

MVC NET CORE

Lunes, 10 de febrero de 2025 9:06

INYECCION DE DEPENDENCIAS

Es la parte más importante dentro de Net Core. Cada objeto es llamado de forma Automática dentro de las clases. Sin esta arquitectura, no podríamos realizar ciertas Acciones dentro de Net Core.

La inyección de dependencias permite poder crear objetos en un punto y poder Recuperarlos ya creados en múltiples lugares.
Si modificamos el objeto en el punto inicial, podemos seguir trabajando en el resto De capas sin necesidad de tocar código.
El punto donde se generan los objetos se llama **Container**

El concepto nos va a permitir "cambiar" una clase por otra sin que afecte al resto.
A esta arquitectura se le llama principios SOLID.

```
CLASE BICICLETA
public class Bicicleta { ... }
```

```
CLASE CICLISTA
public class Ciclista {
    Bicicleta bici;

    public Ciclista() {
        this.bici = new Bicicleta();
    }
}
```

¿Qué sucede si necesitamos que el Ciclista utilice una Bicicleta de carreras?

La inyección nos permite "cambiar" objetos de forma sencilla.

Los objetos son recibidos en el constructor de las clases:

```
CLASE BICICLETACARRERAS
public class BicicletaCarreras { ... }
```

```
CLASE CICLISTA
public class Ciclista {
    BicicletaCarreras bici;

    public Ciclista(BicicletaCarreras biciRecibida) {
        this.bici = biciRecibida;
    }
}
```

Ahora mismo, en nuestro código, no podemos crear Ciclistas sin una bicicleta.

Para que los objetos se puedan recibir en los códigos necesitamos un punto de inicio

De creación de objetos llamado **Container**

Este es el concepto de **Inversión de Control (IoC)**

```
public class Container {
    BicicletaCarreras bici = new BicicletaCarreras();
    Services.AddTransient(bici);
}
```

En Mvc Net Core tenemos un contenedor ya definido: **Program** y es el encargado

De crear los servicios para nuestra App

¿Es lo mismo una clase Bicicleta que una clase BicicletaCarreras?

Estos objetos son distintos aunque tengan las mismas propiedades.

```
BicicletaCarreras bici1 = new Bicicleta();
Bicicleta bici2 = new Bicicleta();
Podríamos solucionar esto mediante la Herencia. Si un objeto hereda de otro
es el mismo objeto, pero no tendrá las mismas características.
```

Necesitamos **obligar** a que los objetos tengan los mismos atributos, con independencia
Del tipo de objeto que sea

Para ello se utilizan las **INTERFACES**.

Una interface es un molde de una clase.

Solamente define una serie de propiedades y métodos, pero solamente su definición,

No tiene más código.

Una **interface** obliga a que las clases que hereden tengan que implementar sus métodos y
Propiedades.

MOLDE DE BICICLETA

```
public interface IBicicleta {
    string marca;
    void Acelerar();
}
```

Las clases que heredan están obligadas a implementar esas características:

```
public class BicicletaCarreras: IBicicleta {
    public string Marca {get;set;}
    public void Acelerar() {
        //CODIGO
    }
}
```

```
public class BicicletaMontaña: IBicicleta {
    public string Marca {get;set;}
    public void Acelerar() {
        //CODIGO
    }
}
```

```

IBicicleta bici1 = new BicicletaCarreras();
IBicicleta bici2 = new BicicletaMontaña();

CLASE CICLISTA
public class Ciclista {
    IBicicleta bici;

    public Ciclista(IBicicleta biciRecibida) {
        this.bici = biciRecibida;
    }
}

public class Container {
    IBicicleta bici = new BicicletaCarreras();
    Services.AddTransient(bici);
}

```

Para probar esto vamos a crear un nuevo proyecto llamado `NetCoreLinqToSqlInjection`

Vamos a comenzar con la inyección de un objeto Coche sobre un Controller

Dependency Injection

Utilitario Utilitario



Velocidad actual: 10 Km/h
Velocidad Máxima: 120 Km/h

[Acelerar](#) [Frenar](#)

Creamos una carpeta llamada `images` dentro de `wwwroot` y copiamos alguna imagen de Coches

Sobre la carpeta `Models` creamos una nueva clase llamada `Coche`

COCHE

```

public class Coche
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
    public string Imagen { get; set; }
    public int Velocidad { get; set; }
    public int VelocidadMaxima { get; set; }
    public Coche()
    {
        this.Marca = "Utilitario";
        this.Modelo = "Utilitario";
        this.Imagen = "utilitario.jpg";
        this.Velocidad = 0;
        this.VelocidadMaxima = 120;
    }

    public void Acelerar()
    {
        this.Velocidad += 10;
        if (this.Velocidad >= this.VelocidadMaxima)
        {
            this.Velocidad = this.VelocidadMaxima;
        }
    }

    public void Frenar()
    {
        this.Velocidad -= 10;
        if (this.Velocidad < 0)
        {
            this.Velocidad = 0;
        }
    }
}

```

Sobre Controllers creamos un nuevo controlador llamado `CochesController`

COCHESCONTROLLER

```

public class CochesController : Controller
{
    private Coche car;

    public CochesController()
    {

```

```

        this.car = new Coche();
    }

    public IActionResult Index()
    {
        return View(this.car);
    }

    [HttpPost]
    public IActionResult Index(string accion)
    {
        if (accion.ToLower() == "acelerar")
        {
            this.car.Acelerar();
        }
        else
        {
            this.car.Frenar();
        }
        return View(this.car);
    }
}

```

Creamos una carpeta llamada **Coches** dentro de **Views** y la vista **Index.cshtml**

INDEX.CSHTML

```

@model Coche

<h1>Dependency Injection</h1>

<h1 style="color:blue">
    @Model.Marca @Model.Modelo
</h1>


<h2>
    Velocidad actual:
    <span style="color:green">@Model.Velocidad</span> Km/h
</h2>
<h2>Velocidad Máxima:
    <span style="color:red">@Model.VelocidadMaxima</span> Km/h
</h2>

<form method="post">
    <button name="accion" value="acelerar" class="btn btn-outline-info">
        Acelerar
    </button>
    <button name="accion" value="frenar" class="btn btn-outline-danger">
        Frenar
    </button>
</form>

```

Una vez que hemos visto que es funcional vamos a probar este concepto con **Dependency Injection**

Lo primero será injectar, dentro de **CochesController** el objeto **Coche** para que la Clase lo reciba

COCHESCONTROLLER

```

public class CochesController : Controller
{
    private Coche car;

    0 references
    public CochesController(Coche car)
    {
        this.car = car;
    }
}

```

Al ejecutar nos dará un Error indicando que no puede resolver el Servicio (Coche)

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'NetCoreLinqToSqlInjection.Models.Coche' while attempting to activate 'NetCoreLinqToSqlInjection.Controllers.CochesController'.

Microsoft.Extensions.DependencyInjection.ActivatorUtilities.ThrowHelperUnableToResolveService(Type type, Type requiredBy)

Stack **Query** **Cookies** **Headers** **Routing**

InvalidOperationException: Unable to resolve service for type 'NetCoreLinqToSqlInjection.Models.Coche' while attempting to activate 'NetCoreLinqToSqlInjection.Controllers.CochesController'.

El siguiente concepto es resolver el servicio del objeto Coche mediante **Inversión de Control (IoC)**

Tenemos dos formas de resolver los servicios:

- 1) **TRANSIENT**: Instancia el objeto cada vez que realizamos una petición al Controller
- 2) **SINGLETON**: Instancia el objeto solamente una vez en el Container

PROGRAM

```

using NetCoreLinqToSqlInjection.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

//RESOLVEMOS EL SERVICIO Coche
//builder.Services.AddTransient<Coche>();
builder.Services.AddSingleton<Coche>();

var app = builder.Build();

```

El siguiente paso será crearnos una clase **Deportivo** dentro de **Models** que heredará de **Coche**

DEPORTIVO

```

public class Deportivo: Coche
{
    0 references
    public Deportivo()
    {
        this.Marca = "Batmovil";
        this.Modelo = "Clásico";
        this.Imagen = "deportivo.png";
        this.Velocidad = 0;
        this.VelocidadMaxima = 320;
    }
}

```

Y podremos visualizar que va a ser tan sencillo como sustituir en la inyección de Programa Nuestro coche por un Deportivo.

PROGRAMA

```

using NetCoreLinqToSqlInjection.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

//RESOLVEMOS EL SERVICIO Coche
//builder.Services.AddTransient<Coche>();
//builder.Services.AddSingleton<Coche>();
builder.Services.AddSingleton<Deportivo>();

var app = builder.Build();

```

Veremos que nos está dando un error de nuevo porque no puede resolver el servicio.

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'NetCoreLinqToSqlInjection.Models.Coche' while attempting to activate 'NetCoreLinqToSqlInjection.Controllers.CochesController'.

Microsoft.Extensions.DependencyInjection.ActivatorUtilities.ThrowHelperUnableToResolveService(Type type, Type requiredBy)

[Stack](#) [Query](#) [Cookies](#) [Headers](#) [Routing](#)

InvalidOperationException: Unable to resolve service for type 'NetCoreLinqToSqlInjection.Models.Coche' while attempting to activate 'NetCoreLinqToSqlInjection.Controllers.CochesController'.

Microsoft.Extensions.DependencyInjection.ActivatorUtilities.ThrowHelperUnableToResolveService(Type type, Type requiredBy)

Aunque la clase **Deportivo** sea un **Coche**, lo que está esperando el constructor es un Objeto de la clase **Coche** de forma explícita

```

public CochesController(Coche car)
{
    this.car = car;
}

```

Necesitamos que se pueda recibir cualquier objeto Coche, sin importar su Forma.
Ahora mismo, nuestro objeto Deportivo es "modificable", es decir, podríamos
Quitar el método Acelerar()

Tenemos que **obligar** a que todos los coches tengan los mismos atributos.

Para poder obligar se utilizan las **INTERFACES** que son los moldes para las clases.

Vamos a crear una nueva interface llamada **Icoche** dentro de **Models**

ICOCHE

```

public interface ICoché
{
    //LAS INTERFACES NO TIENEN AMBITO NI CODIGO
    //EN SUS PROPIEDADES/METODOS.
    //SOLO DECLARACIONES
    0 references
    string Marca { get; set; }
    0 references
    string Modelo { get; set; }
    0 references
    string Imagen { get; set; }
    0 references
    int Velocidad { get; set; }
    0 references
    int VelocidadMaxima { get; set; }
    0 references
    void Acelerar();
    0 references
    void Frenar();
}

```

El siguiente paso será que las clases que deseemos hereden de la **Interface**

COCHE

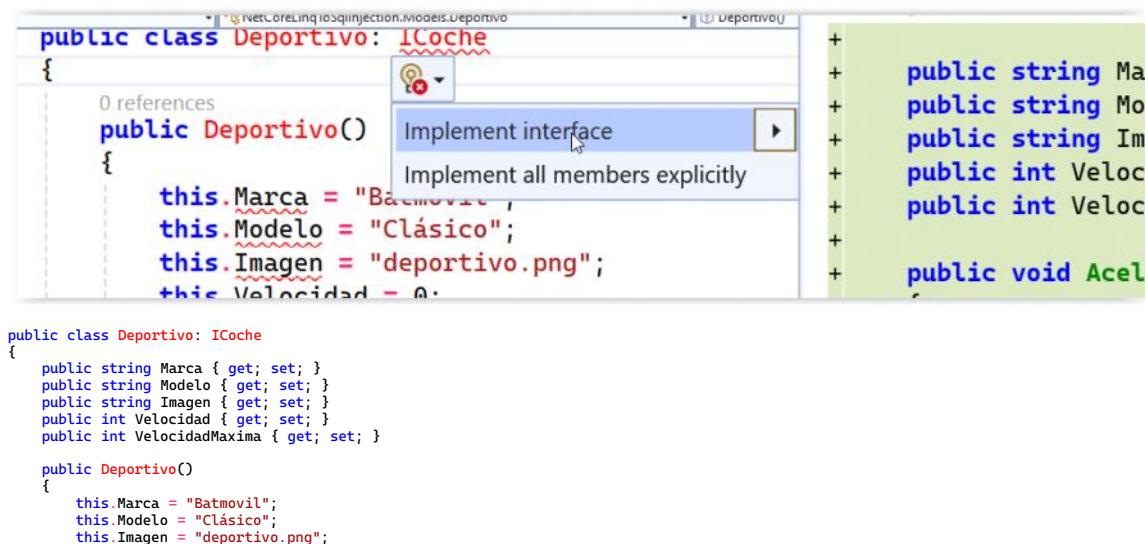
```

public class Coche: ICoché
{
    4 references
    public string Marca { get; set; }
    4 references
    public string Modelo { get; set; }
    4 references
    public string Imagen { get; set; }
    10 references
}

```

En todas las clases que hereden de una interface estamos obligados a implementar sus métodos
Y propiedades

DEPORTIVO



```

        this.Velocidad = 0;
        this.VelocidadMaxima = 320;
    }

    public void Acelerar()
    {
        this.Velocidad += 30;
        if (this.Velocidad >= this.VelocidadMaxima)
        {
            this.Velocidad = this.VelocidadMaxima;
        }
    }

    public void Frenar()
    {
        this.Velocidad -= 30;
        if (this.Velocidad < 0)
        {
            this.Velocidad = 0;
        }
    }
}

```

La inyección de dependencias debe realizarse mediante **Interfaces**, es decir, lo que recibirá Nuestras clases en el constructor debe ser la Interface, no la clase.

COCHESCONTROLLER

```

public class CochesController : Controller
{
    private ICoches car;

    0 references
    public CochesController(ICoches car)
    {
        this.car = car;
    }
}

```

Por último, al resolver el servicio debemos indicar el Molde (interface) y la clase que enviamos Con dicho Molde.

```
builder.Services.AddTransient<Interface, Clase>();
```

PROGRAM

```

using NetCoreLinqToSqlInjection.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

//RESOLVEMOS EL SERVICIO Coche
//builder.Services.AddTransient<Coche>();
//builder.Services.AddSingleton<Coche>();
builder.Services.AddSingleton<ICoches, Deportivo>();

var app = builder.Build();

```

Al ejecutar, podremos visualizar otro error diferente

InvalidOperationException: The model item passed into the ViewDataDictionary is of type 'NetCoreLinqToSqlInjection.Models.Deportivo', but this ViewDataDictionary instance requires a model item of type 'NetCoreLinqToSqlInjection.Models.Coches'.

Lo que nos está diciendo es que, en la vista, no estamos recibiendo lo que debemos.

```

@model ICoches


# Dependency Injection



# @Model.Marca @Model.Modelo



```

El último concepto es poder crear objetos dentro del Container y enviarlos desde ahí.

PROGRAM

```
    //DENTRO DEL SERVICES HABEMOS UN COCHE, DEPENDIENTE DE,  
    Coche car = new Coche();  
    car.Marca = "PONTIAC";  
    car.Modelo = "RAYO";  
    car.Imagen = "rayo.jpg";  
    car.Velocidad = 0;  
    car.VelocidadMaxima = 280;  
    //PARA ENVIAR NUESTRO OBJETO PERSONALIZADO SE UTILIZA LAMBDA  
    builder.Services.AddSingleton<ICoche, Coche>(x => car);
```

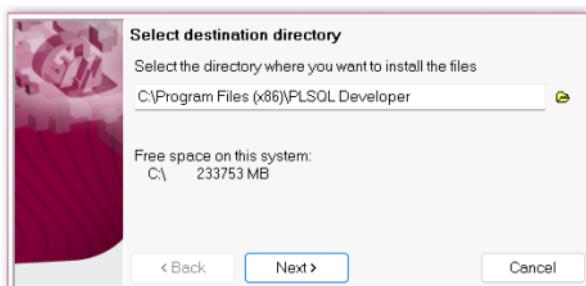
Vamos a visualizar este mismo concepto pero con acceso a datos.

Primero vamos a realizar una práctica en la que mostraremos los doctores y Podremos crear un Doctor.

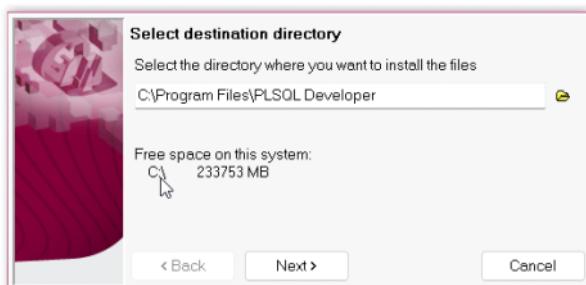
Posteriormente, debemos hacer la misma práctica, pero con Oracle.

El siguiente programa a instalar es nuestro cliente para acceder a Oracle.
PL/SQL Developer

En la instalación, lo único que tenemos que tener en cuenta es que debemos quitar los Paréntesis de (x86) de la ruta.

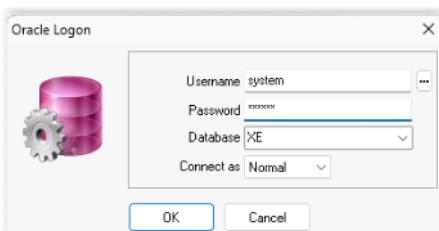


Lo dejamos así:

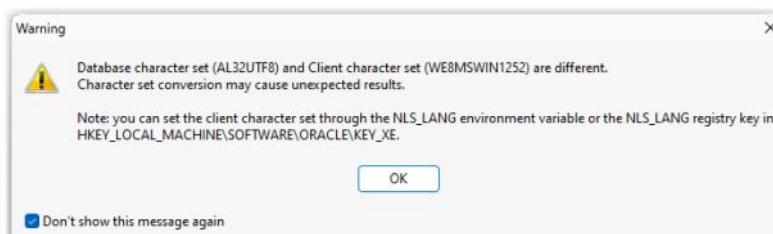


CREDENCIALES DE ORACLE

Usuario: **SYSTEM**
Password: **oracle**
BBDD: **XE**



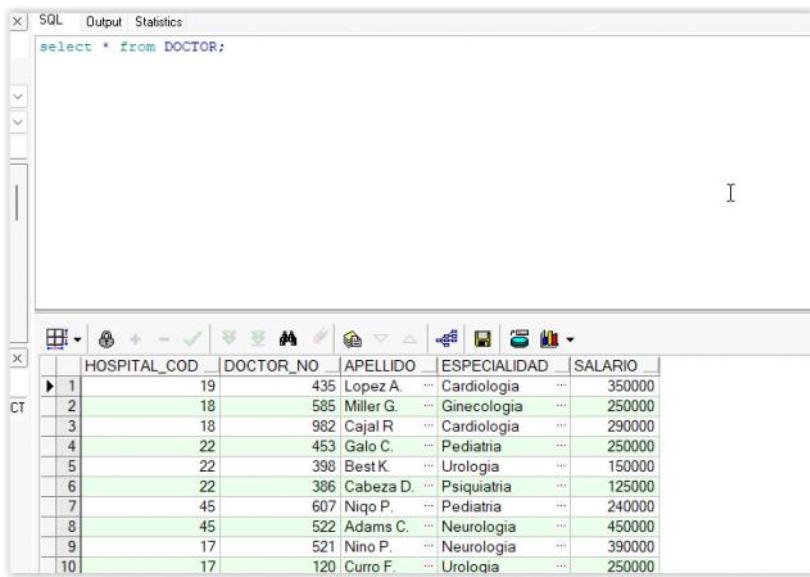
Indicamos que no queremos más mensajes



Agregamos nuestra base de datos Hospital.

