Creación de aplicaciones MVC con EF

1. Crear un proyecto web asp.net entityframework, y seleccionar las opciones MVC, WebAPi y autenticación de usuarios individuales
2. Correr el proyecto para ver la funcionalidad que tiene (sin haber realizado nada)
3. Crear la cadena de conexión con la base de datos. Abrir el archivo webconfig y crear la cadena de conexión como se muestra a continuación. (Adsi\_2021)

<connectionStrings>

<add name="DefaultConnection" connectionString="Data Source=DESKTOP- M0E608B\SQLEXPRESS;Initial Catalog=Agenda2026693;Integrated Security=True"

providerName="System.Data.SqlClient" />

</connectionStrings>

1. Correr nuevamente el proyecto y revisar que la base de datos se cree en el servidor
2. Crear los modelos teniendo en cuenta que EntityFramework requiere una propiedad de tipo entera int, con nombreId, la cual será utilizada como clave primaria

public class Aprendiz

{

[Key]

public int AprendizId { get; set; } //identity

public string Documento { get; set; }

public string Nombres { get; set; }

public string Apellidos { get; set; }

public string Celular { get; set; }

public string Correo { get; set; }

}

1. Luego se debe crear la clase principal, que será la encargada de manejar las transacciones con la base de datos, esta clase se debe llamar Agenda2026693Context y heredar de la súper clase DbContext.

A esta clase se le crea un constructor por defecto que hace un llamado al constructor de la súper clase DbContext, enviándole como argumento el nombre de la conexión

public class Agenda2026693Context : DbContext

{

public Agenda2026693Context()

: base("DefaultConnection")

{

}

}

1. Luego a la clase Agenda2026693Context le debemos adicionar una propiedad de tipo DbSet, por cada una de las clases que se van adicionando al modelo, así

public class Agenda2026693Context : DbContext

{

public Agenda1Context()

: base("DefaultConnection")

{

}

public DbSet<Aprendiz> Aprendizs { get; set; }

}

1. Aquí se recomienda compilar el proyecto antes de empezar a crear los controladores
2. Para crear los controladores damos click derecho en la carpeta controller, new controlador y seleccionamos controlador vacío, el nombre del controlador es nombre del modelo pluralizado más la palaba controller

De forma original el controlador queda de la siguiente manera

public class AprendizsController : Controller

{

// GET: Aprendizs

public ActionResult Index()

{

return View();

}

}

1. El controlador es el encargado de manejar toda la lógica del negocio para ello se debe conectar con la base de datos, para poderse conectar se debe instanciar un objeto de tipo Agenda2026693Context

public class AprendicesController : Controller

{

private Agenda1Context db = new Agenda1Context();

// GET: Aprendices

public ActionResult Index()

{

return View();

}

}

1. También es muy recomendable cerrar la conexión con la base de datos al finalizar cada transacción, para ello adicionamos el siguiente método al controlador, el método dispose

public class AprendicesController : Controller

{

private Agenda1Context db = new Agenda1Context();

// GET: Aprendices

public ActionResult Index()

{

return View();

}

protected override void Dispose(bool disposing)

{

if (disposing)

{

db.Dispose();

}

base.Dispose(disposing);

}

}

1. Al trabajar con MVC, debemos tener en cuenta el concepto de acciones, que son los diferentes eventos que se pueden presentar cuando el usuario interactúa con el proyecto y las diferentes respuestas o acciones que el proyecto realiza ante las dichas acciones.

La primera acción que se considera dentro del controlador, es la acción Index la cual se encarga de enviar a una vista un listado de todos los objetos que hay almacenado en base de datos (objetos de cada modelo), la lógica de la acción Index es la siguiente:

public ActionResult Index()

{

return View(db.Aprendizs.ToList());

}

1. Ahora para poder que el usuario interactúe con la aplicación, debemos crear la vista para la acción determinada, para ello damos click derecho sobre el nombre de la acción y luego agregar vista.

Hay diferentes maneras de adicionar la vista; pero en este caso vamos a utilizar un aplantilla predefinida (créate, details, List, Delete) las plantillas predefinidas deben tener asociado un modelo y el contexto de la base de datos.

