

Tema 5: El Modelo

Asignatura: Desarrollo web en entorno servidor

CS Desarrollo de Aplicaciones Web



Introducción

- En este capítulo veremos aspectos como:
 - Objetivo, alcance y contenido
 - Tipos de anotaciones
 - Restricciones del modelo
 - Restricciones personalizadas
 - Trabajo con Micro-ORMs

Introducción

- El Modelo de un sistema consta de:
 - **Entidades de negocio**, como la clase Post, que vimos en el tema anterior
 - **Las acciones para gestionar las entidades en el sistema.** Por ejemplo, la clase BlogManager con operaciones como Listar entidades de tipo Post.
 - **Las reglas y restricciones para asegurar la consistencia del sistema**, por ejemplo, impedir que se almacene un post vacío.
 - **La comunicación con el almacén de datos**, necesario para hacer persistente su estado.
 - **Los propios mecanismos de almacenamiento**

Introducción

- Como ya vimos, un ejemplo básico de un modelo es la clase **Post**:

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Code { get; set; }
    public string Text { get; set; }
    public DateTime Date { get; set; }
    public string Author { get; set; }
}
```

- Esta entidad se puede complicar mucho más aplicándole restricciones, así como relacionándola con otras entidades.

Introducción

- Para trabajar la persistencia de datos con estas entidades tendremos varias opciones:
 - **Utilizar un ORM** (Por ejemplo, Entity Framework, como vimos en el tema anterior)
 - **Utilizar un Micro-ORM** (Trabajaremos con Dapper)
- Volveremos sobre esto más adelante

RESTRICCIONES CON DATA ANNOTATIONS

Data Annotations

- Una de las responsabilidades del Modelo es **mantener el sistema en un estado correcto y coherente**, asegurando para ello que la información que va a gestionar cumple las restricciones impuestas por el entorno y el dominio de la aplicación.
- Para ello, ASP.NET Core MVC nos ofrece el mecanismo de **Data Annotations**
- Este mecanismo nos permitirá hacer una doble validación de los datos:
 - **En el servidor**, desde los controladores, para determinar la validez de la información recibida
 - Generando código script en la vista, de forma estas comprobaciones también se realicen **en el cliente**.

Data Annotations

- Las anotaciones de datos son atributos que podemos aplicar a las propiedades de las entidades del Modelo y a la propia entidad, para indicar restricciones en la información que van a contener.
- Más adelante, cuando se ejecute el proceso de validación, una entidad se considerará correcta si el resultado de la comprobación de cada una de dichas restricciones es correcto también.

Data Annotations

- Veamos un ejemplo:

```
public class Friend
{
    [Required, StringLength(50)]
    public string Name { get; set; }

    [RegularExpression(@"\w+([-.\'])\w+@\w+([-.\'])\w+\.\w+([-.\'])\w+*")]
    [Required, StringLength(100)] public string Email { get; set; }

    [Range(18, 99)]
    public int Age { get; set; }

    public string Description { get; set; }
}
```

Data Annotations

- Veamos cada uno de los casos:

- La propiedad **Name**, se indica que es requerida y que como máximo tendrá 50 caracteres
- La propiedad **Email**, se indica su obligatoriedad, el tamaño máximo, y también estamos especificando el formato de texto que aceptaremos mediante una expresión regular.
- La propiedad **Age**, se indica un rango de edades válidos
- La propiedad **Description**, permitiremos cualquier tipo de valor, incluso nulos, al no haber indicado ninguna limitación.

Data Annotations

Las clases “Buddy”

- En el ejemplo anterior vimos que es bastante sencillo el aplicarle restricciones a un modelo. Sin embargo, cuando la lógica sea más complicada, conviene separar el modelo de la validación de los atributos.
- En otras palabras, en lugar de agregar atributos de validación directamente en la clase de modelo, se puede crear **una clase separada que contenga la lógica de validación y que se asocie a la clase de modelo**

Data Annotations

Las clases “Buddy”

- Además, como vamos a trabajar con Entity Frameworks y otras herramientas ORM, corremos el riesgo de que esa entidad se autogenera y nos desaparezcan todos los atributos de validación que con tanto cuidado hemos estado colocando.
- Para solucionar esto, debemos:
 - Tener una clase **[Entidad]**, donde se definan cada uno de los atributos
 - Tener una clase buddy **[Entidad]Metadata**, donde definamos cada una de las validaciones.

Data Annotations

- Veremos ahora un ejemplo con la entidad Friend:

```
public class Friend
{
    public string Name { get; set; }
    public string Email { get; set; }
    public int Age { get; set; }
    public string Description { get; set; }
}
```

```
// File: FriendMetadata.cs
public class FriendMetadata
{
    [Required, StringLength(50)] public string Name { get; set; }

    [RegularExpression(@"\w+([-.\'])\w+@\w+([-.\'])\w+([-.\'])\w+")]
    [Required, StringLength(100)] public string Email { get; set; }

    [Range(18, 99)]
    public int Age { get; set; }
}
```

- Como se puede observar, contiene exclusivamente las propiedades de la entidad sobre las que queremos realizar anotaciones (Description, por ejemplo, no ha sido incluida).

Data Annotations

Las clases “Buddy”

- Por último, debemos modificar la clase Friend para decirle cuál es la clase que contiene sus metadatos, ayudándola a validar los atributos.
- Utilizaremos la propiedad **MetadataType**:



```
[MetadataType(typeof(FriendMetadata))]  
public class Friend  
{  
    public string Name { get; set; }  
    public string Email { get; set; }  
    public int Age { get; set; }  
    public string Description { get; set; }  
}
```

Actividad 1

Utilizando la base de datos Futbol, vamos a crear las clases correspondientes y sus clases buddy:

- *Futbolista y FutbolistaMetadata*
- *Equipo y EquipoMetadata*

10 MINUTOS

TIPOS DE ANOTACIONES

Data Annotations

Tipos de anotaciones

- A continuación enumeramos los tipos de anotaciones o restricciones que podemos encontrar en el espacio de nombres **System.ComponentModel.DataAnnotations**.

Data Annotations

Required

- Esta anotación **se aplica a propiedades que deben presentar algún contenido.**
- En concreto:
 - La validación será negativa cuando se trate de un tipo referencia y contenga el valor *null*.
 - Si se trata de un tipo valor, como *int* o *bool*, su contenido se considerará siempre correcto.
 - Cuando sea un string no nulo, se observará la propiedad *AllowEmptyStrings*. Si es false, su valor por defecto, tampoco se permitirán cadenas compuestas por espacios o con longitud cero.

Data Annotations

```
[Required]
public string Nombre { get; set; } // Empty string are not allowed.

[Required(ErrorMessage = "The surname is required")]
public string Apellidos { get; set; }

[Required(AllowEmptyStrings=true, ErrorMessage="The DNI is required")] // This property will allow spaces
public string DNI { get; set; }
```

Data Annotations

Range

- Esta anotación indica que el valor de la propiedad a la que se aplica debe encontrarse en un rango de valores determinado.
- OJO: Esta restricción no funcionará si un atributo contiene un null.

```
[Range(0.0, 300.0)]
public double Peso { get; set; }

[Range(0, 23, ErrorMessage = "Between {1} and {2}")]
public int Hora { get; set; }

[Range(typeof(DateTime), "1/1/1990", "5/5/2023")]
public DateTime FechaNacimiento { get; set; }
```

Data Annotations

RegularExpression

- Valida el contenido de una propiedad contra una expresión regular, tras convertirlo a string

```
[RegularExpression(@"\w+([-.\'])\w+*@(\w+([-.\'])\w+)*\.( \w+([-.\'])\w+)*")]
public string Email { get; set; }

[RegularExpression("...", ErrorMessage = "Must match the RegEx{1}")]
public string Value { get; set; }
```

Data Annotations

StringLength

- Permite comprobar que la **longitud en caracteres de un texto se encuentre dentro de un rango** determinado.
- Las cadenas con valor nulo se considerarán como de longitud cero

```
[StringLength(50)] // Max 50 chars
public string Nombre { get; set; }

[StringLength(9, MinimumLength = 9)]    // Exactly 9 chars length
public string NumeroTelefono { get; set; }

[StringLength(50, MinimumLength = 10)]   // Between 10 and 50 chars length
public string Password { get; set; }
```

Data Annotations

Compare

- Esta anotación permite **comparar la igualdad de propiedades incluidas en el mismo objeto.**
- Es habitual en funcionalidades que requieren la doble introducción del mismo valor, como una clave o una dirección de correo electrónico.

