

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería del Software

APIs de seguridad en .NET Framework
Security APIs for .NET Framework

Realizado por
Abel Domínguez Ruiz
Tutorizado por
Isaac Agudo Ruiz
Departamento
Lenguajes Y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Noviembre de 2014

Fecha defensa:
El Secretario del Tribunal

Resumen

Este trabajo hace un estudio de algunas de las herramientas de seguridad disponibles en .Net Framework así como la forma de usarlas en un desarrollo web bajo la metodología de desarrollo de ASP.NET siguiendo el modelo Vista-Controlador y usando como entorno de desarrollo Visual Studio. Además de repasar las herramientas disponibles y la forma de uso se ha desarrollado también una aplicación de ejemplo: **ItemCoteca-Web**; en la que se demuestra cómo resolver el registro de usuarios, la autenticación y autorización de forma segura. En paralelo se ha implementado un cliente Android: **Itemcoteca-app** que realiza autenticación, registro y dispone de un chat seguro de incidencias para hablar con administradores Online

Este trabajo no solo presenta las APIs disponibles sino también las buenas prácticas que define Microsoft para el uso de sus herramientas. En particular nos centramos en los tres tipos de controladores que podemos encontrar en una aplicación web Asp.net que son Controllers, WebApi y SignalR, comentando sus diferencias y su uso para clientes Web y móviles.

Palabras clave.

Asp.net – Active Server Pages

Visual Studio (VS) – Entorno de desarrollo creado por Microsoft

Summary

This document makes a study about the security tools in .Net Framework and how to use them in a web development under the ASP.NET development methodology Model-View-Controller and using Visual Studio as development environment. In addition to reviewing the available tools and how to use them, a sample application has also been developed: ItemCoteca-Web; in which it is shown how to solve user registration, authentication and authorization in a secure way. In parallel we have implemented an Android client: Itemcoteca-app that performs authentication, registration and has a secure online chat with managers in order to resolve incidences.

This work not only presents the APIs available, but also the good practices defined by Microsoft for their tools. In particular we focus on three types of controllers that can be found in a Asp.net web application: Controllers, WebAPI and SignalR, discussing their differences and their use for Web and mobile clients.

Key words.

Asp.net: Active Server Pages

Visual Studio (VS) – Development Environment made by Microsoft

Contenido

Introducción.....	1
Capítulo 1 Definición y estructuración correcta de un proyecto ASP.NET MVC	3
1.1 Convention over Configuration.....	3
1.2 Tabla de rutas	4
1.3 Definiendo la estructura del proyecto	5
Capítulo 2 Data annotations y Validaciones Remotas	8
2.1 Las data annotations, validaciones cliente	8
2.2 Validaciones remotas.....	11
2.3 IValidatableObject	13
2.4 EntityFramework para consultas a base de datos.....	14
2.5 Defensa contra las inyecciones SQL	16
Capítulo 3 Cifrado y Hash en C#. Paquete de clases Security.	18
3.1 Funciones Hash	18
3.2 MD5.....	19
3.3 SHA1	20
3.4 GUID, Globally Unique Identifier	21
3.5 Cifrado simétrico	22
3.6 Cifrado Asimétrico.....	24
3.7 Generación de claves para cifrado y descifrado.....	25
3.8 Ejemplo de cifrado AES	27
Capítulo 4 Control de Acceso en .NET.....	28
4.1 Explicación e introducción al contexto	28
4.2 Autorización.....	30
4.3 La interfaz IAttribute	30
Capítulo 5 Descripción de la aplicación de DEMO, Caso de uso.	33
5.1 Validación de Registro.	33
5.2 Diagrama de secuencia generación token.....	34
5.3 Diagrama de secuencia confirmación token	35
5.4 Código de ejemplo de la generación del token.....	35
Capítulo 6 Servicios REST Seguros en Android y C#.....	36
6.1 Web API	37
6.2 Consumo de un recurso Web Api en Android.....	38
Capítulo 7 SignalR.....	40
7.1 Primeros pasos a SignalR.....	41
7.2 Definición de nuestro Hub.....	42
7.3 Configuración de la parte cliente web en JQuery	43
7.4 Los métodos Override de los Hub.....	45
7.5 Mensajes privados por usuarios	46
7.6 Mensaje privado a grupos	46
7.7 Como usar Android-SignalR-client.....	47
Conclusiones.....	52
Anexo	53

Introducción

Para aclarar exactamente la motivación de este trabajo fin de grado, el lector ha de tener ciertos conocimientos sobre arquitecturas por capas, siguiendo el modelo Modelo-Vista-Controlador en el desarrollo de aplicaciones web, el haber desarrollado con anterioridad este tipo de aplicaciones web le será más fácil su comprensión.

La evolución de ASP.NET MVC ha sufrido a lo largo de la historia de C# una corta, pero gran evolución debido al fracaso de SilverLight. Para entrar en contexto, SilverLight fue una tecnología basada en una arquitectura MV-VM, Modelo vista – Vista Modelo, que para ser usada era necesario la instalación de un Plugin en nuestro navegador. Pero al haber nacido los dispositivos móviles como los SmarthPhones y tablets, hizo que esta tecnología desapareciese debido a su incompatibilidad con estos dispositivos y Microsoft diese un paso al frente ofreciendo un producto nuevo y totalmente reorganizado, creando así el ASP.NET MVC que ahora se conoce y del que trata éste proyecto.

El objetivo de este proyecto es que un usuario que nunca haya programado en ASP.NET MVC una aplicación empresarial, tenga una guía de referencia rápida o le sirva como aprendizaje autodidacta a la hora de desarrollar de forma correcta aquellos conceptos de seguridad básicos que debe de tenerse en cuenta en su desarrollo. Para ello vamos a explicar en primera instancia, las herramientas necesarias que hay actualmente en ASP.NET MVC para desarrollar la seguridad en una aplicación web sin entrar en gran detalle de: validaciones en servidor y cliente, evitar ataques de inyección SQL, las librerías criptográficas para cifrado y Hash, autorización, autenticación, uso de cookies, creación de recursos web, consumición de recursos a través de un cliente Android, ejemplos de cómo se deben de implementar todos y cada uno de los conceptos y además una introducción a SignalR, un framework dedicado a sistemas en tiempo reales en la web con un ejemplo entre el servidor ASP.NET MVC y un cliente Android.

El contenido de la memoria se dividirá en dos partes, una de ellas será el cuerpo principal, donde explicaremos cada uno de los conceptos y herramientas de seguridad con ejemplos y la otra será un anexo donde explicaremos todo aquello que tenga relación con el contenido principal, pero que no sea relevante con el tema de seguridad o recursos web, como por ejemplo, configuración del entorno o explicación de algunos patrones.

Todo lo que se explique a continuación no solo puede ser usado en una aplicación Web, si usted quiere desarrollar la lógica de negocio en web y esta ser usada además en un cliente remoto, puede utilizar WPF, o Windows Phone utilizando el mismo lenguaje, lo único que debería de cambiar es las interfaces gráficas de usuario.

La estructura que sigue éste documento consistirá en una introducción a las herramientas de desarrollo, tras haber realizado ese paso hablaremos sobre como configurarlas y preparar nuestros proyectos. Tras estar listo para el trabajo duro, iremos desarrollando los casos de

uso que se encuentran en el anexo y explicando aquellas cosas que sean importantes para el propósito del trabajo fin de grado.

Dado que es imposible en 50 páginas hablar de todo lo referido con C# en el anexo se podrá encontrar referencias a los patrones de diseños utilizados como información adicional, algunas de las buenas prácticas, y herramientas interesantes que trae C#. Para más información siempre se puede consultar la web de msdn en la que se puede encontrar casi todo lo referente a los frameworks y nuggets que usaremos.

Capítulo 1 Definición y estructuración correcta de un proyecto ASP.NET MVC

Los primeros pasos que daremos en este proyecto será tratar de los aspectos más importantes de la convención entre los programadores C#. Todos los lenguajes tienen sus metodologías y sus formas correctas de definir sus componentes, clases, atributos, etc. C# también tiene su convención realizada por los desarrolladores, y a nivel mundial se ha declarado esta formalidad como el CoC Convention Over Configuration. Dado que no vamos a entrar en detalle cómo se debería de hacer correctamente cada una de ellas, explicaremos las más importantes para un proyecto en ASP.NET MVC

1.1 Convention over Configuration

Convención sobre configuración es la metodología y filosofía que sigue un desarrollo mvc que busca reducir el número de decisiones que un desarrollador necesita tomar ganando así simplicidad pero sin perder flexibilidad.

Esto significa que en la estructura de paquetes que tenemos nada más crear el proyecto debe de seguir esa convención y nunca salirnos de ella.

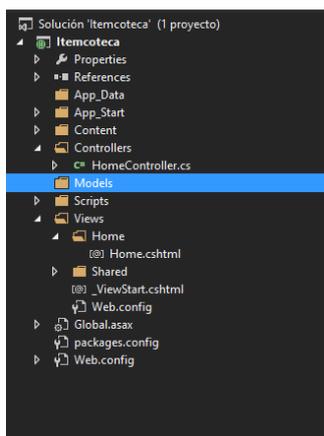


Figura 1.1

En esta imagen vemos que tenemos una carpeta para los controladores (el nombre de sus clases debe de acabar en Controller) los modelos en la carpeta Models y las vistas en la carpeta Views, siguiendo así la estructura MVC.

Cada vez que la aplicación necesite un controlador, hará la búsqueda en la carpeta raíz controllers, si la aplicación necesita acceder a un modelo de datos lo hará en la carpeta de datos, si nuestra aplicación necesita acceder a una vista lo hará en la carpeta de Views. Será en la tabla de rutas donde definamos que patrones de configuración necesitan las URL de la página para indicarle como hacer las búsquedas.

A diferencia del desarrollo dirigido por test (TDD), asp.net mvc está diseñado para seguir el desarrollo dirigido por dominio (DDD, Domain Drive Design) el cual hace que nos centremos en el dominio siendo éste el corazón de la aplicación y no nos obligue solamente a usar una tecnología en concreto, sino que podamos desacoplar esa capa y hacer uso de la lógica de negocio desde otro sistemas o tecnologías, aunque a su vez también permita usar el framework de .NET. No obstante, también se puede aplicar TDD ya que los patrones Repository y DependencyInjection nos permitirán hacer mocking de nuestras estructuras.

A la hora de implantar la seguridad, debemos de hacerlo tras haber definido bien nuestro modelo del domino, lo que recomiendan las buenas prácticas es hacer el trabajo en paralelo, primero diseñando nuestra arquitectura, y luego ir implementando la seguridad por capas.

Otra anotación importante respecto al CoC es la definición de nuestras tablas en las bases de datos y el nombre de las clases y atributos de dichas clases. Las tablas han de llamarse en plural, y las clases en singular. Los atributos de nuestras clases deben de llamarse exactamente igual que en nuestra base de datos.

1.2 Tabla de rutas

La tabla de ruta es la clase que va a encargarse de regir el orden y jerarquía de todas las vistas y controladores de nuestro proyecto. ¿Por qué es importante definir la tabla de rutas?, a diferencia de una aplicación de escritorio o dispositivo móvil, el programador es capaz de dirigir el flujo de la aplicación, obligando fácilmente a un usuario que pase por una serie de formularios o pantallas antes de llegar a un dialogo en concreto.

En una aplicación web no ocurre esto, el usuario puede acceder desde la URL de nuestro navegador a cualquier otra vista o controlador si conoce las rutas, saltándose de esta manera el orden del flujo que queremos definir.

La tabla de rutas nos ayudará junto con el fichero WebConfig y Global.asax a que esto no llegue a ocurrir, y un usuario sin haberse registrado y comprobar su autorización pueda acceder a zonas restringidas.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{action}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Figura 1.2

En éste ejemplo de una WebApi podemos ver la definición de una ruta. Lo que declaramos en éste método Register es el nombre de la ruta que vamos a definir, el template y parámetros opcionales. Lo más destacado es el routeTemplate que define la estructura de la URL que debe de seguir nuestras llamadas desde los clientes para la consumición de recursos, en esta imagen la url que nos define sería la siguiente:

<Http://www.mysitioweb.com/api/<NombreController>/<NombreDelMetodo>/<Params>>

El nombre del controlador debe de ir sin el prefijo "Controller" ya que por Convention over Configuration, el framework es lo suficientemente inteligente como para encontrar ese recurso.

1.3 Definiendo la estructura del proyecto

La buena definición de nuestro proyecto deberá de seguir una arquitectura por capas separando la lógica de negocio, de la interacción con el usuario a través de las GUIs. Por lo que nuestro proyecto ha de tener 3 partes.

1 - La primera parte vamos a denominarla Domain o dominio, ésta será la parte que contenga la lógica de negocio, sus reglas, entidades, y consultas a base de datos y en la que vamos a centrarnos desde el primer momento. La razón por la que ésta parte está desacoplada del resto es porque si en un futuro queremos implementar la lógica con una WEB API para ofrecer servicios terceros, no tendremos que volver a implementarla, simplemente diseñar una interfaz web y crear los métodos de accesos. Si por ejemplo queremos un cliente en WPF tendremos la lógica preparada, lo mismo ocurriría si queremos un proyecto en Windows Phone. Delegando el control de acceso La parte de seguridad que implementaremos en esta capa será de criptografía de datos, cifrando las contraseñas y datos que nos hayan llegado de la capa superior. Haremos comprobaciones de usuarios registrados o no con anterioridad, aquellos campos que no se puedan validar de parte del cliente los validaremos aquí, como por ejemplo la inyección SQL o duplicación de identidades.

2- La segunda parte que vamos a desarrollar en nuestro proyecto será la parte de cara al cliente la denominaremos .GUI/Aplication o aplicación. Aquí es donde definiremos nuestras reglas de validación, llamadas a otros WebServices si fuera necesario, comunicación de Hubs, etc.

La parte de seguridad que implementaremos en esta capa será de validaciones y autorizaciones, para controlar el acceso de los usuarios a zonas restringidas. Debemos de controlar en esta capa que los datos introducidos siguen el formato deseado, comprobación de campos nulos.

3 – Para finalizar, la última capa que desarrollaremos será la WEB API. La WEB API estará integrada junto con la parte de los controladores de la capa GUI. La diferencia entre un controlador de vista y uno de api es que las clases de GUI extienden de Controller, mientras que las de WebApi extienden de ApiController. Unas devuelven vistas y otras devuelven contenido serializado.

En esta capa se diseñará un acuerdo de interfaz para comunicarnos con cualquier cliente con soporte REST, con una serie de funciones básicas para que se haga una demostración de la facilidad que se puede obtener a la hora de comunicar sistemas terceros siguiendo ésta estructura.

En esta capa desarrollaremos una seguridad semejante a la de la capa Aplicación, a diferencia de que las validaciones no podemos hacerlas de parte cliente, ha de ser la aplicación Android la que haga esas validaciones.

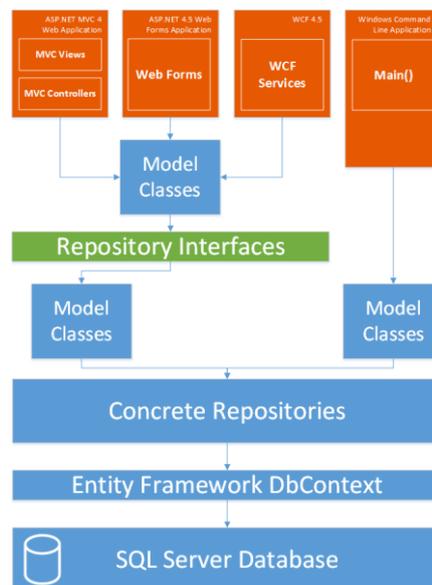


Figura 1.3

Tras haber estructurado nuestro proyecto lo único que nos queda es ir desarrollando por casos de uso todo nuestro proyecto, y a su vez, ir aplicando la seguridad que veamos necesaria en cada una de las capas.

La buena metodología que seguiremos es resumida en el diagrama de la figura X.X, la cual para finalizar solo nos quedaría comentar el funcionamiento de las Interfaces Repository que se explicará junto con la creación de clases del dominio.

Para finalizar el capítulo nos queda decir que la parte cliente será lo último a implementar pero básicamente podemos resumir la estructura de nuestro proyecto en el siguiente diagrama de arquitectura.

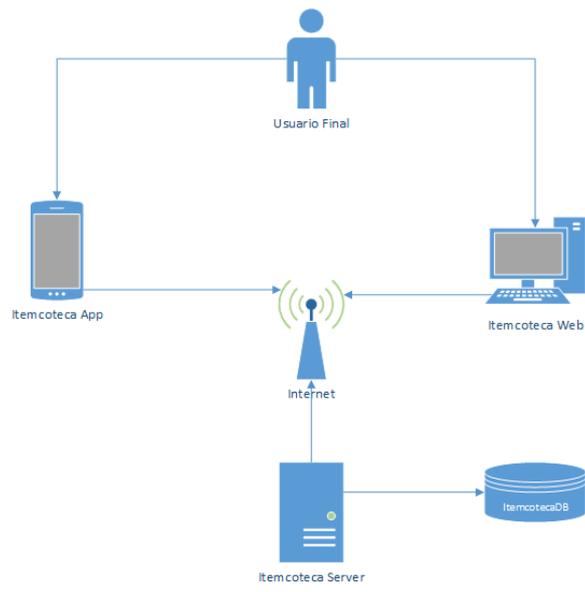


Figura 1.4

Capítulo 2 Data annotations y Validaciones Remotas

En este capítulo explicaremos la forma de validar datos de nuestras vistas a través de dos elementos fundamentales, las Data annotations y las validaciones remotas. Éste tipo de validaciones tanto cliente como servidor pueden hacerse desde código javascript, pero dado que son situaciones comunes y típicas ASP.NET MVC ya tiene integrado en el framework herramientas que nos ayudarán a implementarlas.

2.1 Las data annotations, validaciones cliente

Los data annotations son validaciones cliente, conocidas como experiencia de usuario, validaciones de datos que no requieren información o comunicación con el servidor. Un caso común de éste tipo de validaciones se puede encontrar en un formulario de registro de usuario en el que requiere una contraseña, esta contraseña debe de tener un tamaño mínimo y máximo, y además debe de tener un campo Repetir contraseña en el que deben de coincidir.