Las acciones de tipo Index por general se relacionan con la plantilla List, esta es la que vamos a seleccionar y obtendremos la siguiente vista en formato CSHTML

@model IEnumerable<Agenda1.Models.Aprendiz>

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

</p>

<table class="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.Documento)

</th>

<th>

@Html.DisplayNameFor(model => model.Nombres)

</th>

<th>

@Html.DisplayNameFor(model => model.Apellidos)

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Documento)

</td>

<td>

@Html.DisplayFor(modelItem => item.Nombres)

</td>

<td>

@Html.DisplayFor(modelItem => item.Apellidos)

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id=item.AprendizId }) |

@Html.ActionLink("Details", "Details", new { id=item.AprendizId }) |

@Html.ActionLink("Delete", "Delete", new { id=item.AprendizId })

</td>

</tr>

}

</table>

1. Para poder acceder a las nuevas acciones que se van adicionando al programa, deber ir actualizando el menú, para ello vamos a la carpeta vistas, subcarpeta shared, archivo \_layout.cshtml y adicional al menú las opciones que sean necesarias, de la siguiente manera

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li>@Html.ActionLink("Inicio", "Index", "Home")</li>

<li>@Html.ActionLink("Acerca de", "About", "Home")</li>

<li>@Html.ActionLink("Contacto", "Contact", "Home")</li>

<li>@Html.ActionLink("Aprendices", "Index", "Aprendices")</li>

</ul>

@Html.Partial("\_LoginPartial")

</div>

1. La siguiente acción es Create, la cual sirve para crear un registro; es importante tener en cuenta que esta acción tiene dos momentos, el primero se desencadena cuando el usuario da clic sobre el vínculo crear, ahí la aplicación lo debe enviar a una vista que le permita ingresar los datos del nuevo registro, este momento se denomina HttGet; luego cuando la nueva vista está llena, con los datos del nuevo registro se debe disparar una nueva acción en el momento HttPost, la cual captura los datos de la vista y los lleva a la base de datos por medio del controlador

[HttpGet]

public ActionResult Create()

{

return View();

}

1. Antes de generar la acción del HttPost, debemos crear la vista Create, ya sabemos que para ello damos clic derecho en el nombre de la acción, agregar vista, y para esta vista utilizaremos la plantilla créate, obviamente seleccionando el modelo y contexto adecuados

@model Agenda1.Models.Aprendiz

@{

ViewBag.Title = "Create";

}

<h2>Create</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Aprendiz</h4>

<hr />

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

<div class="form-group">

@Html.LabelFor(model => model.Documento, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Documento, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Documento, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Nombres, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Nombres, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Nombres, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Apellidos, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Apellidos, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Apellidos, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Create" class="btn btn-default" />

</div>

</div>

</div>

}

<div>

@Html.ActionLink("Back to List", "Index")

</div>

@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

}

1. El HttpPost de la acción Create es el siguiente

[HttpPost]

public ActionResult Create(Aprendiz aprendiz)

{

if (ModelState.IsValid)

{

db.Aprendices.Add(aprendiz);

db.SaveChanges();

}

else

{

return View(aprendiz);

}

return RedirectToAction("Index");

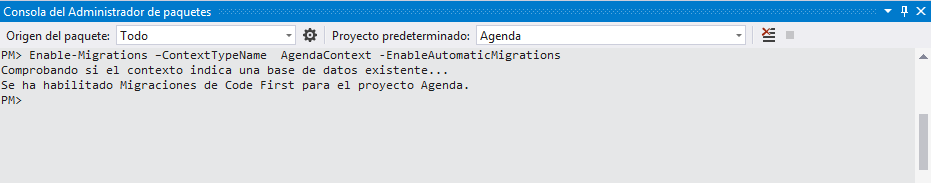
}

1. Como ya está creada la relación directa entre el modelo y la base de datos debemos tener en cuenta que los cambios en el modelo deben ser transparentes a la base de datos, y modificaciones no planeadas pueden afectar el funcionamiento del proyecto, por esta razón el framework debe ser configurado para aceptar migraciones automáticas a la base de datos de la siguiente manera.

