewSummaryRowItem summaryRowItem = new GridViewSummaryRowItem();

summaryRowItem.Add(summaryItem);

this.gridView.SummaryRowsTop.Add(summaryRowItem);

En el .Name ponemos el nombre de la columna a la que se va a referir.

En el .Aggregate usamos la función que queremos usar.

Cuando lo añadimos al grid podemos usar SummaryRowsTop o SummaryRowsBottom

Tendremos que usar un if sobre CurrentUICulture para saber si nos referimos al nombre del campo en español o en ingles.

Ejemplo de los Summary Rows de Stock:

GridViewSummaryItem summaryItemTop = new GridViewSummaryItem();

GridViewSummaryItem summaryItemBottom = new GridViewSummaryItem();

if (CultureInfo.CurrentUICulture.Name == "es-ES")

{

summaryItemTop.Name = "Cantidad";

summaryItemTop.Aggregate = GridAggregateFunction.Count;

GridViewSummaryRowItem summaryRowItemTop = new GridViewSummaryRowItem();

summaryRowItemTop.Add(summaryItemTop);

this.gridView.SummaryRowsTop.Add(summaryRowItemTop);

summaryItemBottom.Name = "Cantidad";

summaryItemBottom.Aggregate = GridAggregateFunction.Sum;

GridViewSummaryRowItem summaryRowItemBottom = new GridViewSummaryRowItem();

summaryRowItemBottom.Add(summaryItemBottom);

this.gridView.SummaryRowsBottom.Add(summaryRowItemBottom);

}

else

{

summaryItemTop.Name = "Quantity";

summaryItemTop.Aggregate = GridAggregateFunction.Count;

GridViewSummaryRowItem summaryRowItemTop = new GridViewSummaryRowItem();

summaryRowItemTop.Add(summaryItemTop);

this.gridView.SummaryRowsTop.Add(summaryRowItemTop);

summaryItemBottom.Name = "Quantity";

summaryItemBottom.Aggregate = GridAggregateFunction.Sum;

GridViewSummaryRowItem summaryRowItemBottom = new GridViewSummaryRowItem();

summaryRowItemBottom.Add(summaryItemBottom);

this.gridView.SummaryRowsBottom.Add(summaryRowItemBottom);

}

Las funciones agregadas que se pueden usar en el GridAggregateFunction.X son

.None

.Sum

.Min

.Max.

.Last

.First

.Count

.Avg

.StDev

.Var

**APUNTE: Si quitamos la columna asociada mediante el selector de columnas, la summary row seguirá en su sitio pero el valor desaparecerá**

## RadContextMenu

Menú contextual que se despliega al pulsar click derecho en el grid.

Se crea un objeto RadContextMenu,y se crea un método para el evento radGridView1.ContextMenuOpening en el que asignamos nuestro contextMenu.Luego se crean objetos RadMenuItem y le asignamos un texto, también debemos crear métodos para los eventos item.Click . Por último añadimos los ítem al objeto contextMenu.

Ejemplo:

RadMenuItem bomMenuItem = new RadMenuItem();

RadMenuItem histArticulosMenuItem = new RadMenuItem();

bomMenuItem.Text = "Ver formulario Bom";

histArticulosMenuItem.Text = "Hist. Articulos";

this.gridViewControl.ContextMenuOpening += gridView\_ContextMenuOpeningEvent;

bomMenuItem.Click += new EventHandler(bomMenuItem\_Event);

histArticulosMenuItem.Click +=new EventHandler(histArticulosMenuItem\_Event);

contextMenu.Items.Add(bomMenuItem);

contextMenu.Items.Add(histArticulosMenuItem);

void gridView\_ContextMenuOpeningEvent(object sender, Telerik.WinControls.UI.ContextMenuOpeningEventArgs e)

{

e.ContextMenu = contextMenu.DropDown;

}

## RadScheduler

(Esto no se usa pero se ha hablado de utilizarlo por lo que vendría bien mirárselo, estos son apuntes de cuando le eché yo un vistazo en su momento)

RadScheduler muestra todos sus eventos en una "Tabla de tiempo". Esta tabla consiste en:

La cabecera que muestra las fechas en las que se pueden crear citas

Debajo de la cabecera se muestran las citas que duran todo el día, es decir que no tiene una duración especificada