ASP.NET MVC nos trae implementado unas data annotations que nos ayudará a implementar todo este tipo de validaciones. Es de destacar que no se pueden sobrescribir las validaciones de experiencia de usuario, si tuviésemos que hacer comprobaciones clientes customizadas, deberíamos de usar código javascript o hacerlo a través de las validaciones remotas.

Para éste ejemplo vamos a usar un formulario de registro de usuario para itemcoteca web. Vamos a tener nuestro ViewModel los datos de registro como se muestra a continuación:

```
public class RegistroUserViewModel
{
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public string Correo { get; set; }
    public string Telefono { get; set; }
    public string Movil { get; set; }
    public string Password { get; set; }
    public string RepetirPassowrd { get; set; }
}
```

Figura 2.1

Una vez creado nuestro ViewModel debemos de configurar la vista para que reciba ésta información y además compruebe que las data annotation que afiliemos a cada atributo se cumplen.

Para crear un formulario en asp.net mvc que reciba o envíe datos al servidor con un objeto de tipo RegistroUserViewModel, debemos de hacer un “using” apuntando a la clase contenedora.

```
@model Itemcoteca.Domain.ViewModels.RegistroUserViewModel

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title></title>
</head>
<body>
  <div id="toolBar"> Usuario</div>
```

Figura 2.2

De ésta manera cuando la vista se comunice con el servidor enviará solo objetos de la clase importada.

Para crear un formulario usando razor y comprobar las validaciones debemos de iniciar el formulario con un `@using` (`Html.BeginForm("MétodoAccion", "Controlador")`)

```
<link href="@Url.Content("~/CSS/CSSItemcotecaMain.css")" rel="stylesheet" type="text/css" />
<h2>Creación de cuenta</h2>
<div class="Forms">
<h1><span>Formulario</span></h1>

  @using (Html.BeginForm("RegistroUsuario", "Registro"))
  {
    @Html.ValidationSummary(true)
    <div class="validation-field">
      @Html.TextBoxFor(m => m.Nombre, new { @placeholder = "Nombre" })
      @Html.ValidationMessageFor(m => m.Nombre)
    <br /><br />
    </div>
```

Figura 2.3

`@Html.ValidationSummary(true)` sirve para mostrar un resumen de las validaciones erróneas en el caso de que no hayan sido superadas y `@Html.ValidationMessageFor(m => m.Nombre)` sirve para mostrar el mensaje en el lugar que queramos.

```

[HttpPost]
public ActionResult RegistroUsuario(RegistroUserViewModel registro)
{
    //Registramos al usuario
    if(ModelState.IsValid)
    {
        string token = _service.RegistrarCliente(registro);

        if (token != null && !token.Equals("")) {
            RegisterValidation emailValidation = new RegisterValidation();

            //Preparamos el formato del correo
            Correo correo = new Correo(registro.Correo,"Itemcoteca PFG Registro");
            correo.MensajeStandard(token);

            //Enviamos el correo al usuario registrado
            emailValidation.EnviarCorreoVerificacion(correo);
            ViewBag.Correo = registro.Correo;
        }

        return View("RegistroEnviadoEmail");
    }
    else
    {
        return View(registro);
    }
}

```

Figura 2.4

Con `ModelState.IsValid` sabremos si todos los campos del formulario han pasado las validaciones en el cliente. Realmente hacemos dos validaciones, una en el cliente y el `ModelState` valida lo mismo pero de parte del servidor, esto es así para asegurarnos de que si se accede al método de acción del controlador a través de un programa externo que no sea un navegador, le inyectemos los datos de forma incorrecta.

Gracias al `ModelState`, se pueden evitar ataques del tipo SQL Injection, ya que internamente el framework generará expresiones irregulares para impedir la formación de cadenas que no concuerden con la definición del Data Annotation y además los valores son parametrizados.

Hay decenas de Data annotations en asp.net mvc, pero se explicará a continuación las usadas en nuestro proyecto.

```

public class RegistroUserViewModel
{
    [Required(ErrorMessage = "Campo requerido")]
    public string Nombre { get; set; }

    [Required(ErrorMessage = "Campo requerido")]
    public string Apellidos { get; set; }

    [Required(ErrorMessage = "El email es requerido")]
    [EmailAddress(ErrorMessage = "El email no tiene el formato correcto")]
    public string Correo { get; set; }

    [Required(ErrorMessage = "Teléfono es requerido")]
    [Phone(ErrorMessage = "El telefono no es válido")]
    public string Telefono { get; set; }

    public string Movil { get; set; }

    [Required(ErrorMessage = "La contraseña es requerida")]
    [MinLength(8, ErrorMessage = "La password debe de tener al menos 8 caracteres")]
    [MaxLength(16, ErrorMessage = "La password debe de tener como máximo 16 caracteres")]
    public string Password { get; set; }

    [Compare("Password", ErrorMessage = "Los campos Password y Repetir Password no coinciden")]
    [Required(ErrorMessage = "Repita la contraseña")]
    public string RepetirPassowrd { get; set; }
}

```

Figura 2.5

Required: El campo es requerido

EmailAddress: Debe de cumplir un formato de dirección de correo electrónico

Phone: Debe de ser numérico y además tener 9 dígitos

MinLength: Tiene que ser al menos X caracteres

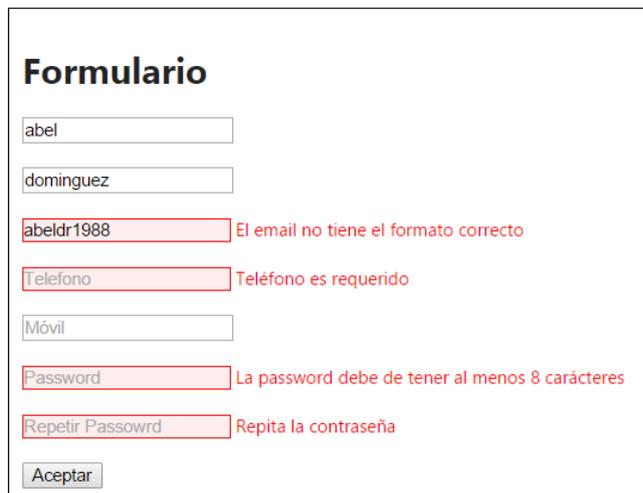
MaxLength: Tiene que tener menos de Y caracteres

Compare: Compara dos atributos del ViewModel, si son iguales devuelve true, en otro caso false.

Range: El campo debe de comprenderse entre dos valores. Es útil para números de teléfonos o rangos de edad.

Así es como se mostrarían finalmente las validaciones que no se han cumplido.

Creación de cuenta



The screenshot shows a registration form titled "Formulario". It contains the following fields and validation messages:

- Field:
- Field:
- Field: with a red border and a red message: "El email no tiene el formato correcto"
- Field: with a red border and a red message: "Teléfono es requerido"
- Field:
- Field: with a red border and a red message: "La password debe de tener al menos 8 caracteres"
- Field: with a red border and a red message: "Repita la contraseña"
- Button: "Aceptar"

Figura 2.6

2.2 Validaciones remotas

En este apartado vamos a ver cómo funcionan las validaciones remotas en asp.net mvc, las validaciones remotas son unas validaciones cliente, es decir, que se ejecutan en cliente vía javascript pero que lo que hace es realizar una petición http post para hacer peticiones clientes que aceptan a datos del servidor.

Por ejemplo, un caso muy simple que se puede resolver con validaciones remotas, es en nuestro formulario de registro queremos comprobar si un email de usuario está

registrado ya o no según vamos escribiendo el nombre o según cambiamos de textbox, realmente esta validación necesita comprobar datos de servidor porque el cliente no contiene esa información

Tenemos varias maneras de realizar esto, una puede ser con Ajax, pasando como nombre de usuario cada vez que escribimos una letra al método del controlador para que haga la comprobación en la base de datos. Aunque podríamos hacerlo usando Ajax nativo, asp.net mvc nos lo trae integrado para poder hacerlo sin necesidad de usar Ajax.

Para mostrar el ejemplo usaremos el anterior ViewModel de registro y utilizaremos el atributo email.

Lo primero que debemos de hacer es utilizar el Data Annotation remote en nuestro atributo email de la siguiente manera

```
[System.Web.Mvc.Remote("CheckUsuario", "Registro", ErrorMessage="Este email ya existe")]
[Required(ErrorMessage = "El email es requerido")]
[EmailAddress(ErrorMessage = "El email no tiene el formato correcto")]
public string Correo { get; set; }
```

Figura 2.7

Realmente el servidor no comprobará el validation remote hasta que la validación cliente esté correcta. Una vez se hayan cumplido las otras validaciones se hará la validación remota.

```
public ActionResult CheckUsuario(string Correo)
{
    var existeUsuario = _service.FindUserByEmail(Correo);
    if (existeUsuario == null)
    {
        return Json(true, JsonRequestBehavior.AllowGet);
    }

    return Json(false, JsonRequestBehavior.AllowGet);
}
```

Figura 2.8

Este sería nuestro método de la validación remota que se encargará de comprobar si el usuario ya existe o no en la base de datos, Si obtenemos null devolveremos una respuesta json que valdrá true, en otro caso devolverá false y ya el framework es lo suficientemente inteligente como para tratar la respuesta.

De nuestra vista no debemos de cambiar absolutamente nada respecto al ejemplo anterior y esto será lo que obtengamos cuando el email ya exista en nuestra base de datos

abeldr1988@gmail.com Este email ya existe

Telefono

Figura 2.9

2.3 IValidatableObject

Cuando queramos hacer validaciones complejas, que no solo acepte a un atributo, sino a varios de ellos o querramos hacer una validación implementando cierta lógica de negocio, con solo las Data Annotations explicadas anterior mente no nos bastaría.

Para ello nace la interfaz IValidatableObject, que al implementarla nos aparecerá como sobrescritura el método validate. Éste método no devuelve true o false por si ha funcionado bien, devolvemos una colección de ValidationResult que usará el framework para rellenar el ModelState.

En éste ejemplo haremos un caso de uso en el que el usuario podrá cambiar su contraseña, para ello deberá introducir su email y su nueva password. Para realizar esta funcionalidad implementaremos la interfaz IValidatableObject en nuestro ViewModel correspondiente, en éste caso será el LoginViewModel.

```
public class LoginViewModel : IValidatableObject
{
    [EmailAddress(ErrorMessage = "El email no tiene el formato correcto")]
    public string Correo { get; set; }

    [Required(ErrorMessage = "La contraseña es requerida")]
    [MinLength(8, ErrorMessage = "La password debe de tener al menos 8 caracteres")]
    [MaxLength(16, ErrorMessage = "La password debe de tener como máximo 16 caracteres")]
    public string Password { get; set; }

    [Compare("Password", ErrorMessage = "Los campos Password y Repetir Password no coinciden")]
    [Required(ErrorMessage = "Repita la contraseña")]
    public string RepetirPassword { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        ILoginRepository service = new LoginRepository();
        var result = service.CambiarPassword(this);
        if(result == "KO"){
            yield return new ValidationResult("El usuario no existe", new[]{"Correo"});
        }else if (result.Equals("ERROR")){
            yield return new ValidationResult("Ha habido un error con la base de datos");
        }
    }
}
```

Figura 2.10

Si el método no devuelve nada, se trabaja como si no hubiese habido ningún error y el ModelState nos aceptará toda la validación como si fuese correcta y realizaríamos la lógica de negocio en nuestro controlador.

abell123@gmail.com El usuario no existe

Password

RePassword

Cambiar

Figura 2.11

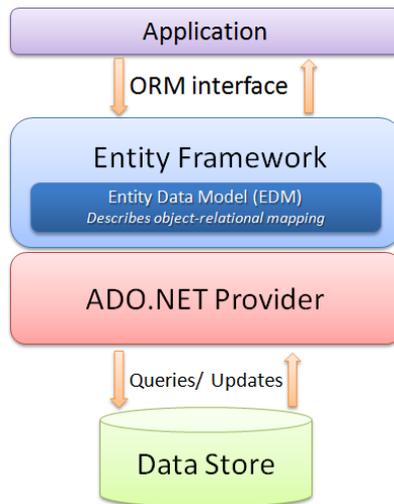
Éste sería el resultado de error tras comprobar nuestra validación y ver que es errónea

2.4 EntityFramework para consultas a base de datos.

EntityFramework es la librería que nos permitirá trabajar con conexiones a base de datos y sus persistencias. La primera versión de EntityFramework apareció en 2008 y fue altamente criticada y anteriormente a EntityFramework, las consultas se realizaban nativamente en C# y se tenían que parametrizar las query nativamente de la misma manera que en Java., usando ADO.NET para updates, select, deletes e insert.

La razón por la que nace EntityFramework es, una capa subyacente a nuestra lógica de negocio que se abstraiga del sistema gestor de base de datos que usemos. Si por ejemplo hacemos una consulta compleja en SQL y en un futuro debemos de cambiar la base de datos a Oracle, no tendríamos que cambiar todas las consultas en nuestro código.

EntityFramework es una interfaz ORM que se comunica con la aplicación y por debajo con ADO.NET, que sería el encargado de realizar las llamadas a la base de datos.



Siguiendo la convención, las clases deben de llamarse en singular, las tablas en plural. A veces EntityFramework no es capaz de reconocer una clase y asociarla a su tabla correspondiente, haciendo que salten excepciones de código al ejecutarse. Para solucionar este problema se debe de usar el Data Annotation Table, el cual recibe como parámetro el nombre exacto de la tabla a la que hace referencia esa entidad.

```

[Table("HistorialChat")]
public class HistorialChat
{
    public HistorialChat()
    {
    }

    public int ID { get; set; }
    public Nullable<System.DateTime> Fecha{ get; set; }
    public string CorreoUsuarioEnvia {get;set;}
    public string Mensaje { get; set; }
    public string ID_Grupo { get; set; }
    public Nullable<int> Recibido { get; set; }
}
  
```

Figura 2.12

La clase que realmente utilizaremos para realizar las consultas y la comunicación extiende de DbContext, la cual sigue la siguiente estructura que hemos comentado siguiendo el CoC.

```

public class ItemcotecaDBEntities : DbContext
{
    public ItemcotecaDBEntities() : base("name=ItemcotecaDBEntities")
    {
    }

    public DbSet<Articulo> Articulos { get; set; }
    public DbSet<Estados_Articulos> Estados_Articulos { get; set; }
    public DbSet<Estados_Pedidos> Estados_Pedidos { get; set; }
    public DbSet<Factura> Facturas { get; set; }
    public DbSet<Lista_Articulos_Comprados> Lista_Articulos_Comprados { get; set; }
    public DbSet<Log_Acceso> Log_Acceso { get; set; }
    public DbSet<Logs_Estados_Articulos> Logs_Estados_Articulos { get; set; }
    public DbSet<Pedido> Pedidos { get; set; }
    public DbSet<Role> Roles { get; set; }
    public DbSet<Seguimientos_Pedidos> Seguimientos_Pedidos { get; set; }
    public DbSet<Tarjeta> Tarjetas { get; set; }
    public DbSet<Usuario> Usuarios { get; set; }
    public DbSet<Comprado> comprados { get; set; }
    public DbSet<HistorialChat> HistorialChat { get; set; }
}

```

Figura 2.13

2.5 Defensa contra las inyecciones SQL

A menudo, las aplicaciones obtienen entradas externas (de un usuario o de otro agente externo) y realizan acciones en función de dichas entradas. Cualquier entrada derivada directa o indirectamente del usuario o de un agente externo puede inyectar contenido que aproveche la sintaxis del lenguaje de destino para realizar acciones no autorizadas. Cuando el lenguaje de destino es del tipo de Lenguaje de consulta estructurado (SQL), como Transact-SQL, esta manipulación se conoce como ataque de inyección de SQL. Un usuario malintencionado puede inyectar comandos directamente en la consulta y quitar una tabla de la base de datos, provocar un ataque de denegación de servicio o cambiar de alguna otra forma la naturaleza de la operación que se está realizando.

Los ataques de inyección de SQL se pueden realizar en Entity SQL proporcionando entradas malintencionadas a los valores que se utilizan en un predicado de consulta y en los nombres de los parámetros. Para evitar el riesgo de inyección de SQL, nunca debería combinar los datos proporcionados por el usuario con el texto de comandos de Entity SQL.

Para ello usaremos las llamadas expresiones lambda a la hora de hacer nuestras consultas. Las expresiones lambda tienen implementadas realmente LINQ to Entities. LINQ to Entities proporciona la capacidad de realizar consultas integradas en lenguajes (LINQ) que permite a los desarrolladores de software escribir consultas contra el modelo conceptual de Entity Framework mediante Visual Basic o Visual

C#. Las consultas con Entity Framework se representan mediante consultas de árboles de comandos, que se ejecutan en el contexto del objeto. LINQ to Entities convierte las consultas de Language-Integrated Queries (LINQ) en consultas de árboles de comandos, ejecuta las consultas en Entity Framework y devuelve objetos que se pueden usar tanto en Entity Framework como en LINQ. A continuación se muestra el proceso para crear y ejecutar una consulta de LINQ to Entities.

Una expresión lambda es una función anónima que se puede usar para crear tipos delegados o de árbol de expresión. Utilizando expresiones lambda, puede escribir funciones locales que se pueden pasar como argumentos o devolverse como valor de llamadas a funciones. Las expresiones lambda son especialmente útiles para escribir expresiones de consulta LINQ o EntityFramework.

Pues bien, el framework ya está adaptado para que intercepte todas aquellas cadenas de caracteres propias del lenguaje que sean SQL para interceptarlas y decidir si son válidas o no. Actualmente no se recomienda usar ADO.NET para realizar las consultas, porque con expresiones lambda o LINQ obtenemos dos cosas:

- 1) Seguridad anti inyecciones SQL o código malicioso
- 2) Rapidez a la hora de realizar consultas.

Sería imposible hablar de todas las funcionalidades del EntityFramework y hay libros muy reconocidos para ello, pero como ejemplo, expondremos una consulta básica:

```
var existe = context.Usuarios.Where(x => x.Correo == user.Correo).SingleOrDefault();
```

Figura 2.14

Context sería nuestro contexto de datos, una instancia de la clase ItemcotecaDBEntities, que tendrá como propiedades los DbSet que se muestran en la figura 3.2. Usuarios sería la equivalencia a la tabla Usuarios, Where nuestra acción y aquí es donde entra en detalle las expresiones lambda.