Ir al menú al menú herramientas y a la opción administrador de paquetes NuGet, consola Nuget y ejecutar el comando

PM> Enable-Migrations –ContextTypeName Agenda2026693Context –EnableAutomaticMigrations -Force

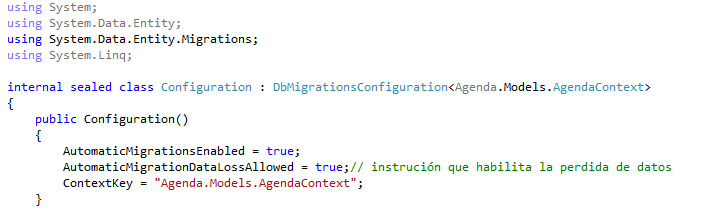
De este comando Agenda1Context es el nombre del contexto el cual se encuentra en la carpeta Models



Cuando sale el mensaje presentado en la imagen anterior se ha finalizado el primer paso para habilitar las migraciones automáticas, en este punto el Framework ha creado una carpeta llamada Migrations con una clase llamada Configuration, en la cual se deben agregar algunas líneas de código para terminar ésta configuración



En esta clase lo primero que debemos hacer es habilitar la perdida de datos teniendo en cuenta que es un entorno de desarrollo; con la siguiente instrucción la cual va debajo de AutomaticMigrationsEnabled=true y es la siguiente AutomaticMigrationDataLossAllowed = true; la clase debe quedar como se muestra a continuación



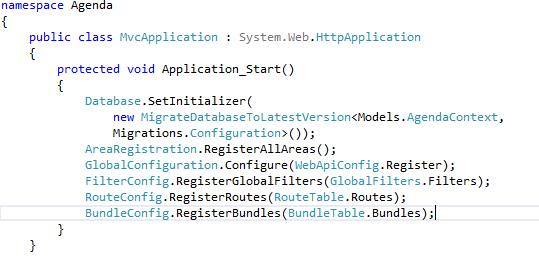
Con esto han quedado habilitadas las migraciones automáticas pero falta hacer algo para verificar la integridad o versionamiento de la base de datos y que de esta manera al iniciar la aplicación se verifique si la base de datos debe ser migrada, esto se hace en el archivo de configuración global.asax. Agregar la siguiente línea de código

Database.SetInitializer(

new MigrateDatabaseToLatestVersion<Models.Agenda2026693Context,

Migrations.Configuration>());

El método MigrateDatabaseToLatestVersion recibe 2 parámetros el primero es la carpeta Models.nombre del contexto el cual para nuestro caso es Agenda2026693Context, el segundo es la carpeta Migrations.Configuración que es la que se crea automáticamente cuando se ejecuta el comando para habilitar las migraciones automáticas. La vista final del global.asax es la siguiente



A partir de este momento ya podemos hacer modificaciones en los modelos y estas serán asumidas por la base de datos, aquí hay que tener en cuenta que las vistas no se modifican automáticamente por lo tanto debemos ir al código fuente y realizar las modificaciones del caso.

1. Ahora se deben crear los métodos para el resto de las acciones CRUD, inicialmente vamos a crear la acción Edit, esta acción en estado HttpGet se encarga de buscar en la base de datos un registro específico con base en el parámetro que recibe, este parámetro normalmente es la llave primaria del modelo al cual hacemos referencia. La acción Edit debe quedar de la siguiente manera.

public ActionResult Edit(int? id)

{

if (id.Equals(null))

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

//var aprendiz = db.Aprendices.Find(id);

Aprendiz aprendiz = db.Aprendizs.Find(id);

if (aprendiz.Equals(null))

{

return HttpNotFound();

}

return View(aprendiz);

}

1. Para continuar con la secuencia normal del proyecto debemos crear la vista, para ello damos clic derecho sobre el nombre de la acción, adicionar vista y seleccionamos la plantilla, acompañada de los campos modelo y contexto; para obtener la siguiente vista.

