Utiliza las tablas Brands y Cervezas que ya tengo.

ASP NET con MVC

StartApp desaparece. Todo se hace en el program.Cs

Lo 1° es instalar entity Framework. ORM.

NuGet es una herramienta para instalar bibliotecas.

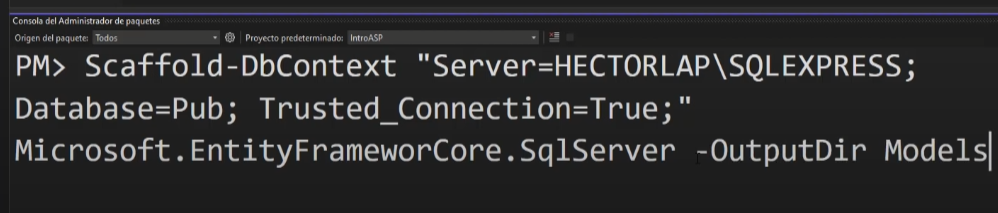




Modelo es la representación de las entidades que vamos a trabajar

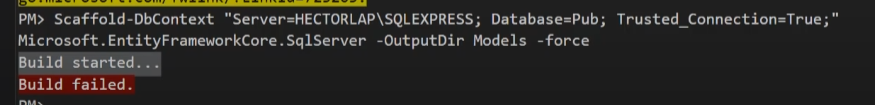
Mapear BBDD

Consola de administrador de paquetes Nuget



Con esto se genera una clase por cada tabla de la BBDD

Si luego cambia la BBDD y se desea cambiar el ORM hay que agregar el forcé. Señala que eso hace que se compile el proyecto, por lo que si hay cualquier error en el mismo, la actualización del ORM va a fallar.



Mi texto:

PM> Scaffold-DbContext "Server=Desktop-MSJO6R8;DataBase=BolsaDeDatos; Trusted\_Connection=True;TrustServerCertificate=True;" Microsoft.EntityFrameworkCore.SqlServer -OutPutDir Models

PM> Scaffold-DbContext "Server=Desktop-MSJO6R8;DataBase=TPN4; Trusted\_Connection=True;TrustServerCertificate=True;" Microsoft.EntityFrameworkCore.SqlServer -OutPutDir Models

Cadena de conexión

Dice que es inseguro tirarla allí por donde anda el código. Cuando se ejecuta el comando aparece una leyenda amarilla repitiendo esto mismo:

#warning To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid scaffolding the connection string by using the Name= syntax to read it from configuration - see https://go.microsoft.com/fwlink/?linkid=2131148. For more guidance on storing connection strings, see <http://go.microsoft.com/fwlink/?LinkId=723263>.

Y tecodea la cadena de conexión.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

=> optionsBuilder.UseSqlServer("Server=Desktop-MSJO6R8;DataBase=BolsaDeDatos; Trusted\_Connection=True;TrustServerCertificate=True;");

Por eso las pone en el appSettings.Json

"ConnectionStrings": {

"ConexionPablo": "Server=Desktop-MSJO6R8;DataBase=BolsaDeDatos; Trusted\_Connection=True;TrustServerCertificate=True;"

}

Inyección de dependencias (Cadena de conexión)

Ayuda a resolver un principio Solid que es la inversión de dependencias.

Los objetos se crean en un contenedor y luego se los inyecta, allí donde se los necesite.

builder.Services.AddDbContext<BolsaDeDatosContext>(opciones =>

{

opciones.UseSqlServer(builder.Configuration.GetConnectionString("ConexionPablo"));

});

Ahí se está inyectando la cadena de conexión. No olvidar los 2 using:

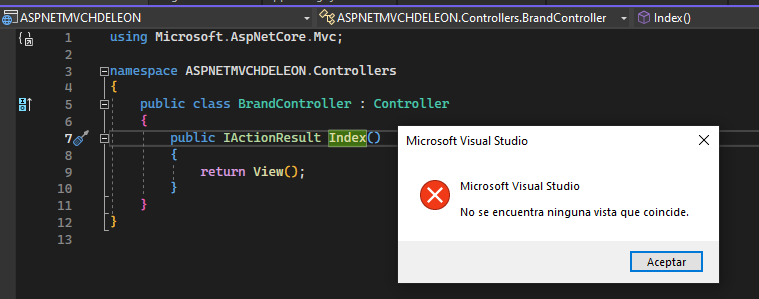
using ASPNETMVCHDELEON.Models;

using Microsoft.EntityFrameworkCore;

Controlador

Agrega controlador MVC en blanco.