```
[Required]
public string Password { get; set; }

[Required, Compare("Password", ErrorMessage = "Passwords must match")]
public string PasswordRepetido { get; set; }
```

Data Annotations

CreditCard

- Esta anotación **permite comprobar si el contenido de la propiedad a la que se aplica es un número de tarjeta de crédito válido.**

```
[Required]  
[CreditCard]  
public string TarjetaCredito { get; set; }
```

Data Annotations

EmailAddress

- Esta anotación indica que el valor de la propiedad a la que se aplica debe ser una dirección de correo electrónico.
- Es equivalente a utilizar una anotación de tipo RegularExpression suministrando la expresión adecuada.

```
[EmailAddress]  
public string Email { get; set; }
```

Data Annotations

FileExtensions

- Este atributo se aplica a un campo de texto cuyo **contenido se espera sea un nombre de archivo**, y comprueba que su extensión sea una de las indicadas explícitamente en su declaración.
- Si no se especifica ninguna extensión se asumirán válidas las extensiones png, jpg, jpeg y gif

```
[FileExtensions(Extensions = "png, jpg")]
public string ImagenPerfil { get; set; }
```

Data Annotations

Phone

- Comprueba que la propiedad contenga un número de teléfono válido.
- Es equivalente a utilizar una anotación de tipo RegularExpression suministrando una expresión regular que impida el uso de caracteres no habituales en este tipo de dato.

```
[Phone]  
public string NumeroTelefono { get; set; }
```

Data Annotations

Url

- Esta anotación comprueba que **el contenido de la propiedad a la que se aplica es una URL válida.**
- Como en los casos anteriores, internamente lo único que se hace es validar el contenido usando una expresión regular.

```
[Url]  
public string DireccionWeb { get; set; }
```

Actividad 2

Aplicaremos las validaciones correspondientes a cada uno de los atributos:

- *Todos los campos son obligatorios*
 - *La edad irá de 0 a 100*
 - *Ningún campo numérico puede ser inferior a 0*
- *Las cadenas de caracteres deben tener como máximo 50 caracteres*

10 MINUTOS

CREACIÓN DE RESTRICCIONES PERSONALIZADAS

Data Annotations personalizados

- Hemos visto que los DataAnnotations nos permiten indicar muy fácilmente las restricciones habituales en propiedades de entidades del Modelo, como asegurar que no se introduzcan nulos o definir el tamaño máximo de una cadena de texto.
- Sin embargo, **es habitual encontrar escenarios en los que esto no es suficiente.**
- Veremos distintas formas de crear reglas personalizadas, más específicas del dominio de la aplicación, que nos ayuden a asegurar la validez de nuestras entidades del Modelo.

Data Annotations personalizados

Restricciones personalizadas

- El atributo **[CustomValidation]** nos permite indicar un método estático definido por nosotros que validará el atributo al que se lo hayamos especificado.
- Por ejemplo:

```
[CustomValidation(typeof(ValidacionesPersonalizadas), "EsPar")]
public int EsPar { get; set; }
```

```
public static class ValidacionesPersonalizadas
{
    public static ValidationResult EsPar(int valor)
    {
        if (valor % 2 == 0) return ValidationResult.Success;
        else return new ValidationResult("Debería ser un número par");
    }
}
```

Data Annotations personalizados

Creación de atributos personalizados

- Podemos definir atributos que implementen directamente la lógica de comprobación. Para ello, simplemente hay que crear una clase que herede de ValidationAttribute e implementar su método abstracto IsValid()

```
[EsPar]
public int NumeroPar { get; set; }
```

```
public class EsParAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        int i = Convert.ToInt32(value);
        return i % 2 == 0;
    }
}
```

Data Annotations personalizados

El interfaz **IValidatableObject**

- Esta técnica de validación se basa en la implementación del interfaz **IValidatableObject**.
- En él se define una única operación, **Validate()**, donde implementamos la lógica de validación completa de la entidad.
- **Comprobaremos la validez del objeto analizando el contenido de sus propiedades de instancia; los errores, en caso de existir, se retornan en una colección de objetos ValidationResult.**

Data Annotations personalizados

- Veremos un ejemplo para la clase Usuario:

```
public class Usuario : IValidatableObject
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string RetypedPassword { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (string.IsNullOrWhiteSpace(Username))
        {
            yield return new ValidationResult("El usuario no puede estar vacío", new[] { "Username" });
        }

        if (Password != RetypedPassword)
        {
            yield return new ValidationResult("Password inválido", new[] { "Password", "RetypedPassword" });
        }

        if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
        {
            yield return new ValidationResult("Los Domingos estamos cerrados :)");
        }
    }
}
```

Data Annotations personalizados

- Veremos un ejemplo para la clase Usuario (otra forma de hacerlo) :

```
public class Usuario : IValidatableObject
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string RetypedPassword { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        List<ValidationResult> listaValidaciones = new List<ValidationResult>();
        if (string.IsNullOrWhiteSpace(Username))
        {
            listaValidaciones.Add(new ValidationResult("El usuario no puede estar vacío", new[] { "Username" }));
        }

        if (Password != RetypedPassword)
        {
            listaValidaciones.Add(new ValidationResult("Password inválido", new[] { "Password",
                "RetypedPassword" }));
        }

        if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
        {
            listaValidaciones.Add(new ValidationResult("Los Domingos estamos cerrados :D"));
        }
        return listaValidaciones;
    }
}
```

Actividad 3

Crear una validación para comprobar si el futbolista está en activo:

- *Debe tener un CódigoEquipo*
- *Debe tener Edad menor de 45*
 - *Debe tener Minutos > 0*
- *Debe tener un Dorsal entre 1 y 25*
- *Debe tener PrecioMercado > 0*

10 MINUTOS

Vídeo: *Validaciones con DataAnnotations Avanzado*

https://youtu.be/rCYGd3_ADn4

Práctica guiada

Práctica guiada 3: Validaciones del Modelo en ASP.NET

Core MVC

TRABAJO CON MICRO-ORMs

Concepto de ORM

- Un **ORM** es una utilidad que nos permitirá formatear los datos de nuestros objetos para poder guardar la información en una base de datos (**mapeo**)
- De esta manera, se crea una base de datos virtual donde los datos que se encuentran en nuestra aplicación, quedan vinculados a la base de datos (**persistencia**)

Concepto de ORM

- Utilizando un ORM, permitiremos que el proceso de mapeo se haga automáticamente
- ¿En qué consiste esto del mapeo? Pues **consiste en transformar toda la información que recibamos de la base de datos, sobre todo las tablas, en objetos de nuestra aplicación y viceversa.**

Concepto de ORM

- Además, gracias a utilizar ORM nos abstraemos del motor de base de datos
- El cambio entre motores de bases de datos es muy sencillo, a veces solamente cambiando un par de líneas estaremos cambiando de motor de datos

Concepto de ORM

- Algunos ORM que existen son:
 - **Hibernate** (Java)
 - **MyBatis** (Java)
 - **Ebean** (Java)
 - **Entity Framework** (.NET)
 - **NHibernate** (.NET)
 - **MyBatis.NET** (.NET)
 - **Doctrine** (PHP)
 - **Rock**s (PHP)
 - **Torpor** (PHP)

Concepto de ORM

- Veremos, a continuación, un vídeo de un experto acerca del concepto de ORM:

<https://www.youtube.com/watch?v=TdXZIXqRGU8>

Concepto de Micro-ORM

- Un Micro-ORM es el mismo concepto que un ORM, pero podríamos decir que es una versión Lite.
- Ofrece una carga más ligera y rápida y sacrifica algunas facilidades y características para garantizar el rendimiento en el acceso a datos

Concepto de Micro-ORM

- Nos permite realizar un mapeo de nuestro modelo relacional a objetos de nuestra aplicación, pero de una forma más simple que el ORM
- Elimina algunas características como:
 - Manejo de relaciones entre tablas
 - Manejo de concurrencia
 - Diseñador del modelo, etc.

Concepto de Micro-ORM

- Algunos Micro-ORM que existen son:
 - **PetaPoco**
 - **Insight**
 - **NPoco**
 - **Dapper** (utilizado por StackOverflow)

ORM vs Micro-ORM

- ¿Qué diferencias hay entre ORM y Micro-ORM?

- Los beneficios de Micro-ORM son:

- Simplicidad de uso
 - Gran rendimiento

ORM vs Micro-ORM

- ¿Qué diferencias hay entre ORM y Micro-ORM?
 - **Las desventajas de un Micro-ORM son:**
 - No posee almacenamiento en caché
 - Las relaciones entre tablas son limitadas
 - No tiene un diseñador de tablas

ORM vs Micro-ORM

- Resumiendo, ¿cuándo utilizar ORM o Micro-ORM?

Micro-ORM

- Se busca rendimiento
- Es una aplicación de prueba
- Código heredado de SqlDataReader

ORM

- Son necesarias relaciones entre tablas
- El proyecto es lo suficientemente grande

Dapper

- Ahora que ya conocemos que es un Micro-ORM y sabemos cuál es su principal diferencia con un ORM, vamos a ver cómo funciona y cómo es su implementación.
- Utilizaremos el micro-ORM **Dapper**

Dapper

- Lo primero que haremos será instalar el paquete de Dapper en Visual Studio
- Para instalarlo, dentro del proyecto accedemos a “Dependencias – Administrar paquetes NuGet”
- Nos vamos a “Examinar” y buscamos el paquete Dapper.
- Lo instalamos

Dapper

 **Dapper** por Sam Saffron, Marc Gravell, Nick Craver, **126M** descargas v2.0.123
A high performance Micro-ORM supporting SQL Server, MySQL, Sqlite, SqlCE, Firebird etc..

 **Dapper.Contrib** por Sam Saffron, Johan Danforth, **12M** descargas v2.0.78
The official collection of get, insert, update and delete helpers for Dapper.net. Also handles lists of entities and optional "dirty" tracking of interface-based entities.

 **Dapper.FluentMap** por Henk Mollema, **2,71M** descargas v2.0.0
Simple API to fluently map POCO properties to database columns when using Dapper.

 **DapperExtensions** por Thad Smith, Page Brooks, Valfrid Couto, **1,84M** descargas v1.7.0
A small library that complements Dapper by adding basic CRUD operations (Get, Insert, Update, Delete) for your POCOs. For more advanced querying scenarios, Dapper Extensions provides a predicate system.

- Lo importaremos a nuestra aplicación con un using:

```
using Dapper;
```

Dapper. Conexión a la BD

- Para **conectarnos con la base de datos** a través de Dapper solo necesitamos especificar la cadena de conexión, como ya hemos visto.
- Definimos la cadena de conexión “**ConexionMontecastelo**”, dentro de la sección “**ConnectionString**s”, en el fichero **appsettings.json**

```
"ConnectionStrings": {  
    "ConexionMontecastelo": "Data Source=localhost;Initial Catalog=BD_Articulos;User  
    ID=sa;Password=root;Trusted_Connection=True;MultipleActiveResultSets=true;TrustServerCertificate=True"  
},
```

Dapper. Conexión a la BD

- Creamos una nueva clase llamada **Conexion** dentro de Models

```
public class Conexion
{
    private readonly string _connectionString;

    public Conexion(string valor)
    {
        _connectionString = valor;
    }

    public SqlConnection ObtenerConexion()
    {
        var conexion = new SqlConnection(_connectionString);
        conexion.Open();
        return conexion;
    }
}
```

Dapper. Definir clases del modelo

- Definimos las clases del modelo:

```
namespace Ejercicio1.Models
{
    public class Fabricante
    {
        public int codigo { get; set; }
        public string nombre { get; set; }
    }
}
```

Dapper. Añadimos el repositorio

- Creamos una nueva carpeta llamada **Repository** y creamos una interfaz **IFabricanteRepository**, donde definimos las operaciones que necesitamos

```
public interface IFabricanteRepository
{
    Task<IEnumerable<Fabricante>> GetFabricantes();
    Task<Fabricante> GetFabricante(int? codigo);
    Task CreateFabricante(Fabricante fabricante);
    Task UpdateFabricante(Fabricante fabricante);
    Task DeleteFabricante(Fabricante fabricante);
}
```

Dapper. Implementamos el repositorio

- Creamos una clase

FabricanteRepository

donde implementamos

cada una de las

operaciones definidas:

```
private readonly Conexion _conexion;  
  
public FabricanteRepository(Conexion conexion)  
{  
    _conexion = conexion;  
}
```

Dapper. Implementamos el repositorio

- Creamos una clase

FabricanteRepository

donde implementamos

cada una de las

operaciones definidas:

```
public async Task<IEnumerable<Fabricante>> GetFabricantes()
{
    var query = "SELECT * FROM fabricante";

    using (var connection = _conexion.ObtenerConexion())
    {
        var companies = await connection.QueryAsync<Fabricante>(query);
        return companies.ToList();
    }
}
```

Dapper. Implementamos el repositorio

- Creamos una clase

FabricanteRepository

donde implementamos

cada una de las

operaciones definidas:

```
public async Task<Fabricante> GetFabricante(int? codigo)
{
    var query = "SELECT * FROM fabricante WHERE codigo = @codigo";

    using (var connection = _conexion.ObtenerConexion())
    {
        var Fabricante = await connection.QuerySingleOrDefaultAsync<Fabricante>(query, new { codigo });
        return Fabricante;
    }
}
```

Dapper. Implementamos el repositorio

- Creamos una clase

FabricanteRepository

donde implementamos

cada una de las

operaciones definidas:

```
public async Task CreateFabricante(Fabricante Fabricante)
{
    var query = "INSERT INTO fabricante (codigo, nombre) VALUES (@codigo, @nombre)";

    var parameters = new DynamicParameters();
    parameters.Add("codigo", Fabricante.codigo, DbType.Int32);
    parameters.Add("nombre", Fabricante.nombre, DbType.String);

    using (var connection = _conexion.ObtenerConexion())
    {
        await connection.ExecuteAsync(query, parameters);
    }
}
```

Dapper. Implementamos el repositorio

- Creamos una clase

FabricanteRepository

donde implementamos

cada una de las

operaciones definidas:

```
public async Task UpdateFabricante(Fabricante Fabricante)
{
    var query = "UPDATE fabricante SET nombre = @nombre WHERE codigo = @codigo";
    var parameters = new DynamicParameters();
    parameters.Add("nombre", Fabricante.nombre, DbType.String);
    parameters.Add("codigo", Fabricante.codigo, DbType.Int32);
    using (var connection = _conexion.ObtenerConexion())
    {
        await connection.ExecuteAsync(query, parameters);
    }
}
```

Dapper. Implementamos el repositorio

- Creamos una clase

FabricanteRepository

donde implementamos

cada una de las

operaciones definidas:

```
public async Task DeleteFabricante(Fabricante Fabricante)
{
    var query = "DELETE FROM fabricante WHERE codigo = @codigo";
    using (var connection = _conexion.ObtenerConexion())
    {
        await connection.ExecuteAsync(query, new { Fabricante.codigo });
    }
}
```

Dapper. Configuramos la conexión

- Configuramos la conexión en el fichero **Program.cs**
 - Establecemos a qué cadena de conexión conectarnos
 - Mostramos cómo se va a crear un Fabricante

```
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton(new Conexion(builder.Configuration.GetConnectionString("ConexionMontecastelo")));
builder.Services.AddScoped<IFabricanteRepository, FabricanteRepository>();
```

Dapper. Creamos el controlador

- A continuación, creamos las diferentes acciones en el controlador:
 - Mostrar todos los fabricantes
 - Obtener un fabricante por código
 - Crear un fabricante
 - Actualizar la información de un fabricante
 - Borrar un fabricante

Dapper. Creamos el controlador

- A continuación, creamos las diferentes acciones en el controlador:
 - Mostrar todos los fabricantes

```
public async Task<IActionResult> Index()
{
    return RedirectToAction("GetAll");
}

[HttpGet]
public async Task<IActionResult> GetAll()
{
    var Data = await _fabricanteRepository.GetFabricantes();
    return View("Index", Data);
}
```

Dapper. Creamos el controlador

- A continuación, creamos las diferentes acciones en el controlador:
 - Obtener un fabricante por código

```
[HttpGet]
public async Task<IActionResult> GetById(int codigo)
{
    var fabricante = await _fabricanteRepository.GetFabricante(codigo);
    return View("VerFabricante", fabricante);
}
```

Dapper. Creamos el controlador

- A continuación, creamos las diferentes acciones en el controlador:
 - Crear un fabricante

```
public IActionResult Create()
{
    return View();
}

[HttpPost]
public async Task<IActionResult> Create(Fabricante fabricante)
{
    await _fabricanteRepository.CreateFabricante(fabricante);
    return RedirectToAction("Index");
}
```

Dapper. Creamos el controlador

- A continuación, creamos las diferentes acciones en el controlador:
 - Actualizar la información de un fabricante

```
public async Task<IActionResult> Update(int codigo)
{
    var fabricante = await _fabricanteRepository.GetFabricante(codigo);
    return View(fabricante);
}