Sobre PL/SQL, File --> New --> SQL Window

Copiamos nuestro fichero descargado de la base de datos y pulsamos sobre la **Rueda** o La tecla **F8**



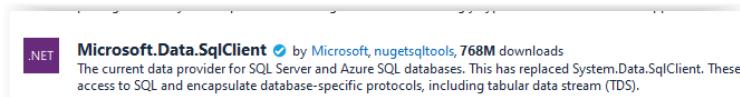
HOSPITAL_COD	DOCTOR_NO	APELLIDO	ESPECIALIDAD	SALARIO
1	19	435 Lopez A.	Cardiología	350000
2	18	585 Miller G.	Ginecología	250000
3	18	982 Cajal R.	Cardiología	290000
4	22	453 Galo C.	Pediatria	250000
5	22	398 Best K.	Urología	150000
6	22	386 Cabeza D.	Psiquiatría	125000
7	45	607 Niño P.	Pediatria	240000
8	45	522 Adams C.	Neurología	450000
9	17	521 Niño P.	Neurología	390000
10	17	120 Curro F.	Urología	250000

Sobre **Models** creamos una nueva clase llamada **Doctor**

DOCTOR

```
public class Doctor
{
    public int IdDoctor { get; set; }
    public string Apellido { get; set; }
    public string Especialidad { get; set; }
    public int Salario { get; set; }
    public int IdHospital { get; set; }
}
```

Agregamos el Nuget para acceder a SQL Server



Vamos a tener un Repositorio para Oracle y otro para SQL Server

Sobre el proyecto, creamos una nueva carpeta llamada **Repositories** y una clase llamada **RepositoryDoctoresSQLServer**

REPOSITORYDOCTORES

```
public class RepositoryDoctoresSQLServer
{
    private DataTable tableDoctores;
    private SqlConnection cn;
    private SqlCommand com;

    public RepositoryDoctoresSQLServer()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Encrypt=True;Trust Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
        this.tableDoctores = new DataTable();
        SqlDataAdapter ad = new SqlDataAdapter("select * from DOCTOR", connectionString);
        ad.Fill(this.tableDoctores);
    }

    public List<Doctor> GetDoctores()
    {
        var consulta = from datos in this.tableDoctores.AsEnumerable()
                      select datos;
        List<Doctor> doctores = new List<Doctor>();
        foreach (var item in consulta)
        {
            Doctor d = new Doctor();
            d.IdDoctor = item["IDDOCTOR"].ToString();
            d.Apellido = item["APELLIDO"].ToString();
            d.Especialidad = item["ESPECIALIDAD"].ToString();
            d.Salario = item["SALARIO"].ToString();
            d.IdHospital = item["IDHOSPITAL"].ToString();
            doctores.Add(d);
        }
        return doctores;
    }
}
```

```

        foreach (var row in consulta)
    {
        Doctor doc = new Doctor();
        doc.IdDoctor = row.Field<int>("DOCTOR_NO");
        doc.Apellido = row.Field<string>("APELIDO");
        doc.Especialidad = row.Field<string>("ESPECIALIDAD");
        doc.Salario = row.Field<int>("SALARIO");
        doc.IdHospital = row.Field<int>("HOSPITAL_COD");
        doctores.Add(doc);
    }
    return doctores;
}

public void InsertDoctor
    (int idDoctor, string apellido, string especialidad
    , int salario, int idHospital)
{
    string sql = "insert into DOCTOR values (@idhospital, @iddoctor "
    + ", @apellido, @especialidad, @salario)";
    this.com.Parameters.AddWithValue("@iddoctor", idDoctor);
    this.com.Parameters.AddWithValue("@apellido", apellido);
    this.com.Parameters.AddWithValue("@especialidad", especialidad);
    this.com.Parameters.AddWithValue("@salario", salario);
    this.com.Parameters.AddWithValue("@idhospital", idHospital);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    this.com.CommandText = sql;
    this.com.Open();
    this.com.ExecuteNonQuery();
    this.com.Close();
    this.com.Parameters.Clear();
}

```

Sobre Controllers creamos un nuevo controlador llamado **DoctoresController**

DOCTORESCONTROLLER

```

public class DoctoresController : Controller
{
    RepositoryDoctoresSQLServer repo;

    public DoctoresController(RepositoryDoctoresSQLServer repo)
    {
        this.repo = repo;
    }

    public IActionResult Index()
    {
        List<Doctor> doctores = this.repo.GetDoctores();
        return View(doctores);
    }

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

    [HttpPost]
    public IActionResult Create(Doctor doc)
    {
        this.repo.InsertDoctor(doc.IdDoctor, doc.Apellido
            , doc.Especialidad, doc.Salario, doc.IdHospital);
        return RedirectToAction("Index");
    }
}

```

Creamos una carpeta llamada **Doctores** dentro de **Views** y una vista **Index.cshtml** y otra Vista llamada **Create.cshtml**

INDEX.CSHTML

```

@model List<Doctor>

<h1>Doctores SQL Server/Oracle</h1>
<p>
    <a href="#" asp-controller="Doctores"
       asp-action="Create">
        Create New Doctor
    </a>
</p>
<table class="d-lg-table table-active">
    <thead>
        <tr>
            <th>Id doctor</th>
            <th>Apellido</th>
            <th>Especialidad</th>
            <th>Salario</th>
            <th>Hospital</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Doctor doctor in Model){
            <tr>
                <td>@doctor.IdDoctor</td>
                <td>@doctor.Apellido</td>
                <td>@doctor.Especialidad</td>
                <td>@doctor.Salario</td>
                <td>@doctor.IdHospital</td>
            </tr>
        }
    </tbody>
</table>

```

CREATE.CSHTML

```

<h1>Create</h1>
<p><a href="#" asp-controller="Doctores" asp-action="Index">
    Back to list
</a></p>
<form method="post">
    <label>Id doctor</label>
    <input type="text" name="iddoctor"
           class="form-control"/>
    <label>Apellido</label>
    <input type="text" name="apellido"
           class="form-control"/>
    <label>Especialidad</label>
    <input type="text" name="especialidad"
           class="form-control"/>

```

```

<label>Salario</label>
<input type="text" name="salario" class="form-control"/>
<label>Id hospital</label>
<input type="text" name="idhospital" class="form-control"/>
<button class="btn btn-info">Create</button>
</form>

```

El último paso que tenemos que realizar es resolver las dependencias de nuestros servicios

PROGRAM

```

using NetCoreLinqToSqlInjection.Models;
using NetCoreLinqToSqlInjection.Repositories;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddTransient<RepositoryDoctoresSQLServer>();

```

Una vez que hemos visto que es funcional, vamos a realizar exactamente lo mismo, pero accediendo a Oracle

Lo que necesitamos es poder "cambiar" de forma sencilla desde SQL Server a Oracle
En una sola linea

Las clases de acceso a datos de SQL Server y de Oracle son distintas, es decir,
Necesitamos una clase Especial para Oracle y otra clase especial para SQL Server,
Dos repos.

Esto lo podemos realizar mediante **Interfaces**

Sobre **Repositories** creamos una nueva interface llamada **IRepositoryDoctores**

Nota: Los nombres de los métodos tienen que ser iguales, además de los nombres
De los parámetros que recibamos en cada método.

```

public void InsertDoctor
    (int idDoctor, string apellido, string especialidad
     , int salario, int idHospital)

```

IREPOSITORYDOCTORES

```

0 references | 0 changes | 0 authors, 0 changes
public interface IRepositoryDoctores
{
    0 references | 0 changes | 0 authors, 0 changes
    List<Doctor> GetDoctores();

    0 references | 0 changes | 0 authors, 0 changes
    void InsertDoctor
        (int idDoctor, string apellido, string especialidad
         , int salario, int idHospital);
}

```

Indicamos, en la clase **RepositoryDoctoresSQLServer** que heredamos de dicha Interface

```

4 references | serraguti, 15 minutes ago | 1 author, 2 changes
public class RepositoryDoctoresSQLServer : IRepositoryDoctores
{

```

El **Controlador** tiene que recibir la Interface, no la clase

DOCTORESCONTROLLER

```

1 reference | 1 edit | 1 minute ago | Author: | Change
public class DoctoresController : Controller
{
    IRepositoryDoctores repo;

    0 references | 0 changes | 0 authors, 0 changes
    public DoctoresController(IRepositoryDoctores repo)
    {
        this.repo = repo;
    }
}

```

Por último, inyectamos la Interface con la clase de SQL Server

PROGRAM

```

builder.Services
    .AddTransient<IRepositoryDoctores, RepositoryDoctoresSQLServer>();

```

Connection String =
Data Source=LOCALHOST:1521/XE; Persist Security Info=True; User Id=SYSTEM; Password=oracle

Agregamos el siguiente Nuget para conectar con Oracle.



Sobre la carpeta **Repositories** creamos una nueva clase llamada **RepositoryDoctoresOracle**

REPOSITORYDOCTORESOracle

```

public class RepositoryDoctoresOracle : IRepositoryDoctores
{
    private DataTable tablaDoctores;
    private OracleConnection cn;
    private OracleCommand com;

    public RepositoryDoctoresOracle()
    {
        string connectionString =
            @"Data Source=LOCALHOST:1521/XE; Persist Security Info=True; User
Id=SYSTEM; Password=oracle";
        this.tablaDoctores = new DataTable();
        this.cn = new OracleConnection(connectionString);
        this.com = new OracleCommand();
        this.com.Connection = this.cn;
        OracleDataAdapter ad =
            new OracleDataAdapter("select * from DOCTOR", connectionString);
        ad.Fill(this.tablaDoctores);
    }

    public List<Doctor> GetDoctores()
    {
        var consulta = from datos in this.tablaDoctores.AsEnumerable()
                      select datos;
        List<Doctor> doctores = new List<Doctor>();
        foreach (var row in consulta)
        {
            Doctor doc = new Doctor();
            doc.IdDoctor = row.Field<int>("DOCTOR_NO");
            doc.Apellido = row.Field<string>("APELLIDO");
            doc.Especialidad = row.Field<string>("ESPECIALIDAD");
            doc.Salario = row.Field<int>("SALARIO");
            doc.IdHospital = row.Field<int>("HOSPITAL_COD");
            doctores.Add(doc);
        }
        return doctores;
    }

    public void InsertDoctor(int idDoctor, string apellido, string especialida
d, int salario, int idHospital)
    {
        string sql = "insert into DOCTOR values (:idhospital, :iddoctor "
                    + ", :apellido, :especialidad, :salario)";
        //ORACLE TIENE EN CUENTA EL ORDEN DE LOS PARAMETROS
        OracleParameter pamHospital = new OracleParameter(":idhospital", idhos
pital);
        this.com.Parameters.Add(pamHospital);
        OracleParameter pamIdDoctor =
            new OracleParameter(":iddoctor", idDoctor);
        this.com.Parameters.Add(pamIdDoctor);
        OracleParameter pamApellido =
            new OracleParameter(":apellido", apellido);
        this.com.Parameters.Add(pamApellido);
        OracleParameter pamEspecialidad =
            new OracleParameter(":especialidad", especialidad);
        this.com.Parameters.Add(pamEspecialidad);
        OracleParameter pamSalario = new OracleParameter(":salario", salario);
        this.com.Parameters.Add(pamSalario);

        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        this.cn.Open();
        this.com.ExecuteNonQuery();
        this.cn.Close();
        this.com.Parameters.Clear();
    }
}

```

Modificamos la inyección dentro de Program

```
builder.Services
    .AddTransient< IRepositoryDoctores , RepositoryDoctoresOracle>();
```

El siguiente paso será crear una funcionalidad para Eliminar doctores.

Vamos a realizar dicha funcionalidad con Procedimientos almacenados.

SQL SERVER

```
create procedure SP_DELETE_DOCTOR
(@iddoctor int)
as
    delete from DOCTOR where DOCTOR_NO=@iddoctor
go
```

ORACLE

Los procedimientos almacenados en Oracle deben ser de tipo consulta de acción.
Un procedimiento de Oracle NO puede ejecutar consultas SELECT.
Si que podría, pero utilizando parámetros de salida como Cursos.

La declaración de los parámetros en su tipado pueden ser estáticos (int) o indicar el
Tipo de dato asociado a la tabla y a la columna que represente.
Los nombres de los parámetros en la cabecera no llevan @ ni :

TABLA.COLUMN%TYPE

En Oracle las consultas de acción son transaccionales, es decir, solamente se ejecutan
Mientras al sesión esté abierta. Si cerramos la sesión, no se guardan los cambios.
Para que se almacenen los cambios, debemos utilizar la palabra **commit**

Sintaxis:

```
create or replace procedure NOMBRE_PROCEDURE
(parametros)
as
begin
    //NUESTROS CODIGOS
end;

-- LENGUAJE PL/SQL
create or replace procedure sp_delete_doctor
(p_iddoctor DOCTOR.DOCTOR_NO%TYPE)
as
begin
    delete from DOCTOR where DOCTOR_NO=p_iddoctor;
    commit;
end;
```

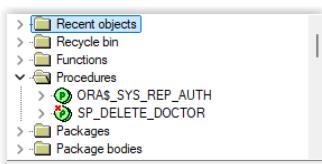
Todas las bases de datos tienen sus propias características.
Por ejemplo, Oracle, si tenemos errores en los código PL/SQL le da igual...

Debemos comprobar siempre si ha creado bien el objeto

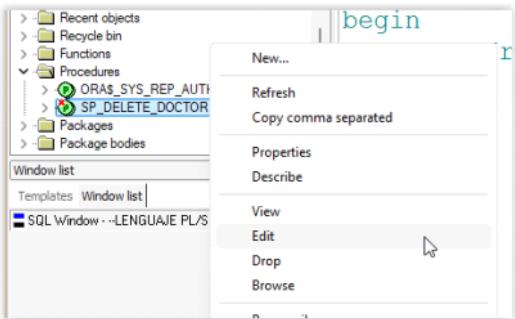
```
-- LENGUAJE PL/SQL
create or replace procedure sp_delete_doctor
(p_iddoctor DOCTOR.DOCTOR_NO%TYPE)
as
begin
    delete from DOCTOR where DOCTOR_NO=PACO;
    commit;
end;
```

En el menú de la izquierda, veremos los Procedimientos almacenados

Si el objeto tiene un aspa roja, quiere decir que no se ha creado correctamente



Tendremos que **Editar** de nuevo el Procedimiento



En esta ventana podremos visualizar qué es lo que sucede y solucionarlo

```

sp_delete_doctor
Code section | Delete
Parameter list | Code section
1 create or replace procedure sp_delete_doctor
2   (p_iddoctor DOCTOR.DOCTOR_NO%TYPE)
3   as
4   begin
5     delete from DOCTOR where DOCTOR_NO=PACO;
6     commit;
7   end;

```

PL/SQL:ORA-00904: "PACO": invalid identifier
PL/SQL: SQL Statement ignored

Nos llevamos los Procedimientos a cada Repo correspondiente.

Debemos crear un nuevo método dentro de la Interface

IREPOSITORYDOCTORES

```

public interface IRepositoryDoctores
{
    3 references | 0 changes | 0 authors, 0 changes
    List<Doctor> GetDoctores();

    3 references | 0 changes | 0 authors, 0 changes
    void InsertDoctor
        (int idDoctor, string apellido, string especialidad
         , int salario, int idHospital);

    0 references | 0 changes | 0 authors, 0 changes
    void DeleteDoctor(int idDoctor);
}

```

Implementamos el método en cada clase **RepositoryDoctores**

REPOSITORYDOCTORESSQLSERVER

```

public void DeleteDoctor(int idDoctor)
{
    string sql = "sp_delete_doctor";
    this.com.Parameters.AddWithValue("@iddoctor", idDoctor);
    this.com.CommandType = CommandType.StoredProcedure;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}

```

REPOSITORYDOCTORESORACLE

```

public void DeleteDoctor(int idDoctor)
{
    string sql = "sp_delete_doctor";
    OracleParameter pamId =
        new OracleParameter(":p_iddoctor", idDoctor);
    this.com.Parameters.Add(pamId);
    this.com.CommandType = CommandType.StoredProcedure;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}

```

Implementamos la funcionalidad dentro de nuestro Controlador

DOCTORESCONTROLLER

```

public IActionResult Delete(int iddoctor)
{
    this.repo.DeleteDoctor(iddoctor);
    return RedirectToAction("Index");
}

```

Realizamos el dibujo dentro de Index.cshtml

INDEX.CSHTML

```

<td>
    <a asp-controller="Doctores"
        asp-action="Delete"
        asp-route-iddoctor="@doctor.IdDoctor"
        class="btn btn-danger">
        Delete
    </a>
</td>

```

El siguiente paso es realizar dos implementaciones más en SQL Server y en Oracle:

- 1) UpdateDoctores mediante un procedimiento almacenado.
- 2) Buscar doctores por su especialidad con Linq.

