# MVC Tutorial – Notas Transcritas

## *Controllers*

Un $controller$ tiene métodos del tipo $ActionResult$, que corresponden a distintos 'directorios' dentro de la página Web.

Por ejemplo, en el $controller\ Store$, el método $Browse$ se activa cuando se ingresa a la URL $/Store/Browse$.

El método $Index$ es especial, porque se activa con el nombre del $Controller$, no es necesario explicitarlo.

Los métodos del $Controller$ pueden también recibir parámetros y usarlos en sus respuestas.

Por ejemplo, el método $public\ string\ Browse(string\ genre)$ recibe el parámetro $Disco$ cuando el URL ingresado es $/Browse?genre = Disco$.

Un caso especial de esto último es cuando el parámetro tiene el nombre $id$. En este caso, la URL se anida con un simple $'/'$, en vez de necesitar el nombre del parámetro.

## *Views*

Las $Views$ son el resultado de una acción ejecutada en el $Controller$. Cada método tiene su propia $View$ asociada. Esta $View$ puede además recibir parámetro en su llamado, de forma de mostrarlos en la página.

Para poder usar los parámetros, estos deben estar definidos como una clase en la carpeta $Models$. Además, se debe especificar en la primera línea de la $View$ el tipo de parámetro que recibirá, esto se hace de la siguiente forma:

$@model\ Namespace.ClassName$

A continuación, se puede referir al parámetro recibido mediante la sintaxis $@Model$. De esta forma se puede acceder a todos los campos y métodos del objeto.

Una extensión de esto consiste en recibir como parámetro una lista de objetos, lo que se puede lograr definiendo el modelo como $IEnumerable < Namespace.ClassName >$.

## *Links Entre Páginas*

Es mala práctica escribir los links por su URL usando un $< a\ href >$, porque es poco mantenible. Para esto, se utiliza un $@Html.ActionLink$ que recibe los parámetros que se quieren poner.

- $@Html.ActionLink("Link\ Text\ goes\ here","ActionName")$ se puede usar cuando se quiere hacer un link al mismo $Controller$.
- $@Html.ActionLink("Link\ Text\ goes\ here","ActionName",parametros)$ permite agregar parámetros, similar al método $GET$.

Ejemplo:

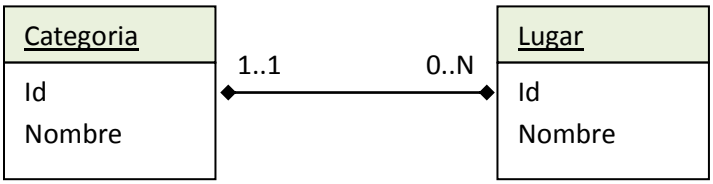
$$@Html.ActionLink(cat.Name, \text{Browse}, new\{\ categoria = cat.Name\})$$

## *Acceso a Bases de Datos con Entity Framework*

$Entity\ Framework$ permite acceder a la base de datos con una abstracción de las sentencias de $SQL$ que se necesitan para realizar las distintas operaciones.

El primer paso es definir clases en la carpeta $Models$ para las entidades del modelo $entidad - relación$. Estas clases deben tener los mismos atributos de las entidades, y con el mismo tipo y nombre para que sean reconocidas. El nombre de la clase deberá ser también el mismo que el de la tabla en la base de datos.

Supongamos que el siguiente es nuestro modelo $E - R$:




Luego, las clases definidas serían:

```
public class Categoria
{
    public int Id { get; set; }
    public string Nombre { get; set; }

    public List<Lugar> Lugar { get; set; }
}

public class Lugar
{
    public int Id { get; set; }
    public string Nombre { get; set; }

    public Categoria Categoria { get; set; }
}
```

El acceso a la base de datos se maneja mediante una clase especial en la carpeta $Models$ que debe heredar de $DbContext$. Para este ejemplo, definiremos la clase $DbEntities$.

```
public class DbEntities : DbContext
{
    public DbSet<Categoria> Categoria { get; set; }
    public DbSet<Lugar> Lugar { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

La clase especial contiene $DbSets$ de las clases Modelo que quieren ser pobladas con la base de datos.

Al inicializarse la clase $DbEntities$ las properties $Categoria$ y $Lugar$ son inmediatamente accesibles y contienen la información de las tuplas.

**El método definido en la parte baja de la clase deshabilita la convención de $Entity\ Framework$ de buscar las tablas respectivas en la base de datos agregando una $s$ al final del nombre de la clase.*

El estándar de $Entity\ Framework$ para las llaves foráneas es $NombreEntidadId$. Si se usa algún otro nombre se deberá explicitar esta relación.

## *Carga de Entidades Relacionadas*

Para usar los atributos de las clases que hacen referencia a otras entidades relacionadas, estos deben cargarse anteriormente.  Para este tema vea: aquí.

## *Manejo de Relaciones N a N*

Las relaciones N a N requieren un manejo distinto. En primer lugar, cada clase debe contener una lista de la otra clase.

En este caso, la tabla que maneja la relación no tiene un homólogo en código, pero ésta debe explicitarse al momento de crear el modelo.

Suponemos para el ejemplo una relación N a N entre Lugar y Categoría.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

    modelBuilder.Entity<Categoria>()
    .HasMany(x => x.Lugar)
    .WithMany(x => x.Categoria)
    .Map(m =>
    {
        m.ToTable("Lugar_En_Categoria"); // Nombre de la tabla
        m.MapLeftKey("CategoriaId");  // Nombre de la llave de Categoria
        m.MapRightKey("LugarId");     // Nombre de la llave de Lugar

    });
}
```

*Ver fuentes aquí.

## *Detalles Técnicos*

Para poder usar $Entity\ Framework$ se debe agregar la referencia al archivo $.dll$ que contiene a la librería que viene adjunto con este tutorial.

La clase $DbContext$ es accesible al incluir el $namespace\ System.Data.Entity$.

El nombre del $ConnectionString$ declarado en el archivo $Web.config$ debe corresponder al nombre de la clase especial, en este caso $DbEntities$.

```
  <connectionStrings>
    <add name="DbEntities"
    connectionString="Data Source=|DataDirectory|TryoutDB.sdf"
    providerName="System.Data.SqlServerCe.4.0"/>
  </connectionStrings>
```

## *Distintos Mecanismos de Carga de Datos*:

***Lazy Loading***, consiste en no hacer nunca una carga explícita de los datos. Al momento de acceder a una entidad relacionada a otra, los datos para ésta se cargan si aún no se ha hecho. Cada acceso implica un $query$ separado, por lo que si se realizan muchos puede resultar en una ejecución lenta.

### Ejemplo Lazy Loading:

$var\ cat = db.Categoria.Single(a => a.Id == 1);$
$var\ lugares = cat.Lugares;$

Para habilitar $Lazy\ Loading$, se debe modificar el método $OnModelCreating$ de la clase $DbEntities$ de forma de que habilite $LazyLoading$ como se ve en el recuadro de abajo.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

    Configuration.ProxyCreationEnabled = true;
    Configuration.LazyLoadingEnabled = true;
}
```

***Eager Loading***, en cambio, es una sentencia explícita que hace todas las asociaciones en un solo $query$. Puede ser muy lento si se está cargando muchos datos y es poco eficiente si no todos ellos van a ser consultados.

### Ejemplo Eager Loading:

$var\ categorias = db.Categoria.Include("Lugar");$

*Ver más sobre este tema aquí..*