De esta forma parametrizamos totalmente los valores, ya que recordemos, que por debajo está ADO.NET que es el encargado de motorizar estas consultas.

SingleOrDefault() devuelve el valor de la base de datos cuando realmente se sabe que solo existe uno o ninguno. Si por ejemplo existiesen dos correos exactamente iguales, SingleOrDefault devolvería un Exception, porque no sabría cuál de los dos devolver, para ese tipo de situaciones deberemos de usar el método ToList();

Para más información de EntityFramework, se puede consultar la bibliografía del anexo.

Capítulo 3 Cifrado y Hash en C#. Paquete de clases Security.

En éste capítulo hablaremos de las primitivas criptográficas implementadas en .NET. Se explicará muy resumidamente una pequeña introducción a los tipos de cifrado y que posibilidades y herramientas ofrece .NET Framework

En primera instancia hablaremos sobre las funciones hash que se usaran como checksum y para el almacén seguro de contraseñas y acto seguido sobre funciones de cifrado simétricas y asimétricas.

3.1 Funciones Hash

Los hash que utiliza C# podemos obtenerlos desde el paquete de clases System.Security.Cryptography. Antes de empezar con la implementación de las funciones hash haremos un pequeño recordatorio de que son.

Las funciones hash asignan cadenas binarias de una longitud arbitraria a cadenas binarias pequeñas de una longitud fija. Una función hash criptográfica tiene la propiedad de que, mediante el cálculo, no es posible encontrar dos entradas distintas que generen aleatoriamente el mismo valor; es decir, los valores hash de dos conjuntos de datos deben coincidir si los datos correspondientes también coinciden. Pequeñas modificaciones en los datos ocasionan grandes cambios imprevisibles en el valor hash.

Estas funciones son obligatorias para almacenar las contraseñas en nuestras bases de dato y está legalmente penado si no se cumple este requisito no funcional. Tanto MD5 como SHA1 pueden usarse para esta finalidad, pero en nuestro caso particular hemos decidido usar SHA1 para una mayor protección.

Siguiendo lo establecido por la AGPD (Agencia Española de Protección de Datos) las contraseñas nunca deben de estar en texto claro y entendible por el ojo humano

Toda la información respecto a estas funciones se puede consultar en el siguiente link a la Agencia Española de Protección de Datos.

http://www.agpd.es/portalwebAGPD/canaldocumentacion/publicaciones/common/Guias/GUIA_SEGURIDAD_2010.pdf

3.2 MD5

El ejemplo de código siguiente calcula el valor hash MD5 de una cadena y devuelve el hash cómo una cadena hexadecimal de 32 caracteres. La cadena hash creada en éste ejemplo de código es compatible con cualquier función hash MD5 (en cualquier plataforma) que cree una cadena hash hexadecimal de 32 caracteres, el tamaño del valor hash del algoritmo MD5 es de 128 bits.

```
public static string cadena(string source)
{
    string result = "";
    using (MD5 md5Hash = MD5.Create())
    {
        string hash = GetMd5Hash(md5Hash, source);
        result+="El hash md5 de "+ source +" es: "+ hash +".";
        result+="Verificación del hash: .";

        if (VerifyMd5Hash(md5Hash, source, hash))
        {
            result+="Hashes iguales.";
        }
        else
        {
            result+="Hashes distintos.";
        }
    }
    return result;
}

private static string GetMd5Hash(MD5 md5Hash, string input)
{
    //Convertimos la cadena string de entrada en una array de byte para computar su hash
    byte[] data = md5Hash.ComputeHash(Encoding.UTF8.GetBytes(input));

    //Creamos un nuevo StringBuilder para agrupar los bytes y creamos una string
    StringBuilder sBuilder = new StringBuilder();

    //Iteramos cada byte de la función hash y sacamos su valor en hexadecimal
    for (int i=0; i<data.Length; i++)
    {
        sBuilder.Append(data[i].ToString("x2"));
    }

    //Devolvemos su valor hexadecimal en forma de string
    return sBuilder.ToString();
}

// Verificación de un hash y una string
private static bool VerifyMd5Hash(MD5 md5Hash, string input, string hash)
{
    // Obtenemos el hash en forma de string
    Boolean resultado=false;
    String hashOfInput= GetMd5Hash(md5Hash, input);

    // Comparamos los dos hash, el de entrada y el calculado por la cadena nueva.
    StringComparer comparer =StringComparer.OrdinalIgnoreCase;

    if (0==comparer.Compare(hashOfInput, hash))
    {
        resultado = true;
    }

    return resultado;
}
```

Figura 3.1

Resultado obtenido desde la llamada de un web service:

```

▼<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
  EL hash md5 de Holaaaaa ess: 8e853a4cae4f582a7023501b9f92234e.Verificación del hash: .Hashes iguales.
</string>

```

Figura 3.2

3.3 SHA1

El Sha1 es una función hash que nos devolverá el valor con un tamaño de 160 bytes el cual nosotros después convertiremos a cadena de string.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Web;

namespace GPSFriends.Controllers.Seguridad
{
    public class Seguridad
    {
        public static String EncriptarPassword(String password)
        {
            String result = "";
            SHA1 sha1 = SHA1Managed.Create();
            ASCIIEncoding encoding = new ASCIIEncoding();
            byte[] stream = null;
            StringBuilder sb = new StringBuilder();
            stream = sha1.ComputeHash(encoding.GetBytes(password));
            for (int i = 0; i < stream.Length; i++)
            {
                sb.AppendFormat("{0:x2}", stream[i]);
            }
            result = sb.ToString();
            return result;
        }
    }
}

```

Figura 3.3

Resultado de la función SHA1 insertando el valor de la contraseña en una base de datos:

Nombre	Apellidos	Password
Abel	Dominguez Ruiz	626a3e0deaaa419144915fd534e88363755299de
David	Oracle	7c4a8d09ca3762af61e59520943dc26494f8941b

Figura 3.4

Comparativa entre tamaños de md5 y sha1:

SHA1: 7c4a8d09ca3762af61e59520943dc26494f8941b

MD5: 8e853a4cae4f582a7023501b9f92234e

3.4 GUID, Globally Unique Identifier

GUID es un acrónimo para Globally Unique Identifier o identificador único global, es un número de 128 bits que es producido por el sistema operativo Windows o por algunas aplicaciones Windows para identificar un componente particular, una aplicación, un archivo, un registro en una base de datos y/o un usuario. Por ejemplo, podemos tener un sitio web que genere un GUID y se lo asigne a un usuario para grabar las acciones de este usuario en la sesión (lo que se conoce como session tracking).

GUID se utiliza también en el registro de Windows para identificar dlls COM. Algunos DBAs incluso utilizan un GUID como claves primarias de sus bases de datos. Un ejemplo de GUID generado es el siguiente: {ded53e2b-91e9-4682-b673-862ca6503b2e}

Si bien no está garantizado que un GUID generado sea único, el total de claves que se pueden generar (2 elevado a la 128 potencia) es tan grande que la probabilidad de que se repita es realmente muy pequeña. Para dar una idea de esto podemos decir que una aplicación que genere 10 mil millones de GUID la probabilidad de que se repita una clave generada es de 1 en un quintillón.

La clase GUID nos proporciona un método de acción llamado newGuid que lo que nos devolverá será un identificador único con una probabilidad muy baja de que se repita. Estos Guid's tienen muchas utilidades, podríamos usarlos de token para verificar cuentas, podríamos haberlos usado para resetear passwords, etc. El framework nos asegura que la probabilidad de que se repitan dos Guid's simultáneamente es casi imposible.

Ejemplo de código para uso de un GUID:

```
Guid g = Guid.NewGuid();  
var guid = g.ToString();
```

g	{bdf0a1b6-2915-41e7-8cdb-355bf9409dae}
guid	"bdf0a1b6-2915-41e7-8cdb-355bf9409dae"

Figura 3.5

Ejemplo de TeamViewer, una de las empresas que usan el GUID para la verificación del registro de usuario vía email:

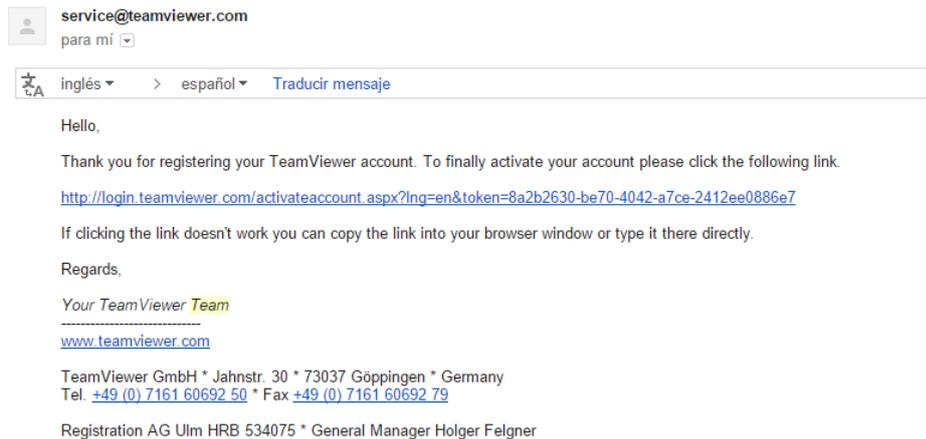


Figura 3.6

Para finalizar y como anotación histórica el algoritmo empleado para generar nuevos GUID ha sido ampliamente criticado. Al principio, la dirección MAC de la tarjeta de red del usuario se usaba como base para varios dígitos del identificador, lo que significa, por ejemplo, que un documento podía ayudar a averiguar qué ordenador lo había creado. Este agujero en la privacidad se utilizó para localizar al creador del gusano Melissa. Después de que esto se descubriera, Microsoft cambió el algoritmo, por lo que ya no se usa la dirección MAC.

3.5 Cifrado simétrico

Las clases de criptografía simétrica administrada se utilizan con una clase de secuencia especial denominada IAttribute que cifra los datos leídos en la secuencia. La clase CryptoStream se inicializa con una clase de secuencia administrada, una clase implementa la interfaz ICryptoTransform (creada a partir de una clase que implementa un algoritmo criptográfico) y una enumeración CryptoStreamMode que describe el tipo de acceso permitido en CryptoStream. La clase CryptoStream puede inicializarse utilizando cualquier clase que se derive de la clase Stream, incluidas FileStream, MemoryStream y NetworkStream. Si se utilizan estas clases, se puede realizar el cifrado simétrico en diversos objetos de secuencia.

```
//Creamos un proceso TCP con un oyente del tipo TCP.  
//Usaremos localhost para especificar nuestro PC y reemplazaremos por la IP  
del oyente.  
TcpClient TCP = newTcpClient("localhost",11000);  
//Creamos una conexión del tipo NetStream para trabajar desde TCP.  
NetworkStreamNetStream = TCP.GetStream();
```

```

//InstanciamosRijndaelManaged y encriptamos el canal.
RijndaelManagedRMCrypto = newRijndaelManaged();

byte[] Key = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
0x11, 0x12, 0x13, 0x14, 0x15, 0x16};
byte[] IV = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

//Create a CryptoStream, pass it the NetworkStream, and encrypt
//it with the Rijndael class.
CryptoStreamCryptStream = newCryptoStream(NetStream,
RMCrypto.CreateEncryptor(Key, IV),
CryptoStreamMode.Write);

//Create a StreamWriter for easy writing to the
//network stream.
StreamWriterSWriter = newStreamWriter(CryptStream);

//Write to the stream.
SWriter.WriteLine("Hello World!");

SWriter.Close();
CryptStream.Close();
NetStream.Close();
TCP.Close();

```

3.5.1 Ejemplo de cifrado AES

El cifrado AES viene de la interfaz AES, y la clase que lo implementa es AesCryptoServiceProvider. La forma de cifrar los bytes es necesario usar las clases CryptoStream y StreamWriter. CryptoStream definirá el modo de AES que son, modo cifrado modo descifrado como se muestra en el siguiente ejemplo:

```

String plainText = "Cíframe";
ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
MemoryStream msEncrypt = new MemoryStream();
CryptoStream CryptStream(msEncrypt, encryptor, CryptoStreamMode.Write));

StreamWriter swEncrypt = new StreamWriter(csEncrypt);

swEncrypt.Write(plainText);

```

La manera de descifrar cambia sutilmente, ya no podemos hacer directamente el cifrado desde un tipo string, necesitamos la array de bytes para hacer la operación inversa:

```
String plainText = "";
ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
MemoryStream msDecrypt = new MemoryStream(cipherText)
CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)

StreamReader srDecrypt = new StreamReader(csDecrypt)
plaintext = srDecrypt.ReadToEnd();
```

3.6 Cifrado Asimétrico

Los algoritmos asimétricos se suelen usar para cifrar pequeñas cantidades de datos como el cifrado de una clave simétrica y un IV. Normalmente, un individuo que realiza cifrado asimétrico utiliza la clave pública que genera otro individuo, .NET Framework proporciona la clase RSACryptoServiceProvider para este fin.

```
byte[] PublicKey =
{214,46,220,83,160,73,40,39,201,155,19,202,3,11,191,178,56,
    74,90,36,248,103,18,144,170,163,145,87,54,61,34,220,222,
    207,137,149,173,14,92,120,206,222,158,28,40,24,30,16,175,
    108,128,35,230,118,40,121,113,125,216,130,11,24,90,48,194,
    240,105,44,76,34,57,249,228,125,80,38,9,136,29,117,207,139,
    168,181,85,137,126,10,126,242,120,247,121,8,100,12,201,171,
    38,226,193,180,190,117,177,87,143,242,213,11,44,180,113,93,
    106,99,179,68,175,211,164,116,64,148,226,254,172,147};

byte[] Exponent = {1,0,1};

//Creamos valores para almacenarlos en las claves simetricas encriptadas.
byte[] EncryptedSymmetricKey;
byte[] EncryptedSymmetricIV;

//Creamos una nueva instancia de la claseRSACryptoServiceProvider.
RSACryptoServiceProvider RSA = newRSACryptoServiceProvider();

//Creamos una nueva instancia de la estructuraRSAParameters.
RSAParametersRSAKeyInfo = newRSAParameters();

//Insertamos los valores de la llave pública a la estructuraRSAKeyInfo.
RSAKeyInfo.Modulus = PublicKey;
RSAKeyInfo.Exponent = Exponent;

//Importamos los parámetros de la llave al algoritmo RSA.
RSA.ImportParameters(RSAKeyInfo);

//Creamos una nueva instancia de la clase RijndaelManaged.
RijndaelManaged RM = newRijndaelManaged();

//Encrioptamos la llave simétrica y el IV.
EncryptedSymmetricKey = RSA.Encrypt(RM.Key, false);
```

```
EncryptedSymmetricIV = RSA.Encrypt(RM.IV, false);
```

3.7 Generación de claves para cifrado y descifrado

La creación y administración de claves es una parte importante del proceso criptográfico. Los algoritmos simétricos requieren la creación de una clave y un IV (Initialization Vector, vector de inicialización). La clave debe mantenerse en secreto y a salvo de quienes no deban descifrar los datos. No es necesario que el vector de inicialización sea secreto, pero debe cambiarse para cada sesión. Los algoritmos asimétricos requieren la creación de una clave pública y una clave privada. La clave pública puede revelarse a cualquiera, mientras que la privada debe conocerla sólo la parte que descifrá los datos cifrados con la clave pública. En esta sección se describe cómo generar y administrar claves para algoritmos simétricos y asimétricos.

3.7.1 Claves simétricas

Las clases de cifrado simétrico que proporciona .NET Framework requieren una clave y un nuevo vector de inicialización (IV) para cifrar y descifrar datos. Siempre que cree una nueva instancia de una de las clases criptográficas simétricas administradas utilizando el constructor predeterminado, se crean automáticamente una clave y un vector de inicialización nuevos. Cualquier persona a la que permita descifrar sus datos debe poseer la misma clave y el mismo IV, y utilizar el mismo algoritmo. Normalmente, debe crearse una nueva clave y vector de inicialización para cada sesión, y ni la clave ni el vector deberían almacenarse para utilizarlos en una sesión posterior.

Para comunicar una clave simétrica y un IV a una parte remota, la clave simétrica normalmente se cifra usando cifrado asimétrico. El envío de la clave a través de una red insegura sin cifrarla no es seguro, porque cualquiera que intercepta la clave y el IV podrán descifrar los datos.

Un esquema criptográfico simple para cifrar y descifrar datos podría especificar los pasos siguientes:

1. Cada parte genera un par de claves pública y privada.
2. Las partes intercambian las claves públicas.
3. Cada parte genera una clave secreta para el cifrado TripleDES, por ejemplo, y cifra la clave recién creada utilizando la clave pública de la otra parte.

4. Cada parte envía los datos a la otra parte y combina la clave secreta de la otra parte con la suya propia, en un orden particular, para crear una nueva clave secreta.
5. Después, las partes inician una conversación utilizando el cifrado simétrico.

En ocasiones tendrá que generar varias claves. En este caso, puede crear una nueva instancia de una clase que implemente un algoritmo simétrico y después crear una nueva clave e IV mediante una llamada a los métodos **GenerateKey** y **GenerateIV**. En el ejemplo de código siguiente se ilustra cómo crear nuevas claves y vectores de inicialización una vez creada una nueva instancia de la clase criptográfica asimétrica.

Ejemplo:

```
TripleDESCryptoServiceProvider TDES = new TripleDESCryptoServiceProvider();  
TDES.GenerateIV();  
TDES.GenerateKey();
```

.NET Framework proporciona las clases `RSACryptoServiceProvider`, `DSACryptoServiceProvider` y `AESCryptoServiceProvider` para el cifrado asimétrico. Estas clases crean un par de claves pública y privada cuando utiliza el constructor predeterminado para crear una nueva instancia. Las claves asimétricas pueden almacenarse para utilizarse en sesiones múltiples o bien generarse para una sesión únicamente. Aunque la clave pública puede ponerse a disposición general, la clave privada debe guardarse bien.

Un par de claves pública y privada se genera siempre que se crea una nueva instancia de una clase de algoritmo asimétrico. Una vez creada una nueva instancia de la clase, la información de la clave puede extraerse mediante uno de estos dos métodos:

- El método `ToXMLString`, que devuelve una representación XML de la información de la clave.
- Método `ExportParameters` que devuelve una estructura `RSAPParameters` que contiene la información de la clave.

Ambos métodos aceptan un valor booleano que indica si hay que devolver sólo la información de la clave pública o si se devuelve también la correspondiente a la clave privada. El valor de una clase `RSACryptoServiceProvider` puede inicializarse con una estructura `RSAPParameters` mediante el método `ImportParameters`.

Las claves privadas asimétricas nunca deben almacenarse literalmente o en texto sin formato en el equipo local. Si debe almacenar una clave privada, utilice un contenedor de claves.

En el ejemplo de código siguiente se crea una nueva instancia de la clase **RSACryptoServiceProvider**, se crea un par de claves pública y privada, y se guarda la información de la clave pública en una estructura **RSAParameters**.