ORACLE

```

create or replace procedure sp_update_doctor
(p_iddoctor DOCTOR.DOCTOR_NO%TYPE
,p_apellido DOCTOR.APELLIDO%TYPE
,p_especialidad DOCTOR.ESPECIALIDAD%TYPE
,p_salario DOCTOR.SALARIO%TYPE
,p_idhospital DOCTOR.HOSPITAL_COD%TYPE)
as
begin
update DOCTOR set APELLIDO=p_apellido, ESPECIALIDAD=p_especialidad
,SALARIO=p_salario, HOSPITAL_COD=p_idhospital
where DOCTOR_NO=p_iddoctor;
commit;
end;

```

SQL SERVER

```

create procedure SP_UPDATE_DOCTOR
(@iddoctor int, @apellido nvarchar(50)
,@especialidad nvarchar(50)
,@salario int, @idhospital int)
as
    update DOCTOR set APELLIDO=@apellido, ESPECIALIDAD=@especialidad
    , SALARIO=@salario, HOSPITAL_COD=@idhospital
    where DOCTOR_NO=@iddoctor
    commit;
go

```

Implementamos la Interface

IRepositoryDoctores

```

public interface IRepositoryDoctores
{
    List<Doctor> GetDoctores();

    void InsertDoctor
        (int idDoctor, string apellido, string especialidad
        , int salario, int idHospital);
}

```

```

    void DeleteDoctor(int idDoctor);
    List<Doctor> GetDoctoresEspecialidad(string especialidad);
    void UpdateDoctor(int idDoctor, string apellido
                      , string especialidad, int salario, int idHospital);
    Doctor FindDoctor(int idDoctor);
}

```

REPOSITORYDOCTORESSQLSERVER

```

public void UpdateDoctor(int idDoctor, string apellido
                         , string especialidad, int salario, int idHospital)
{
    string sql = "SP_UPDATE_DOCTOR";
    this.com.Parameters.AddWithValue("@iddoctor", idDoctor);
    this.com.Parameters.AddWithValue("@apellido", apellido);
    this.com.Parameters.AddWithValue("@especialidad", especialidad);
    this.com.Parameters.AddWithValue("@salario", salario);
    this.com.Parameters.AddWithValue("@idhospital", idHospital);
    this.com.CommandType = CommandType.StoredProcedure;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}

public Doctor FindDoctor(int idDoctor)
{
    var consulta = from datos in this.tableDoctores.AsEnumerable()
                  where datos.Field<int>("DOCTOR_NO") == idDoctor
                  select datos;
    var row = consulta.First();
    Doctor doc = new Doctor();
    doc.IdDoctor = row.Field<int>("DOCTOR_NO");
    doc.Apellido = row.Field<string>("APELLIDO");
    doc.Especialidad = row.Field<string>("ESPECIALIDAD");
    doc.Salario = row.Field<int>("SALARIO");
    doc.IdHospital = row.Field<int>("HOSPITAL_COD");
    return doc;
}

```

REPOSITORYDOCTORESORACLE

```

public void UpdateDoctor(int idDoctor, string apellido
                         , string especialidad, int salario, int idHospital)
{
    string sql = "sp_update_doctor";
    OracleParameter pamId =
        new OracleParameter(":p_iddoctor", idDoctor);
    this.com.Parameters.Add(pamId);
    OracleParameter pamApe =
        new OracleParameter(":p_apellido", apellido);
    this.com.Parameters.Add(pamApe);
    OracleParameter pamEspe =
        new OracleParameter(":p_especialidad", especialidad);
    this.com.Parameters.Add(pamEspe);
    OracleParameter pamSal = new OracleParameter(":p_salario", salario);
    this.com.Parameters.Add(pamSal);
    OracleParameter pamHosp =
        new OracleParameter(":p_idhospital", idHospital);
    this.com.Parameters.Add(pamHosp);
    this.com.CommandType = CommandType.StoredProcedure;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}

public Doctor FindDoctor(int idDoctor)
{
    var consulta = from datos in this.tablaDoctores.AsEnumerable()
                  where datos.Field<int>("DOCTOR_NO") == idDoctor
                  select datos;
    var row = consulta.First();
    Doctor doc = new Doctor();
    doc.IdDoctor = row.Field<int>("DOCTOR_NO");
    doc.Apellido = row.Field<string>("APELLIDO");
    doc.Especialidad = row.Field<string>("ESPECIALIDAD");
    doc.Salario = row.Field<int>("SALARIO");
    doc.IdHospital = row.Field<int>("HOSPITAL_COD");
    return doc;
}

```

Implementamos las acciones dentro del Controlador

DOCTORESCONTROLLER

```

public IActionResult Edit(int iddoctor)
{
    Doctor doctor = this.repo.FindDoctor(iddoctor);
    return View(doctor);
}

[HttpPost]
public IActionResult Edit(Doctor doctor)
{
    this.repo.UpdateDoctor(doctor.IdDoctor
                          , doctor.Apellido, doctor.Especialidad
                          , doctor.Salario, doctor.IdHospital);
    return RedirectToAction("Index");
}

```

EDIT.CSHTML

```

@model Doctor
<h1 style="color:blue">Edit</h1>
<p>
    <a asp-controller="Doctores" asp-action="Index">
        Back to list
    </a>
</p>

```

```

<form method="post">
    <input type="hidden" name="iddoctor" value="@Model.IdDoctor"
           class="form-control" />
    <label>Apellido</label>
    <input type="text" name="apellido" value="@Model.Apellido"
           class="form-control" />
    <label>Especialidad</label>
    <input type="text" name="especialidad" value="@Model.Especialidad"
           class="form-control" />
    <label>Salario</label>
    <input type="text" name="salario" value="@Model.Salario"
           class="form-control" />
    <label>Id hospital</label>
    <input type="text" name="idhospital" value="@Model.IdHospital"
           class="form-control" />
    <button class="btn btn-info">Update</button>
</form>

```

En la vista **Index.cshtml** incluimos el Link para **Edit.cshtml**

INDEX.CSHTML

```

<a asp-controller="Doctores"
    asp-action="Edit"
    asp-route-iddoctor="@doctor.IdDoctor"
    class="btn btn-info">
    Edit
</a>

```

A continuación, implementamos la búsqueda de doctores por especialidad

```

public List<Doctor> GetDoctoresEspecialidad(string especialidad)
{
    var consulta = from datos in this.tableDoctores AsEnumerable()
                   where datos.Field<string>("ESPECIALIDAD") == especialidad
                   select datos;
    List<Doctor> doctores = new List<Doctor>();
    foreach (var row in consulta)
    {
        Doctor doc = new Doctor();
        doc.IdDoctor = row.Field<int>("DOCTOR_NO");
        doc.Apellido = row.Field<string>("APELLIDO");
        doc.Especialidad = row.Field<string>("ESPECIALIDAD");
        doc.Salario = row.Field<int>("SALARIO");
        doc.IdHospital = row.Field<int>("HOSPITAL_COD");
        doctores.Add(doc);
    }
    return doctores;
}

```

DOCTORESCONTROLLER

```

public IActionResult BuscarDoctores()
{
    List<Doctor> doctores = this.repo.GetDoctores();
    return View(doctores);
}

[HttpPost]
public IActionResult BuscarDoctores(string especialidad)
{
    List<Doctor> doctores =
        this.repo.GetDoctoresEspecialidad(especialidad);
    return View(doctores);
}

```

BUSCARDOCTORES.CSHTML

```

@model List<Doctor>

<h1>Buscador Doctores</h1>
<form method="post">
    <label>Especialidad</label>
    <input type="text" name="especialidad" class="form-control"/>
    <button class="btn btn-dark">
        Buscar doctores
    </button>
</form>
<table class="table table-warning">
    <thead>
        <tr>
            <th>Id doctor</th>
            <th>Apellido</th>
            <th>Especialidad</th>
            <th>Salario</th>
            <th>Hospital</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Doctor doctor in Model)
        {
            <tr>
                <td>@doctor.IdDoctor</td>
                <td>@doctor.Apellido</td>
                <td>@doctor.Especialidad</td>
                <td>@doctor.Salario</td>
                <td>@doctor.IdHospital</td>
            </tr>
        }
    </tbody>
</table>

```

```
    {
        var consulta = from datos in this.tablaDoctores.AsEnumerable()
where (datos.Field<string>("ESPECIALIDAD")).ToUpper()
        == especialidad.ToUpper()
        select datos;
    }
```

ENTITY FRAMEWORK

miércoles, 12 de febrero de 2025 9:10

EF es la tecnología más moderna de acceso a datos.
Es el acceso a datos puro utilizando Linq.
EF depende del fabricante, es decir, que la base de datos que estamos consumiendo
Tenga soporte para EF, si no lo tiene, tenemos que bajar al barro y volver a Ado.

Es transparente para el programador, ya no vamos a utilizar consultas SQL, directamente,
Mediante LINQ, se harán las consultas y recuperaremos los datos.

También tenemos consultas de Acción dentro de EF.

Necesitamos montar una serie de características en nuestro proyecto

- 1) Inyección de dependencias
- 2) Modelos de la base de datos **mapeados**
- 3) Objeto **Context** que contendrá los modelos mapeados
- 4) Repository para las consultas Linq sobre el objeto Context

Necesitamos una Nuget llamado **EntityFrameworkCore**

Cuando trabajamos con Net Core estamos utilizando Nuget propios de Net Core y

Dependemos de la versión.

Ahora mismo estamos trabajando con la versión 9 de Net Core.

Los Nuget que descargamos están en dicha versión (actual)

Tenemos que utilizar los Nuget Core de la versión en la que estamos trabajando

Necesitamos mapear los modelos con las siguientes decoraciones:

- **[Table]**: Indica el nombre la tabla del Modelo
- **[Column]**: El nombre de la columna asociada a la propiedad
- **[Key]**: Es la columna PK de la tabla asociada
- **[DatabaseGenerated]**: Indica si la tabla tiene en su PK un campo de Autoincremento

Los modelos estarán mapeados dentro de un **Context** mediante colecciones de tipo
DbSet<T>

Creamos un nuevo proyecto llamado **MvcCoreEF**

The screenshot shows the NuGet Package Manager interface. It displays two packages:
Microsoft.EntityFrameworkCore (version 9.0.2) by aspnet, dotnetframework, EntityFra...
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works wit...
Microsoft.EntityFrameworkCore.SqlServer (version 9.0.2) by aspnet, dotnetframew...
Microsoft SQL Server database provider for Entity Framework Core.

Los Models deben tener el mismo tipo de dato que las columnas asociadas.

Sobre **Models** creamos una nueva clase llamada **Hospital**

HOSPITAL

```
[Table("HOSPITAL")]
public class Hospital
{
    [Key]
    [Column("HOSPITAL_COD")]
    public int IdHospital { get; set; }
    [Column("NOMBRE")]
    public string Nombre { get; set; }
    [Column("DIRECCION")]
    public string Dirección { get; set; }
    [Column("TELEFONO")]
    public string Telefono { get; set; }
    [Column("NUM_CAMA")]
    public int Camas { get; set; }
}
```

El siguiente paso es crear un **Context** que es el objeto que se comunica con la base de datos
A través de los modelos.

Sobre el proyecto, creamos una nueva carpeta llamada **Data** y una clase llamada **HospitalContext**

HOSPITALCONTEXT

```

public class HospitalContext: DbContext
{
    //TENDREMOS UN CONSTRUCTOR QUE RECIBIRA LAS
    //OPCIONES DE INICIO PARA EL CONTEXT, COMO LA
    //CADENA DE CONEXION POR EJEMPLO
    0 references
    public HospitalContext(DbContextOptions<HospitalContext> options)
        :base(options)
    { }
    //EN ESTA CLASE ESTARAN LAS COLECCIONES DE LOS MODELOS
    //QUE SERAN LAS QUE UTILIZAREMOS MEDIANTE LINQ
    0 references
    public DbSet<Hospital> Hospitales { get; set; }
}

```

Sobre el proyecto, creamos una nueva carpeta llamada **Repositories** y una clase
Llamada **RepositoryHospital**

REPOSITORYHOSPITAL

```

public class RepositoryHospital
{
    private HospitalContext context;
    public RepositoryHospital(HospitalContext context)
    {
        this.context = context;
    }
    public List<Hospital> GetHospitales()
    {
        var consulta = from datos in this.context.Hospitales
                      select datos;
        return consulta.ToList();
    }
}

```

Sobre **Controllers** creamos un nuevo controlador llamado **HospitalesController**

HOSPITALESCONTROLLER

```

public class HospitalesController : Controller
{
    private RepositoryHospital repo;
    public HospitalesController(RepositoryHospital repo)
    {
        this.repo = repo;
    }
    public IActionResult Index()
    {
        List<Hospital> hospitales = this.repo.GetHospitales();
        return View(hospitales);
    }
}

```

El siguiente paso es ir a **Program** y resolver las dependencias del Repo y del Context

En EF las inyecciones deben ser **Transient** a la fuerza

PROGRAM

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddTransient<RepositoryHospital>();
string connectionString = @"Data Source=LOCALHOST\DESARROLLO;Initial Ca
//PARA INYECTAR UN SERVICIO CONTEXT SE UTILIZA EL METODO
//AddDbContext CON LAS OPCIONES QUE NECESITE LA BBDD
builder.Services.AddDbContext<HospitalContext>
    Options => options.UseSqlServer(connectionString));

var app = builder.Build();

```

Sobre **Views/Hospitales** creamos una nueva vista llamada **Index.cshtml**

INDEX.CSHTML

```

@model List<Hospital>
<h1>Hospitales EF</h1>
<table class="table table-striped">
    <thead>

```

```

<tr>
    <th>Id</th>
    <th>Nombre</th>
    <th>Dirección</th>
    <th>Teléfono</th>
    <th>Camas</th>
</tr>
</thead>
<tbody>
    @foreach (Hospital hospital in Model){
        <tr>
            <td>@hospital.IdHospital</td>
            <td>@hospital.Nombre</td>
            <td>@hospital.Direccion</td>
            <td>@hospital.Telefono</td>
            <td>@hospital.Camas</td>
        </tr>
    }
</tbody>
</table>

```

El siguiente paso es descentralizar nuestra cadena de conexión, es decir, incluir nuestra Cadena de conexión dentro de **Settings**.

Por suerte, en este tipo de proyectos tenemos el fichero de settings nativo, es decir, No tenemos que indicar dónde está el fichero ni nada, ya viene dentro del servidor asociado.

APPSETTINGS.JSON

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Ini"
  }
}

```

Ahora mismo tenemos dentro de un objeto llamado **IConfiguration** las claves que hemos puesto dentro de **appsettings.json**

Dicho objeto ya está inyectado en el proyecto, podríamos recuperarlo en cualquier clase que tengamos.

Si lo recuperamos desde **Program** se realiza mediante el objeto **builder**

PROGRAM

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddTransient<RepositoryHospital>();
string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital");
//PARA INYECTAR UN SERVICIO CONTEXT SE UTILIZA EL METODO
//AddDbContext CON LAS OPCIONES QUE NECESITE LA BBDD
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseSqlServer(connectionString));

```

Por supuesto, vamos a utilizar consultas asíncronas.

Para realizar consultas asíncronas se realiza con un método llamado **ToListAsync()** de Entity Framework Core
Modificamos nuestro método dentro del **RepositoryHospital**

REPOSITORYHOSPITAL

```

public async Task<List<Hospital>> GetHospitalesAsync()
{
    var consulta = from datos in this.context.Hospitales
                  select datos;
    return await consulta.ToListAsync();
}

```

Modificamos nuestro Controller también

```

public async Task<IActionResult> Index()
{
    List<Hospital> hospitales =
        await this.repo.GetHospitalesAsync();
    return View(hospitales);
}

```

CONSULTAS DE ACCIÓN ENTITY FRAMEWORK

No todas las bases de datos se comportan igual.
Si realizamos consultas de acción, no es obligatorio, pero deberíamos tener una PK
Dentro de la tabla de la base de datos.

Si tenemos una PK doble, debemos declararla dentro del **Context**, no se realiza mediante Decoraciones [**Key**]

Ya no estamos trabajando con consultas SQL, estamos trabajando con clases que son Mapeadas hacia la base de datos.

Las consultas de acción se realizan sobre dichas clases a través del **Context** y de las Colecciones **DbSet<T>**

Las consultas de acción son temporales, no irán nunca a la base de datos hasta que se lo Indiquemos (**commit**)

Para poder finalizar una consulta de acción debemos hacerlo mediante un método Llamado **SaveChanges()** del objeto **Context**

INSERTAR

```

ClaseModel model = new ClaseModel();
Model.Propiedad1= valor1;
Model.Propiedad2 = valor;
Context.DbSet.Add(ClaseModel);
Context.SaveChanges();

```

MODIFICAR

```

ClaseModel model = Context.DbSet.BuscaElModelo();
Model.Propiedad2 = cambiando;
Context.SaveChanges();

```

ELIMINAR

```

ClaseModel model = Context.DbSet.BuscaElModelo();
Context.DbSet.Remove(model);
Context.SaveChanges();

```

Hospitales EF

[New Hospital](#)

Id	Nombre	Dirección	Teléfono	Camas			
19	Provinciales	O' Donell 50	964-4256	502	Details	Delete	Edit
18	General	Atocha s/n	595-3111	987	Details	Delete	Edit
22	La Paz	Castellana 1000	923-5411	412	Details	Delete	Edit
45	San Carlos	Ciudad Universitaria	597-1500	845	Details	Delete	Edit
17	Ruberico	Juan Bravo, 51	91-4027100	217	Details	Delete	Edit

Comenzamos creando un método para buscar Hospitales por su ID dentro de **RepositoryHospital**

REPOSITORYHOSPITAL

```
public async Task<Hospital> FindHospitalAsync(int idHospital)
{
    var consulta = from datos in this.context.Hospitales
                  where datos.IdHospital == idHospital
                  select datos;
    //CUANDO ESTAMOS BUSCANDO ALGO, QUE HEMOS APRENDIDO?
    //SI NO ENCUENTRA ALGO, DEBE DEVOLVER UN NULL
    if (consulta.Count() == 0)
    {
        return null;
    }
    else
    {
        return await consulta.FirstAsync();
    }
}
```

Tenemos un método llamado **FirstOrDefaultAsync()**

El método **FirstOrDefault** devuelve el primer elemento y, si no existen elementos, Devuelve el valor por defecto de la búsqueda

```
public async Task<Hospital> FindHospitalAsync(int idHospital)
{
    var consulta = from datos in this.context.Hospitales
                  where datos.IdHospital == idHospital
                  select datos;
    return await consulta.FirstOrDefaultAsync();
}
```

Sobre **HospitalesController** implementamos el método **Details()**

HOSPITALESCONTROLLER

```

public async Task<IActionResult> Details(int idhospital)
{
    Hospital hospital =
        await this.repo.FindHospitalAsync(idhospital);
    return View(hospital);
}

```

Creamos una nueva vista llamada Details.cshtml

DETAILS.CSHTML

```

@model Hospital
<h1 style="color:blue">Details</h1>
<p>
    <a asp-controller="Hospitales"
       asp-action="Index">Back to list</a>
</p>
<ul class="list-group">
    <li class="list-group-item">
        Id hospital @Model.IdHospital
    </li>
    <li class="list-group-item">
        Nombre @Model.Nombre
    </li>
    <li class="list-group-item">
        Dirección @Model.Direccion
    </li>
    <li class="list-group-item">
        Teléfono @Model.Telefono
    </li>
    <li class="list-group-item">
        Camas @Model.Camas
    </li>
</ul>

```

Sobre Index.cshtml incluimos un nuevo Link para mostrar los detalles

INDEX.CSHTML

```

<td>
    <a asp-controller="Hospitales"
       asp-action="Details"
       asp-route-idhospital="@hospital.IdHospital"
       class="btn btn-warning">
        Details
    </a>
</td>

```

El siguiente paso será realizar un INSERT.

Creamos un nuevo método dentro de RepositoryHospital

REPOSITORYHOSPITAL

```

public async Task
    InsertHospitalAsync(int idHospital, string nombre
        , string direccion, string telefono, int camas)
{
    //CREAMOS UN MODEL
    Hospital hospital = new Hospital();
    //ASIGNAMOS SUS PROPIEDADES
    hospital.IdHospital = idHospital;
    hospital.Nombre = nombre;
    hospital.Direccion = direccion;
    hospital.Telefono = telefono;
    hospital.Camas = camas;
    //AÑADIMOS NUESTRO MODEL A LA COLECCION DBSET DEL CONTEXT
    await this.context.Hospitales.AddAsync(hospital);
    //INDICAMOS QUE ALMACENE LOS DATOS EN LA BBDD
    await this.context.SaveChangesAsync();
}

```

Sobre nuestro Controller creamos un nuevo método para insertar

HOSPITALESCONTROLLER