@model Agenda1.Models.Aprendiz

@{

ViewBag.Title = "Edit";

}

<h2>Edit</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Aprendiz</h4>

<hr />

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

@Html.HiddenFor(model => model.AprendizId)

<div class="form-group">

@Html.LabelFor(model => model.Documento, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Documento, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Documento, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Nombres, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Nombres, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Nombres, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Apellidos, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Apellidos, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Apellidos, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Save" class="btn btn-default" />

</div>

</div>

</div>

}

<div>

@Html.ActionLink("Back to List", "Index")

</div>

@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

}

1. Debemos tener en cuenta que la acción Edit también tiene su estado de HttpPost, por tal motivo debemos implementarlo.

[HttpPost]

public ActionResult Edit(Aprendiz aprendiz)

{

if (ModelState.IsValid)

{

db.Entry(aprendiz).State = EntityState.Modified;

db.SaveChanges();

}

else

{

return View(aprendiz);

}

return RedirectToAction("Index");

}

1. Continuando con el desarrollo del CRUD, vamos a crear una acción que aunque no hace parte de éste, es importante tener una acción y una vista que nos permitan ver todos los elementos de modelos que pueden ser muy grandes y no es funcional mostrarlos completos en la vista Index; esta acción la llamaremos Details y se crea como se ve a continuación. (esta acción sólo tiene estado HttpGet)

[HttpGet]

public ActionResult Details(int? id)

{

if (id.Equals(null))

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

//var aprendiz = db.Aprendices.Find(id);

Aprendiz aprendiz = db.Aprendizs.Find(id);

if (aprendiz == null)

{

return HttpNotFound();

}

return View(aprendiz);

}

1. Paso a seguir es crear la vista Details, para lo cual damos clic derecho sobre el nombre de la acción, agregar vista, seleccionamos la plantilla Details acompañada del modelo y el contexto adecuados.

@model Agenda1.Models.Aprendiz

@{

ViewBag.Title = "Details";

}

<h2>Details</h2>

<div>

<h4>Aprendiz</h4>

<hr />

<dl class="dl-horizontal">

<dt>

@Html.DisplayNameFor(model => model.Documento)

</dt>

<dd>

@Html.DisplayFor(model => model.Documento)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Nombres)

</dt>

<dd>

@Html.DisplayFor(model => model.Nombres)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Apellidos)

</dt>

<dd>

@Html.DisplayFor(model => model.Apellidos)

</dd>

</dl>

</div>

<p>

@Html.ActionLink("Edit", "Edit", new { id = Model.AprendizId }) |

@Html.ActionLink("Back to List", "Index")

</p>

1. Para finalizar el mantenimiento de tabla vamos a crear la acción Delete, esta acción consta de los estados HttpGet y HttPost, para darle al usuario la opción de confirma si desea hacer uso de la eliminación del registro. Inicialmente crearemos la acción con el estado HttpGet

public ActionResult Delete(int? id)

{

if (id.Equals(null))

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

//var aprendiz = db.Aprendices.Find(id);

Aprendiz aprendiz = db.Aprendizs.Find(id);

if (aprendiz == null)

{

return HttpNotFound();

}

return View(aprendiz);

}

1. La vista de Delete, se crea de la misma manera que las anteriores y queda de la siguiente manera

@model Agenda1.Models.Aprendiz

@{

ViewBag.Title = "Delete";

}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>

<div>

<h4>Aprendiz</h4>

<hr />

<dl class="dl-horizontal">

<dt>

@Html.DisplayNameFor(model => model.Documento)

</dt>

<dd>

@Html.DisplayFor(model => model.Documento)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Nombres)

</dt>

<dd>

@Html.DisplayFor(model => model.Nombres)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Apellidos)

</dt>

<dd>

@Html.DisplayFor(model => model.Apellidos)

</dd>

</dl>

@using (Html.BeginForm()) {

@Html.AntiForgeryToken()

<div class="form-actions no-color">

<input type="submit" value="Delete" class="btn btn-default" /> |

@Html.ActionLink("Back to List", "Index")

</div>

}

</div>

1. Por último creamos la acción Delete con su estado HttpPost, la cual queda de la siguiente manera

[HttpPost]

public ActionResult Delete(int id, Aprendiz aprendiz)