```
//Generamos un par de llaves públicas y privadas.
RSACryptoServiceProvider RSA = newRSACryptoServiceProvider();
//Guardamos la información de la llave pública en una estructura del tipo
RSAParameters.
RSAParametersRSAKeyInfo = RSA.ExportParameters(false);
```

3.8 Ejemplo de cifrado AES

El cifrado AES viene de la interfaz AES, y la clase que lo implementa es AesCryptoServiceProvider. La forma de cifrar los bytes es necesario usar las clases CryptoStream y StreamWriter. CryptoStream definirá el modo de AES que son, modo cifrado modo descifrado como se muestra en el siguiente ejemplo:

```
String plainText = "Cíframe";
ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
MemoryStream msEncrypt = new MemoryStream();
CryptoStream CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write));

StreamWriter swEncrypt = new StreamWriter(csEncrypt);

swEncrypt.Write(plainText);
```

La manera de descifrar cambia sutilmente, ya no podemos hacer directamente el cifrado desde un tipo string, necesitamos la array de bytes para hacer la operación inversa:

```
String plainText = "";
ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
MemoryStream msDecrypt = new MemoryStream(cipherText)
CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)

StreamReader srDecrypt = new StreamReader(csDecrypt)
plaintext = srDecrypt.ReadToEnd();
```

Capítulo 4 Control de Acceso en .NET

Una vez explicado en el capítulo anterior la forma de cifrar el texto podemos introducirnos a la aplicación de todo lo anterior. En este capítulo hablaremos sobre cómo se realiza una autenticación de usuario cliente que venga desde un navegador web o un cliente Android.

Este capítulo está dedicado al Control de Acceso Basado en Roles (RBAC) es una función de seguridad para controlar el acceso de usuarios a tareas que normalmente están restringidas. Mediante la aplicación de atributos de seguridad a procesos y usuarios, RBAC puede dividir las capacidades de acceso a recursos entre varios tipos de roles. La gestión de derechos de procesos se implementa a través de data annotations.

Para hacer una correcta autenticación es importante guardar la información mínima imprescindible en las cookies, que nos servirá para controlar los accesos de sesión y zonas restringidas en nuestra aplicación. La forma de realizar una autenticación y almacenar la información es con la clase `HttpCookie`.

La clase `HttpCookie` tendrá para nosotros el mismo funcionamiento que un diccionario de datos, dado una key contendrá un determinado valor el cual podremos leer y escribir. Además podremos inyectarle el tiempo de expiración de la cookie en el servidor. Hay otra forma de realizar este funcionamiento con la clase `FormsAuthentication` pero no se puede trabajar con ella con la misma flexibilidad que `HttpCookie`, por lo que para aplicaciones que no requieran fuertes restricciones de datos `FormsAuthentication` nos arreglaría todos los problemas, pero para una aplicación con cierta lógica compleja se nos quedaría corta.

4.1 Explicación e introducción al contexto

Imaginemos que un usuario inicia sesión en nuestra aplicación y necesitamos un cierto control para que estos usuarios no puedan acceder a determinados métodos o acciones de nuestro controlador o api, para poder implementar más tarde estas restricciones es importante almacenar la información de nuestro cliente, para ello vamos a realizar 3 acciones.

La primera de ella será configurar el tiempo de inicio de sesión máximo de nuestras cookies de sesión, incluyendo además la redirección a nuestra URL de la vista login o que queramos para obligar al usuario que se autentique. Esto se hará en el fichero `Web.config` de nuestro proyecto y muy importante dentro del contenido de la etiqueta `<system.web>`

La segunda de ellas será redefinir el método `Application_AuthenticateRequest` en nuestro fichero `Global.asax`. Éste método de acción se ejecuta antes de cargar cualquiera solicitud a través de nuestro servidor web y lo utilizaremos para comprobar si hay actualmente una cookie de usuario asociado al usuario que inicia la petición y si es así almacenaremos la información en el servidor en una propiedad llamada `user` en nuestro contexto de la siguiente forma `Context.user`.

La tercera de ellas será redefinir una clase que implemente interfaz `IPrincipal`. Ésta interfaz proporcionada por el framework está pensada para comprobar a posteriori los roles asociados a los usuarios y es obligatorio hacerlo para poder hacer muros de autorización.

`Context.user` solo aceptará tener como parámetro una clase que implemente ésta interfaz.

Ejemplo de autenticación

Paso 1:

```
<system.web>
  <authentication mode="Forms">
    <forms loginUrl="~/Login/Login" timeout="2880" />
  </authentication>
```

Figura 4.1

Paso 2:

```
protected void Application_AuthenticateRequest(Object sender, EventArgs e)
{
    HttpCookie authCookie = Request.Cookies[Constantes.COOKIEES];
    if (authCookie != null)
    {
        try
        {
            UserDataCookie newUser = new UserDataCookie(authCookie.Values["Correo"]);
            Context.User = newUser;
        }
        catch (Exception ex)
        {
        }
    }
}
```

Figura 4.2

Paso 3:

```
public class UserDataCookie : IPrincipal
{
    public IIdentity Identity { get; private set; }

    public UserDataCookie(string email)
    {
        this.Identity = new GenericIdentity(email);
    }

    //Método que comprueba si el usuario identificado contiene el rol requerido
    public bool IsInRole(string role)
    {
        CRUDRolesRepository roles = new CRUDRolesRepository();
        var rol = roles.GetRolByUser(Identity.Name);

        return rol.Nombre_Rol.Equals(role) ? true : false;
    }
}
```

Figura 4.3

Asp.net mvc implementa de una forma elegante la forma de controlar nuestras sesiones en el servidor, pudiendo separar la responsabilidad de cada acción en su clase y métodos correspondientes.

Uno de los problemas que puede ocasionarnos quizás el guardar información de las cookies es que por ejemplo el nombre del usuario sea modificado y al realizar una petición engañemos al framework haciendo que haga la comprobación de autenticación con otro usuario, para solucionar esto hay varias maneras de hacerlo, desde usar JS, CA en las cookies etc. Pero realmente hay una forma muy sencilla de hacerlo que es asociar a un ID de sesión un Guid, es decir, en vez de guardar el correo del usuario en las cookies, generar un Guid en su sesión y asociársela en un campo de la tabla correspondiente a este usuario como un identificador más.

4.2 Autorización

En este apartado vamos a comentar como se puede realizar una autorización de recursos en asp.net mvc. A diferencia de java, C# puede hacer comprobaciones de autorizaciones antes de entrar a los métodos de acción de un controlador, haciendo que sea imposible ejecutar la más mínima línea de código si no el usuario no está autorizado y además de una forma muy elegante, haciendo de forma fácil y estructurada la separación de responsabilidades

Éste capítulo está dedicado al Control de Acceso Basado en Roles (RBAC) es una función de seguridad para controlar el acceso de usuarios a tareas que normalmente están restringidas. Mediante la aplicación de atributos de seguridad a procesos y usuarios, RBAC puede dividir las capacidades de acceso a recursos entre varios tipos de roles. La gestión de derechos de procesos se implementa a través de data annotations.

4.3 La interfaz IAttribute

Las IAttribute son una serie de interfaces que nos dejan customizar e implementar el comportamiento del control a un determinado recurso, ya sea método, atributo de clase, etc. Estas interfaces se pueden usar para hacer restricciones como por ejemplo que la edad de una persona sea mayor de 18, que un campo no puede estar vacío, o contenga un determinado valor.

Para las autorizaciones debemos de usar una en concreto llamada AuthorizeAttribute, ésta interfaz en concreto nos proporciona unos métodos de pre, post, in acción. Esto significa que tendrá un método que se ejecute antes de la llamada, otro durante la

llamada y otro después de la llamada. En mi caso particular he sobrescrito el método `AuthorizeCore`.

El método `AuthorizeCore` recibe como parámetro un objeto de tipo `HttpContext`, este objeto guarda toda la información necesaria entre el usuario y el servidor. Podríamos acceder al debugger de nuestro VS y ver toda la información detallada, pero nosotros nos centraremos en la autorización.

En el siguiente diagrama se muestra la secuencia que seguiría un usuario que intenta acceder a un sitio restringido sin haber logueado siquiera en nuestra aplicación.

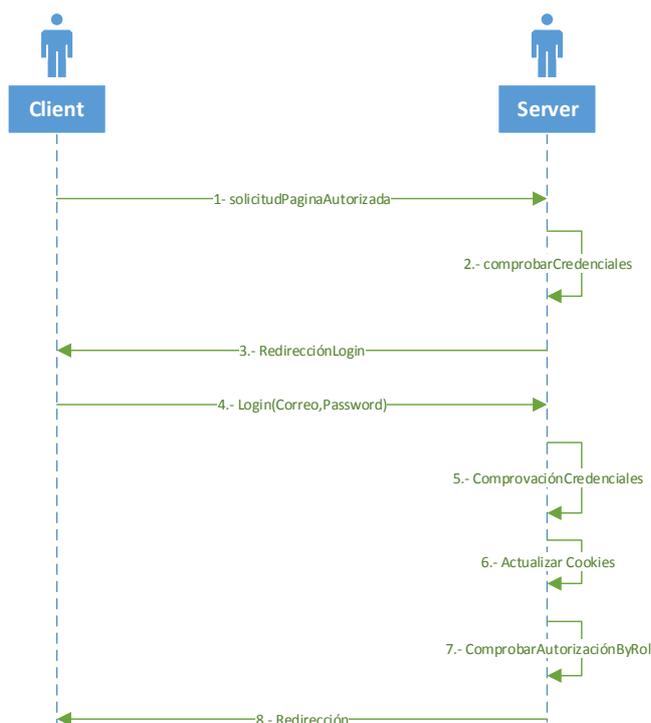


Figura 4.4

En este código de ejemplo la acción que realizamos es relativamente sencilla, consiste en obtener por el constructor aquellos roles que van a poder acceder a un determinado recurso, estos roles serán comparados uno a uno por el método que sobrescribimos en el capítulo anterior en un objeto que implementa la interfaz `IPrincipal`.

```

namespace Itemcoteca.UI.LogicGUI
{
    public class Autorizaciones : AuthorizeAttribute
    {
        IEnumerable<Role> roles;
        string[] rolesAdmitidos;

        public Autorizaciones(params string[] values)
        {
            if(values != null)
            {
                rolesAdmitidos = new string[values.Length];
                for (int i = 0; i < values.Length;i++)
                {
                    rolesAdmitidos[i] = values[i];
                }
            }
        }

        protected override bool AuthorizeCore(HttpContextBase httpContext)
        {
            var result = false;
            if(httpContext.Request.IsAuthenticated)
            {
                CRUDRolesRepository repo = new CRUDRolesRepository();
                Guid g = Guid.NewGuid();
                var guid = g.ToString();
                foreach(string item in rolesAdmitidos)
                {
                    if (httpContext.User.IsInRole(item))
                    {
                        result = true;
                        break;
                    }
                }
            }

            return result;
        }
    }
}

```

Figura 4.5

La forma de poner estas restricciones se hace de la siguiente manera:

```

// GET: /Tienda/
[Autorizaciones("Admin","Usuario")]
public ActionResult Index()
{
    return View("MainArticulos");
}

```

Figura 4.6

Para finalizar es importante destacar que estas validaciones de autorización y autenticación se pueden usar también como etiquetas en nuestra Web Api y en signalR como veremos más adelante.

Capítulo 5 Descripción de la aplicación de DEMO, Caso de uso.

Tras haber hecho una pequeña mención sobre en qué consiste una función Hash y ver varios tipos, vamos a mostrar su uso implementado en nuestro proyecto. La utilidad de estas funciones nos ha servido para autenticar el registro y proteger las contraseñas de los usuarios para que no sean entendibles al ojo humano.

5.1 Validación de Registro.

Antes de comenzar con la explicación de la validación de registro hay que destacar que tabla y campos en la base de datos se han tenido que crear y explicar el porqué. Para el registro de usuario vamos a tener los siguientes campos para su validación:

Atributo	Descripción	Posibles Valores	Tipo
Estado_Cuenta	Es el estado de la cuenta de usuario el cual puede tener tres estados	APOR – “Account Confirmar” ACON – “Account Confirmada” ABLO - “Account Bloqueada”	String
Token	Es el token generado apartir del ID de la tabla usuario, el email con el que se registra el usuario y un número aleatorio generado por el servidor	Hexadecimal obtenido de SHA1	String
Random	El número aleatorio con el que se ha generado el token	Rango: 0 - 100.000 (int)	Int

La validación de registro es una manera de hacer que un usuario malicioso no pueda registrarse en un sitio web con la cuenta email de otra persona y haciéndose pasar por ella. Por ejemplo, si mi nombre es Abel y conozco el correo electrónico de Javier, podría registrarme con su email y bloquearle el acceso a la web con su correo. Para ello haremos uso de una comunicación de intercambio de tokens, a continuación explicaremos los pasos del protocolo:

- 1) El usuario envía el formulario con los campos correctos
- 2) El servidor comprueba que el email no existe en la base de datos
- 3) El servidor inserta los datos de usuario en la tabla Usuarios e inicializa el estado de la cuenta a "APOR"
- 4) El servidor obtiene el ID de usuario de la base de datos, genera un número random y concatena estos dos datos con el email que acaba de registrar el usuario

SHA1(random+id+email) = Token

- 5) Generamos el SHA1 de esta concatenación y guardamos en la base de datos.
- 6) Creamos un email con formato de presentación
- 7) En el email creamos una URL con dirección al controlador que se encargará de completar el registro y además le pasamos como parámetro el token que generamos en el paso 4.
- 8) Cuando el usuario clickea en el link se vuelve a comunicar con el servidor
- 9) El servidor comprueba que efectivamente, ese token fue enviado y está a la espera de completar el registro
- 10) Cambia el estado de la cuenta de "APOR" a "ACON"

5.2 Diagrama de secuencia generación token

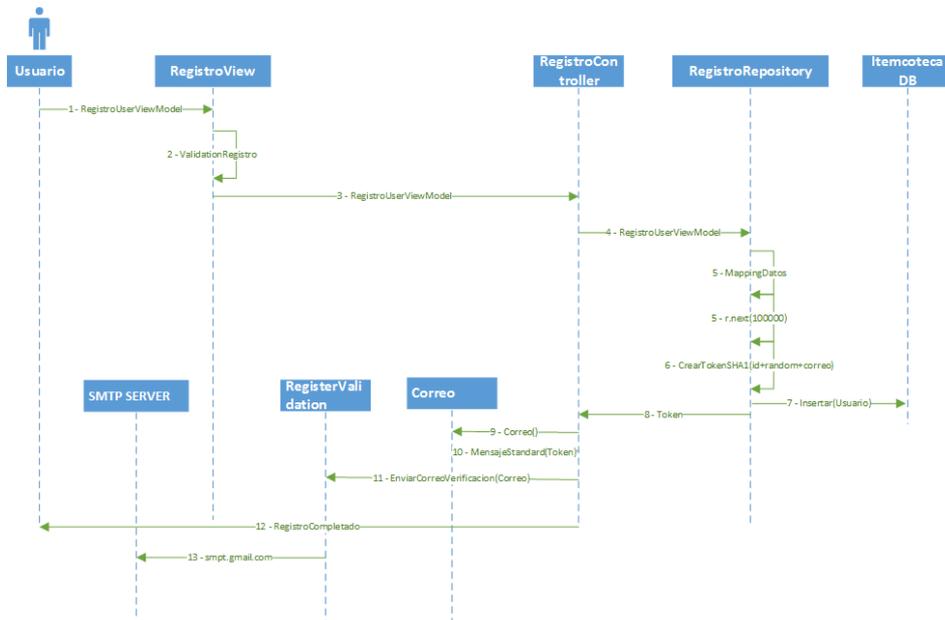


Figura 5.1

En nuestra bandeja de entrada del correo electrónico aparecerá un mensaje con el siguiente formato.



Figura 5.2

5.3 Diagrama de secuencia confirmación token

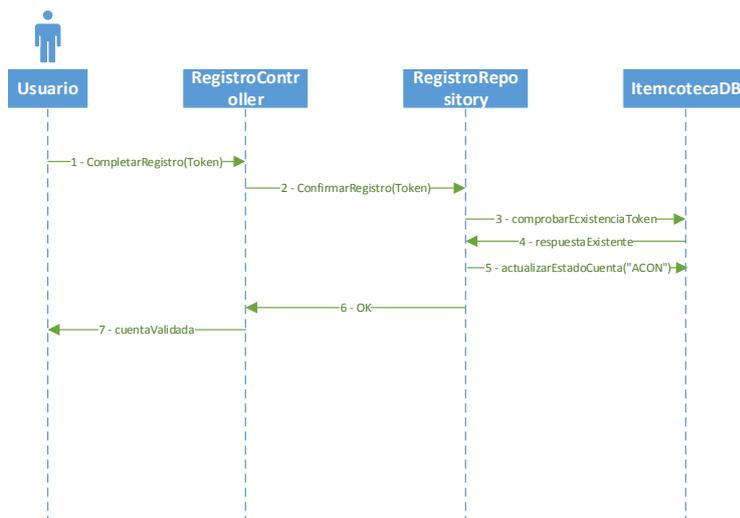


Figura 5.3

5.4 Código de ejemplo de la generación del token