```

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

[HttpPost]
0 references
public async Task<IActionResult> Create(Hospital hospital)
{
    await this.repo.InsertHospitalAsync
        (hospital.IdHospital, hospital.Nombre
         , hospital.Direccion, hospital.Telefono, hospital.Camas);
    return RedirectToAction("Index");
}

```

Creamos una nueva vista llamada Create.cshtml

CREATE.CSHTML

```

<h1>Create</h1>
<p>
    <a asp-controller="Hospitales"
       asp-action="Index">
        Back to Index</a>
</p>
<form method="post">
    <label>Id hospital</label>
    <input type="text" name="idhospital"
           class="form-control"/>
    <label>Nombre</label>
    <input type="text" name="nombre"
           class="form-control"/>
    <label>Dirección</label>
    <input type="text" name="direccion"
           class="form-control"/>
    <label>Teléfono</label>
    <input type="text" name="telefono"
           class="form-control"/>
    <label>Camas</label>
    <input type="text" name="camas"
           class="form-control"/>
    <button class="btn btn-dark">Create</button>
</form>

```

A continuación, vamos a realizar la funcionalidad para Delete

REPOSITORYHOSPITAL

```

0 references
public async Task DeleteHospitalAsync(int idHospital)
{
    //BUSCAMOS EL MODEL PARA ELIMINARLO
    Hospital hospital =
        await this.FindHospitalAsync(idHospital);
    //ELIMINAMOS DE LA COLECCION DbSet<T> DEL CONTEXT
    this.context.Hospitales.Remove(hospital);
    //ACTUALIZAMOS LA BASE DE DATOS
    await this.context.SaveChangesAsync();
}

```

Implementamos un nuevo método dentro del Controller

HOSPITALESCONTROLLER

```

0 references
public async Task<IActionResult> Delete(int idhospital)
{
    await this.repo.DeleteHospitalAsync(idhospital);
    return RedirectToAction("Index");
}

```

Ponemos un Link dentro de Index.cshtml

INDEX.CSHTML

```

<a asp-controller="Hospitales"
    asp-action="Delete"
    asp-route-idhospital="@hospital.IdHospital"
    class="btn btn-danger">
    Delete
</a>

```

Por último aplicamos la funcionalidad de Update

REPOSITORYHOSPITAL

```

public async Task
{
    UpdateHospitalAsync(int idHospital, string nombre
        , string direccion, string telefono, int camas)
    {
        //BUSCAMOS EL OBJETO HOSPITAL A MODIFICAR
        Hospital hospital =
            await this.FindHospitalAsync(idHospital);
        //PODEMOS MODIFICAR TODO LO QUE DESEEMOS EXCEPTO
        //EL CAMPO [Key]
        hospital.Nombre = nombre;
        hospital.Direccion = direccion;
        hospital.Telefono = telefono;
        hospital.Camas = camas;
        //NO TENEMOS NINGUN METODO PARA REALIZAR UN UPDATE
        //DENTRO DEL CONTEXT Y DbSet<T>
        await this.context.SaveChangesAsync();
    }
}

```

Implementar la funcionalidad dentro del Controller

HOSPITALESCONTROLLER

```

public async Task<IActionResult> Edit(int idhospital)
{
    Hospital hospital = await
        this.repo.FindHospitalAsync(idhospital);
    return View(hospital);
}

[HttpPost]
public async Task<IActionResult> Edit(Hospital hospital)
{
    await this.repo.UpdateHospitalAsync
        (hospital.IdHospital, hospital.Nombre, hospital.Direccion
        , hospital.Telefono, hospital.Camas);
    return RedirectToAction("Index");
}

```

Creamos una nueva vista llamada Edit.cshtml

EDIT.CSHTML

```

@model Hospital


# Edit



Back to Index


<form method="post">
    <input type="hidden" name="idhospital" value="@Model.IdHospital"/>
    <label>Nombre</label>
    <input type="text" name="nombre" value="@Model.Nombre"
        class="form-control" />
    <label>Dirección</label>
    <input type="text" name="direccion" value="@Model.Direccion"
        class="form-control" />
    <label>Teléfono</label>
    <input type="text" name="telefono" value="@Model.Telefono"
        class="form-control" />
    <label>Camas</label>
    ...

```

```

<input type="text" name="camas" value="@Model.Camas"
       class="form-control" />
<button class="btn btn-info">Update</button>
</form>

```

Por último, creamos un nuevo Link dentro de **Index.cshtml**

INDEX.CSHTML

```

<a asp-controller="Hospitales"
    asp-action="Edit"
    asp-route-idhospital="@hospital.IdHospital"
    class="btn btn-info">
    Edit
</a>

```

PROCEDIMIENTOS ALMACENADOS ENTITY FRAMEWORK

Para poder llamar a procedimientos almacenados dentro de EF todo depende de la base de datos

Por ejemplo, en Oracle, se utilizan cursos con parámetros de salida para poder llamar a procedimientos almacenados con SELECT.

La forma de llamar a los procedimientos se realiza de forma **manual**, es decir, creamos un código diferente para las consultas. Es muy parecido a ADO.NET.

Tenemos dos métodos para llamar a procedimientos:

- 1) **ExecuteSqlRaw**: Ejecuta una consulta de acción
- 2) **FromSqlRaw**: Ejecuta una consulta de selección

Utilizaremos los siguientes objetos:

- **DbCommand**: Ejecuta una consulta
- **DbConnection**: Nuestra conexión a la base de datos
- **DbDataReader**: Lector de consultas de selección

Vamos a realizar un proyecto nuevo donde solamente utilizaremos PROCEDIMIENTOS ALMACENADOS.

Creamos un nuevo proyecto llamado **MvcNetCoreProceduresEF**

PROCEDIMIENTOS ALMACENADOS

```

create procedure SP_TODOS_ENFERMOS
as
    select * from ENFERMO
go
create procedure SP_FIND_ENFERMO
(@inscripcion nvarchar(50))
as
    select * from ENFERMO where INSCRIPCION=@inscripcion
go
create procedure SP_DELETE_ENFERMO
(@inscripcion nvarchar(50))
as
    delete from ENFERMO where INSCRIPCION=@inscripcion
go

```

Sobre el proyecto, agregamos los siguientes Nuget



Sobre **Models** creamos una nueva clase llamada **Enfermo**

```

[Table("ENFERMO")]
public class Enfermo
{
    [Key]
    [Column("INSCRIPCION")]
    public string Inscripcion { get; set; }
    [Column("APELLIDO")]
    public string Apellido { get; set; }
    [Column("DIRECCION")]
    public string Direccion { get; set; }
    [Column("FECHA_NAC")]
    public DateTime FechaNacimiento { get; set; }
    [Column("SEXO")]
    public string Genero { get; set; }
}

```

Creamos una nueva carpeta llamada **Data** y una clase llamada **EnfermosContext**

ENFERMOSCONTEXT

```

public class EnfermosContext: DbContext
{
    0 references
    public EnfermosContext(DbContextOptions<EnfermosContext> options)
        : base(options) { }

    0 references
    public DbSet<Enfermo> Enfermos { get; set; }
}

```

Creamos una carpeta llamada **Repositories** y una clase llamada **RepositoryEnfermos**

REPOSITORYENFERMOS

```

public class RepositoryEnfermos
{
    private EnfermosContext context;

    public RepositoryEnfermos(EnfermosContext context)
    {
        this.context = context;
    }

    public async Task<List<Enfermo>> GetEnfermosAsync()
    {
        //PARA CONSULTAS DE SELECCION DEBEMOS MAPEAR
        //MANUALMENTE LOS DATOS.
        using (DbCommand com =
            this.context.Database.GetDbConnection().CreateCommand())
        {
            string sql = "SP_TODOSENFERMOS";
            com.CommandType = System.Data.CommandType.StoredProcedure;
            com.CommandText = sql;
            //ABRIMOS LA CONEXION A TRAVES DEL COMMAND
            await com.Connection.OpenAsync();
            //EJECUTAMOS NUESTRO READER
            DbDataReader reader = await com.ExecuteReaderAsync();
            List<Enfermo> enfermos = new List<Enfermo>();
            while (await reader.ReadAsync())
            {
                Enfermo enfermo = new Enfermo
                {
                    Inscripcion = reader["INSCRIPCION"].ToString(),
                    Apellido = reader["APELLIDO"].ToString(),
                    Direccion = reader["DIRECCION"].ToString(),
                    FechaNacimiento =
                        DateTime.Parse(reader["FECHA_NAC"].ToString()),
                    Genero = reader["S"].ToString()
                };
                enfermos.Add(enfermo);
            }
            await reader.CloseAsync();
            await com.Connection.CloseAsync();
            return enfermos;
        }
    }
}

```

Agregamos sobre **appsettings.json** nuestra cadena de conexión

APPSETTINGS.JSON



```

{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "SqlHospital": "Data Source=LOCALHOST\\DESARROL"
    }
}

```

Sobre **Program** resolvemos las dependencias

PROGRAM

```
// Add services to the container.
builder.Services.AddControllersWithViews();

string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital");
builder.Services.AddTransient<RepositoryEnfermos>();
builder.Services.AddDbContext<EnfermosContext>
    (options => options.UseSqlServer(connectionString));

var app = builder.Build();
```

Sobre **Controllers** creamos un nuevo controlador llamado **EnfermosController**

ENFERMOSCONTROLLER

```
public class EnfermosController : Controller
{
    private RepositoryEnfermos repo;

    public EnfermosController(RepositoryEnfermos repo)
    {
        this.repo = repo;
    }

    public async Task<IActionResult> Index()
    {
        List<Enfermo> enfermos =
            await this.repo.GetEnfermosAsync();
        return View(enfermos);
    }
}
```

VISTAS SCAFFOLDING

Nota: Esta nueva funcionalidad no la podemos utilizar con Errores en el proyecto

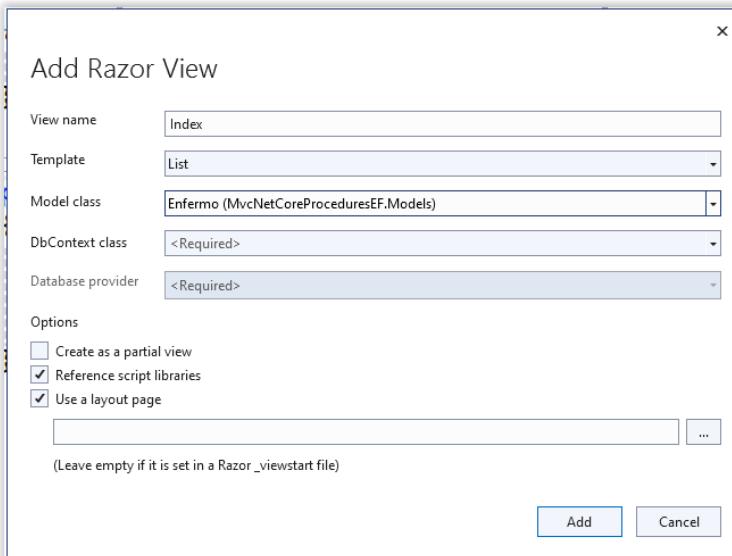
Esto es una herramienta para los desarrolladores y poder generar plantillas de una forma Sencilla.

Tenemos unas cuantas plantillas:

- **List:** Genera una vista con una colección
- **Details:** Genera una vista con un Model
- **Create:** Genera una vista con elementos para INSERT un Model
- **Edit:** Genera una vista con elementos para UPDATE de un Model

Nos sirve como pruebas para el back y generar los dibujos de forma dinámica.

Sobre el **IActionResult** del Controller de **Index**, botón derecho y **Add View**



A continuación vamos a llamar al procedimiento de Buscar un enfermo para Details

REPOSITORYENFERMOS

```
public Enfermo FindEnfermo(string inscripcion)
{
    //PARA LLAMAR A PROCEDIMIENTOS ALMACENADOS
    //CON PARAMETROS LA LLAMADA SE REALIZA MEDIANTE
```

```

//EL NOMBRE DEL PROCEDIMIENTO Y CADA PARAMETRO
//A CONTINUACION SEPARADO MEDIANTE COMAS
// SP_PROCEDIMIENTO @PARAM1, @PARAM2
string sql = "SP_FIND_ENFERMO @INSCRIPCION";
//DEBEMOS CREAR LOS PARAMETROS
SqlParameter pamInscripcion =
    new SqlParameter("@INSCRIPCION", inscripcion);
//SI LOS DATOS QUE DEVUELVE EL PROCEDIMIENTO
//ESTAN MAPEADOS CON UN MODEL, PODEMOS UTILIZAR
//EL METODO FromSqlRaw CON LINQ
//CUANDO UTILIZAMOS LINQ CON PROCEDIMIENTOS ALMACENADOS
//LA CONSULTA Y LA EXTRACCION DE DATOS SE REALIZAN EN
//DOS PASOS.
//NO SE PUEDE HACER LINQ Y EXTRAER A LA VEZ
var consulta =
    this.context.Enfermos.FromSqlRaw(sql, pamInscripcion);
//PARA EXTRAER LOS DATOS NECESITAMOS TAMBIEN
//EL METODO AsEnumerable()
Enfermo enfermo =
    consulta.AsEnumerable().FirstOrDefault();
return enfermo;
}

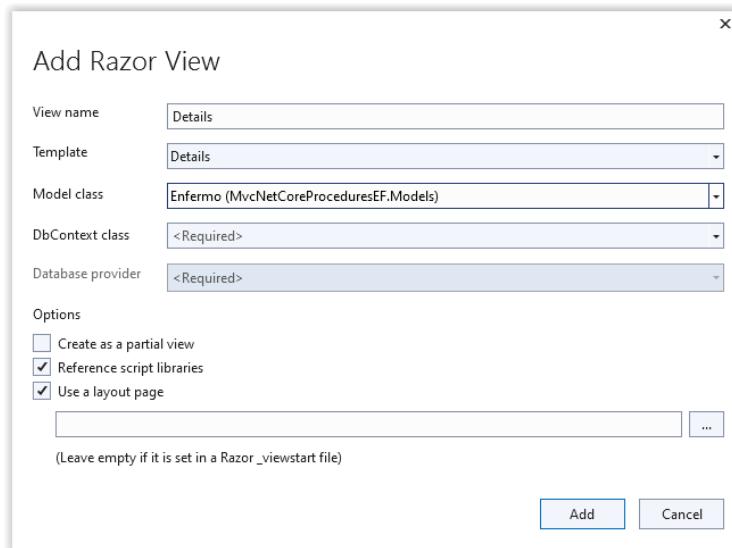
```

Agregamos un nuevo método **Details** dentro de **EnfermosController**

```

public IActionResult Details(string inscripcion)
{
    Enfermo enfermo = this.repo.FindEnfermo(inscripcion);
    return View(enfermo);
}

```



El siguiente paso será llamar a un procedimiento de acción mediante parámetros

REPOSITORYENFERMOS

Vamos a probarlo mediante **DbCommand**

```

string sql = "SP_DELETE_ENFERMO @INSCRIPCION";
SqlParameter pamInscripcion =
    new SqlParameter("@INSCRIPCION", inscripcion);
using (DbCommand com =
    this.context.Database.GetDbConnection().CreateCommand())
{
    com.CommandType = System.Data.CommandType.StoredProcedure;
    com.CommandText = sql;
    com.Parameters.Add(pamInscripcion);
    com.Connection.Open();
    com.ExecuteNonQuery();
    com.Connection.Close();
    com.Parameters.Clear();
}

```

```

        com.Connection.Open();
        com.ExecuteNonQuery(); ✖
        com.Connection.Close();
    }

Exception User-Unhandled
Microsoft.Data.SqlClient.SqlException: 'No se encontró el procedimiento almacenado 'SP_DELETE_ENFERMO @INSCRIPCION'.'

Ask Copilot | Show Call Stack | View Details | Copy Details | Start Live Share session

```

CODIGO CORRECTO

```

string sql = "SP_DELETE_ENFERMO";
SqlParameter pamInscripcion =
    new SqlParameter("@INSCRIPCION", inscripcion);
using (DbCommand com =
    this.context.Database.GetDbConnection().CreateCommand())
{
    com.CommandType = System.Data.CommandType.StoredProcedure;
    com.CommandText = sql;
    com.Parameters.Add(pamInscripcion);
    com.Connection.Open();
    com.ExecuteNonQuery();
    com.Connection.Close();
    com.Parameters.Clear();
}

```

También podemos realizarlo con un método llamado ExecuteSqlRaw

```

public void DeleteEnfermoRaw(string inscripcion)
{
    string sql = "SP_DELETE_ENFERMO @INSCRIPCION";
    SqlParameter pamInscripcion =
        new SqlParameter("@INSCRIPCION", inscripcion);
    //DENTRO DEL CONTEXT TENEMOS UN METODO PARA
    //PODER LLAMAR A PROCEDIMIENTOS DE CONSULTAS DE ACCION
    this.context.Database.ExecuteSqlRaw(sql, pamInscripcion);
}

```

Lo convertimos a asíncrono

```

public async Task DeleteEnfermoRawAsync(string inscripcion)
{
    string sql = "SP_DELETE_ENFERMO @INSCRIPCION";
    SqlParameter pamInscripcion =
        new SqlParameter("@INSCRIPCION", inscripcion);
    //DENTRO DEL CONTEXT TENEMOS UN METODO PARA
    //PODER LLAMAR A PROCEDIMIENTOS DE CONSULTAS DE ACCION
    await this.context.Database
        .ExecuteSqlRawAsync(sql, pamInscripcion);
}

```

Creamos un nuevo método **Delete** dentro de **EnfermosController**

ENFERMOSCONTROLLER

```

public IActionResult Delete(string inscripcion)
{
    this.repo.DeleteEnfermo(inscripcion);
    return RedirectToAction("Index");
}

```

Necesito un procedimiento almacenado para INSERTAR un ENFERMO.

En el procedimiento, le damos la seguridad social (NSS) manual

En el procedimiento realizamos el MAX ID de INSCRIPCION

```
create procedure SP_INSERT_ENFERMO
(@apellido nvarchar(50), @direccion nvarchar(50)
, @fechanac datetime, @genero nvarchar(1))
as
    --MAXIMO ID INSCRIPCION DEL ENFERMO
    declare @maxinscripcion int
    select @maxinscripcion = cast(max(inscripcion) as int) + 1
    from ENFERMO
    insert into ENFERMO values
    (@maxinscripcion, @apellido, @direccion,
    @fechanac, @genero, '12334')
go
```

Implementamos el método en el Repository y dibujamos la vista con Scaffolding

REPOSITORYENFERMOS

```
public async Task InsertEnfermoAsync
    (string apellido, string direccion,
    DateTime fechaNacimiento, string genero)
{
    string sql = "SP_INSERT_ENFERMO @apellido, @direccion "
        + ", @fechanac, @genero";
    SqlParameter pamApellido =
        new SqlParameter("@apellido", apellido);
    SqlParameter pamDireccion =
        new SqlParameter("@direccion", direccion);
    SqlParameter pamFecha =
        new SqlParameter("@fechanac", fechaNacimiento);
    SqlParameter pamGen =
        new SqlParameter("@genero", genero);
    await this.context.Database
        .ExecuteSqlRawAsync(sql, pamApellido, pamDireccion
        , pamFecha, pamGen);
}
```

ENFERMOSCONTROLLER

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

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public async Task<IActionResult> Create(Enfermo enfermo)
{
    await this.repo.InsertEnfermoAsync
        (enfermo.Apellido, enfermo.Direccion
        , enfermo.FechaNacimiento, enfermo.Genero);
    return RedirectToAction("Index");
}
```

Quiero FINDENFERMO de forma ASINCRONA

```
1 reference | 0 changes | 0 authors, 0 changes
public async Task<Enfermo> FindEnfermoAsync(string inscripcion)
{

public async Task<Enfermo> FindEnfermoAsync(string inscripcion)
{
    //PARA LLAMAR A PROCEDIMIENTOS ALMACENADOS
    //CON PARAMETROS LA LLAMADA SE REALIZA MEDIANTE
    //EL NOMBRE DEL PROCEDIMIENTO Y CADA PARAMETRO
    //A CONTINUACION SEPARADO MEDIANTE COMAS
    // SP_PROCEDIMIENTO @PARAM1, @PARAM2
    string sql = "SP_FIND_ENFERMO @INSCRIPCION";
    //DEBEMOS CREAR LOS PARAMETROS
    SqlParameter pamInscripcion =
        new SqlParameter("@INSCRIPCION", inscripcion);
    //SI LOS DATOS QUE DEVUELVE EL PROCEDIMIENTO
    //ESTAN MAPEADOS CON UN MODEL, PODEMOS UTILIZAR
```

```
//EL METODO FromSqlRaw CON LINQ
var consulta =
    await this.context.Enfermos.FromSqlRaw(sql, pamInscripcion)
    .ToListAsync();
Enfermo enfermo = consulta.FirstOrDefault();
return enfermo;
}
```

En esta práctica reutilizamos el Context, es decir, añadimos un nuevo DbSet

Necesito una página para incrementar el salario de los doctores por Especialidad.