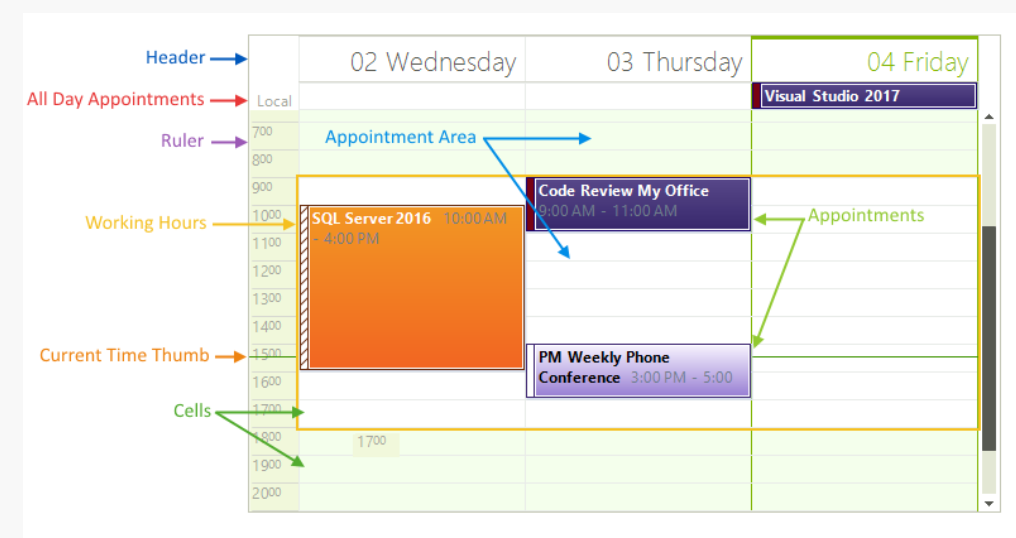
Después está la zona de citas que contiene cualquier número de citas

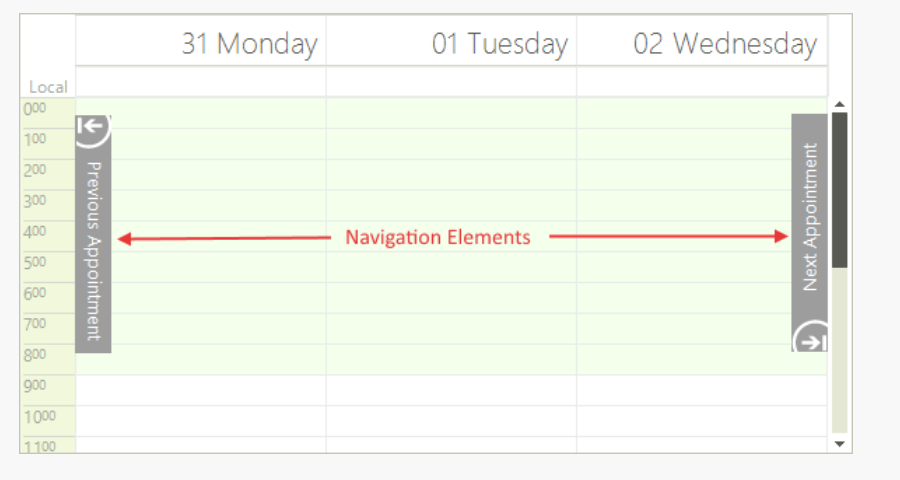
En el borde izquierdo del scheduler encontramos la "regla de tiempos"

Se pueden definir "Horas de trabajo" para facilitar la visualización, estas horas tienen un color mas claro.

En la mayoría de las vistas del scheduler la fecha actual se destaca para una mejor presentación visual.

Además, los elementos de navegación permiten al usuario saltar a la cita mas cercana tanto hacia adelante como hacia atrás





Month View

La vista por mes solamente tiene la cabecera, celdas para cada día del mes y 0 o mas citas en cada celda.

Agenda View

Es una tabla estructurada como una lista simple en la que se listan las citas para un periodo de tiempo determinado que se define en la propiedad "DayCount". En esta vista cada cita es una fila y a diferencia de las otras vistas no tiene celdas/filas vacías ya que los días sin citas no son mostrados, por lo tanto el usuario no puede mover o redimensionar las citas.