```

public string RegistrarCliente(RegistroUserViewModel data)
{
    Security seguridad = new Security();
    ItemcotecaDBEntities context = new ItemcotecaDBEntities();
    Usuario user = new Usuario();
    user.Correo = data.Correo;

    var existe = context.Usuarios.Where(x => x.Correo.Equals(user.Correo)).SingleOrDefault();
    if (existe == null)
    {
        user.Nombre = data.Nombre;
        user.Apellidos = data.Apellidos;
        user.Movil = data.Movil;
        user.Telefono = data.Telefono;
        user.Password = Security.CifradoSHA1Password(data.Password);
        user.Estado_Cuenta = "APOR";
        user.Fecha_Alta = DateTime.Now;

        Random random = new Random();
        user.Random = random.Next(100000);
        var idMax = 1;
        try
        {
            idMax = context.Usuarios.Max(x => x.ID);
        } catch (Exception e) {}
        user.Token = Security.CifradoSHA1Password(user.Random + idMax + user.Correo + "");

        context.Usuarios.Add(user);
        context.SaveChanges();
    }

    return user.Token;
}
    
```

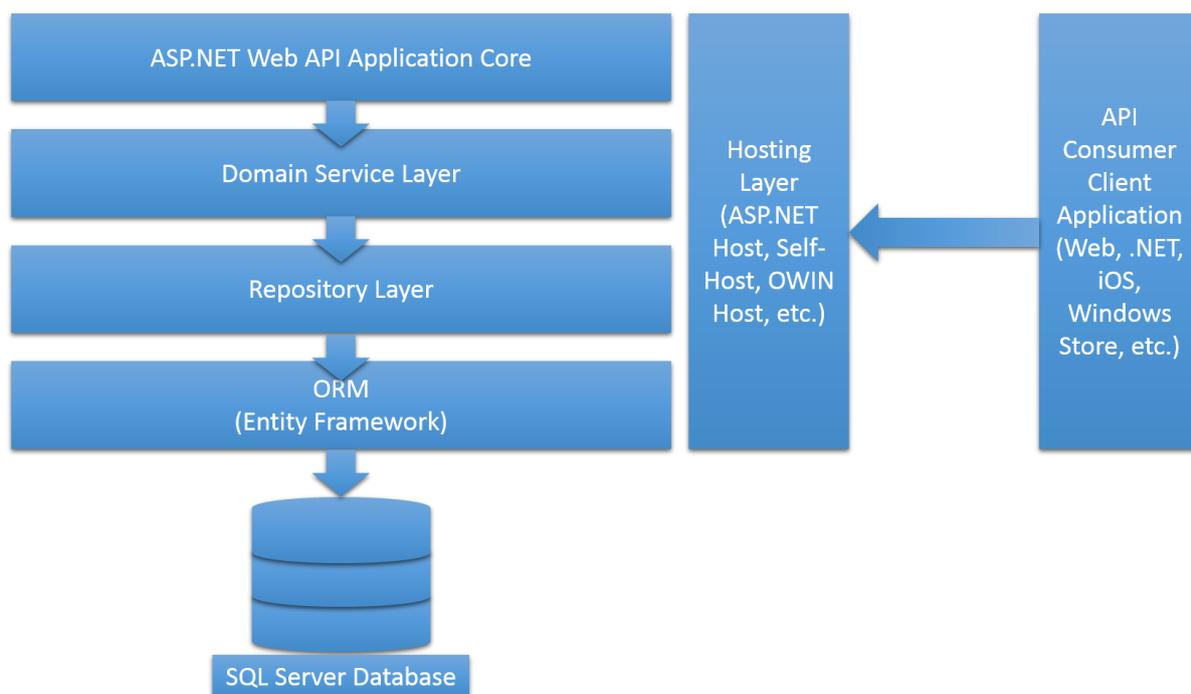
Figura 5.4

Capítulo 6 Servicios REST Seguros en Android y C#

Dado que ya hemos explicado básicamente casi todo respecto a la seguridad que se puede implementar en asp.net mvc nos queda hacer una verdadera demostración de los servicios REST, su implementación y la comunicación con un dispositivo móvil.

Para este apartado haremos una introducción a la web API y como se forman y se envían los mensajes, además de cómo funciona el cliente Android para comunicarse con asp.net mvc y la buena metodología de acción.

Si tuviésemos que definir la estructura de nuestro proyecto en un diagrama lo haríamos de la siguiente manera:



Tenemos nuestra base de datos SQL server con nuestras clases entidades, son aquellas que tienen el mismo nombre con los mismos campos que en la base de datos, siguiendo el Convention over Configuration que Microsoft propone. Justo después tenemos nuestras clases Repository que son aquellas que realmente realizan la lógica del negocio con sus interfaces correspondientes. Para terminar los controladores WebApi, que tienen que ser definidos con las menores líneas de código posible, recordemos que las buenas prácticas del DDD nos dicen que es importante abstraerse de la tecnología que usemos, por lo que no es óptimo que un controlador contenga lógica de negocio.

6.1 Web API

Para utilizar la webApi en un proyecto asp.net mvc hay que tener clara una cosa, Convention over Configuration, la misma filosofía que hemos seguido en los controladores asp.net mvc lo aplicaremos de la misma manera en la WebApi.

Para hacer una comunicación entre una aplicación ASP.NETMVC y otros dispositivos debemos ante todo declarar un acuerdo de interfaz. Vamos a tratar en este ejemplo nuestro registro de usuario y comentar ciertos matices:

Lo primero que debemos de hacer es escribir en la tabla de enrutamiento el formato que deben de seguir las URL para acceder a nuestros controladores WebApi:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{action}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Figura 6.1

Lo segundo es definir nuestra web Api como queramos. En este caso vamos a usar un controlador llamado Usuario y tendrá dos métodos, el registro de usuario y el Login. Vamos a centrarnos en el Registro.

```
public class UsuarioController : ApiController
{
    [HttpPost]
    public string RegistroUsuario(RegistroUserViewModel registro)
    {
        //Registramos al usuario
        IRegistroRepository service = new RegistroRepository();
        string result = "KO";
        string token = service.RegistrarCliente(registro);

        if (token != null && !token.Equals(""))
        {
            try
            {
                RegisterValidation emailValidation = new RegisterValidation();

                //Preparamos el formato del correo
                Correo correo = new Correo(registro.Correo, "Itemcoteca PFG Registro");
                correo.MensajeStandard(token);

                //Enviamos el correo al usuario registrado
                emailValidation.EnviarCorreoVerificacion(correo);
            } catch (Exception e) {
                result = "ERROR";
            }
            result = "OK";
        }

        return result;
    }
}
```

Figura 6.2

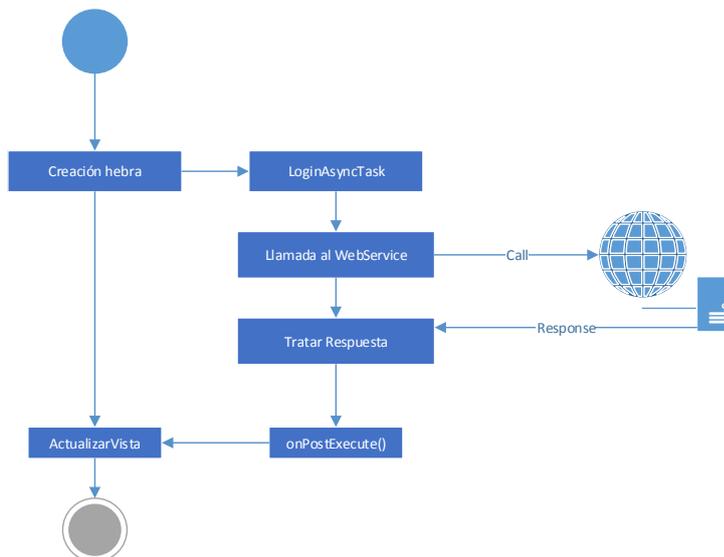
Para hacer una llamada a nuestro controlador deberíamos de usar la URL con el formato definido en nuestra tabla de enrutamiento, además podríamos añadirle las Data Annotation explicadas en los capítulos anteriores. Gracias a la separación de responsabilidad crear controladores WebApi con la misma funcionalidad que los controladores comunes serían tan fácil como copiar y pegar sus implementaciones.

La WebApi nos deserializa de por si las llamadas y nos la prepara para que el cliente sea quien decida como consumirla, sin necesidad de tener que hacer deserializaciones de objetos, el framework es lo suficientemente inteligente para hacer estas transformaciones ya sean JSON,XML,HTML, o texto plano.

6.2 Consumo de un recurso Web Api en Android

Dado que Android nativo usa Java, la forma de hacer la llamada desde Android al Java normal es prácticamente casi idénticas. Lo único que debemos de tener cuidado es con el tipo de conexión, si es http o https y el tipo de acción, POST o GET. En nuestro caso particular haremos los consumos vía POST y Http.

La forma de realizar la llamada es muy sencilla, usaremos la url objetivo y además debemos de hacerlo a través de un tipo de hebra en concreto, este tipo de hebra son las AsyncTask.



Aquí se muestra un código de ejemplo de cómo sería la secuencia de Llamada al Webservice y Tratar Respuesta:

```

public void Registrar(RegistroUserViewModel user) {
    String urlService = "http://www.pfgabeldominguez.rodoxinformatica.net/api/Usuario/RegistroUsuario";
    Cipher cifrado;
    try {
        URL url = new URL(urlService);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    }
  
```

```

connection.setDoOutput(true);
connection.setRequestMethod("POST");
connection.setRequestProperty("Content-Type", "application/json");
connection.setConnectTimeout(5000);

Gson gson = new Gson();
String input = gson.toJson(user, RegistroUserViewModel.class);

OutputStream os = connection.getOutputStream();
os.write(input.getBytes());
os.flush();
BufferedReader responseBuffer = new BufferedReader(new InputStreamReader(
    (connection.getInputStream())));

String output = "";
String contenido = "";
while ((output = responseBuffer.readLine()) != null) {
    contenido = contenido + output;
}
String logueado = gson.fromJson(contenido, String.class);
connection.disconnect();

if(logueado != null){
    resultado = logueado;
    System.out.println("Resultado: "+logueado);
}else{
    resultado = "ERROR";
}

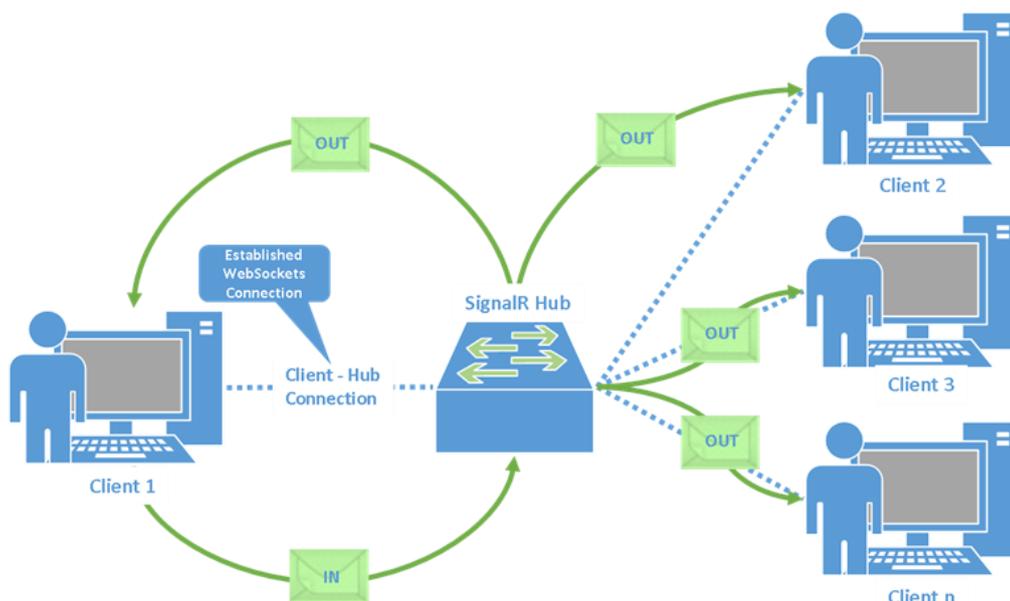
} catch (MalformedURLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
    System.out.println(e.getMessage());
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
    System.out.println(e.getMessage());
}
}
}

```

Capítulo 7 SignalR

SignalR es un framework que trabaja a nivel websocket y sirve para la implementación de recursos web en tiempo real. Esto significa que a diferencia de los recursos REST actuales, los cuales reciben respuesta a cambio de hacer una petición, ahora podemos recibir respuestas sin tener que hacer previamente una petición.

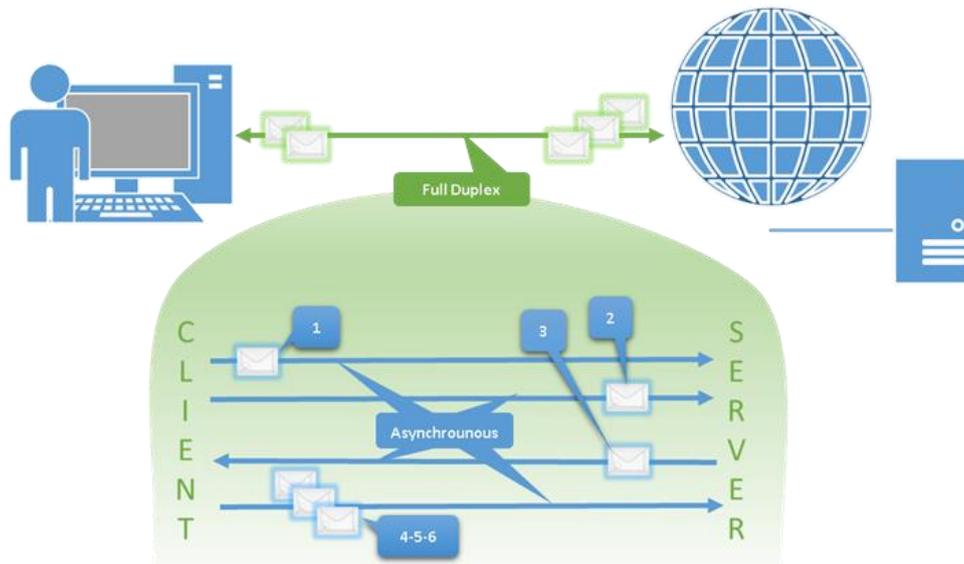
Para entender un poco mejor SignalR vamos a fijarnos en el siguiente diagrama:



Para utilizar SignalR debemos de pensar en el funcionamiento de los controladores WebApi, van a ser controladores de consumo de recursos solo que en vez de realizar peticiones y cerrar la conexión con el servidor una vez realizados, esta vez no la cerraremos, se quedará una conexión persistente que esta suscrita a escuchas.

Los pasos que se realizan son los siguientes:

- 1) Request de conexión con el servidor.
- 2) Una vez que se ha conectado, SignalR establece conexiones de persistencia.
- 3) Crea un canal asíncrono persistente que además es full dúplex.



Imaginemos que queremos crear un chat desde nuestro servidor y los clientes pueden ser usuarios tanto de Android, iOS como un navegador web. Si quisiésemos interactuar con todos ellos a la vez deberíamos de usar una manera de comunicarnos con el servidor, como por ejemplo un REST service, cada vez que alguien envía un mensaje, este lo recibiría el servidor, pero no habría forma posible de comunicarle a todos los clientes que otro usuario ha enviado un mensaje. La solución a este tipo de problemas se soluciona fácilmente usando signalR, haciendo que con esperas activas, todos los clientes conectados a nuestro servidor puedan recibir información sin tener que gastar consumo computacional alguno.

Para poder trabajar con SignalR debemos de instalar su nugget correspondiente, para ello vamos a *Herramientas -> Administrador de paquetes Nugget -> Consola*. Debemos de escribir como contexto el proyecto principal que contiene las vistas la siguiente línea de comando

```
PM> Install-Package Microsoft.AspNet.SignalR
```

Esto nos instalará todo el framework necesario para poder desarrollar una funcionalidad SignalR, entre los paquetes instalados podremos destacar los .js para poder trabajar con un cliente web a través de jquery, los paquetes OWIN y las funcionalidades de la clase Startup.

7.1 Primeros pasos a SignalR

La clase Startup es una clase con un método de ruteo que su funcionalidad es mapear el contenido de nuestro proyecto en la web para que el framework sea capaz de localizar los hubs de conexión que signalR proporciona. Más tarde se verá que aunque los métodos y sus llamadas parezcan imperativas realmente son

declarativas y sin esta clase de configuración jamás sería posible realizar esa comunicación entre cliente y servidor.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(Itemcoteca.UI.Startup))]
namespace Itemcoteca.UI
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }
    }
}
```

Figura 7.1

Éste podría ser perfectamente un ejemplo de la configuración de nuestras clases Hub, lo que es realmente relevante es la etiqueta `assembly` que invoca al constructor de la clase `OwinStartup` indicándole que esta clase debe de ejecutarse antes de cargar el contenido de nuestra web del servidor al cliente para configurar los hubs. El `typeof` indica la ruta donde está la clase, por defecto y recomendación es dejarla en ninguna carpeta o subdomino y que quede en la raíz del proyecto como el fichero `Global.asax`.

7.2 Definición de nuestro Hub

El hub será la clase que contenga todos los métodos que se comunicarán entre el servidor y el cliente, actualmente `signalR` solo manda mensajes de texto, numéricos o booleans, es decir, no podemos mandar sonido o mandar imágenes.

Para crear un Hub necesitamos crearnos una clase controlador que implementará la interfaz Hub, no es necesario crearla en una ruta específica pero la recomendación de los desarrolladores del framework es ponerla en una carpeta llamada `Hub` en la raíz del proyecto y ahí es donde desarrollaremos sus mensajes y su tratamiento como un controlador más.

```
namespace Itemcoteca.UI.Hubs
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message)
        {
            // Call the broadcastMessage method to update clients.
            Clients.All.broadcastMessage(name, message);
        }
    }
}
```

Figura 7.2

Éste ejemplo tiene declarado un Hub que tendrá como método un envío de mensaje global a todos los clientes que se subscriban a él. Realmente podríamos crear tantos hubs como quisiésemos para organizar nuestro proyecto, pero la recomendación es que solo tengamos uno donde englobemos todos los clientes para cerrar más de una conexión a nuestro servidor. Esto no quiere decir que siempre sea así, si por ejemplo nuestro proyecto se encarga de mandar mensajes a través de una aplicación llamada Tienda de Ropa, tendrá su propio hub, pero si tenemos además una funcionalidad de otra SubAplicación como Administradores de tienda, si es recomendable tener otro hub declarado impidiendo que nunca se conecten los dos hub a la vez desde una misma aplicación, ya que serían dos cosas totalmente distintas.