[HttpPost]
public async Task<IActionResult> Update(Fabricante fabricante)
{
    await _fabricanteRepository.UpdateFabricante(fabricante);
    return RedirectToAction("Index");
}
```

Dapper. Creamos el controlador

- A continuación, creamos las diferentes acciones en el controlador:
 - Borrar un fabricante

```
public async Task<IActionResult> Delete(int codigo)
{
    var fabricante = await _fabricanteRepository.GetFabricante(codigo);
    return View(fabricante);
}

[HttpPost]
public async Task<IActionResult> Delete(Fabricante fabricante)
{
    await _fabricanteRepository.DeleteFabricante(fabricante);
    return RedirectToAction("Index");
}
```

Dapper. Creamos las vistas

- A continuación, creamos las diferentes vistas asociadas a las acciones:
 - Index
 - Ver fabricante
 - Crear fabricante
 - Actualizar fabricante
 - Borrar fabricante

Dapper. Vista Index

```
@model IEnumerable<Ejercicio1.Models.Fabricante>

{@
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.codigo)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.nombre)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.codigo)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.nombre)
                </td>
                <td>
                    @Html.ActionLink("Ver detalles", "GetById", new { codigo=item.codigo })
                    @Html.ActionLink("Actualizar", "Update", new { codigo=item.codigo })
                    @Html.ActionLink("Eliminar", "Delete", new { codigo=item.codigo })
                </td>
            </tr>
        }
    </tbody>
</table>
```

Dapper. Vista Ver fabricante

```
@model Ejercicio1.Models.Fabricante

<h1>Detalle del fabricante</h1>

<dl class="row">
    <dt class="col-sm-2">Codigo</dt>
    <dt class="col-sm-10">@Model.codigo</dt>

    <dt class="col-sm-2">Nombre</dt>
    <dt class="col-sm-10">@Model.nombre</dt>
</dl>
<p>
    <a asp-action="Index" class="btn btn-secondary">Regresar a la lista de fabricantes</a>
</p>
```

Dapper. Vista Crear fabricante

```
@model Ejercicio1.Models.Fabricante

<h1>Crear nuevo producto</h1>
<form asp-action="Create" method="post">
    <div class="form-group">
        <label asp-for="codigo" class="control-label"></label>
        <input asp-for="codigo" class="form-control" />
        <span asp-validation-for="codigo" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="nombre" class="control-label"></label>
        <input asp-for="nombre" class="form-control" />
        <span asp-validation-for="nombre" class="text-danger"></span>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-primary">Guardar</button>
        <a asp-action="Index" class="btn btn-secondary">Cancelar</a>
    </div>
</form>
```

Dapper. Vista Actualizar fabricante

```
@model Ejercicio1.Models.Fabricante

<h1>Editar fabricante</h1>
<form asp-action="Update" method="post">
    <input type="hidden" asp-for="@Model.codigo" />
    <div class="form-group">
        <label asp-for="nombre" class="control-label"></label>
        <input asp-for="nombre" class="form-control" />
        <span asp-validation-for="nombre" class="text-danger"></span>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-primary">Guardar</button>
        <a asp-action="Index" class="btn btn-secondary">Cancelar</a>
    </div>
</form>
```

Dapper. Vista Borrar fabricante

```
@model Ejercicio1.Models.Fabricante

<h1>Eliminar fabricante</h1>
<h3>¿Estás seguro que deseas eliminar el siguiente fabricante?</h3>
<dl class="row">
    <dt class="col-sm-2">Nombre</dt>
    <dt class="col-sm-10">@Model.nombre</dt>
</dl>
<form asp-action="Delete" method="post">
    <input asp-for="codigo" type="hidden" />
    <div class="form-group">
        <button type="submit" class="btn btn-danger">Eliminar</button>
        <a asp-action="Index" class="btn btn-secondary">Cancelar</a>
    </div>
</form>
```

Actividad 3

Crea una aplicación web que se conecte a base de datos Futbol utilizando Dapper y haga las siguientes acciones:

- *Permitir listar equipos*
- *Permitir crear equipos*
- *Permitir actualizar equipos*
- *Permitir borrar equipos*

20 MINUTOS

Actividad 3B

Repite todo el proceso para añadir la entidad Futbolista:

- *Permitir listar futbolistas*
- *Permitir crear futbolistas*
- *Permitir actualizar futbolistas*
- *Permitir borrar futbolistas*

Comprueba que se cumplen las pruebas de validación

20 MINUTOS

Expresiones LINQ Y LAMBDA

- **Linq** (Language integrated query) **es un conjunto de herramientas para explorar colecciones** (arrays, listas, etc.) para obtener elementos o subconjuntos usando lógica.
- Podemos hacer operaciones como coger elementos de la colección que cumplan con ciertas condiciones, ordenar por diferentes atributos, tomar el primer elemento de la colección que cumpla algo, y varias otras cosas.

Expresiones LINQ Y LAMBDA

- Las **lambda expressions** son funciones anónimas declaradas en la línea de ejecución.
- La **estructura** de las mismas es **definir el nombre de la variable, seguido por el operador de asignación ‘=>’ y luego la expresión a evaluar.**