Retornan una vista. Una vista en MVC es un archivo Razor (cshtml), el cual es parecido a un html.

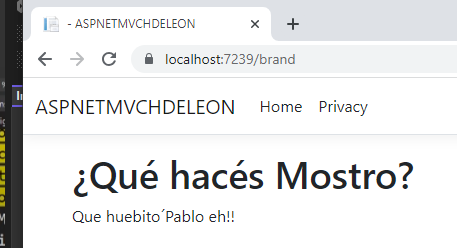


Así se ve un controlador pelado.

Hice click derecho en Index y seleccioné “ir a vista”, ahí salta ese error, ya que no hay ninguna vista (o sea, archivo Razor, asociado). Por eso hay que Agregar vista. Se crea en Views y en una carpeta de igual nombre que el controlador.

Vista

Agrego unas líneas HTML. Compilo y se ve así (/Brand):



En Shared está el header y el footer en layout.cshtml. o también puede ponerse:

@{

Layout = null;

}

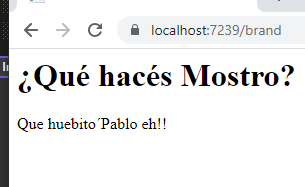
Para no utilizar ningún layout:

\_ViewStart.cshtml tiene solamente esto:

@{

Layout = "\_Layout";

}



Agrega un nuevo itém al header

li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Brand" asp-action="index">Marca</a>

</li>

Inyección de dependencias II (En controlador)

A brandController le agrega un campo privado para tener el contexto, el cual se recibe por parámetro en el constructor.

public class BrandController : Controller

{

private readonly BolsaDeDatosContext \_context;

public BrandController(BolsaDeDatosContext context)

{

\_context = context;

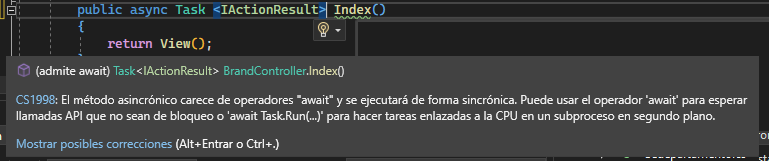
}

Esto se puede hacer porque fue inyectado en program.cs

Controlador Async/Task/Await

Hace que el método sea asíncrono, con lo cual tiene que poner task también. Async hace que se ejecute en un pool independiente, por lo que será mas rápido. Dice tener un video sobre async.

Con eso codeado, aparece la advertencia de que es al pedo poner async Task<> si después no vas a poner un await. Funcionaría sincrónicamente.



Versión posta:

public class BrandController : Controller

{

private readonly BolsaDeDatosContext \_context;

public BrandController(BolsaDeDatosContext context)

{

\_context = context;

}

public async Task <IActionResult> Index()

{

return View(await \_context.Brands.ToListAsync());

}

}



Así puede hacerse cuando hay solo una línea de código.

Dentro del paréntesis está lo que se le manda a la vista.

Vista

Con la primera línea se recibe lo enviado desde el controller:

@model List<Brand>

<h1> Marcas </h1>

<table class="table">

<thead>

<tr>

<th>Nombre</th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>@Html.(m=>item.Name)</td>

</tr>

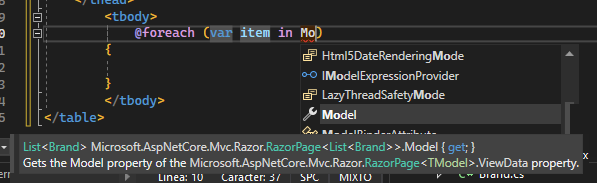
}

</tbody>

</table>

El foreach va a crear un tableRow y tableData por cada bucle que haga, printeando el name del item(brand).