Una vez declarado este hub también podemos configurarlo como los controladores anteriores y poner filtros de autorización

```
public class ChatHub : Hub
{
    [Authorize(Roles = "Admin")]
    [Authorize(Users = "Alice,Bob")]
    public void Send(string name, string message)
    {
        // Método que envía mensaje a todos los clientes
        Clients.All.broadcastMessage(name, message);
    }
}
```

Figura 7.3

Es importante recordar, sí que estas restricciones de autorización se colocan en el nombre de la clase, no se podrá acceder a ningún método pero si se podrá cargar la vista, para que no se cargue la vista debemos de poner el filtro de autorización en la cabecera de nuestro controlador, ya sea su clase o método, que haga acceso a la vista del chat que implementa el Hub.

7.3 Configuración de la parte cliente web en JQuery

Para detallar el funcionamiento haremos una práctica sencilla de chat global, la gente puede conectarse al chat desde cualquier navegador compatible con asp.net mvc y que soporte las librerías jquery 1.6 o más. Con un diálogo le pedimos el Nick al usuario, en éste ejemplo nos conectaremos desde dos pestañas distintas del navegador.

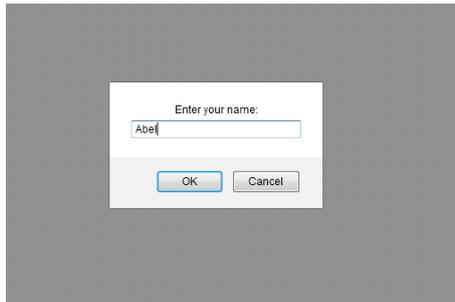


Figura 7.4

Cada vez que escribimos los usuarios reciben el mensaje que cualquier persona escriba, no tenemos ningún control de quien entra o sale, de lo que se dice, o de enviar mensajes privados. Ya depende del desarrollador implementar toda la lógica que vea necesaria, ahora haremos un comentario de cómo funciona realmente las llamadas.

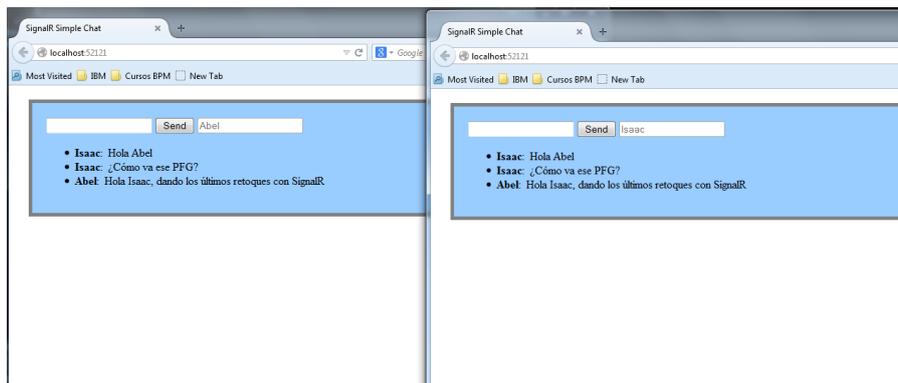


Figura 7.5

Código de ejemplo

```

<script src="/Scripts/jquery-1.6.4.min.js"></script>
<script src="/Scripts/jquery.signalR-2.1.2.min.js"></script>
<script src="/signalr/hubs"></script>

<script type="text/javascript">
$(function () {
    // Declarar un proxy para conectarnos al Hub
    var chat = $.connection.chatHub;

    // Suscripción al mensaje del hub broadcastMessage
    chat.client.broadcastMessage = function (nick, mensaje) {
        // Código Html para obtener el emisor y el receptor
        var nombreEmisor = $('<div />').text(nick).html();
        var mensajeEmisor = $('<div />').text(mensaje).html();
        // Añadir dinámicamente a la página el contenido del mensaje
        $('#discussion').append('<li><strong>' + nombreEmisor + '</strong>&nbsp;&nbsp;&nbsp;' + mensajeEmisor + '</li>');
    };

    // DialogFrame para obtener el nick del usuario
    var nick = prompt('Enter your name: ');
    $('#displayname').val(nick);
    // Inicializar el focus en la ventana de texto.
    $('#message').focus();

    // Iniciamos la conexión al hub
    $.connection.hub.start().done(function () {
        $('#sendmessage').click(function () {
            // Envío de mensaje a nuestro hub cuando pulsamos el botón enviar.
            chat.server.send($('#displayname').val(), $('#message').val());
            // Limpieza del input que contiene el mensaje escrito y le hacemos focus.
            $('#message').val('').focus();
        });
    });
});
</script>

```

Figura 7.6

Cómo podemos ver en el código cliente, cuando se realiza la sentencia `chat.client.broadcastMessage`, este `broadcastMessage` es una subscripción declarativa al método que tenemos en nuestra clase `Hub` en el servidor. Es importante tener cuidado de nombrarlas igual ya que no nos saldrá ningún error de compilación, de la misma manera que cuando creamos el proxy de conexión con la sentencia `$.connection.chathub`, `chathub` hace referencia a la clase controladora que implementa la interfaz `Hub`.

7.4 Los métodos Override de los Hub

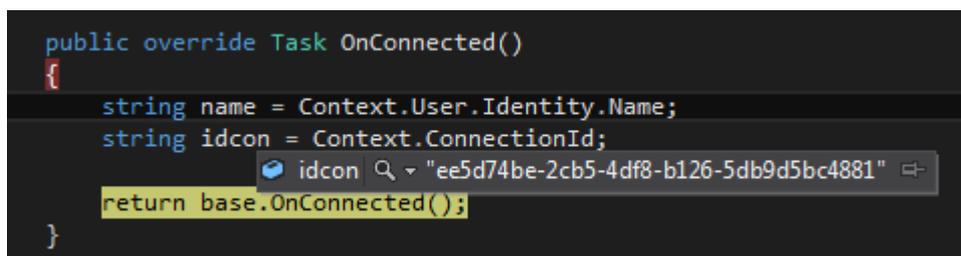
Hay tres métodos que se pueden sobrescribir en SignalR utilizados para la entrada, salida y reconexión de un cliente, estos métodos son respectivamente

```
public override Task OnConnected(){ }

public override Task OnDisconnected(bool stopCalled){}

public override Task OnReconnected(){}
```

El método `OnConnected` es el primero en ejecutarse cuando un cliente comienza su conexión con el `Hub`. Éste método tiene muchas características importantes que puede implementar, si por ejemplo queremos usar un id único para cada usuario o comprobar u autenticación podemos hacerlo de la misma manera con la que autenticábamos en los controladores usando las cookies o usando la clase `Context`.



```
public override Task OnConnected()
{
    string name = Context.User.Identity.Name;
    string idcon = Context.ConnectionId;
    return base.OnConnected();
}
```

Figura 7.7

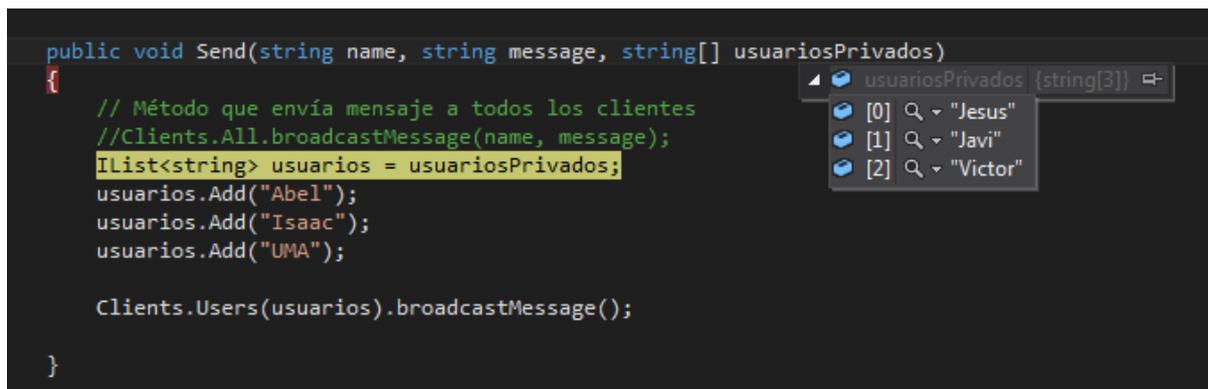
Si por ejemplo queremos un identificador único para cada usuario, el propio framework nos da una `ConnectionId` que hace uso de los GUID para darnos un identificador único para cada usuario conectado al `Hub`.

El método `OnDisconnected` se ejecuta cuando el usuario ha cerrado el navegador o su sesión, y puede reescribirse también para limpiar las cookies, o mandar un mensaje a los demás usuarios advirtiéndoles de que usuario ha abandonado la sala de chat.

El método OnReconnect se ejecuta cuando un usuario ha vuelto a conectarse sin haber pasado previamente por OnDisconnected.

7.5 Mensajes privados por usuarios

Si quisiéramos enviar un mensaje privado a un usuario en concreto o un grupo de usuarios en concreto, SignalR tiene un método llamado Users propio de la clase Clients al que le podemos pasar una array, y todos aquellos usuarios cuyo Context.User.Identity.Name sea igual a uno de esos usuarios recibirá el mensaje:



```
public void Send(string name, string message, string[] usuariosPrivados)
{
    // Método que envía mensaje a todos los clientes
    //Clients.All.broadcastMessage(name, message);
    IList<string> usuarios = usuariosPrivados;
    usuarios.Add("Abel");
    usuarios.Add("Isaac");
    usuarios.Add("UMA");

    Clients.Users(usuarios).broadcastMessage();
}
```

The debugger window shows the array 'usuariosPrivados' with the following elements:

Index	Value
[0]	"Jesus"
[1]	"Javi"
[2]	"Victor"

Figura 7.8

En este ejemplo recibimos por parámetro del mensaje, el Nick de la persona que lo envía, el mensaje y a los usuarios a los que se lo queremos enviar, además de 3 usuarios más.

7.6 Mensaje privado a grupos

El mensaje privado por grupos es otra forma de enviar un mensaje a un cierto número por usuarios, solo que en esta ocasión los receptores del mensaje serán los usuarios suscritos a un grupo al que se le envía el mensaje.

Si por ejemplo tenemos un canal de chat privado llamado "LCC" y enviamos el mensaje al grupo LCC, solo lo recibirán aquellas personas que pertenezcan a ese grupo.

La forma de suscribirse a un grupo en concreto se hace de la siguiente forma:

```

public void subscribirseGrupo(string nick, string grupo)
{
    Groups.Add(nick, grupo);
}

public void enviarMensajeGrupo(string nick, string mensaje, string grupo)
{
    Clients.Group(grupo).mensajePrivadoGrupo(nick, mensaje);
}

```

Figura 7.9

Group.Add() recibe dos parámetros, el primero de ellos es una id por parte del usuario, podemos usar el Guid que nos proporciona el framework o podemos hacer uso del nick del usuario en el caso de que hayamos verificado que nunca se repetirá y lo mismo ocurre con el segundo parámetro que será el nombre del grupo o el identificador asociado a él.

7.7 Como usar Android-SignalR-client

Para utilizar el cliente SignalR en Android debemos de utilizar una librería cliente que ya tiene implementada las conexiones para realizar las peticiones en el servidor. La forma de utilizar esta librería es muy similar a la usada para el cliente web y puede resultarnos bastante familiar.

Para conectarnos al servidor debemos primero configurar un objeto de tipo HubProxy, y configurar la conexión:

```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    String host = "http://www.pfgabeldominguez.rodoxinformatica.net/";

    connection = new HubConnection( host );
    hub = connection.createHubProxy( "ChatHub" );
}

```

Figura 7.10

Una vez configurada los parámetros debemos de realizar la conexión a través de las siguientes sentencias.

```

private void conectar(){
    SignalRFuture<Void> awaitConnection = connection.start();
    for(int i = 0; i< 3 ; i ++){
        try {
            awaitConnection.get();
            escucharGrupos();
            break;
        } catch (InterruptedException e) {
            System.out.println("Estamos DESconectados . . .");
        } catch (ExecutionException e) {
            System.out.println("Estamos DESconectados . . .");
        }
    }
    System.out.println("Numero de intentos: "+i);
}
}

```

Figura 7.11

7.7.1 Cómo realizar las llamadas:

```

public void enviarSolicitudesNotificacion(String emisor, String mensaje){
    hub.invoke("Send", emisor,mensaje);
}

public void sincronizarConexionID(){
    hub.invoke("Conectar", id);
}

public void SolicitarChatPrivado(String titulo, String comentario) {
    // TODO Auto-generated method stub
    hub.invoke("SolicitarChatPrivado", id,titulo,comentario);
}

public void InicializarChatSeguro(String idUsuario) {
    // TODO Auto-generated method stub
    hub.invoke("InicializarChatSeguro", idUsuario);
}

public void EnviarMensajePrivado(String idChat ,String nombreEmisor ,String mensaje){
    hub.invoke("MensajeChatSeguro2", idChat,nombreEmisor,mensaje);
}

```

Figura 7.12

7.7.2 Cómo tratar las respuestas del servidor:

Para tratar las repuestas del servidor debemos de declararnos las interfaces SubscriptionHandler, hay desde 1 hasta 5 elementos por respuesta, pero es raro que usemos más de 5, podemos customizarlas, el código es libre.

```

private SubscriptionHandler1 handlerConectar;
private SubscriptionHandler2 handlerChat, handlerSolicitarChatPrivado, handlerMensajeCifrado;
private SubscriptionHandler3 handlerNuevaSolicitudIncidencia, handlerRecibirClaves;

```

Figura 7.13

Una vez declarado los handlers debemos implementar el funcionamiento de los handlers, o lo que es lo mismo, como vamos a tratar las respuestas del servidor:

```

handlerMensajeCifrado = new SubscriptionHandler2<String, String>() {

    @Override
    public void run(String emisor, String mensaje) {
        // TODO Auto-generated method stub
        Message message = new Message();
        message.what = 2;

        System.out.println("Recibido un mensaje privado(): "+ emisor + " - "+mensaje);
        SingletonPeticionActiva peticion = SingletonPeticionActiva.getInstance();
        mensaje = Recursos.cifradoAes.descifrarMensaje(mensaje);

        ChatIncidenciasActivity._listMensajes.add(new Mensaje(emisor, mensaje));
        ChatIncidenciasActivity.vistaHandler.sendMessage(message);
    }
};

```

Figura 7.14

Para finalizar nos queda solamente dar de alta nuestros handlers a las llamadas del servidor, el método on recibe 3 parámetros,

- 1) El nombre exacto de las respuestas del servidor
- 2) El handler que tratará esas llamadas
- 3) Los tipos de datos que vamos a recibir (el 3º parámetro es de tamaño N)

```

hub.on("conexionRealizada", handlerConnectar, String.class);
hub.on("mensajeBroadCast", handlerChat, String.class, String.class);
hub.on("RespuestaSolicitarChatPrivado", handlerSolicitarChatPrivado, String.class, String.class);

hub.on("NuevaSolicitudIncidencia", handlerNuevaSolicitudIncidencia, String.class, String.class, String.class);
hub.on("RecibirClaves", handlerRecibirClaves, String.class, String.class, String.class);
hub.on("mensajeCifrado", handlerMensajeCifrado, String.class, String.class);

```

Figura 7.15

A continuación mostraremos como es el protocolo de creación simple de un canal seguro con cifrado simétrico y algoritmo AES/CBC.