```
x => x.Peso == 50
```

Expresiones LINQ Y LAMBDA

- A continuación, veremos las operaciones que podemos hacer en Linq:
 - **First()** y **FirstOrDefault()**
 - **Where()**
 - **OrderBy()**
 - **Union()**
 - **Select()** y **SelectMany()**

Expresiones LINQ Y LAMBDA

First()

- Obtiene el primer resultado de la colección, es lo mismo que hacer items[0]. Puede recibir una expresión lambda, permitiéndonos obtener el primer resultado filtrado en una colección.

```
return GetListaPokemon().First();
```

```
return GetListaPokemon().First(x => x.Nombre == "Pikachu");
```

Expresiones LINQ Y LAMBDA

FirstOrDefault()

- Devuelve el primer elemento si se encuentra y si no, devuelve un valor por defecto. En caso de objetos este sería null.

```
return GetListaPokemon().FirstOrDefault();
```

```
return GetListaPokemon().FirstOrDefault(x => x.Nombre == "Charmander");
```

Expresiones LINQ Y LAMBDA

Where()

- Devuelve un subconjunto de la colección original donde todos los elementos cumplen con la condición dada.
- A diferencia del First(), el Where() devuelve una colección numerable de ítems.

```
return (List<Pokemon>) GetListaPokemon().Where(x => x.Numero_pokedex == idPokemon);
```

Expresiones LINQ Y LAMBDA

OrderBy()

- Devuelve la colección ordenada por un criterio dado. Esta operación usa IComparable, de manera que podemos crear comparaciones específicas para objetos complejos. También existe **OrderByDescending()**.

```
return (List<Pokemon>) GetListaPokemon().OrderBy(x => x.Nombre);
```

```
return (List<Pokemon>) GetListaPokemon().OrderByDescending(x => x.Nombre);
```

Expresiones LINQ Y LAMBDA

Union()

- Nos sirve para generar una colección nueva a partir de la unión de dos colecciones, los elementos repetidos se van a pasar como uno solo.

```
    Tipo FUEGO = new Tipo(1, "Fuego", 1);
    Tipo AGUA = new Tipo(2, "Agua", 2);

    return (List<Pokemon>) GetListaPokemonByTipo(FUEGO).Union(GetListaPokemonByTipo(AGUA));
```

Expresiones LINQ Y LAMBDA

Select()

- El objetivo es obtener un conjunto de cosas que no necesariamente sean del tipo de la colección original.
- Por ejemplo, obtener todos los nombres de los ítems sin tener que recorrer la colección para extraer cada uno.

```
List<String> listaNombres;  
listaNombres = (List<String>)GetListaPokemon().Select(x => x.Nombre);
```

Expresiones LINQ Y LAMBDA

SelectMany()

- Funciona igual que **Select()**, pero se usa cuando la propiedad que queremos extraer es de por si una colección, por lo tanto, el resultado va a ser la unión de cada colección extraída por cada ítem.

```
listaEvoluciones = (List<Pokemon>)GetListaPokemon().SelectMany(x => x.Evoluciones);
```

Expresiones LINQ Y LAMBDA

Encadenación de operaciones

- Se pueden encadenar varias operaciones de las que hemos visto, por ejemplo, hacer un Where y luego OrderBy en la misma línea.
- Vemos un ejemplo a continuación:

```
return (List<Pokemon>)db.Fetch<Pokemon>().Where(x => x.Peso >= kilosMenor &&  
x.Peso <= kilosMayor).OrderBy(x=> x.Nombre).ToList();
```



KEEP
CALM
IT'S
KAHOOT
TIME

Tema 5: El Modelo

Asignatura: Desarrollo web en entorno servidor

CS Desarrollo de Aplicaciones Web