{

aprendiz = db.Aprendizs.Find(id);

if (aprendiz == null)

{

return HttpNotFound();

}

else

{

db.Aprendizs.Remove(aprendiz);

db.SaveChanges();

}

return RedirectToAction("Index");

}

1. Los DataAnotation son utilizados para mejorar los niveles de validación frente a las necesidades puntuales de ejecución de proyectos; el atributo [Required] puede enviar mensajes de error utilizado de la siguiente manera

[Required(“Mensaje de error personalizado”)] o

[Required(“Mensaje de {0} error personalizado”)] el cero entre llaves recupera el nombre del campo

El atributo [StringLengtg] sirve para indicar la longitud que puede tener un campo de texto específico y se puede usar con las siguientes variantes

[StringLength(50)]

[StringLength(50, MinimumLength=10)]

[StringLength(50, MinimumLength = 10, ErrorMessage="Remark must have min length of 10 and max Length of 50")]

[Required(ErrorMessage = "El campo {0} es obligatorio")]

[StringLength(80, MinimumLength = 10,

ErrorMessage = "El campo {0} debe tener entre {2} y {1} caracteres")]

El atributo [Range] se utiliza para establecer rangos en valore numéricos y se puede utilizar con las siguientes variantes

[Range(300, 3000)]

[Range(300, int.maxvalue)]

[Range(300, double.maxvalue)]

[Range(10, 1000,

ErrorMessage = "Value for {0} must be between {1} and {2}.")]

[Range(typeof(DateTime), "1/2/2004", "3/4/2004",

ErrorMessage = "Value for {0} must be between {1} and {2}")]

El atributo [Compare] se utiliza para comprar el valor de dos o más campos, puede ser en el caso de emails o contraseñas y funciona con las siguientes variantes

public string EmailAddress {get; set;}

[Compare(nameof(EmailAddress), ErrorMessage = "Emails mismatch")]

public string VerifiedEmailAddress { get; set; }

El atributo [NotMapped] se utiliza para evitar enviar un campo específico a la base de dato

Ela atributo [NotMapped] se puede combinar con [ScaffoldColumn(false)] para que el campo tampoco aparezca en el vista

El atributo [CreditCard] se utiliza para validar el número de tarjetas de crédito y se puedo combinar con el mensaje de error

[CreditCard, ErrorMessage=”Mensaje de error”]

El atributo [Remote] se utiliza para llamar métodos creados desde javascript

[Remote("ActionName", "ControllerName", ErrorMessage = "{0} Is somthing!")]

El atributo [Display] se utiliza para dar formato al nombre de las propiedades y se utilizar así [Display(name=”nombre con espacio”)] lo que está dentro de nombre es lo que se va a mostrar en la vista

Para dar un forma especifico a los campos, se puede utilizar el atributo [DataType], el cual puede tener las siguientes variantes

[DataType(DataType.MultiLineText)]

[DataType(DataType.EmailAddress)]

[DataType(DataType.Passsword)]

[Display (name= “Nombre que se muestra en la vista”]

[DataType(DataType.DateTime)]

[DisplayFormat(DataFormatString=”0: yyyy-MM-dd hh:mm”,ApplyFormatInEditMode0true)]

Hay dataAnotations que permiten manejar campos con valor único dentro de la tabla, por ejemplo emails o nombres de usuario el dataAnotations es el siguiente

[Index(“EmailIndex”, IsUnique=true)] en este caso Email es el nombre del campo por eso queda como EmailIndex

Para poderlo utilizar puede ser necesario actualizar el EF, para ello vamos a herramientas, administrador paquetes nuget, administrador de paquetes nuget de la solución vamos a actualizaciones y se actualiza el paquete EntityFramework

1. Crearemos ahora la clase/entidad ficha con todas las operaciones CRUD para poder realizar su mantenimiento respectivo.
2. Crear el modelo de ficha

public class Ficha

{

[Key]

public int FichaId { get; set; } // llave primaria

[Required(ErrorMessage ="Este es un campo obligatorio")]