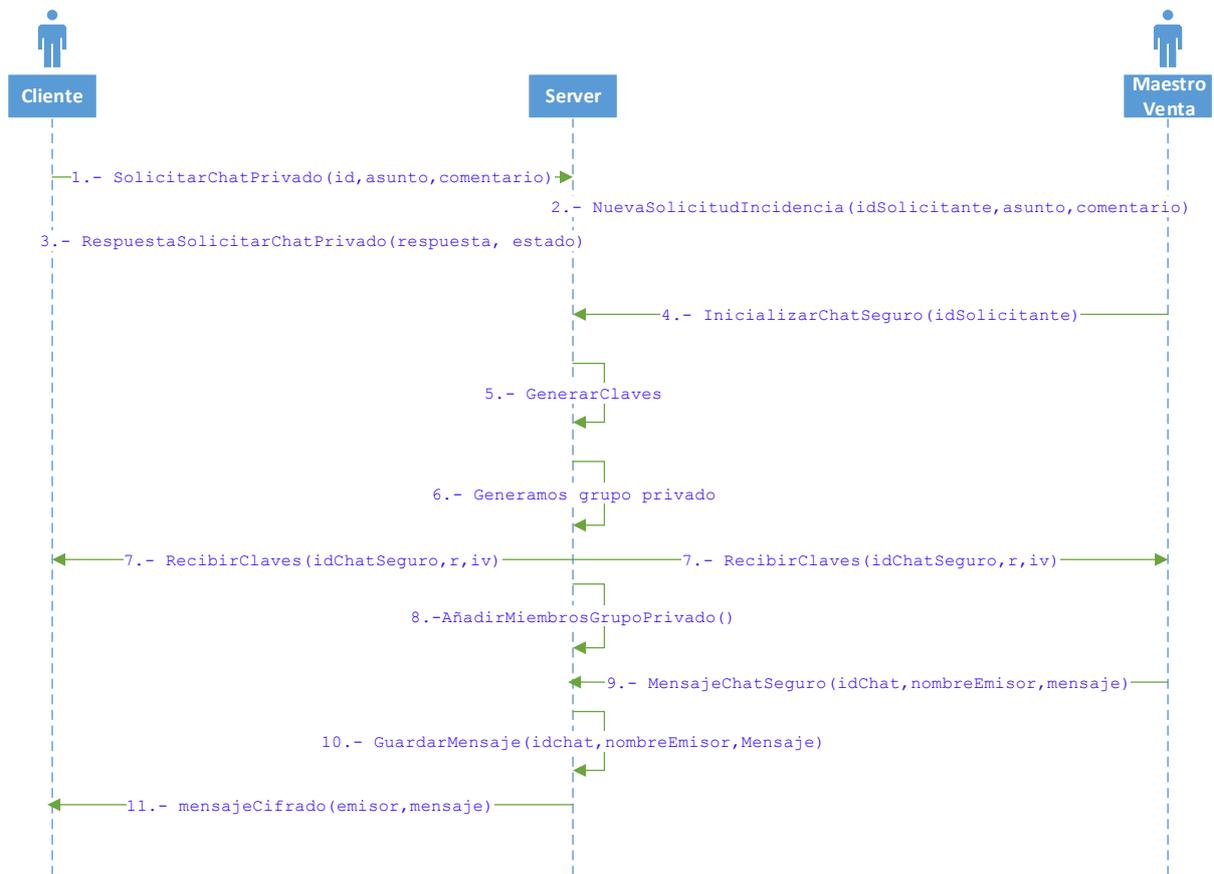


Figura 7.16

7.7.3 Cifrado de los mensajes en el chat seguro

Para cifrar los mensajes al obtener las claves hemos de implementar los métodos de cifrar y descifrar. Es importante destacar que SignalR no acepta arrays de byte por lo que debemos de pasar el mensaje en formato de String y codificarlo en Base64. Si esto no lo hacemos así cuando recibamos el mensaje desde un cliente a otro no podremos descifrarlos y obtendremos un error de ejecución.

```

public String cifrarMensaje(String mensaje)
{
    String result = "";
    try{
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, ivParameterSpec);
        byte[] cifrado = cipher.doFinal(mensaje.getBytes("UTF-8"));

        result = Base64.encodeToString(cifrado, Base64.DEFAULT);
    }catch(Exception e){

    }
    return result;
}

public String descifrarMensaje(String mensajeCifrado){
    String result = null;
    try{
        Cipher cipher2 = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher2.init(Cipher.DECRYPT_MODE, secretKeySpec, ivParameterSpec);

        byte[] data = Base64.decode(mensajeCifrado, Base64.DEFAULT);
        byte[] descifrado= cipher2.doFinal(data);
        result = new String(descifrado,"UTF-8");

    }catch(Exception e){
        System.err.println(e.getMessage());
    }
    return result;
}

```

Figura 7.17

Conclusiones

Tras haber estudiado durante 4 años java en la facultad y haber aplicado estos conocimientos en IBM, mi actual empresa, he de decir que sin duda es un buen lenguaje para desarrollo de aplicaciones web, pero también un lenguaje con muchos problemas a la hora de desarrollar Enterprise Applications debido a su poca escalabilidad y separación de responsabilidades. Aquel que diga que C# es igual que java o parecido es que realmente no ha sabido utilizar la metodología y herramientas de C#. ASP.NET MVC está pensado por ingenieros y para ingenieros, quizás la curva de aprendizaje entre lenguajes sea bastante distinta y opuesta, pero lo que si es cierto es que en C# sabiendo realmente como hacer un buen trabajo puede ahorrarte mucho tiempo en todos los aspectos.

Según iba avanzando en el proyecto he ido también aprendiendo las peculiaridades del lenguaje, las buenas metodologías, leyendo libros y viendo videos de los promotores de ASP.NET MVC y me gustaría invitar a que todo el mundo que realmente le apasione el mundo web pruebe a realizar un pequeño proyecto siguiendo la metodología propia del lenguaje y comparándola con todas las arquitecturas y framework que son demandadas actualmente en el mercado como por ejemplo Python (django), J2EE y PHP.

El EntityFramework es sin duda de las herramientas más bonitas que existe en este lenguaje y la forma de acceder a la persistencia de datos con expresiones lambda para hacer consultas es sin duda difíciles de comprender al principio, pero te dan una soltura y agilidad en la que java está a años luz de alcanzar.

Actualmente el código de los Nuggets en ASP.NET MVC es libre por lo que cualquiera puede inspeccionar su funcionamiento y no hay excusas para decir que Microsoft es cerrado al software.

Ha sido realmente una experiencia muy positiva y espero en un futuro que pueda seguir desarrollando aplicaciones web con este lenguaje, ya que las empresas por Europa cada vez solicitan más personas con conocimientos en esta arquitectura siguiendo sus buenas prácticas

Anexo

Requisitos funcionales para las aplicaciones de ejemplo desarrolladas.

Tabla de prioridades	
Prioridad Muy Baja	1
Prioridad Baja	2
Prioridad Media	3
Prioridad Alta	4
Prioridad Muy Alta	5

ID Requisito: REQ_U01

El **usuario** podrá registrarse en la web. Para ello deberá de introducir unos datos de acceso que servirá al sistema para poder controlar sus privilegios y accesos con restricciones.

Prioridad	3
------------------	---

CASOS DE USO ASOCIADOS	
-------------------------------	--

ID Requisito: REQ_U02

El **usuario registrado** podrá loguearse y desloguearse en la web para realizar sus compras. Para ello el usuario ha de haber realizado el paso previo de registro y activación de cuenta.

Prioridad	3
------------------	---

CASOS DE USO ASOCIADOS	
-------------------------------	--

CASOS DE USO ASOCIADOS	
-------------------------------	--

ID Requisito: REQ_U07

El **usuario** podrá abrir un ticket para resolver una incidencia respecto a un problema con alguna compra, artículo en concreto o su cuenta. Esto hará que un Maestro de venta pueda atender su petición y resolverla.

Prioridad	5
------------------	---

CASOS DE USO ASOCIADOS	
-------------------------------	--

ID Requisito: REQ_M02	
El maestro de venta podrá dar atender peticiones de los clientes, esto requerirá que puedan leer las solicitudes y ponerse en contacto con la persona que inició la solicitud. Para ello el maestro de venta tendrá a su disposición la posibilidad de acceder a datos del cliente respecto su cuenta y pedidos para poder ayudar con las incidencias.	
Prioridad	5
CASOS DE USO ASOCIADOS	

Definición de los Casos de Uso

Caso de Uso UC_A01	Registro de usuario
Versión y cambios	1.0
Descripción	Registrar el usuario en la base de datos para poder realizar compras por sí mismo y a su vez el administrador poder tener más información del sistema.
Actores	Usuario, Itemcoteca
Prioridad	3
Pre-condiciones	El usuario debe de introducir los datos correctamente El usuario debe de utilizar el cliente aportado por Itemcoteca.
Post-Condicion	Quedar registrado el usuario en la base de datos y tener constancia de ello
Escenario principal	<ol style="list-style-type: none"> 1 . El usuario accede a la zona de registro 2. El usuario introduce los datos en los campos correspondientes 3. Itemtoceta valida los datos en el cliente web 4. El usuario pulsa el botón registrar para empezar el proceso 5. El sistema recibe los datos en el servidor para procesarlos. 6. Itemtoceta web comprueba que no existen usuarios registrados con el mismo nombre y email. 7. Itemcoteca web inserta en la base de datos la información recibida por el usuario 8. Itemtoceta web envía un email al correo del registro para poder confirmar la solicitud 9. Itemcoteca web envía un mensaje de texto al navegador cliente que inició el registro para advertirle de que revise su correo electrónico
Escenarios alternativos	<p>3 b – Los datos no son validados correctamente por longitud.</p> <p>6 b – Existe un usuario con ese nombre u email.</p>

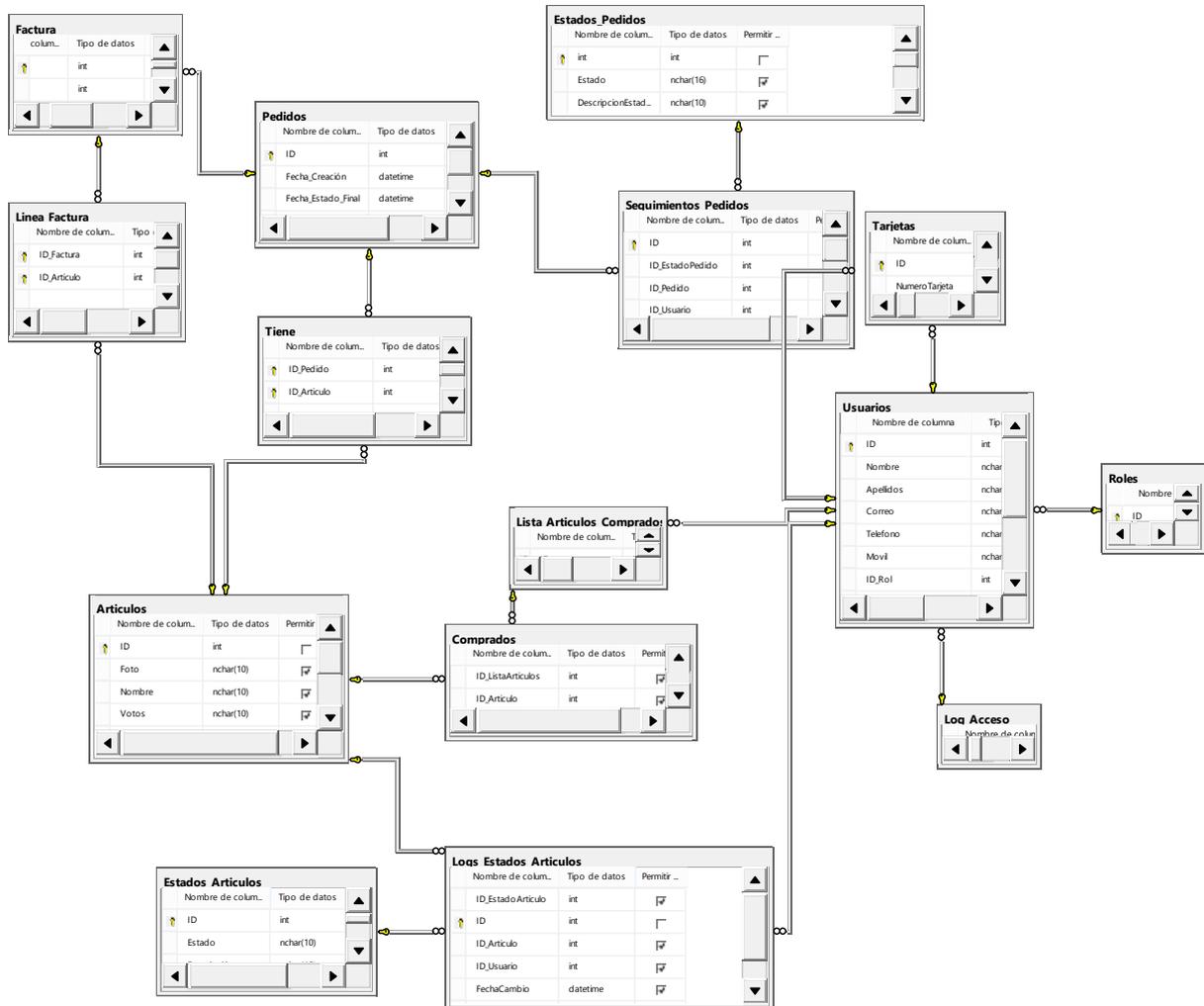
	7 b – Se le envía un mensaje por pantalla al usuario indicando de que su email o correo están registrados y validados.
Clases de análisis	
Entities	Usuario, Rol
Controllers	RegistroController
Views	RegistroView

Caso de Uso UC_A02	Login de usuario
Versión y cambios	1.0
Descripción	El usuario que esté registrado podrá iniciar sesión a la web, esta acción es requerida para todas las demás acciones del servidor
Actores	Usuario, Itemcoteca
Prioridad	3
Pre-condiciones	El usuario debe de introducir su email/usuario y contraseña
Post-Condicion	El usuario podrá acceder a sus sitios personales Queda registrado en el servidor la hora de login para llevar un control de usuarios.
Escenario principal	<ol style="list-style-type: none"> 1. El usuario accede a la zona de login. 2. El usuario introduce los datos en los campos correspondientes 3. El usuario pulsa el botón para comenzar el proceso de login 4. Itemcoteca web recoge los datos 5. Itemcoteca web comprueba que los datos son correctos 6. Itemcoteca web devuelve un mensaje al usuario indicando que los datos son correctos.
Escenarios alternativos	<ol style="list-style-type: none"> 5b. Los datos son incorrectos. 6b. Se devuelve un mensaje de error indicando que los datos introducidos son incorrectos.
Clases de análisis	
Entities	Usuario, Rol, Log_Usuario
Controllers	LoginController
Views	LoginView

Caso de Uso UC_A03	Logout de usuario
Versión y cambios	1.0
Descripción	El usuario que esta logueado podrá cerrar su sesión en el momento que desee.
Actores	Usuario, Itemcoteca
Prioridad	3
Pre-condiciones	El usuario debe de estar logueado
Post-Condiciones	El usuario será deslogueado del servidor, cerrando así su sesión Queda registrado en el servidor la hora de login para llevar un control de usuarios.
Escenario principal	<ol style="list-style-type: none"> 1. El usuario pulsa el botón de logout 2. Itemcotecaciera la sesión del usuario 3. Itemcoteca almacena la información en la tabla Log_Usuario 4. Itemcoteca devuelve al usuario en zonas no restringidas.
Escenarios alternativos	---
Clases de análisis	
Entities	Usuario, Log_Usuario
Controllers	LoginController
Views	---

Caso de Uso UC_A03	Incidencias
Versión y cambios	1.0
Descripción	El usuario podrá iniciar una solicitud de asistencia a un administrador para poder resolver un problema sin necesidad de llamar por teléfono o usar programas terceros
Actores	Usuario, Itemcoteca-web,itemcoteca-app,MaestroVenta
Prioridad	3
Pre-condiciones	El usuario debe de estar logueado
Post-Condicion	Se guardará toda la conversación cifrada en el servidor
Escenario principal	<ol style="list-style-type: none"> 1. El usuario pulsa el botón de de incidencias 2. El usuario escribe su incidencia indicando el título y un breve comentario 3. El usuario envía la petición al servidor 4. El servidor desde SignalR comprueba si hay Maestros de ventas conectados 5. El servidor envía la notificación a un Maestro de Ventas aleatorio que actualmente esté conectado 6. Al usuario se le envía una nota diciéndole que espere su turno 7. El Maestro de ventas recibirá la notificación y podrá acceder a leerla 8. El Maestro de ventas aceptará tratar la solicitud 9. El servidor recibirá la confirmación de la solicitud y le enviará al usuario que envió la notificación al chat para poder hablar con la persona asignada a ello, además generará 3 claves para dárselas a los usuarios implicados. 10. El dispositivo móvil del usuario iniciador del escenario cargará el chat.
Escenarios alternativos	<p>No hay ningún Maestro de ventas conectado</p> <p>El servidor notifica al usuario que no hay Maestros de ventas conectados.</p>
Clases de análisis	
Entities	Usuario, HistorialChat
Controllers	ChatHub
Views	---

Diagrama de Tablas



Instalación y configuración de los productos.

Antes de adentrarnos en lo que serían los detalles técnicos de la aplicación y la seguridad en .net vamos a centrarnos en hacer un recorrido rápido de las herramientas que vamos a necesitar.

Los productos que se han usado para éste proyecto han sido:

- SQL Server 2012 R2 SP 2
- SQL Management Studio 2012
- Visual Studio 2013 SP 3

Todos ellos han sido prestados por licencias estudiantiles de la UMA en la página www.dreamspark.net.

Una vez tengamos todos estos componentes estaremos preparados para poder continuar con la explicación.

SQL Managment Studio.

SQL Managment Studio es la herramienta gráfica que usaremos para definir nuestros diagramas de tablas. Aunque podamos hacer esto también de forma gráfica en Visual Studio es interesante explicar qué ventajas tiene este tipo de herramientas.

Imaginemos que trabajamos en una gran corporación y necesitamos varios entornos para nuestro desarrollo software. Podríamos definir tres de ellos tales como Desarrollo, Preproducción y Producción. Con ésta herramienta podríamos conectarnos a las distintas bases de datos con unos perfiles de configuración.

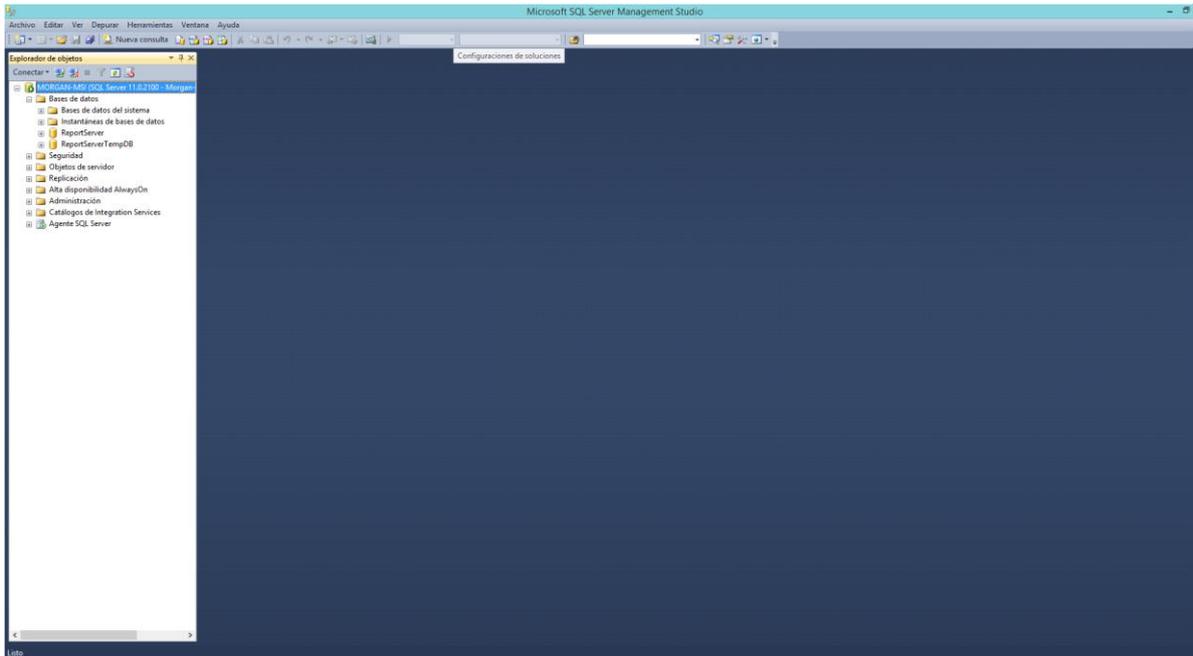


Figura 1.1

Podemos Observar en la figura que hay una carpeta dedicada a “Seguridad”. Esta seguridad está integrada en nuestra base de datos y se abstrae totalmente de la seguridad que implementaremos más adelante en nuestra aplicación web.