Dibujaremos un desplegable con las especialidades y una caja de incremento.

Mostraremos todos los doctores al cargar la página y, al pulsar el botón de incremento, Mostramos los doctores que han sido afectados por la consulta.

IdDoctor	Apellido	Especialidad	Salario	IdHospital
585	Miller G.	Ginecología	479410	18

VISTAS CON ENTITY FRAMEWORK

Sin ninguna duda esta herramienta es algo imprescindible dentro de Cualquier desarrollo.

Las vistas nos permiten mostrar y representar los datos formateados Directamente en nuestro programa.

Las vistas necesitan de una PK para poder ser representadas dentro de EF. Una vista no tiene PK

Tenemos dos opciones:

- 1) Utilizar un campo de la consulta del Select que sea una PK
No repetida y que pongamos [Key] a dicho campo
- 2) Generamos una PK dentro de la consulta de la propia vista

Dicho campo es un campo calculado en una consulta debemos indicar Que NO contendrá nulos para que sea una PK

En este campo calculado ya depende de la propia base de datos para Indicar una función que devuelva campos no null.
En SQL Server es ISNULL
En Oracle es NVL

```
select ISNULL(EMP_NO, 0) AS IDEMPLEADO
, APELLIDO, OFICIO from EMP
```

¿Qué sucede si no tenemos campo clave en nuestra consulta select?

Lo que más se utiliza para generar campos claves no repetidos en las Consultas SELECT es representar el número de fila en la propia consulta.

En SQL Server se representa mediante la función

ROW_NUMBER OVER (ORDER BY COLUMNA)

Esta consulta debemos envolver el campo Key con ISNULL a su vez

```
select
ROW_NUMBER() over (order by APELLIDO) as ID,
APELLIDO, OFICIO from EMP
```

Vamos a crear una vista con empleados y departamentos y la Representamos en nuestro programa mediante EF



Index

Id	Apellido	Oficio	Salario	Departamento	Localidad
1	ALCALA	EMPLEADO	119000	CONTABILIDAD	ELCHE
2	ALONSO	EMPLEADO	143000	INVESTIGACION	MADRID
3	ARROYO	VENDEDOR	208000	VENTAS	BARCELONA
4	CASALES	EMPLEADO	179000	CONTABILIDAD	ELCHE

```
create view V_EMPLEADOS_DEPARTAMENTOS
as
    select ISNULL(ROW_NUMBER() over (order by APELLIDO), 0) as ID
    , EMP.APELLIDO, EMP.OFICIO, EMP.SALARIO
    , DEPT.DNOMBRE AS DEPARTAMENTO
    , DEPT.LOC AS LOCALIDAD
    from EMP
    inner join DEPT
    on EMP.DEPT_NO = DEPT.DEPT_NO
go
```

Sobre **Models** creamos un nuevo modelo llamado **VistaEmpleado**

VISTAEMPLEADO

```
[Table("V_EMPLEADOS_DEPARTAMENTOS")]
public class VistaEmpleado
{
    [Key]
    [Column("ID")]
    public int Id { get; set; }
    [Column("APELIDO")]
    public string Apellido { get; set; }
    [Column("OFICIO")]
    public string Oficio { get; set; }
    [Column("SALARIO")]
    public int Salario { get; set; }
    [Column("DEPARTAMENTO")]
    public string Departamento { get; set; }
    [Column("LOCALIDAD")]
    public string Localidad { get; set; }
}
```

Normalmente solamente se tiene un único **Context** y se van agregando Ahí todos los **DbSet** que necesitemos.

Actualmente tengo **EnfermosContext**. Modificamos el nombre de **EnfermosContext** y lo llamamos como la base de datos que vamos a Representar: **HospitalContext**

Sobre **HospitalContext** agregamos nuestro **DbSet** con la vista

HOSPITALCONTEXT

```
public class HospitalContext: DbContext
{
    0 references | 0 changes | 0 authors, 0 changes
    public HospitalContext(DbContextOptions<HospitalContext> options)
        : base(options) { }

    public DbSet<VistaEmpleado> VistasEmpleados { get; set; }
    1 reference | 0 changes | 0 authors, 0 changes
```

Creamos un nuevo repositorio sobre **Repositories** llamado

RepositoryEmpleados

REPOSITORYEMPLEADOS

```
public class RepositoryEmpleados
{
    private HospitalContext context;
    public RepositoryEmpleados(HospitalContext context)
    {
        this.context = context;
    }
    public async Task<List<VistaEmpleado>>
        GetVistaEmpleadosAsync()
    {
        var consulta = from datos in this.context.VistasEmpleados
                      select datos;
        return await consulta.ToListAsync();
    }
}
```

Sobre **Controllers** creamos un nuevo controlador llamado **EmpleadosController**

EMPLEADOSCONTROLLER

```
public class EmpleadosController : Controller
{
    private RepositoryEmpleados repo;
    public EmpleadosController(RepositoryEmpleados repo)
    {
        this.repo = repo;
    }
    public async Task<IActionResult> Index()
    {
        List<VistaEmpleado> empleados =
            await this.repo.GetVistaEmpleadosAsync();
        return View(empleados);
    }
}
```

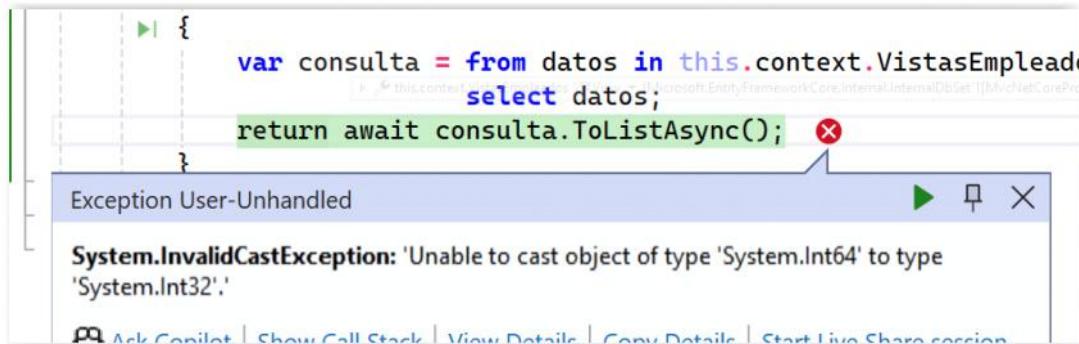
Resolvemos las dependencias dentro de **Program**

PROGRAM

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddTransient<RepositoryEmpleados>();
string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital")
```

Al ejecutar, deberíamos visualizar este error



Cuando generamos campos calculados no sabemos el tipo de datos que vamos a recuperar

Aunque pensemos que son números o textos o fechas.

Debemos indicar, de forma explícita qué tipo de datos queremos.

Tenemos dos opciones:

- 1) Modificar nuestro **Model**

```
[Key]
[Column("ID")]
2 references | 0 changes | 0 authors, 0 changes
public Int64 Id { get; set; }
```

- 2) Modificar el origen de los datos en la Vista indicando qué tipado queremos

En SQL Server tenemos una conversión llamada **CAST(COLUMNAS AS TIPO)**

```

| alter view V_EMPLEADOS_DEPARTAMENTOS
as
    select CAST(
    ISNULL(ROW_NUMBER() over (order by APELLIDO), 0) as int)
    as ID

```

PROCEDIMIENTOS ALMACENADOS PARAMETROS DE SALIDA

Los procedimientos almacenados con parámetros de salida son algo imprescindible en Determinadas situaciones.

Por ejemplo, si tenemos paginación de base de datos, serán necesarios

Si tenemos una base de datos Oracle con consultas select también será necesario

Los parámetros de salida podemos recuperarlos de dos formas diferentes:

- 1) **System.Data.Common**: DbCommand, DbConnection
- 2) **EF**: FromSqlRaw, ExecuteSqlRaw

Vamos a realizar un ejemplo con un procedimiento de salida que nos devolverá La suma, media y personas de todos los trabajadores de la empresa

ID Trabajador	Apellido	Oficio	Salario
120	Curro F.	Urología	211000
386	Cabeza D.	Psiquiatría	152000

Necesitamos una vista primero para todos los trabajadores

```

| create view V_WORKERS
as
    select EMP_NO as IDWORKER
    , APELLIDO, OFICIO, SALARIO from EMP
    union
    select DOCTOR_NO, APELLIDO, ESPECIALIDAD, SALARIO
    from DOCTOR
    union
    select EMPLEADO_NO, APELLIDO, FUNCION, SALARIO
    from PLANTILLA
go

```

Posteriormente haremos el procedimiento con los parámetros

```

create procedure SP_WORKERS_OFICIO
(@oficio nvarchar(50)
, @personas int out
, @media int out, @suma int out)
as
select * from V_WORKERS
where OFICIO=@oficio
select @personas = COUNT(IDWORKER)
, @media = AVG(SALARIO), @suma = SUM(SALARIO)
from V_WORKERS where OFICIO=@oficio
go

```

Sobre **Models** creamos una nueva vista llamada **Trabajador**

TRABAJADOR

```

[Table("V_WORKERS")]
public class Trabajador
{
    [Key]
    [Column("IDWORKER")]
    public int IdTrabajador { get; set; }
    [Column("APELLIDO")]
    public string Apellido { get; set; }
    [Column("OFICIO")]
    public string Oficio { get; set; }
    [Column("SALARIO")]
    public int Salario { get; set; }
}

```

Sobre **HospitalContext** agregamos un nuevo DbSet para esta vista

HOSPITALCONTEXT

```

public class HospitalContext: DbContext
{
    0 references | serraguti, 16 minutes ago | 1 author, 1 change
    public HospitalContext(DbContextOptions<HospitalContext> options)
        : base(options) { }
    0 references | 0 changes | 0 authors, 0 changes
    public DbSet<Trabajador> Trabajadores { get; set; }
    1 reference | serracuti 16 minutes ago | 1 author, 1 change

```

En la vista vamos a representar tanto a los trabajadores como a los Datos de su media salarial y demás.

Creamos un nuevo Model para devolver todos los datos a la vez a nuestra Vista.

Sobre **Models** creamos una nueva clase llamada **TrabajadoresModel**

```

public class TrabajadoresModel
{
    0 references | 0 changes | 0 authors, 0 changes
    public List<Trabajador> Trabajadores { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int Personas { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int SumaSalarial { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int MediaSalarial { get; set; }
}

```

Sobre **Repositories** creamos un nuevo repo llamado **RepositoryTrabajadores**

REPOSITORYTRABAJADORES

```

public class RepositoryTrabajadores
{
    private HospitalContext context;
    public RepositoryTrabajadores(HospitalContext context)
    {
        this.context = context;
    }
    public async Task<TrabajadoresModel>
        GetTrabajadoresModelAsync()
    {
        var consulta =
            from datos in this.context.Trabajadores
            select datos;
        TrabajadoresModel model = new TrabajadoresModel();
        model.Trabajadores = await consulta.ToListAsync();
    }
}

```

```

        model.Personas = await consulta.CountAsync();
        model.SumaSalarial = await consulta.SumAsync(z => z.Salario);
        model.MediaSalarial = (int)
            await consulta.AverageAsync(x => x.Salario);
        return model;
    }

    public async Task<List<string>>
        GetOficiosAsync()
    {
        var consulta = (from datos in this.context.Trabajadores
                        select datos.Oficio).Distinct();
        return await consulta.ToListAsync();
    }
}

```

Sobre Controllers creamos un nuevo controlador llamado **TrabajadoresController**

TRABAJADORESCONTROLLER

```

public class TrabajadoresController : Controller
{
    private RepositoryTrabajadores repo;

    public TrabajadoresController(RepositoryTrabajadores repo)
    {
        this.repo = repo;
    }

    public async Task<IActionResult> Index()
    {
        TrabajadoresModel model =
            await this.repo.GetTrabajadoresModelAsync();
        List<string> oficios = await this.repo.GetOficiosAsync();
        ViewData["OFICIOS"] = oficios;
        return View(model);
    }
}

```

Creamos una vista llamada **Index.cshtml**

INDEX.CSHMTL

```

@model TrabajadoresModel

@{
    List<string> oficios =
        ViewData["OFICIOS"] as List<string>;
}

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

<h1>View Trabajadores</h1>
<form method="post">
    <label>Seleccione oficio</label>
    <select name="oficio" class="form-control">
        @foreach (string oficio in oficios){
            <option value="@oficio">@oficio</option>
        }
    </select>
    <button class="btn btn-danger">Buscar trabajadores</button>
</form>

<ul class="list-group">
    <li class="list-group-item">
        Personas: <b>@Model.Personas</b>
    </li>
    <li class="list-group-item">
        Media salarial: <b>@Model.MediaSalarial</b>
    </li>
    <li class="list-group-item">
        Suma salarial: <b>@Model.SumaSalarial</b>
    </li>
</ul>

<table class="table table-warning">
    <thead>
        <tr>
            <th>ID Trabajador</th>
            <th>Apellido</th>
            <th>Oficio</th>
            <th>Salario</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Trabajadores) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.IdTrabajador)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Apellido)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Oficio)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Salario)
                </td>
            </tr>
        }
    </tbody>
</table>

```

Resolvemos las dependencias dentro de **Program**

PROGRAM

```

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddTransient<RepositoryTrabajadores>();
// ADD ----- +-----+

```

El siguiente paso es llamar a nuestro procedimiento almacenado que filtra por Oficio

REPOSITORYTRABAJADORES

```

public async Task<TrabajadoresModel>
GetTrabajadoresModelOficioAsync(string oficio)
{
    //VAMOS A LLAMAR AL PROCEDIMIENTO CON EF
    //LA UNICA DIFERENCIA RADICA EN QUE TENGO QUE
    //PONER LA PALABRA out EN CADA PARAMETRO DE SALIDA
    //EN LA CONSULTA SQL
    string sql = "SP_WORKERS_OFICIO @oficio, @personas OUT "
        + ", @media OUT, @suma OUT";
    SqlParameter pamOficio = new SqlParameter("@oficio", oficio);
    SqlParameter pamPersonas =
        new SqlParameter("@personas", -1);
    pamPersonas.Direction = ParameterDirection.Output;
    SqlParameter pamMedia =
        new SqlParameter("@media", -1);
    pamMedia.Direction = ParameterDirection.Output;
    SqlParameter pamSuma =
        new SqlParameter("@suma", -1);
    pamSuma.Direction = ParameterDirection.Output;
    //EJECUTAMOS LA CONSULTA DE SELECCION
    var consulta = this.context.Trabajadores.FromSqlRaw
        (sql, pamOficio, pamPersonas, pamMedia, pamSuma);
    TrabajadoresModel model = new TrabajadoresModel();
    //HASTA QUE NO EXTRAEMOS LOS DATOS DEL SELECT
    //NO TENEMOS LOS PARAMETROS DE SALIDA (reader.Close())
    model.Trabajadores = await consulta.ToListAsync();
    model.Personas = int.Parse(pamPersonas.Value.ToString());
    model.MediaSalarial = int.Parse(pamMedia.Value.ToString());
    model.SumaSalarial = int.Parse(pamSuma.Value.ToString());
    return model;
}

```

Sobre **TrabajadoresController** creamos un método Post para llamar a este filtro

TRABAJADORESCONTROLLER

[HttpPost]

0 references | 0 changes | 0 authors, 0 changes

```

public async Task<IActionResult> Index(string oficio)
{
    TrabajadoresModel model =
        await this.repo.GetTrabajadoresModelOficioAsync(oficio);
    List<string> oficios =
        await this.repo.GetOficiosAsync();
    ViewData["OFICIOS"] = oficios;
    return View(model);
}

```

En el siguiente ejemplo vamos a consumir múltiples orígenes de datos (SQL Server y Oracle)

Tendremos una vista para mostrar los datos de empleados junto a departamentos

Podremos visualizar los detalles de un Empleado determinado

Creamos un nuevo proyecto llamado **MvcNetCoreEFMMultiplesBBDD**

SQL SERVER VIEW

```

create view V_EMPLEADOS
as
select EMP_NO as IDEMPLEADO
, EMP.APELLIDO, EMP.OFICIO, EMP.SALARIO
, DEPT.DNOMBRE AS DEPARTAMENTO
, DEPT.LOC AS LOCALIDAD
from EMP
inner join DEPT
on EMP.DEPT_NO = DEPT.DEPT_NO
go

```

Agregamos los Nuget de Entity Framework



Sobre **Models** creamos una nueva clase llamada **EmpleadoView**

EMPLEADOVIEW

```
[Table("V_EMPLEADO")]
public class EmpleadoView
{
    [Key]
    [Column("IDEMPLEADO")]
    public int IdEmpleado { get; set; }
    [Column("APELLIDO")]
    public string Apellido { get; set; }
    [Column("OFICIO")]
    public string Oficio { get; set; }
    [Column("SALARIO")]
    public int Salario { get; set; }
    [Column("DEPARTAMENTO")]
    public string Departamento { get; set; }
    [Column("LOCALIDAD")]
    public string Localidad { get; set; }
}
```

Sobre el proyecto creamos una nueva carpeta llamada **Data** y una clase llamada **HospitalContext**

HOSPITALCONTEXT

```
public class HospitalContext : DbContext
{
    0 references
    public HospitalContext(DbContextOptions<HospitalContext> options)
        : base(options) { }