[StringLength(10, MinimumLength =3,

ErrorMessage =" el campo {0} debe tener entre {2} y {1} caracteres ")]

public string Codigo { get; set; }

public string Especialidad { get; set; }

}

1. Ahora debemos crear el DbSet de ficha en la clase Agenda2026693Context, y queda de la siguiente manera

public class Agenda2026693Context: DbContext

{

// Constructor por defecto

public Agenda2026693Context()

: base("DefaultConnection")

{

}

// Atributos de clase

public DbSet<Aprendiz> Aprendizs { get; set; }

public DbSet<Ficha> Fichas { get; set; }

}

1. Compilar el Proyecto y crear el controlador el blanco FichasController
2. Crear las acciones y las vistas con base en los anteriores puntos de este manual o tomando como referencia las acciones ya creadas en el controlador de aprendices.
3. En cualquier sistema de información lo más relevante son las relaciones que se puedan presentar entre las diferentes entidades, EF Code Firts nos permite establecer las relaciones desde el modelo utilizando propiedades virtuales, las cuales no se verán reflejadas en la base de datos. Supongamos la siguiente relación para nuestro ejemplo: una ficha tiene muchos aprendices, pero un aprendiz sólo puede pertenecer a una ficha



Para dar funcionalidad a esta relación, la llave primaria de la clase con multiplicidad uno debe pasar a ser un atributo de la clase con multiplicidad muchos, este atributo debe ser del mismo tipo y con los mismos dataAnotations; así mismo en el lado muchos de la relación se debe definir un atributo virtual con tipo de datos lado uno, es decir un objeto que identifica dicha clase asi; public virtual Ficha ficha {get; set;}

En el lado uno de la relación se debe crear una propiedad virtual de tipo colección genérica de la clase que representa el lado muchos de la relación así: public virtual ICollection<Aprendiz> aprendices {get; set;}

public class Aprendiz

{

[Key]

public int AprendizId { get; set; }

[Required (ErrorMessage ="El campo {0}, es requerido")]

public string Documento { get; set; }

[Required(ErrorMessage = "El campo {0}, es requerido")]

public string Nombres { get; set; }

public string Apellidos { get; set; }

// elementos para establecer la relación

[Required(ErrorMessage = "El campo {0}, es requerido")]

public int FichaId { get; set; }

public virtual Ficha ficha { get; set; }

}

public class Ficha

{

[Key]

public int FichaId { get; set; }

[Required (ErrorMessage ="El campo {0}, es requerido")]

public string Codigo { get; set; }

[Required(ErrorMessage = "El campo {0}, es requerido")]

public string Especialidad { get; set; }

// elemento para establecer la relacion

public virtual ICollection<Aprendiz> aprendicees { get; set; }

}

1. Para continuar debemos tener en cuenta que se deben hacer algunas modificaciones en el controlador de aprendiz y obviamente en la vista, inicialmente en la acción Index del HttpGet se debe adicionar el código para hacer la consulta cruzada con la entidad que se ha establecido la relación

var aprendizs = db.Aprendizs.Include(a => a.ficha);// recupera la relación entre aprendiz y ficha

[HttpGet]

public ActionResult Index()

{

var aprendizs = db.Aprendizs.Include(a => a.ficha);// recupera la relacion entre aprendiz y ficha

return View(db.Aprendizs.ToList());

}

1. La vista debe incluir un nuevo campo que muestre el registro relacionado con la nueva entidad, para ello lo más fácil es eliminar la vista actual y generar una nueva vista.
2. Ahora realizaremos los cambios en la acción Create; en el estado HttpGet debemos crear un ViewBag que nos permita llevar a la vista el campo que hace la relación con la otra entidad, esta lista se arma teniendo en cuenta los valores de datos y de textos que componen la integridad referencial asi:

[HttpGet]

public ActionResult Create()

{

ViewBag.FichaId = new SelectList(db.Fichas, "FichaId", "Codigo"); // llenar el viewbag con una correspondencia del campo código con el campo fichaId

return View();

}

Tengamos presente corregir la vista, eliminando y creandola nuevamente

1. En el estado HttpPost de la acción Create debemos generar un viewbag que además de los campos de datos y de texto, identifique el valor de la entidad a crear que se relaciona con la entidad externa