Quiero que se note que ese Model refiere a lo que se le paso a la vista:



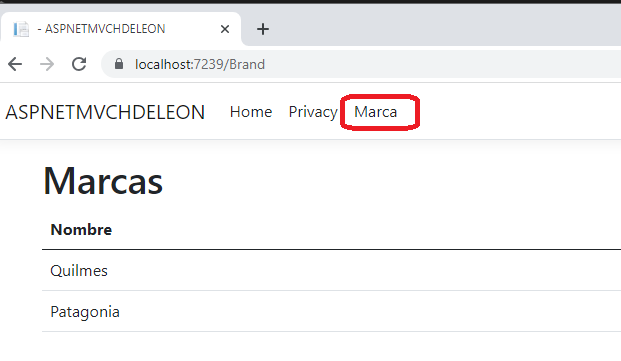
Dice que esto también funcionaría:

<tr>

<td>@item.Name</td>

</tr>

Al hacer click en Marca, se accede a esta bella página:



Modelo (Brand)

La representación de la table es una clase sencilla:

public class Brand

{

public int BrandId { get; set; }

public string Name { get; set; } = null!;

public virtual ICollection<Cerveza> Cervezas { get; set; } = new List<Cerveza>();

}

Inicialmente es una clase partial. Se lo saqué y no explotó.

El ICollection <Cerveza> Cervezas es una lista. Señala que las tablas están unidas; en este caso, dice que una marca, puede tener muchas cervezas.

Traer info de 2 tablas

Agrega otro controlador en blanco para cervezas.

public partial class CervezaController : Controller

{

private readonly BolsaDeDatosContext \_context;

public CervezaController(BolsaDeDatosContext context)

{

\_context = context;

}

public async Task <IActionResult> Index()

{

var beers = \_context.Cervezas.Include(b => b.CervezaNavigation);

return View(await beers.ToListAsync());

}

}

Acá hay un cambio, hay que usar el Include. La “b” es Cervezas y CervezaNavigation es el listado de brands de cada cerveza (no se si debería ser una relación de uno a muchos, sino de uno a uno). Adelante se pone: @Html.DisplayFor(m=>item.CervezaNavigation.Name) para acceder.

Luego agrega la vista correspondiente (prácticamente igual).

@model List<Cerveza>

<h1>Cervezas</h1>

<table class="table">

<thead>

<tr>

<th>Nombre</th>

<th>Marca</th>

<th> </th>

</tr>

</thead>

<tbody>

@foreach (var item in Model)

{

<tr>

<td>

@Html.DisplayFor(m=>item.Name)

</td>

<td>

@Html.DisplayFor(m=>item.CervezaNavigation.Name)

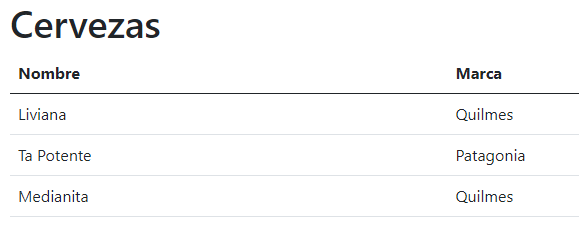
</td>

</tr>

}

</tbody>

</table>



Create

En la vista index, agrega un enlace para un formulario:

<h1>Cervezas</h1>

<p>

<a asp-action="Create"> Crear </a>"

</p>

<table class="table table-striped">

Ese Create llama al controlador de Cerveza, donde hay un método Create que llama a su vista homónima:

public IActionResult Create()

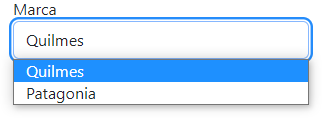
{

ViewData["Brands"] = new SelectList(\_context.Brands, "BrandId", "Name" );

return View();

}

Con lo de ViewData la info queda guardada, pero no en una BBDD, sino en un ComboBox.

ViewModel

Antes de agregar la vista:

ASP trabaja con Data Anotations. Para esto crea una carpeta dentro de models llamada ViewModels y allí una clase llamada cervezaViewModel. Se distancia de la clase de entityFramework (la cual se podría usar, ignorando el viewmodel) ya que dice que es una clase pensada para el formulario:

public class CervezaViewModel

{

[Required]

[Display (Name="Nombre")]

public string Name { get; set; }

[Required]

[Display(Name = "Marca")]

public int BrandId { get; set; }

}

Vista Create

En el cshtml hay que agregar el llamado al ViewModel:

@using ASPNETMVCHDELEON.Models.ViewModel;

@model CervezaViewModel

Con asp-for vincula con algún campo del ModelView

En esta línea, con BrandId se vincula con el ModelView y con el ViewBag.Brands se vincula con el ViewData del controlador de Cerveza. El .Brands debe ser coincidente con lo que se entrecomilló cuando se creó el ViewData.