Esta vista está creada sobre un RadGridView por lo tanto dispone de la función de agrupamiento que usamos en otras tablas.

Por defecto la vista viene agrupada por fecha pero el usuario puede cambiarlo y realizar la agrupación sobre cualquier otra columna.

Para realizar la agrupación por código se deben añadir GroupDescriptors:

GroupDescriptor descriptor = new GroupDescriptor();

descriptor.GroupNames.Add("NombreColumna", ListSortDirection.Ascending);

agendaViewElement.Grid.GroupDescriptors.Add(descriptor);

Para coger el grid que está utilizando el scheduler: SchedulerAgendaViewElement.Grid;

Propiedades Agenda View:

scheduler.GroupByDate = true/false;

Indica si los datos de la agenda se agrupan por fecha.

scheduler.DayCount= (int)

Especifica el número de días mostrados en la vista, por defecto 1.

this.radScheduler1.ActiveViewType = Telerik.WinControls.UI.SchedulerViewType.Agenda;

Cambiar la vista activa, en este caso a Agenda View.

## RadTextBox

Si la propiedad **MultiLine** del TextBox está a True, puedes expandirlo verticalmente para que se puedan escribir en varias líneas.

Para que al pulsar Enter en un TextBox cambie de línea en lugar de activar el botón por defecto del formulario,se debe cambiar la propiedad **AcceptsReturn** a true. Lo mismo con **AcceptsTab** para que el tabulador tabule la línea en lugar de moverse por los controles.

La propiedad **NullText** especifica el texto que se ve en el TextBox mientras no hay nada escrito,ejemplo:

tbContainer.NullText = "Enter a container name";

Para quitar los bordes del TextBox y que se parezca mas a una etiqueta cambiar **Visibility** a “Collapsed”

(tbMultiLine.TextBoxElement.Children[2] as BorderPrimitive).Visibility = ElementVisibility.Collapsed;

**RadMaskedEditBox** te permite crear una plantilla a la que el texto se tiene que adaptar,por ejemplo que solo permita 9 caracteres y que solo sean números,o que estén separados por un guión cada 3 números. En lugar de la propiedad Text hay que usar la propiedad **Value** junto a **MaskType** y **Mask**

MaskType puede estar a None o Standard. La Mask usa ciertos caracteres para especificar los datos,dejo la lista de la documentación:

0 Digit, required. This element takes any number between 0 and 9

9 Digit or space, optional.

L Accepts letters only. Escapes a mask character, turning it into a literal. "\\" is the escape sequence for a back slash. For example, "\# ###", i.e. "# 123".

? Letter, optional.

? Character, required.

? Character, optional.

A Alphanumeric, required. Accepts any symbol.

a Alphanumeric, optional.

. Decimal placeholder.

, Thousands placeholder.

: Time separator.

/ Date separator.

$ Currency symbol.

< Shift down. Converts all characters that follow to lowercase.

> Shift up. Converts all characters that follow to uppercase.

| Disable a previous shift up or shift down.

\ Escape. Escapes a mask character, turning it into a literal. "\\" is the escape sequence for a backslash.

All other characters

All non-mask elements will appear as themselves within RadMaskedEditBox. Literals always occupy a static position in the mask at run time, and cannot be moved or deleted by the user

**AutoCompleteMode** sirve para autocompletar texto,tiene 4 estados

None

Suggest: Da una lista de sugerencias

Append: Autocompleta con la sugerencia mas cercana

SuggestAppend: Las dos cosas

Puedes usar “**AutoCompleteCustomSource**” para definir de donde vienen las sugerencias o crear las tuyas propias.

**Ejemplo de código para crear sugerencias por si fuera necesario:**

tbAutoComplete.AutoCompleteSource = AutoCompleteSource.CustomSource;

AutoCompleteStringCollection containers = new AutoCompleteStringCollection(); containers.Add("Box");

containers.Add("Bundle");

containers.Add("Pallet");

containers.Add("Carton");

tbAutoComplete.AutoCompleteCustomSource = containers; tbAutoComplete.AutoCompleteMode = AutoCompleteMode.SuggestAppend;

## Documentación Telerik

Junto a las herramientas de telerik se te tiene que haber instalado su demo llamada DemoApplications,en esta aplicación tienes ejemplos de código que deberías de complementar con su documentación online, aquí dejo enlaces a la documentación de los controles mas utilizados:

**GridView:**

[**https://docs.telerik.com/devtools/winforms/controls/gridview/overview**](https://docs.telerik.com/devtools/winforms/controls/gridview/overview)

**VirtualGrid:**

[**https://docs.telerik.com/devtools/winforms/controls/virtualgrid/overview**](https://docs.telerik.com/devtools/winforms/controls/virtualgrid/overview)

**TreeView:**

[**https://docs.telerik.com/devtools/winforms/controls/treeview/treeview**](https://docs.telerik.com/devtools/winforms/controls/treeview/treeview)

**RibbonBar:**

[**https://docs.telerik.com/devtools/winforms/controls/ribbonbar/overview**](https://docs.telerik.com/devtools/winforms/controls/ribbonbar/overview)

**RibbonForm:**

[**https://docs.telerik.com/devtools/winforms/controls/forms-and-dialogs/ribbonform/ribbonform**](https://docs.telerik.com/devtools/winforms/controls/forms-and-dialogs/ribbonform/ribbonform)

**ColorDialog:**

[**https://docs.telerik.com/devtools/winforms/controls/forms-and-dialogs/colordialog/colordialog**](https://docs.telerik.com/devtools/winforms/controls/forms-and-dialogs/colordialog/colordialog)

**MultiColumnComboBox:**

[**https://docs.telerik.com/devtools/winforms/controls/multicolumncombobox/multicolumncombobox**](https://docs.telerik.com/devtools/winforms/controls/multicolumncombobox/multicolumncombobox)

**DataFilter:**

[**https://docs.telerik.com/devtools/winforms/controls/datafilter/overview**](https://docs.telerik.com/devtools/winforms/controls/datafilter/overview)

**Progress Bar**

[**https://docs.telerik.com/devtools/winforms/controls/track-and-status-controls/progressbar/overview**](https://docs.telerik.com/devtools/winforms/controls/track-and-status-controls/progressbar/overview)

**RadWaitingBar**

[**https://docs.telerik.com/devtools/winforms/controls/track-and-status-controls/waitingbar/waitingbar**](https://docs.telerik.com/devtools/winforms/controls/track-and-status-controls/waitingbar/waitingbar)

## Apuntes sobre Visual Studio

**Buscar Referencia**: click derecho sobre una palabra>Buscar referencia nos da una lista de momentos del código en que aparece, pudiendo pulsar sobre dichas referencias para que nos lleven a esa sección del código. Se puede filtrar por formulario actual,proyecto actual,documentos abiertos etc.

**Buscar palabras**: CTRL+F abrirá un buscador para buscar palabras concretas en el código, se puede filtrar de la misma manera que en Buscar Referencias.

**Regiones**: Se usan para organizar el código. Se usa la palabra #region antes del código a encapsular y #endregion al final. La ventaja principal es poder contraer dichas regiones de manera que podamos ocultar un gran cantidad de código y sea mas fácil navegar por el.

**Extensiones**: Las herramientas y extensiones de visual studio se pueden instalar desde varios sitios. Primero tienes Herramientas>Extensiones y actualizaciones, desde donde podemos instalar Extensiones oficiales de visual studio. Por otro lado tenemos Herramientas>Administrador de paquetes NuGet>Administrar paquetes NuGet para la solución, donde tendremos acceso a muchas otras herramientas para instalar,como el log4NET o el Newtonsoft. Por último también podemos instalar extensiones desde la página web de Visual Studio Market Place.

**Agregar DLL:** click derecho sobre el proyecto correspondiente>Agregar>Referencia.. Se abrirá una ventana con referencias del proyecto. Si pulsamos en examinar podremos añadir otras DLL al proyecto, si les desmarcamos el check, las quitamos.

**Agregar WebService:** click derecho sobre el proyecto correspondiente>Agregar>Referencia de servicio. Aquí pegaríamos la dirección del WebService,lo nombramos y lo añadimos. Esto creará varias clases para el webservice,la que usaremos cuando queremos llamar a nuestro webservice se llamará igual que el mismo pero con Client al final. Por ejemplo añado una referencia al web service ArticulosWS, para usar sus métodos llamaré a ArticulosWSClient.