    0 references
    public DbSet<EmpleadoView> EmpleadosView { get; set; }
}
```

Creamos una carpeta llamada **Repositories** y una clase llamada **RepositoryEmpleados**

```
public class RepositoryEmpleados
{
    private HospitalContext context;

    public RepositoryEmpleados(HospitalContext context)
    {
        this.context = context;
    }

    public async Task<List<EmpleadoView>> GetEmpleadosAsync()
    {
        var consulta = from datos in this.context.EmpleadosView
                      select datos;
        return await consulta.ToListAsync();
    }

    public async Task<EmpleadoView> FindEmpleadoAsync(int idEmpleado)
    {
        var consulta = from datos in this.context.EmpleadosView
                      where datos.IdEmpleado == idEmpleado
                      select datos;
        return await consulta.FirstOrDefaultAsync();
    }
}
```

Sobre **Controllers** creamos un nuevo controlador llamado **EmpleadosController**

EMPLEADOSCONTROLLER

```
public class EmpleadosController : Controller
{
    private RepositoryEmpleados repo;

    public EmpleadosController(RepositoryEmpleados repo)
    {
        this.repo = repo;
    }

    public async Task<IActionResult> Index()
    {
        List<EmpleadoView> empleados =
            await this.repo.GetEmpleadosAsync();
        return View(empleados);
    }

    public async Task<IActionResult> Details(int id)
    {
        EmpleadoView empleado = await this.repo.FindEmpleadoAsync(id);
        return View(empleado);
    }
}
```

Sobre **appsettings** incluimos la cadena de conexión a SQL Server

APPSETTINGS.JSON

```
istore.org/appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Initial
  }
}
```

En Program resolvemos las dependencias

PROGRAM

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital");
builder.Services.AddTransient<RepositoryEmpleados>();
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseSqlServer(connectionString));

builder.Services.AddControllersWithViews();
```

Generamos las vistas con Scaffolding y comprobamos la funcionalidad

Una vez que hemos visto que es funcional es el momento de probar con Oracle

Abrimos Oracle y creamos la vista en la base de datos

```
create or replace view V_EMPLEADOS
as
  select EMP.EMP_NO as IDEMPLEADO
    , EMP.APELLIDO, EMP.OFICIO
    , EMP.SALARIO, DEPT.DNOMBRE AS DEPARTAMENTO
    , DEPT.LOC AS LOCALIDAD
   from EMP
  inner join DEPT
  on EMP.DEPT_NO=DEPT.DEPT_NO;
```

Agregamos el Nuget para Oracle



Agregamos la cadena de conexión a Oracle dentro de appsettings.json

APPSETTINGS.JSON

```
ALLOWEDHOSTS: " *",
ConnectionStrings": {
  "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Initial Catalog=HOSPITAL;Persis
  "OracleHospital": "Data Source=LOCALHOST:1521/XE; User Id=SYSTEM;Password=oracle"
```

Inyectamos Oracle en lugar de SQL Server en Program

PROGRAM

```
string connectionString =
  builder.Configuration.GetConnectionString("OracleHospital");
builder.Services.AddDbContext<HospitalContext>
  (options => options.UseOracle(connectionString)
  , options => options.UseOracleSQLCompatibility)
```

```
(OracleSQLCompatibility.DatabaseVersion19));
```

Conclusiones: No necesitamos Interface. Simplemente con las consultas Linq son Compatibles entre bbdd.

Cuando trabajamos con múltiples bases de datos, no siempre TODO es lo mismo y no Solamente utilizamos Linq, sino que en cuanto entramos con Procedures, las cosas Cambian.

En SQL Server, utilizar procedimientos almacenados con SELECT es compatible.
En Oracle, utilizar procedures con Select debemos hacerlo con parámetros de salida.

Vamos a realizar un cambio, trabajaremos con Procedimientos almacenados.

Comenzamos con SQL Server. Creamos un nuevo procedimiento llamado SP_ALL_VEMPLEADOS

SQL SERVER

```
create procedure SP_ALL_VEMPLEADOS
as
    select * from V_EMPLEADOS
go|
```

Vamos al Repository y debemos cambiar la llamada Linq por llamada a Procedure

REPOSITORYEMPLEADOS

```
public async Task<List<EmpleadoView>> GetEmpleadosAsync()
{
    string sql = "SP_ALL_VEMPLEADOS";
    var consulta =
        this.context.EmpleadosView
        .FromSqlRaw(sql);
    return await consulta.ToListAsync();
//var consulta = from datos in this.context.EmpleadosView
//                select datos;
//return await consulta.ToListAsync();
}
```

En Program nos conectamos a SQL Server

PROGRAM

```
builder.Services.AddTransient<RepositoryEmpleados>();
string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital");
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseSqlServer(connectionString));
//string connectionString =
```

Una vez que hemos visto que es funcional, debemos crear el mismo procedimiento Pero en Oracle

ORACLE

```
create or replace procedure SP_ALL_VEMPLEADOS
(p_cursor_empleados out sys_refcursor)
as
begin
    open p_cursor_empleados for
        select * from V_EMPLEADOS;
end;|
```

Debemos crear una Interface para poder injectar diferentes Repositories.

Sobre la carpeta **Repositories** creamos una nueva interface llamada **IRepositoryEmpleados**

IRepositoryEMPLEADOS

```
public interface IRepositoryEmpleados
{
    0 references | 0 changes | 0 authors, 0 changes
    Task<List<EmpleadoView>> GetEmpleadosAsync();
    0 references | 0 changes | 0 authors, 0 changes
    Task<EmpleadoView> FindEmpleadoAsync(int idEmpleado);
}
```

Indicamos en nuestro Repo actual que heredamos de dicha interface.

REPOSITORYEMPLEADOS

```

    public class RepositoryEmpleados : IRepositoryEmpleados
{
    private HospitalContext context;
}

```

Sobre **Repositories** creamos una clase llamada **RepositoryEmpleadosOracle** y que Herede de nuestra nueva interface

Si necesitamos llamar a un procedimiento almacenado de Oracle debemos hacerlo Con la siguiente sintaxis:

```

begin
    sp_procedure (:param1, :param2)
end;

```

REPOSITORYEMPLEADOSORACLE

```

public class RepositoryEmpleadosOracle : IRepositoryEmpleados
{
    private HospitalContext context;

    public RepositoryEmpleadosOracle(HospitalContext context)
    {
        this.context = context;
    }

    public async Task<List<EmpleadoView>> GetEmpleadosAsync()
    {
        string sql = "begin ";
        sql += " SP_ALL_VEMPLEADOS (:p_cursor_empleados);";
        sql += " end;";
        OracleParameter pamCursor = new OracleParameter();
        pamCursor.ParameterName = "p_cursor_empleados";
        pamCursor.Value = null;
        pamCursor.Direction = ParameterDirection.Output;
        //DEBEMOS INDICAR EL TIPO DE DATO DE ORACLE CURSOR
        pamCursor.OracleDbType = OracleDbType.RefCursor;
        var consulta = this.context.EmpleadosView
            .FromSqlRaw(sql, pamCursor);
        return await consulta.ToListAsync();
    }

    public async Task<EmpleadoView> FindEmpleadoAsync(int idEmpleado)
    {
        var consulta = from datos in this.context.EmpleadosView
                      where datos.IdEmpleado == idEmpleado
                      select datos;
        var data = await consulta.ToListAsync();
        return data.First();
    }
}

```

Por último, modificamos las inyecciones dentro de **EmpleadosController** recibiendo La interface y dentro de **Program** resolviendo la inyección de la interface

EMPLEADOSCONTROLLER

```

public class EmpleadosController : Controller
{
    private IRepositoryEmpleados repo;

    public EmpleadosController(IRepositoryEmpleados repo)
    {
        this.repo = repo;
    }
}

```

PROGRAM

```

//builder.Services.AddTransient<IRepositoryEmpleados, RepositoryEmpleados>();
//string connectionString =
//    builder.Configuration.GetConnectionString("SqlHospital");
//builder.Services.AddDbContext<HospitalContext>
//    (options => options.UseSqlServer(connectionString));
builder.Services.AddTransient<IRepositoryEmpleados, RepositoryEmpleadosOracle>();
string connectionString =
    builder.Configuration.GetConnectionString("OracleHospital");
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseOracle(connectionString
        , options => options.UseOracleSQLCompatibility
        (OracleSQLCompatibility.DatabaseVersion19)));

```

El siguiente paso es comprobar el funcionamiento con otra base de datos:

MySQL

<https://dev.mysql.com/downloads/installer/>

MySQL Installer 8.0.41

Note: MySQL 8.0 is the final series with MySQL Installer. As of MySQL 8.1, use a MySQL product's MSI or Zip archive for installation. MySQL Server 8.1 and higher also bundle MySQL Configurator, a tool that helps configure MySQL Server.

Select Version: 8.0.41

Select Operating System: Microsoft Windows

Windows (x86, 32-bit), MSI Installer	8.0.41	2.1M	Download
(mysql-installer-web-community-8.0.41.0.msi)		MD5: 22ed92c892160254fbf0f93d811360c2 Signature	
Windows (x86, 32-bit), MSI Installer	8.0.41	352.2M	Download
(mysql-installer-community-8.0.41.0.msi)		MD5: c2e89b80cf89c2214e5ecb9f91b77f10 Signature	

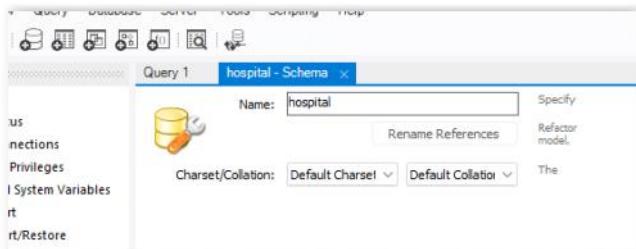
MYSQL WORKBENCH



Datos de MySql

User: **root**
Server: **localhost**
Password: **mysql**
Puerto: **3306**

Creamos un nuevo Schema llamado **hospital**



Connection String para MySql

server=SERVIDOR; port=3306; database=BBDD; user id=root; password=PASSWORD

- 1) Necesito crear una vista en **MySQL** llamada **V_EMPLEADOS**
- 2) Agregamos el Nuget de MySQL: **Pomelo.EntityFrameworkCore.MySql**
- 3) Creamos un nuevo Repo llamado **RepositoryEmpleadosMySql**
- 4) Implementamos los métodos mediante LINQ
- 5) Resolvemos las dependencias para MySQL dentro de **Program**
- 6) Probamos la Aplicación y listo por ahora...

VISTA MYSQL

```
create view V_EMPLEADOS
as
    select EMP.EMP_NO as IDEMPLEADO
    , EMP.APELLIDO, EMP.OFICIO
    , EMP.SALARIO, DEPT.DNOMBRE AS DEPARTAMENTO
    , DEPT.LOC AS LOCALIDAD
    from EMP
    inner join DEPT
    on EMP.DEPT_NO=DEPT.DEPT_NO;
```

Sobre el proyecto, agregamos el siguiente Nuget

A continuación, creamos un nuevo repositorio llamado **RepositoryEmpleadosMySql**
Heredamos de **IRepositoryEmpleados**

REPOSITORYEMPLEADOSMYSQL

```
public class RepositoryEmpleadosMySql : IRepositoryEmpleados
{
    private HospitalContext context;

    public RepositoryEmpleadosMySql(HospitalContext context)
    {
        this.context = context;
    }

    public async Task<List<EmpleadoView>> GetEmpleadosAsync()
    {
        var consulta = from datos in this.context.EmpleadosView
                      select datos;
        return await consulta.ToListAsync();
    }

    public async Task<EmpleadoView> FindEmpleadoAsync(int idEmpleado)
    {
        var consulta = from datos in this.context.EmpleadosView
                      where datos.IdEmpleado == idEmpleado
                      select datos;
        return await consulta.FirstOrDefaultAsync();
    }
}
```

Agregamos la cadena de conexión dentro de **appsettings.json**

APPSETTINGS.JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Encrypt=True;Trust Server Certificate=True",
        "OracleHospital": "Data Source=LOCALHOST:1521/XE; User Id=SYSTEM;Password=oracle",
        " MySqlHospital": "server=LOCALHOST;port=3306;database=hospital;user id=root;password=mysql"
    }
}
```

Sobre **Program** inyectamos nuestro nuevo Repo y cambiamos la cadena de conexión y
El acceso al Context.

PROGRAM

```
//-----MYSQL-----
builder.Services.AddTransient
    < IRepositoryEmpleados, RepositoryEmpleadosMySql>();
string connectionString =
    builder.Configuration.GetConnectionString(" MySqlHospital");
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseMySQL(connectionString));
```

El siguiente nivel: VERSION 2

A continuación, vamos a crear un procedimiento almacenado en **MYSQL** y lo llamaremos
Desde Entity Framework.

El procedimiento que vamos a crear se llamará **SP_ALL_VEMPLEADOS**

SESSION

martes, 18 de febrero de 2025 9:13

La información que tenemos en nuestros controllers o clases dentro de MVC Net Core No es persistente, cada vez que realizamos una petición al servidor, los objetos son Creados de nuevo con todo su contenido inicial.
¿Qué sucede si deseamos almacenar información persistente dentro de nuestro server?

Tenemos múltiples objetos para realizar almacenamiento persistente en el servidor:

- **TEMPDATA:** Esta información solamente se almacena una única vez.
Se utiliza para poder enviar información entre los **Controllers**, aunque también Se puede utilizar en las vistas. Es muy útil para enviar "algo" de **IActionResult** a Otro **IActionResult**
Su contenido es temporal, se almacena hasta que es leída la información de su Interior, en ese momento, es destruida su Key.
Pongamos que tenemos el siguiente código de ejemplo:

ViewData solamente es información para las vistas.
Cada vez que utilizo ViewData, es para enviar información a la propia vista, si Estoy cambiando de **IActionResult** como en el ejemplo, pierdo la información del Mensaje.

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

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult MetodoPost()
{
    ViewData["MENSAJE"] = "He realizado algo....";
    return RedirectToAction("Accion");
}
```

Si necesito enviar el mensaje en una redirección a otro **IActionResult** se utiliza TempData. TempData tiene la misma estructura de declaración de variables que ViewData

TempData["KEY"] = valor

```
public IActionResult Accion()
{
    //AQUI ARRIBA TENEMOS TEMPDATA DE MENSAJE
    return View();
}

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult MetodoPost()
{
    TempData["MENSAJE"] = "He realizado algo....";
    return RedirectToAction("Accion");
}
```

- **CACHE:** La información se almacena de forma persistente en el **Cliente**.
Para almacenar la información de Cache tenemos diversas alternativas, desde Elementos nativos hasta Nugets que realizan la funcionalidad.
Es super útil para minimizar la carga de peticiones de datos al servidor.
- **COOKIES:** Las Cookies se almacenan en el **Cliente**. Necesitamos indicar que El servidor tendrá acceso a las Cookies.
Sirve para mejorar la experiencia del usuario, la información que aquí almacenemos Será una información poco sensible.
Si el usuario borra las cookies, se borra la información.
- **SESSION:** La información se almacena en el Servidor. Por supuesto, cada usuario Tiene su información única, no se comparte con el resto de usuarios.
Debemos ser conscientes de lo que aquí almacenamos, ya que realizamos Un espacio de memoria para cada usuario, debemos almacenar lo mínimo posible.
Podemos indicar un límite de tiempo.

UTILIZAR SESSION CON MVC NET CORE

Session, de forma nativa solamente permite almacenar primitivos, nada más.
Es decir, string o int.
Necesitamos activar Session dentro del Middleware de nuestra App.

Session podemos utilizarlo en diferentes lugares, algunos nativos y otros no.

- **Controllers:** Si queremos utilizar Session dentro de los Controllers, tenemos Un objeto dentro de la clase llamado **HttpContext**
- **Views:** Si queremos utilizar/recuperar Session dentro de las vistas también es Nativo. Mediante código Razor podemos hacerlo con un objeto **Context**
- **Clases:** Las clases son diversas, no forman parte del entorno de MVC, sino que las creamos por necesidad por alguna razón.
Dentro de dichas clases no podemos recuperar la información de Session de Forma nativa. Para poder utilizar Session en cualquier clase no nativa, debemos Hacerlo mediante inyección de dependencias y con un objeto llamado **HttpContextAccessor**.

Su uso es muy sencillo, se realiza mediante **Key/Value**

CONTROLLER

```
HttpContext.Session.Set("KEY", "VALUE");
var dato = HttpContext.Session.Get("KEY");
```

VIEW

```
@{
    var dato = Context.Session.Get("KEY");
}
```

Creamos un nuevo proyecto llamado **MvcNetCoreSession**

Debemos configurar Session dentro del Middleware, dicho orden es importante

PROGRAM

```
app.UseHttpsRedirection();
app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
    .WithStaticAssets();
```

Depende lo que tengamos habilitado en el servidor, el orden es básico, no puedo inventar

Session debe ir antes de **MapRoute**

1. Control de errores y excepciones
2. Protocolo de seguridad de transporte estricta de HTTP
3. Redireccionamiento de HTTPS
4. Servidor de archivos estáticos
5. Cumplimiento de directivas de cookies
6. Autenticación
7. Sesión
8. MVC

Lo primero que vamos a realizar es habilitar Session dentro de nuestro Server.
Debemos indicar el tiempo que deseamos de inactividad para Session.