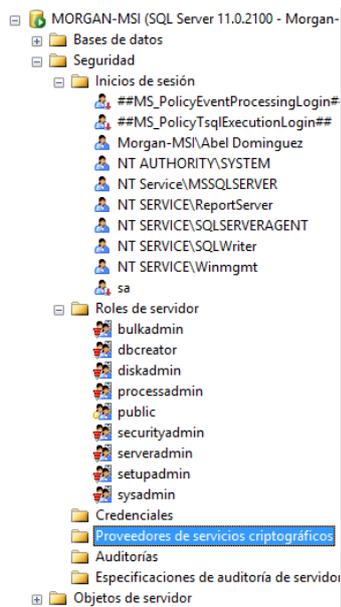


Figura 1.2

Éstos son los usuarios que nos crea por defecto y que para desarrollar en local realmente no necesitaremos cambiarlos, pero si en una gran corporación se necesita un control de quienes acceden a la información de determinados entornos requerirá una auditoría previa para definir bien los roles y privilegios de usuarios.

Creación de nuestra base de datos.

Vamos a crear nuestra base de datos utilizando diagramas de tablas. Estos diagramas podremos sincronizarlos después con nuestro entorno visual studio de forma sincronizada, haciendo que cualquier cambio que hagamos tanto en SQL Management Studio como en Visual Studio afecten de la misma manera a nuestro contenido.

Es importante destacar que trabajar directamente con SQL de forma local como localhost es bastante costoso, es decir, SQL es un gestor de base de datos muy potente pero que a su vez consume bastantes recursos. Si quisiéramos trabajar en un host de desarrollo que tenga hospedado nuestra base de datos y sitio web no tendríamos problemas en hacer un despliegue, pero para trabajar de forma local y con pocos datos hay otros métodos más eficaces. Para ello lo que usaremos será el localdb v11.0

LocalDB es una versión expres de SQL server, esto significa que no tiene el mismo rendimiento ni funcionalidad que esperaríamos del SQL Enterprise pero para trabajar en local es perfecto. La versión 2012 de SQL ya trae integrada esta versión express y no nos hará falta una instalación adicional como es en el caso de las otras versiones Enterprise que no lo traen ya de por sí.

Para poder conectarnos a una localdb es tan sencillo como agregar una nueva conexión a servidor, Click en Archivo -> Conectar con Explorador de Objetos e introducir el nombre del servidor (localdb)\v11.0



Figura 1.3

Si todo ha ido correctamente nos deberá de aparecer la siguiente lista de bases de datos locales: master, model, msdb, tempdb.

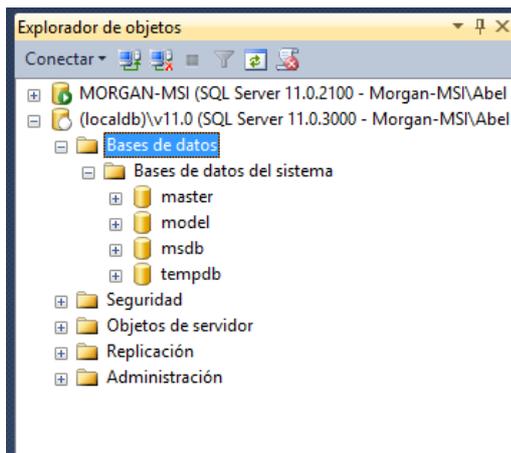


Figura 1.4

Para crear nuestra base de datos solo debemos de hacer click derecho sobre el nuevo nodo que nos ha aparecido (Base de datos) crear nueva base de datos. En nuestro caso particular la hemos llamado ItemcotecaDB. Es importante elegir bien el nombre, ya que luego deberemos de usarla para sincronizarla con VS.

Ahora podremos comenzar con nuestra creación de diagrama de tablas, seleccionamos la carpeta Diagrama de Base de datos que corresponde a la base de datos recién creada, al realizar esta acción nos aparecerá un Draw para poder dibujar el diagrama a nuestro gusto.

Visual Studio

Para sincronizar nuestra base de datos con VS debemos de irnos a la pestaña de explorador de servidores y agregar una nueva conexión. En la ventana de dialogo que nos aparecerá tendremos que introducir el nombre del servidor. Escribiremos el nombre del servidor (localdb)\v11.0 y en seleccionar el nombre de la base de datos debería de aparecernos la base de datos que acabamos de crear.

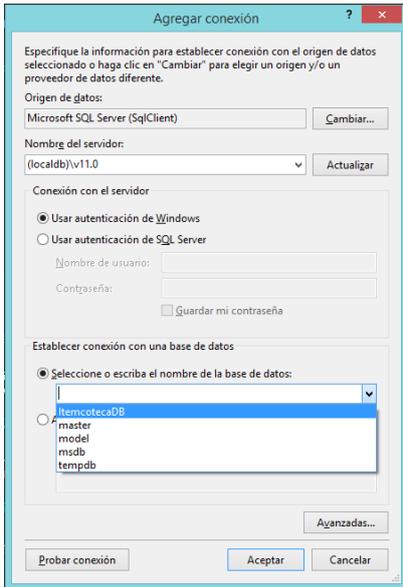


Figura 1.5

Llegados a éste punto, tenemos todo lo necesario para poder trabajar con datos almacenados, podemos hacer consultas de pruebas tanto desde un sitio como desde otro, incluso podemos crear nuestro diagrama de tablas desde el propio visual studio que explicaremos en los siguientes capítulos.

Distinción de los proyectos base en ASP.MVC en Visual Studio

En este apartado explicaremos cómo desarrollar con .Net una aplicación mvc en Visual Studio

Lo primero que vamos a definir es el tipo y nombre de nuestro proyecto. Para ello elegiremos crear un nuevo proyecto en C# tipo web y nos aparecerán las siguientes opciones.

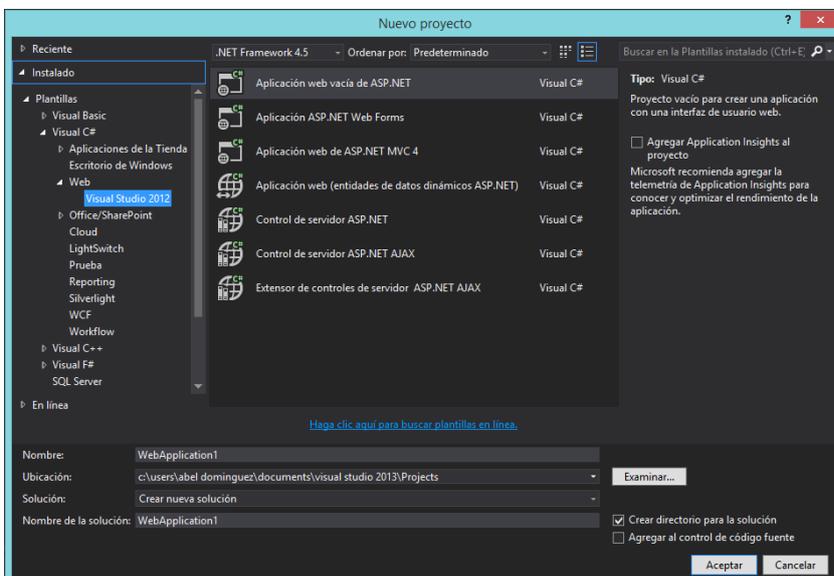


Figura 1.1

La plantilla de Web Forms es una forma reducida de la potencia de mvc, sería algo así como el desarrollo clásico de webs, pero no para un proyecto profesional de gran escala. Trae su propio sistema de seguridad integrado básico, como autorización y autenticación, pero éstos métodos solo nos servirán para trabajar desde la plantilla.

Asp.net MVC será el tipo de proyecto que nosotros debemos de elegir a la hora de trabajar para realizar proyectos empresariales.

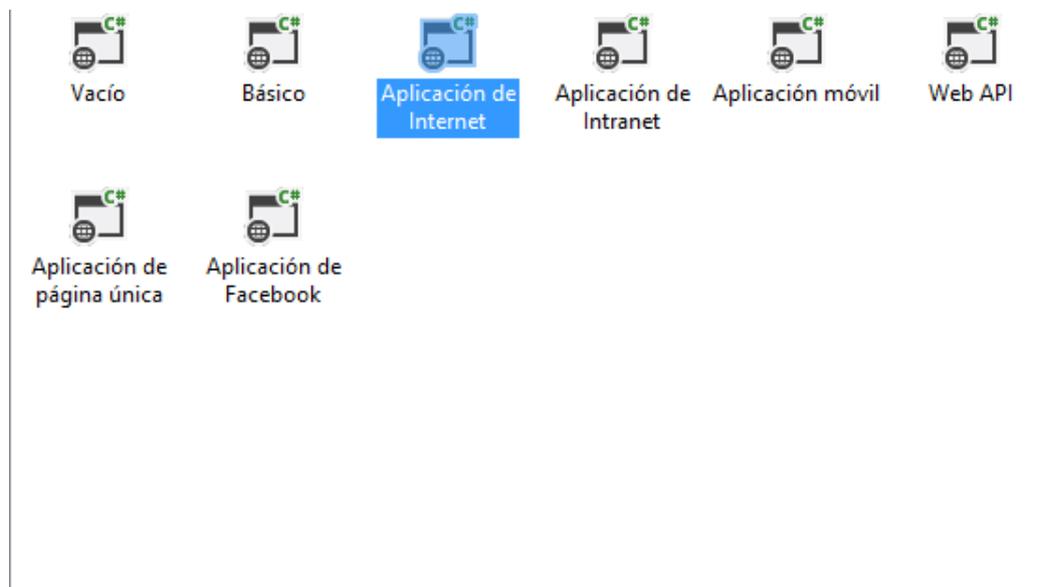


Figura 1.2

De aquí cabe destacar los siguientes:

Aplicación de internet es una plantilla ya definida como web forms pero con estructura mvc 4, con su propio sistema de autorización y autenticación integrado.

Web API Es el proyecto utilizado para definir servicios webs. Éste tipo de proyecto lo usaremos más adelante para definir nuestros servicios y comunicarnos con otros sistemas externos.

Dado que éste proyecto busca utilizar las APIs de seguridad y la metodología de C# usaremos el tipo de proyecto Básico para nuestro entorno.

Una vez creado el proyecto básico vamos hablar un poco respecto a la configuración para separarlo por capas y hacer que la implantación de la seguridad de la aplicación se pueda llevar acabo de forma correcta.

Patrones Inyección de dependencia (DI) y Repository.

El patrón de inyección de dependencia y repository son dos patrones que van unidos de la mano, no se puede implementar el patrón de dependencia sin haber implementado anteriormente el Repository, y no tiene sentido implementar el patrón Repository si posteriormente no se implementa el Inyector de dependencia.

El patrón repository consiste en crear una capa de abstracción entre nuestro modelo del dominio y los controladores que accederán a estas funcionalidades, abstrayéndonos así de la tecnología que haga uso de esta lógica de negocio.

Un ejemplo sencillo de cómo se declararía la interfaz

```
public interface ILoginRepository
{
    string Login(LoginViewModel login);
    string RecuperarContraseña(LoginViewModel login);
    string RecuperarCodigoActivacion(LoginViewModel login);
    string CambiarPassword(LoginViewModel login);
}
```

Un ejemplo de cómo se implementaría la interfaz

```
ItemcotecaDBEntities _context;

public LoginRepository()
{
    _context = new ItemcotecaDBEntities();
}

/// <summary>
/// Método que comprueba si existe un usuario y devolverá el usuario si existe. OK si existe, KO si no existem ERROR si ha habido algun problema
/// en la base de datos
/// </summary>
/// <param name="login"></param>
/// <returns></returns>
public string Login(LoginViewModel login)
{
    string result = "";
    string password = Seguridad.Security.CifradoSHA1Password(login.Password);

    var usuario = _context.Usuarios.Where(x => x.Correo.Equals(login.Correo, StringComparison.OrdinalIgnoreCase) && x.Password.Equals(password)).SingleOrDefault();

    if(usuario != null){
        result = usuario.Estado_Cuenta;
    }
    else
    {
        result = "KO";
    }

    return result;
}
```

Repository

El patrón repository consiste en declarar interfaces o clases abstractas siguiendo el convenio I<NombreInterfaz>Repository, y la clase que implementará esta interfaz se llamará <NombreClase>Repository. Si por ejemplo yo quiero implementar una funcionalidad total referente a un registro de usuarios debería de crearme una

interfaz llamada IRegistroUsuariosRepository y la clase que la implementará se llamará RegistroUsuariosRepository.

Inyección de dependencia

El patrón DI, es un patrón aplicado a los controladores para abstraernos de la implementación de una interfaz y centrarnos simplemente en su funcionalidad.

La forma de implementarla en un controlador es la siguiente:

```
private IRegistroRepository _service;

public RegistroController(IRegistroRepository service) {
    _service = service;
}
```

Cada vez que hagamos la llamada a un controlador éste se instanciará, y además, instanciamos un repositorio con la implementación que nosotros queramos, así si en algún futuro debemos de hacer una modificación de la funcionalidad solo debemos de cambiar el código en un solo punto, ese punto es nuestra clase Bootstrapper

```
public static class Bootstrapper
{
    public static IUnityContainer Initialise()
    {
        var container = BuildUnityContainer();

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        return container;
    }

    private static IUnityContainer BuildUnityContainer()
    {
        var container = new UnityContainer();

        // register all your components with the container here
        // it is NOT necessary to register your controllers

        // e.g. container.RegisterType<ITestService, TestService>();
        RegisterTypes(container);

        return container;
    }

    public static void RegisterTypes(IUnityContainer container)
    {
        container.RegisterType<IRegistroRepository, RegistroRepository>();
        container.RegisterType<ILoginRepository, LoginRepository>();
    }
}
```

En el objeto container debemos de registrar el tipo de interfaz y la clase que lo implementa. Haciendo esto cada vez que queramos cambiar una funcionalidad aquí es donde debemos de cambiar las referencias, y no tener que ir buscando por todo el proyecto cuales controladores usaban nuestra interfaz y cuáles no.

Ejemplo Cifrado AES

```
class AesExample
{
    public static void Main()
    {
        try
        {
            string original = "Here is some data to encrypt!";

            // Create a new instance of the AesCryptoServiceProvider
            // class. This generates a new key and initialization
            // vector (IV).
            using (AesCryptoServiceProvider myAes = new AesCryptoServiceProvider())
            {
                // Encrypt the string to an array of bytes.
                byte[] encrypted = EncryptStringToBytes_Aes(original, myAes.Key,
myAes.IV);

                // Decrypt the bytes to a string.
                string roundtrip = DecryptStringFromBytes_Aes(encrypted, myAes.Key,
myAes.IV);

                //Display the original data and the decrypted data.
                Console.WriteLine("Original: {0}", original);
                Console.WriteLine("Round Trip: {0}", roundtrip);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e.Message);
        }
    }
}

static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
{
    // Check arguments.
    if (plainText == null || plainText.Length <= 0)
        throw new ArgumentNullException("plainText");
    if (Key == null || Key.Length <= 0)
        throw new ArgumentNullException("Key");
    if (IV == null || IV.Length <= 0)
        throw new ArgumentNullException("Key");
    byte[] encrypted;
    // Create an AesCryptoServiceProvider object
    // with the specified key and IV.
    using (AesCryptoServiceProvider aesAlg = new AesCryptoServiceProvider())
    {
        aesAlg.Key = Key;
        aesAlg.IV = IV;

        // Create a decryptor to perform the stream transform.
        ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

        // Create the streams used for encryption.
        using (MemoryStream msEncrypt = new MemoryStream())
        {
            using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor,
CryptoStreamMode.Write))
            {
                using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
```

```

        {
            //Write all data to the stream.
            swEncrypt.Write(plainText);
        }
        encrypted = msEncrypt.ToArray();
    }
}

// Return the encrypted bytes from the memory stream.
return encrypted;
}

static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
{
    // Check arguments.
    if (cipherText == null || cipherText.Length <= 0)
        throw new ArgumentNullException("cipherText");
    if (Key == null || Key.Length <= 0)
        throw new ArgumentNullException("Key");
    if (IV == null || IV.Length <= 0)
        throw new ArgumentNullException("IV");

    // Declare the string used to hold
    // the decrypted text.
    string plaintext = null;

    // Create an AesCryptoServiceProvider object
    // with the specified key and IV.
    using (AesCryptoServiceProvider aesAlg = new AesCryptoServiceProvider())
    {
        aesAlg.Key = Key;
        aesAlg.IV = IV;

        // Create a decryptor to perform the stream transform.
        ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

        // Create the streams used for decryption.
        using (MemoryStream msDecrypt = new MemoryStream(cipherText))
        {
            using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read))
            {
                using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                {
                    // Read the decrypted bytes from the decrypting stream
                    // and place them in a string.
                    plaintext = srDecrypt.ReadToEnd();
                }
            }
        }
    }

    return plaintext;
}
}

```

Bibliografía

Programming Entity Framework, 2nd Edition - Julia Lerman

Pro ASP.NET MVC 4, 4th Edition - Adam Freeman

Link de descargas

Nuggets

Install-Package EntityFramework -Version 5.0.0

Install-Package Unity.Mvc4

Install-Package Unity.WebAPI

Install-Package Microsoft.AspNet.SignalR

SignalR y cliente Android

<http://whatthecode.wordpress.com/2014/03/20/getting-started-with-the-java-signalr-sdk/>

<http://www.mediafire.com/download/8qun6hbs3onnbb1/signalr-client-sdk.zip>

<https://code.google.com/p/google-gson/>