<**select** **asp-for**="BrandId" class="form-control" **asp-items**="ViewBag.Brands" ></**select**>

CÓDIGOS!!

@using ASPNETMVCHDELEON.Models.ViewModel;

@model CervezaViewModel

<h1> Crear cerveza</h1>

<hr />

<div class="row">

<div class="col-md-4">

<form asp-action="Create">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<div class="form-group">

<label asp-for="Name" class="control-label"></label>

<input asp-for="Name" class="form-control" />

<span asp-validation-for="Name" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="BrandId" class="control-label"></label>

<select asp-for="BrandId" class="form-control" asp-items="ViewBag.Brands"></select>

<span asp-validation-for="BrandId" class="text-danger"></span>

</div>

<div class="form-group">

<input type="submit" value="Crear" class="btn btn-primary" />

</div>

</form>

</div>

</div>

@section Scripts {

@{

await Html.RenderPartialAsync("\_ValidationScriptsPartial");

}

}

Al final hay 2 métodos Create

Uno es Get (implícito o no) y el otro Post

[HttpGet]

public IActionResult Create()

{

ViewData["Brands"] = new SelectList(\_context.Brands, "BrandId", "Name" );

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public async Task <IActionResult> Create(CervezaViewModel model)

{

if(ModelState.IsValid)

{

var beer = new Cerveza()

{

Name = model.Name,

BrandId = model.BrandId,

};

//Opción larga y opción corta:

\_context.Cervezas.Add(beer);

//\_context.Add(beer);

//Con esta última se agrega a la BBDD con EF:

await \_context.SaveChangesAsync();

//Index es el método de arriba, que te lleva a la viste de la tabla de cervezas

return RedirectToAction(nameof(Index));

}

ViewData["Brands"] = new SelectList(\_context.Brands, "BrandId", "Name", model.BrandId);

return View(model);

}

Al final agrega un partial view (no se bien para qué):

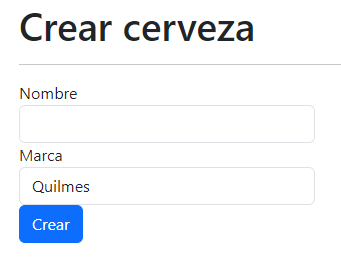
@section Scripts {

@{

await Html.RenderPartialAsync("\_ValidationScriptsPartial");

}

}



Esto para que las validaciones se realicen sobre datos provenientes del mismo dominio. Por una cuestión de seguridad.

[ValidateAntiForgeryToken]

Por parámetro le llega eso, ya que las validaciones están allí, en viewmodel:

(CervezaViewModel model)

ModelState.IsValid devuelve si el modelo ha pasado o no las validaciones. Dice HDeLeón que hay validaciones FrontEnd y las BackEnd. DataAnottations actúa en 2 tiempos:



Eso, por ejemplo, es frontEnd.

API

Agrega controlador de API. El cual hereda de controllerBase. En tanto que los otros, heredan de controller.

Estos son para trabajar en BackEnd. Devuelven un JSON, no una vista.

[Route("api/[controller]")]

[ApiController]

public class ApiBeerController : ControllerBase

{

private readonly BeerYbrandContext \_context;

public ApiBeerController(BeerYbrandContext context)

{

\_context = context;

}

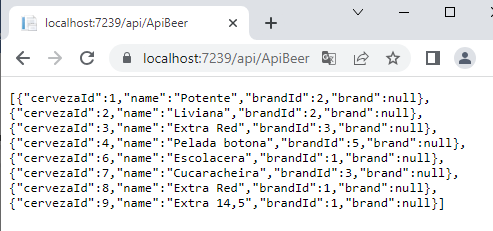
public async Task <List <Cerveza>> GetCerveza()

{

return await \_context.Cervezas.ToListAsync();

}

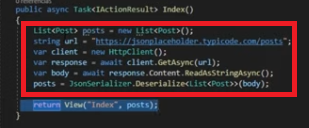
}



Capa de servicios

<https://www.youtube.com/watch?v=9-TvHe-hHeY>

Dice que la documentación de Microsoft te sugiere hacer una capa de servicios==dominio==aplicación==negocio.