Session simple

[Almacenar Session](#)

[Mostrar Session](#)

Programeitor

10:04:17

PROGRAM

```
// Add services to the container.  
builder.Services.AddControllersWithViews();  
  
//DEBEMOS HABILITAR EL SERVICIO DE MEMORIA CACHE  
//PORQUE COMPARTEN CARACTERISTICAS  
builder.Services.AddDistributedMemoryCache();  
//CONFIGURAMOS SESSION CON UN TIEMPO DE INACTIVIDAD  
✓ builder.Services.AddSession(options =>  
{  
    options.IdleTimeout = TimeSpan.FromMinutes(10);  
});  
  
var app = builder.Build();|
```

El orden es IMPORTANTE

```
app.UseAuthorization();  
  
app.MapStaticAssets();  
  
//HABILITAMOS SESSION PARA EL SERVIDOR  
app.UseSession();  
✓ app.MapControllerRoute(  
    name: "default",  
    pattern: "{controller=Home}/{action=Index}/{id?}")  
    .WithStaticAssets();
```

Sobre **Controllers** creamos un nuevo controlador llamado **EjemploSessionController**

Vamos a comenzar con un ejemplo simple en el que almacenaremos la hora y recuperaremos Dicha hora almacenada

EJEMPLOSESSIONCONTROLLER

```
public IActionResult SessionSimple(string accion)  
{  
    if (accion != null)  
    {  
        if (accion.ToLower() == "almacenar")  
        {  
            //GUARDAMOS DATOS EN SESSION  
            HttpContext.Session.SetString("nombre", "Programeitor");  
            HttpContext.Session.SetString("hora",  
                DateTime.Now.ToString());  
            ViewData["MENSAJE"] = "Datos almacenados en Session";  
        }  
        else if (accion.ToLower() == "mostrar")  
        {  
            //RECUPERAMOS LOS DATOS DE SESSION  
            ViewData["USUARIO"] =  
                HttpContext.Session.GetString("nombre");  
            ViewData["HORA"] =  
                HttpContext.Session.GetString("hora");  
        }  
    }  
}
```

```

        return View();
    }

SESSIONSIMPLE.CSHTML

<h1 style="color:blue">
    Session simple
</h1>

<p>
    <a asp-controller="EjemploSession"
        asp-action="SessionSimple"
        asp-route-accion="almacenar">
        Almacenar Session
    </a>
</p>
<p>
    <a asp-controller="EjemploSession"
        asp-action="SessionSimple"
        asp-route-accion="mostrar">
        Mostrar Session
    </a>
</p>
<h3>@ViewData["MENSAJE"]</h3>
<h2 style="color:fuchsia">@ViewData["USUARIO"]</h2>
<h2 style="color:green">@ViewData["HORA"]</h2>

```

La información puede ser leída en múltiples lugares, es decir, la ventaja está en que La información está accesible desde las Vistas y otros controllers.
Por ejemplo, vamos a dibujar session dentro de **Index.cshtml**

INDEX.CSHTML

```

<h1>Ejemplos Session</h1>
<h2 style="color:blue">
    @Context.Session.GetString("nombre")
</h2>
<h2 style="color:red">
    @Context.Session.GetString("hora")
</h2>
<ul class="list-group">
    <li class="list-group-item">
        <a asp-controller="EjemploSession"
            asp-action="SessionSimple">
            Session simple
        </a>
    </li>
</ul>

```

Y podremos comprobar que, si no tenemos Session no nos dará ningún error
Porque estamos manejando tipos primitivos

Ejemplos Session

Programeitor

10:24:17

[Session simple](#)

Solamente podemos almacenar tipos String.
Y si deseamos/necesitamos almacenar algún objeto???

Para almacenar objetos completos debemos hacerlo mediante el método **Set** de **Session**

Necesitamos convertir los objetos a **byte[]** y, posteriormente, volver a convertir los Objetos a su clase cuando los recuperemos.

Necesitamos **Serialización**

Para utilizar esta funcionalidad haremos **Helpers**

Comenzamos creando un **Model** llamado **Mascota**

MASCOTA

```

0 references
public class Mascota
{
    0 references
    public string Nombre { get; set; }
    0 references
    public string Raza { get; set; }
    0 references
    public int Edad { get; set; }
}

```

Sobre el proyecto, creamos una nueva carpeta llamada **Helpers** y una clase llamada **HelperBinarySession**

HELPERSSESSION

```

public class HelperBinarySession
{
    //VAMOS A CREAR DOS METODOS static
    //PORQUE NO NECESITAMOS REALIZAR NEW PARA
    //UTILIZAR LOS METODOS DE CONVERSIÓN QUE CREAREMOS
    //EN ESTA CLASE
    //CONVERTIMOS UN OBJETO A BYTE[]
    public static byte[] ObjectToByte(Object objeto)
    {
        BinaryFormatter formatter =
            new BinaryFormatter();
        using (MemoryStream stream = new MemoryStream())
        {
            formatter.Serialize(stream, objeto);
            return stream.ToArray();
        }
    }

    //CONVERSOR DE BYTE[] A OBJETO
    public static Object ByteToObject(byte[] data)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        using (MemoryStream stream = new MemoryStream())
        {
            stream.Write(data, 0, data.Length);
            stream.Seek(0, SeekOrigin.Begin);
            Object objeto = (Object)
                formatter.Deserialize(stream);
            return objeto;
        }
    }
}

```

Sobre nuestro **Controller** agregamos un nuevo método para almacenar en Session Objetos complejos.

EJEMPLOSESSIONCONTROLLER

```

public IActionResult SessionMascota(string accion)
{
    if (accion != null)
    {
        if (accion.ToLower() == "almacenar")
        {
            Mascota mascota = new Mascota();
            mascota.Nombre = "Flounder";
            mascota.Raza = "Pez";
            mascota.Edad = 5;
            //PARA ALMACENAR OBJETOS Mascota DEBEMOS
            //CONVERTIRLOS A byte[]
            byte[] data =
                HelperBinarySession.ObjectToByte(mascota);
            //ALMACENAMOS EL OBJETO EN SESSION MEDIANTE Set
            HttpContext.Session.Set("MASCOTA", data);
            ViewData["MENSAJE"] = "Mascota almacenada en Session";
        }
        else if (accion.ToLower() == "mostrar")
        {
            //RECUPERAMOS LOS BYTES DE MASCOTA DE SESSION
            byte[] data = HttpContext.Session.Get("MASCOTA");
            //CONVERTIMOS LOS BYTES RECUPERADOS A OBJETO MASCOTA
            Mascota mascota = (Mascota)
                HelperBinarySession.ByteToObject(data);
            ViewData["MASCOTA"] = mascota;
        }
    }
    return View();
}

```

Creamos nuestra vista **SessionMascota.cshtml**

SESSIONMASCOTA.CSHTML

```

@{
    Mascota mascota =
        ViewData["MASCOTA"] as Mascota;
}
<h1>Session Binary Mascota</h1>
<h2>@ViewData["MENSAJE"]</h2>
<p>
    <a asp-controller="EjemploSession"
       asp-action="SessionMascota"
       asp-route-accion="almacenar">
        Almacenar
    </a>

```

```

        </a>
    </p>
    <a asp-controller="EjemploSession"
       asp-action="SessionMascota"
       asp-route-accion="mostrar">
        Mostrar
    </a>
</p>
@if (mascota != null){
    <h1 style="color:blue">
        @mascota.Nombre, Raza: @mascota.Raza, Edad: @mascota.Edad
    </h1>
}

```

Al ejecutar, nos encontraremos con errores debido a que la versión 9 de Net Core ya no Soporta la serialización binaria, ya que tiene errores de seguridad conocidos.

Instalamos el siguiente Nuget



Debemos marcar con una decoración la clase **Mascota** para que sea Serializada mediante este Método

MASCOTA

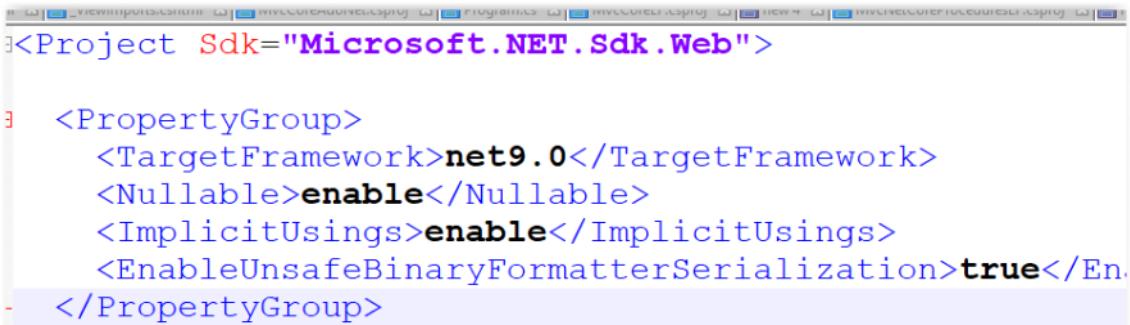
```

[Serializable]
6 references
public class Mascota
{
    2 references
    public string Nombre { get; set; }
    2 references
    public string Raza { get; set; }
    2 references
    public int Edad { get; set; }
}

```

En el fichero **CSPROJ** del proyecto están todos los Nuget y características de nuestro proyecto Debemos abrir dicho fichero e incluir la siguiente línea:

```
<EnableUnsafeBinaryFormatterSerialization>true</EnableUnsafeBinaryFormatterSerialization>
```



Y ya podremos visualizar nuestra página activa

Session Binary Mascota

[Almacenar](#)

[Mostrar](#)

Flounder, Raza: Pez, Edad: 5

El siguiente ejemplo es comprobar si podemos almacenar de forma binaria algo más complejo, Por ejemplo, una colección de mascotas.

[Almacenar colección](#)

[Mostrar colección](#)



Session Collection

Nombre	Raza	Edad
Nala	Leona	10
Olaf	Nieve	14
Rafiki	Mono	20
Sebastian	Cangrejo	12

EJEMPOSESSIONCONTROLLER

```
public IActionResult SessionCollection(string accion)
{
    if (accion != null)
    {
        if (accion.ToLower() == "almacenar")
        {
            List<Mascota> mascotas = new List<Mascota>
            {
                new Mascota {Nombre = "Nala", Raza = "Leona", Edad = 10},
                new Mascota {Nombre = "Olaf", Raza = "Nieve", Edad = 14},
                new Mascota {Nombre = "Rafiki", Raza = "Mono", Edad = 20},
                new Mascota {Nombre = "Sebastian", Raza = "Cangrejo", Edad = 1
            };
            byte[] data =
                HelperBinarySession.ObjectToByte(mascotas);
            HttpContext.Session.Set("MASCOTAS", data);
            ViewData["MENSAJE"] = "Colección almacenada en Session";
            return View();
        }
        else if (accion.ToLower() == "mostrar")
        {
            byte[] data = HttpContext.Session.Get("MASCOTAS");
            List<Mascota> mascotas = (List<Mascota>)
                HelperBinarySession.ByteToObject(data);
            return View(mascotas);
        }
    }
    return View();
}
```

SESSIONCOLLECTION.CSHTML

```
@model IEnumerable<MvcNetCoreSession.Models.Mascota>

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



Almacenar colección



Mostrar colección



# Session Collection



### @ViewData["MENSAJE"]


@if (Model != null){
    <table class="table table-bordered">
        <thead>
            <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Nombre)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Raza)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Edad)
                </th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model)
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem => item.Nombre)
                    </td>
                    <td>

```

```

        @Html.DisplayFor(modelItem => item.Raza)
    </td>
    <td> @Html.DisplayFor(modelItem => item.Edad)
    </td>
</tr>
}
</tbody>
</table>
}

```

Pongamos, como supuesto, que os pido lo siguiente:

- Dibujar el NOMBRE y la HORA de Session en **Index.cshtml** (HECHO)
- Quiero dibujar una Mascota de Session en **Index.cshtml**

¿Cómo lo hacemos?

¿Qué tenemos que hacer?

```

@using MvcNetCoreSession.Helpers

 @{
    byte[] data =
        Context.Session.Get("MASCOTA");
    Mascota mascota = null;
    if (Context.Session.Get("MASCOTA") != null){
        mascota =
            HelperBinarySession.ByteToObject(data) as Mascota;
    }
}

<h1>Ejemplos Session</h1>

@if (mascota != null){
    <h2 style="color: red">
        @mascota.Nombre, @mascota.Raza, @mascota.Edad
    </h2>
}

```

El coste que hemos tenido para dibujar un objeto simple dentro de nuestras vistas ha sido muy grande y debemos minimizarlo.

- **Debemos modificar el tipo de serialización:** Vamos a utilizar el método **GetString()** pero no directamente, lo haremos a partir de objetos Serializados en un Helper de forma que, recuperaremos el objeto y, a través de **GetString()** nos dará el propio objeto almacenado.
- **Métodos de extensión:** Esta arquitectura nos permite tener una clase ya definida Y poder AÑADIR métodos a dicha clase.

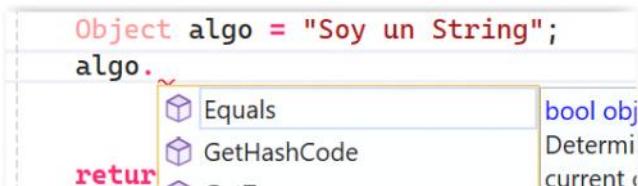
Cuando estamos serializando/deserializando actualmente, estamos enviando o recibiendo Un **Object**, es un standard, pero no sabemos realmente que tenemos dentro, tenemos Que estar haciendo Casting continuamente.

Podríamos tener alguna característica que realice dicha acción por nosotros, es decir, Si tenemos una **Mascota**, que nos devuelva una mascota y, si tenemos una **Persona**, Que nos devuelva una **Persona**.

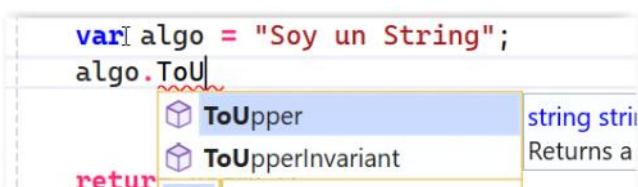
```
Mascota mascota = HttpContext.Session.Get("MASCOTA");
```

La solución es utilizar genéricos. Es un tipo de datos reconocible para el compilador, Es decir, es un tipo **var**

Si utilizamos **Object**, no reconoce el objeto:



En cambio, si utilizamos **var**, reconoce el objeto el compilador:



Mediante un genérico, podremos indicar que el método nos devolverá el objeto

Que deseemos y reconocible por el compilador, sin Casting de ningún tipo.

El problema está en que no podemos utilizar **var** para devolver algo en un método,
Solamente para almacenar valores de variables.

```
STICKY NOTES  
public static var ByteToObject(byte[] data)  
{  
    ...  
}
```

La solución está en utilizar el genérico de los métodos llamado **T**
Mediante la letra **T** en los métodos le estaremos indicando que devolverá un Objeto
Reconocible:

```
STICKY NOTES  
public static T ByteToObject<T>(byte[] data)  
{  
    ...  
}
```

La utilización del objeto, la conocéis perfectamente:

```
string texto =  
    HelperBinarySession.ByteToObject<string>(data);  
Mascota mascota =  
    HelperBinarySession.ByteToObject<Mascota>(data);  
List<Mascota> mascotas =  
    HelperBinarySession.ByteToObject<List<Mascota>>(data);  
...  
...
```

Para serializar/deserializar vamos a utilizar **JSON**

Instalaremos el siguiente Nuget



Sobre **Helpers** creamos una nueva clase llamada **HelperJsonSession**

HELPERSSESSION

```
public class HelperJsonSession  
{  
    //VAMOS A UTILIZAR EL METODO GetString() COMO HERRAMIENTA  
    //ALMACENAREMOS OBJETOS CON Serialize de JSON { nombre: "aa", raza: ""}  
    public static string SerializeObject<T>(T data)  
    {  
        //CONVERTIMOS EL OBJETO A STRING MEDIANTE Newton  
        string json = JsonConvert.SerializeObject(data);  
        return json;  
    }  
  
    //RECIBIREMOS UN string Y LO CONVERTIMOS A CUALQUIER OBJETO T  
    public static T DeserializeObject<T>(string data)  
    {  
        //DESERIALIZAMOS EL STRING A CUALQUIER OBJETO  
        T objeto = JsonConvert.DeserializeObject<T>(data);  
        return objeto;  
    }  
}
```

Creamos un nuevo método dentro del Controller

Y podremos comprobar la funcionalidad

Session Mascota JSON

[Almacenar Mascota Json](#)

[Mostrar Mascota Json](#)

Nombre: Simba, Raza: Leon

Edad: 9

Me gustaría mostrar Mascota dentro de **Index.cshtml**

Quitamos el código de **byte[]** y utilizamos lo nuevo
Solo código de la vista.

```
@using MvcNetCoreSession.Helpers

@{
    string data = Context.Session.GetString("MASCOTA");
    Mascota mascota = null;
    if (Context.Session.GetString("MASCOTA") != null){
        mascota =
            HelperJsonSession.DeserializeObject<Mascota>(data);
    }
}
```

¿Qué diferencia tenemos con este código?

```
@{
    Mascota mascota = null;
    if (Context.Session.Get("MASCOTA") != null)
    {
        byte[] data = Context.Session.Get("MASCOTA");
        mascota = (Mascota) HelperBinarySession.ByteToObject(data);
    }
}
```

Los códigos son iguales, no tenemos diferencia.

La clase Session es una chapuza, tenemos GetBytes(), GetString(), GetInt()

Estaría bien poder incluir un método que fuera **GetObject()** y que nos devuelva un Objeto

¿Qué opción tengo actual conocida?

- **Herencia:** Nos permite recuperar un objeto y poder implementar métodos en su Interior.

Estamos realizando lo siguiente:

HttpContext.Session

Nosotros, creamos nuestra clase: **MiSessionEstupenda**

HttpContext.MiSessionEstupenda

NO PODEMOS SUSTITUIR UNA CLASE DEL FRAMEWORK NATIVA POR OTRA

- **Extensions:** Nos permite agregar métodos a clases ya existentes sin necesidad De utilizar Herencia ni clases propias, es decir, me permite AGREGAR un método A cualquier clase.

```
string frase = "Hoy es martes todavía";
int numero = frase.GetNumeroPalabras();
```

Para extender métodos debemos seguir unas normas:

- 1) Los métodos deben ser **static**
- 2) La clase que implementa dichos métodos también debe ser **static**

Dentro del método static, debemos recibir la clase que estamos implementando mediante la Extensión.

```
public static void MetodoExtension
    (this CLASE_A_EXTENDER variable)
```

Si queremos traducir esto a extensión:

```

string frase = "Hoy es martes todavía";
int numero = frase.GetNumeroPalabras();

```

```

0 references
public static int GetNumeroPalabras(
    this string texto)
{
    //HACEMOS BUCLES
    return 5;
}

```

La clase que utiliza Session es una interface llamada **ISession**.
Dicha clase es la que debemos extender con nuestros métodos

Vamos a crear un método en Session llamado **GetObject<T>(KEY)**
Creamos otro método para almacenar nuestro objeto **SetObject<T>(KEY, OBJETO)**

Posteriormente, en nuestros códigos nativos:

RECUPERAR UN OBJETO

```
var dato = HttpContext.Session.GetObject("MASCOTA")
```

ALMACENAR UN OBJETO

```
HttpContext.Session.SetObject("MASCOTA", mascota)
```

Session Mascota Object

[Almacenar Mascota Object](#)

[Mostrar Mascota Object](#)

Nombre: Olaf, Raza: Muñeco

Edad: 19

Sobre el proyecto, creamos una nueva carpeta llamada **Extensions** y una clase llamada **SessionExtension**

SESSIONEXTENSION

```

public static class SessionExtension
{
    //CREAMOS UN METODO PARA RECUPERAR CUALQUIER OBJETO
    public static T GetObject<T>
        (this ISession session, string key)
    {
        //YA ESTAREMOS TRABAJANDO CON
        //HttpContext.Session
        //DEBEMOS RECUPERAR LO QUE TENEMOS ALMACENADO
        //DENTRO DE Session
        string json = session.GetString(key);
        //QUE SUCEDA SI RECUPERAMOS ALGO DE SESSION
        //QUE NO EXISTE???
        //SI NO TENEMOS NADA ALMACENADO, DEBEMOS DEVOLVER EL
        //VALOR POR DEFECTO
        if (json == null)
        {
            return default(T);
        }
        else
        {
            //RECUPERAMOS EL OBJETO QUE TENEMOS ALMACENADO DENTRO
            //DE NUESTRA KEY
            T data = HelperJsonSession.DeserializeObject<T>(json);
            return data;
        }
    }

    public static void SetObject
        (this ISession session, string key, object value)
    {
        string data = HelperJsonSession.SerializeObject(value);
        //ALMACENAMOS EL JSON DENTRO DE SESSION
        session.SetString(key, data);
    }
}