[HttpPost]

public ActionResult Create(Aprendiz aprendiz)

{

if (ModelState.IsValid)

{

db.Aprendizs.Add(aprendiz);

db.SaveChanges();

//return RedirectToAction("Index");

}

else

{

return View(aprendiz);

}

ViewBag.FichaId = new SelectList(db.Fichas, "FichaId", "Codigo", aprendiz.FichaId);

return RedirectToAction("Index");

}

1. Ahora modificaremos la acción Edit en su estado HttpGet, la modificación es básicamente la misma que se hizo en el punto anterior, es decir agregar la siguiente línea de código

ViewBag.FichaId = new SelectList(db.Fichas, "FichaId", "Codigo", aprendiz.FichaId);

Para al final tener el controlador como se ve a continuación

[HttpGet]

public ActionResult Edit(int? id)

{

if (id.Equals(null))

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

//var aprendiz = db.Aprendizs.Find(id);

Aprendiz aprendiz = db.Aprendizs.Find(id);

if (aprendiz.Equals(null))

{

return HttpNotFound();

}

ViewBag.FichaId = new SelectList(db.Fichas, "FichaId", "Codigo", aprendiz.FichaId);

return View(aprendiz);

}

No podemos olvidar en este punto modificar la vista para que se adapte a los nuevos cambios efectuados sobre el controlador y tengamos en cuenta que es mejor eliminarla y crearla nuevamente

1. Esta misma será la modificación que haremos en el estado HttpPost de la acción Edit y dicha acción se verá de la siguiente manera

[HttpPost]

public ActionResult Edit(Aprendiz aprendiz)

{

if (ModelState.IsValid)

{

db.Entry(aprendiz).State = EntityState.Modified;

db.SaveChanges();

}

else

{

return View(aprendiz);

}

ViewBag.FichaId = new SelectList(db.Fichas, "FichaId", "Codigo", aprendiz.FichaId);

return RedirectToAction("Index");

}

1. Las acciones Delete y Details no se modifican, pero es recomendable eliminar y volver a crear las vistas de éstas acciones para que el código sea limpio y trazable.
2. En un sistema de información no es recomendable permitir la eliminación en cascada, debido a que esto puede ocasionar pérdidas importantes de información, por tal motivo debemos configurar el Framework para impedir esta operación. Para ello vamos a la clase que representa el contexto en la carpeta Models y adicionamos el siguiente método después del constructor y antes de los DbSet

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();

}

Con la implementación de este método ya se impide la eliminación en cascada, pero hay que adaptar la acción Delete para que el programa no se reviente

[HttpPost]

public ActionResult DeleteConfirmed(int id)

{

Ficha ficha = db.Fichas.Find(id);

try

{

db.Fichas.Remove(ficha);

db.SaveChanges();

}

catch (Exception ex)

{

if (ex.InnerException != null && ex.InnerException.InnerException != null && ex.InnerException.InnerException.Message.Contains("REFERENCE"))

{

ViewBag.Error = "No se permite elimminar registros con integridad refferencial";

}

else

{

ViewBag.Error = ex.Message;

}

return View(ficha);

}

return RedirectToAction("Index");

}

1. El mensaje de error que se almacena en el elemento ViewBag.Error se debe enviar a la vista para que el usuario lo pueda ver

@model Agenda2026693.Models.Ficha

@{

ViewBag.Title = "Delete";

}

<h2>Delete</h2>

<h3>@ViewBag.Error</h3> // esta es la mdificacion que se hace en la vista delete

<h3>Are you sure you want to delete this?</h3>

<div>

<h4>Ficha</h4>

<hr />

<dl class="dl-horizontal">

<dt>

@Html.DisplayNameFor(model => model.Codigo)

</dt>

<dd>

@Html.DisplayFor(model => model.Codigo)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Especialidad)

</dt>

<dd>

@Html.DisplayFor(model => model.Especialidad)

</dd>

</dl>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-actions no-color">

<input type="submit" value="Delete" class="btn btn-default" /> |

@Html.ActionLink("Back to List", "Index")

</div>

}

</div>