Dice que un controlador solo debe tener lo que debe tener: una redirección a vista, algún direccionamiento, pero la funcionalidad en la capa de servicio.

Recomiendo crear un nuevo proyecto: biblioteca de clases. MVC.Services. Crea luego otro proyecto biblioteca mas, para separar los modelos: MVC.Models.

En MVC.Models CORTA LAS CLASES de la carpeta Models (no MVC.Models). Ahí, el proyecto original falla, por lo que hay que agregarle los 2 nuevos proyectos como su dependencia y todos los usings que hagan falta.

Hasta acá el programa funciona a las mil maravillas, aunque no se ganó nada.

Agrega una interfaz (en mi caso podría no hacerlo) al proyecto Services.

internal interface ICervezaServices

{

public IEnumerable<Cerveza> Create();

}

Y una clase que la implementa.

Luego, va al controlador y corta lo siguiente:



Y eso que está en azul lo pone en la clase de servicio.

En la clase de servicio establece los usings. La URL la manda a la clase, fuera del método, y crea var posts.

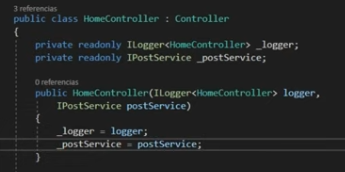
Finalmente pone, return posts.

Program.Cs (lo hace en StarUp)

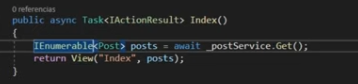
Hace inyección de las dependencias del servicio. Lo que allí se ponga, podría utilizarse en cualquier controlador.



Controlador



Define un campo y le da valor vía el contructor.

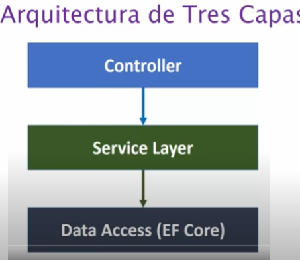


Así queda su clase. Llena el enumerable, llamando al método de la capa de servicio.

Capa de servicios (con contexto)

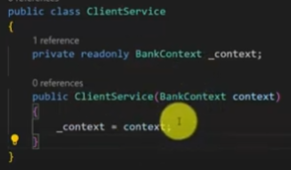
https://www.youtube.com/watch?v=uRx-SdprMSY [el-jamesaranda](https://www.youtube.com/@el-jamesaranda)

Context es parte de la capa de acceso a datos. Dice que no es lo ideal que un controlador se comunique directamente con el contexto.

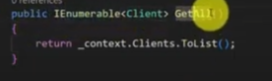


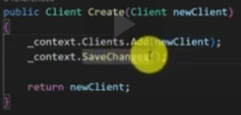
No crea un proyecto nuevo sino una carpeta y ahí ClientService.

No utiliza interfaces. Hace lo que hacía en los controladores: variable+constructor.



Este método GetAll simplemente lo cortó y lo pegó allí.

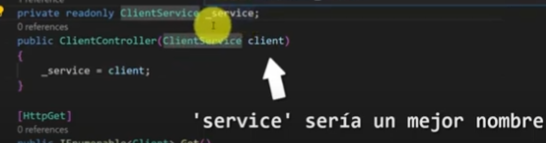
  
Este, un pelín mas complejo, también lo pega tal cual.



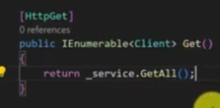
Inyección de dependencias en program.Cs



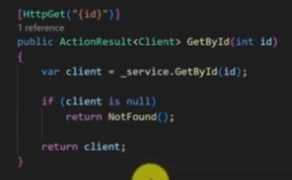
Elimina toda referencia en el controlador al contexto.



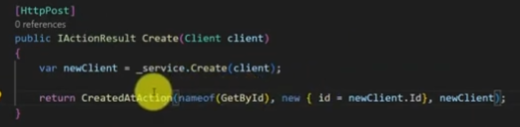
Así queda el método Get del controlador:



Con este Get by Id si que queda algo de código en el controlador.



Así queda el créate:



**https://localhost:7250/api/Users**

**https://localhost:7250/api/Users/1**

**https://localhost:7250/api/Users/pablokrh%40hotmail.com**