```

El siguiente paso será implementar un método dentro del Controller

EJEMPLOSESSIONCONTROLLER

```
public IActionResult SessionMascotaObject(string accion)
{
    if (accion != null)
    {
        if (accion.ToLower() == "almacenar")
        {
            Mascota mascota = new Mascota
            {
                Nombre = "Olaf", Raza = "Muñeco", Edad = 19
            };
            HttpContext.Session.SetObject("MASCOTAOBJECT", mascota);
            ViewData["MENSAJE"] = "Mascota como Object almacenada";
        }else if (accion.ToLower() == "mostrar")
        {
            Mascota mascota =
                HttpContext.Session.GetObject<Mascota>("MASCOTAOBJECT");
            ViewData["MASCOTA"] = mascota;
        }
    }
    return View();
}
```

SESSIONMASCOTAOBJECT.CSHTML

```
@{
    Mascota mascota =
        ViewData["MASCOTA"] as Mascota;
}
<h1 style="color:red">
    Session Mascota Object
</h1>
<p>
    <a asp-controller="EjemploSession"
        asp-action="SessionMascotaObject"
        asp-route-accion="almacenar">
        Almacenar Mascota Object
    </a>
</p>
<p>
    <a asp-controller="EjemploSession"
        asp-action="SessionMascotaObject"
        asp-route-accion="mostrar">
        Mostrar Mascota Object
    </a>
</p>
<h3>@ViewData["MENSAJE"]</h3>
@if (mascota != null)
{
    <h1 style="color:red">
        Nombre: @mascota.Nombre, Raza: @mascota.Raza
    </h1>
    <h3 style="color:green">Edad: @mascota.Edad</h3>
}
```

Quiero visualizar la Mascota dentro de **Index.cshtml**

INDEX.CSHTML

```
@using MvcNetCoreSession.Extensions
[

@{
    Mascota mascota =
        Context.Session.GetObject<Mascota>("MASCOTAOBJECT");
}

<h1>Ejemplos Session</h1>

@if (mascota != null){
    <h2 style="color:red">
        @mascota.Nombre, @mascota.Raza, @mascota.Edad
    </h2>
}
```

Como vemos, Session no es nativo en las vistas si implementamos métodos Extension
Si queremos utilizar la extensión en todas las vistas, tendremos que incluir el using
Dentro de **_ViewImports.cshtml**

```
@using MvcNetCoreSession
@using MvcNetCoreSession.Models
@using MvcNetCoreSession.Extensions
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

El siguiente paso será comprobar si funciona igual con colecciones.
Vamos a crear una colección de Mascotas, almacenar dicha colección dentro de Session y ver si funciona.

EJEMPLOSESSIONCONTROLLER

```
public IActionResult SessionMascotaCollection(string accion)
{
    if (accion != null)
    {
        if (accion.ToLower() == "almacenar")
        {
            List<Mascota> mascotas = new List<Mascota>
            {
                new Mascota { Nombre = "Patricio", Raza = "Estrella de mar",
Edad = 4},
                new Mascota { Nombre = "Apu", Raza = "Monito", Edad = 10},
                new Mascota { Nombre = "Donald", Raza = "Pato", Edad = 50 },
                new Mascota { Nombre = "Pluto", Raza = "Perro", Edad = 60}
            };
            HttpContext.Session.SetObject("MASCOTAS", mascotas);
            ViewData["MENSAJE"] = "Colección Mascotas almacenada";
            return View();
        }else if (accion.ToLower() == "mostrar")
        {
            List<Mascota> mascotas =
            HttpContext.Session.GetObject<List<Mascota>>("MASCOTAS");
            return View(mascotas);
        }
    }
    return View();
}
```

Y podremos visualizar el resultado

[Almacenar colección Mascotas](#)

[Almacenar colección Mascotas](#)



Session Collection

Nombre	Raza	Edad
Patricio	Estrella de mar	4
Apu	Monito	10
Donald	Pato	50
Pluto	Perro	60

Por último, estamos utilizando Helpers dentro de Extension, cuando no es necesario.

Lo hicimos porque fuimos escalando hasta encontrar la solución final, pero si miramos el Código, podemos perfectamente utilizar la conversión del Helper en Extensions.

```
else
{
    //RECUPERAMOS EL OBJETO QUE TENEMOS ALMACENADO DENTRO
    //DE NUESTRA KEY
    T data = HelperJsonSession.DeserializeObject<T>(json);
    return data;
}
```

SESSIONEXTENSION

```
public static class SessionExtension
{
    //CREAMOS UN METODO PARA RECUPERAR CUALQUIER OBJETO
    public static T GetObject<T>
        (this ISession session, string key)
    {
        //YA ESTAREMOS TRABAJANDO CON
        //HttpContext.Session
        //DEBEMOS RECUPERAR LO QUE TENEMOS ALMACENADO
        //DENTRO DE Session
        string json = session.GetString(key);
        //QUE SUCEDA SI RECUPERAMOS ALGO DE SESSION
        //QUE NO EXISTE???
        //SI NO TENEMOS NADA ALMACENADO, DEBEMOS DEVOLVER EL
        //VALOR POR DEFECTO
        if (json == null)
        {
            return default(T);
        }
        else
        {
```

```

    //RECUPERAMOS EL OBJETO QUE TENEMOS ALMACENADO DENTRO
    //DE NUESTRA KEY
    T data = JsonConvert.DeserializeObject<T>(json);
    return data;
}

public static void SetObject
    (this ISession session, string key, object value)
{
    string data = JsonConvert.SerializeObject(value);
    //ALMACENAMOS EL JSON DENTRO DE SESSION
    session.SetString(key, data);
}
}

```

Actualmente, podemos recuperar de forma nativa Session en clases del entorno MVC Net Core.

Pero no siempre estaremos dentro de clases Net core y puede que necesitemos acceder a Session, por ejemplo, en un Repo.

Debemos recuperar dicha clase realizando dos pasos:

- 1) Inyectar dentro de **Program** la interface **IHttpContextAccessor**
- 2) Recuperar dicho objeto en las clases que no son Nativas.

Vamos a crear una clase normal e intentar recuperar session de mascotas en dicha clase.

Sobre Helpers, creamos una nueva clase llamada **HelperSessionContextAccessor**

HELPERSSESSIONCONTEXTACESSOR

```

public class HelperSessionContextAccessor
{
    private IHttpContextAccessor contextAccessor;

    public HelperSessionContextAccessor
        (IHttpContextAccessor contextAccessor)
    {
        this.contextAccessor = contextAccessor;
    }

    public List<Mascota> GetMascotasSession()
    {
        List<Mascota> mascotas =
            this.contextAccessor.HttpContext
                .Session.GetObject<List<Mascota>>("MASCOTAS");
        return mascotas;
    }
}

```

En **Program** inyectamos la interface

PROGRAM

```

using MvcNetCoreSession.Helpers;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddSingleton<HelperSessionContextAccessor>();
builder.Services.AddSingleton<IHttpContextAccessor
    , HttpContextAccessor>();

```

Probamos la funcionalidad en nuestro Controller

EJEMPLOSESSIONCONTROLLER

```

public class EjemploSessionController : Controller
{
    HelperSessionContextAccessor helper;

    0 references | 0 changes | 0 authors, 0 changes
    public EjemploSessionController(HelperSessionContextAccessor helper)
    {
        this.helper = helper;
    }

    0 references | serraguti, 19 hours ago | 1 author, 1 change
    public IActionResult Index()
    {
        List<Mascota> mascotas = this.helper.GetMascotasSession();
        return View(mascotas);
    }
}

```

En la vista, incluimos un Mensaje en el caso que hayamos recuperado las Mascotas

INDEX.CSHTML

```

@model List<Mascota>

@using MvcNetCoreSession.Helpers
 @{
    Mascota mascota =
        Context.Session.GetObject<Mascota>("MASCOTAOBJECT");
}

<h1>Ejemplos Session</h1>

@if (Model != null){
    <h1 style="color:blue">Mascotas recuperadas: @Model.Count</h1>
}else{
    <h1 style="color:red">No tenemos mascotas del Helper</h1>
}

```

Y podremos recuperar cualquier elemento de Session en cualquier clase

Ejemplos Session

Mascotas recuperadas: 4

[Session simple](#)

[Session Mascota Object](#)

[Session Mascota Collection](#)

FICHEROS SERVIDOR

jueves, 20 de febrero de 2025 13:49

SUBIR FICHEROS AL SERVIDOR

Los ficheros que subamos deben estar almacenados dentro de mi Server, no dentro De mi proyecto.
Esto quiere decir que tenemos que subirlo a nuestra zona del FRONT, es decir, la carpeta **wwwroot**

Tenemos múltiples formas de trabajar con ficheros:

- 1) **Base de datos:** Dentro de cualquier base de datos, existen tipados llamados BLOC o VARBINARY que almacenan **Stream** dentro de un campo de la tabla. La ventaja radica en que siempre tendremos los ficheros, aunque nuestra App Sea modificada de Servidor, no pierdo los ficheros.
- 2) **Ficheros en Server Net Core:** Podemos tener los ficheros dentro de nuestro Servidor y acceder a ellos de forma local dentro de la aplicación.
Desventajas: Los ficheros no están centralizados, tenemos que copiar los ficheros Si nos mudamos de Server.
- 3) **Repositorios Cloud:** Dentro de cualquier nube existen funcionalidades para almacenar Ficheros como si de un disco duro habláramos.

Nosotros vamos a utilizar el punto 2 por ahora.
Tendremos nuestras imágenes dentro del servidor en la carpeta **wwwroot** y, para nuestro Proyecto, lo que tendremos es el nombre de la imagen en la columna: **1.png**

Para poder subir ficheros al servidor necesitamos dos elementos:

- 1) `<form enctype="form-data">`
- 2) `<input type="file">`

Cuando tenemos estas dos combinaciones, podremos recibir en el Controller una Clase llamada **IFormFile** que contiene los datos asociados a nuestro fichero.

Sobre la carpeta **Views** creamos una nueva carpeta llamada **UploadFiles**
Y una vista llamada **SubirFichero.cshtml**

```
<h1>Upload Files Net Core</h1>

<form method="post"
      enctype="multipart/form-data">
    <label>Seleccione un fichero</label>
    <input type="file" name="fichero" class="form-control"/>
    <button class="btn btn-info">
        Subir fichero
    </button>
</form>

<h2 style="color: red">@ViewData["MENSAJE"]</h2>
```

Si estamos hablando de Files debemos tener en cuenta que las peticiones serán Asíncronas, como **async/await** en los controllers

Sobre **Controllers** creamos un nuevo controlador llamado **UploadFilesController**

UPLOADFILESCONTROLLER

```
public async Task<IActionResult>
SubirFichero(IFormFile fichero)
```

```

    {
        //COMENZAMOS ALMACENANDO EL FICHERO EN LOS
        //ELEMENTOS TEMPORALES
        string tempFolder = Path.GetTempPath();
        string fileName = fichero.FileName;
        //CUANDO HABLAMOS DE FICHEROS Y DE RUTAS DE SISTEMA
        //ESTAMOS PENSANDO EN LO SIGUIENTE
        //C:\miruta\carpeta\file.txt
        //TENEMOS QUE TENER EN CUENTA QUE ESTAMOS DENTRO DE NET CORE
        //PODEMOS MONTAR EL SERVIDOR DONDE DESEEMOS
        //C:/miruta/carpeta/file.txt
        //..miruta\carpeta\file.txt
        //LAS RUTAS DE FICHEROS NO DEBO ESCRIBIRLAS, TENGO QUE GENERAR
        //DICHAS RUTAS CON EL SISTEMA DONDE ESTOY TRABAJANDO
        string path = Path.Combine(tempFolder, fileName);
        //PARA SUBIR EL FICHERO SE UTILIZA Stream CON IFormFile
        using (Stream stream = new FileStream(path, FileMode.Create))
        {
            await fichero.CopyToAsync(stream);
        }
        ViewData["MENSAJE"] = "Fichero subido a " + path;
        return View();
    }
}

```

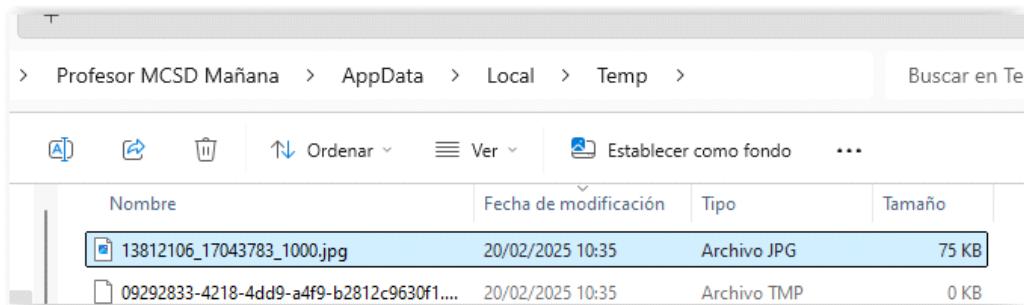
Upload Files Net Core

Seleccione un fichero

Ningún archivo seleccionado

Fichero subido a C:\Users\Profesor MCSD
Mañana\AppData\Local\Temp\batman.jpg

Y podremos comprobar que sube el fichero en la ruta indicada



Actualmente estamos subiendo el fichero a una ruta temporal de un equipo.
Esto, en **Producción**, es imposible, no tendremos acceso al sistema de ficheros del Servidor, solamente dónde esté alojada nuestra App.

El único lugar dónde podemos almacenar ficheros es dentro de la carpeta **wwwroot** de nuestro Web Server

Tenemos que utilizar dos características en nuestro proyecto para poder Acceder a Ficheros y la carpeta **wwwroot** en código C#

- 1) **StaticFiles:** Habilita el uso de ficheros en nuestro servidor. Por defecto NO está activo en La clase **Program**

PROGRAM

```

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseAuthorization();

        app.MapStaticAssets();

    < app.MapControllerRoute(
        "Index", "Home", new { controller = "Home", action = "Index" }
    )>

```

- 1) **IWebHostEnvironment**: Dicha interface nos permite acceder al sistema de carpetas Dentro de nuestro servidor.
Accede a los ficheros del servidor de la carpeta **wwwroot** con código C#

Este objeto lo recuperamos mediante la inyección.

Nota: Como nuestros proyectos son Multi Server, deberíamos escribir todo en Minúsculas. (ficheros)

Sobre la carpeta **wwwroot** creamos una nueva carpeta llamada **uploads**

Modificamos el código de **UPLOADFILESCONTROLLER**

```

public class UploadFilesController : Controller
{
    private IWebHostEnvironment hostEnvironment;

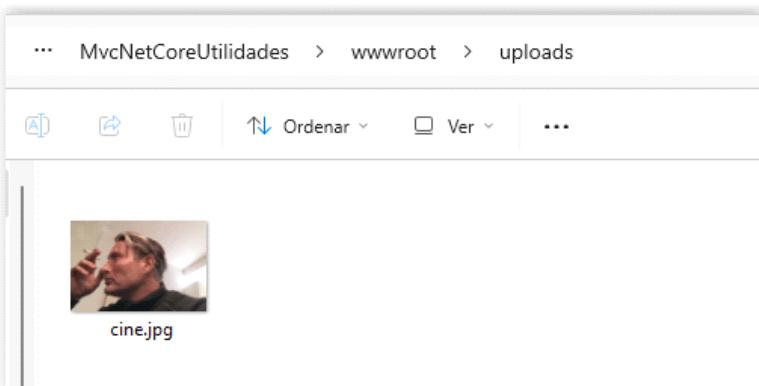
    public UploadFilesController(IWebHostEnvironment hostEnvironment)
    {
        this.hostEnvironment = hostEnvironment;
    }

    public IActionResult SubirFichero()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult> SubirFichero(IFormFile fichero)
    {
        //NECESITAMOS LA RUTA A NUESTRO WWWROOT DEL SERVER
        string rootFolder =
            this.hostEnvironment.WebRootPath;
        //COMENZAMOS ALMACENANDO EL FICHERO EN LOS
        //ELEMENTOS TEMPORALES
        string fileName = fichero.FileName;
        //LAS RUTAS DE FICHEROS NO DEBO ESCRIBIRLAS, TENGO QUE GENERAR
        //DICHAS RUTAS CON EL SISTEMA DONDE ESTOY TRABAJANDO
        string path = Path.Combine(rootFolder, "uploads", fileName);
        //PARA SUBIR EL FICHERO SE UTILIZA Stream CON IFormFile
        using (Stream stream = new FileStream(path, FileMode.Create))
        {
            await fichero.CopyToAsync(stream);
        }
        ViewData["MENSAJE"] = "Fichero subido a " + path;
        return View();
    }
}

```

Y podremos comprobar que ahora sí está subido a nuestro Server



Ya podemos acceder perfectamente a las imágenes desde las vistas.

```

```

Pongamos que tenemos múltiples rutas para subir y lo realizamos en múltiples Controllers.

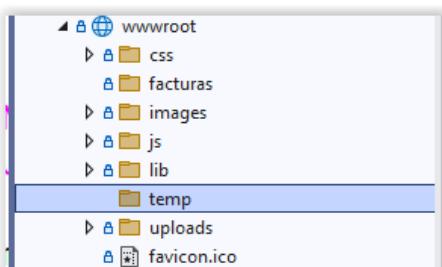
No será lo mismo subir imágenes de mis productos que de mis usuarios, deberías de Organizarlo por Carpetas.

Si queremos subir algo de usuarios, tendremos que utilizar alguna carpeta llamada users

```
string path = Path.Combine(rootFolder, "users", fileName);
```

Si necesitamos subir facturas lo mismo

```
string path = Path.Combine(rootFolder, "facturas", fileName);
```



Vamos a centralizar la subida de ficheros con una clase

Si tenemos cambiar algo, no lo haremos en los Controllers, sino que lo haremos en nuestra Clase.

Sobre el proyecto, creamos una nueva carpeta llamada **Helpers**
Y creamos una clase llamada **HelperPathProvider**

Esta clase se encargará de proporcionar la ruta a las carpetas de nuestros Server

HELPERPATHPROVIDER

Sobre **Program** resolvemos las dependencias de nuestro Helper

PROGRAM

```
// Add services to the container.
builder.Services.AddSingleton<HelperPathProvider>();
```

Modificamos el código de nuestro Controller

UPLOADFILESCONTROLLER

```
public class UploadFilesController : Controller
{
    private HelperPathProvider helperPath;

    public UploadFilesController(HelperPathProvider helperPath)
    {
        this.helperPath = helperPath;
    }

    public IActionResult SubirFichero()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        SubirFichero(IFormFile fichero)
    {
        string fileName = fichero.FileName;
        //LAS RUTAS DE FICHEROS NO DEBO ESCRIBIRLAS, TENGO QUE GENERAR
        //DICHAS RUTAS CON EL SISTEMA DONDE ESTOY TRABAJANDO
        string path =
            this.helperPath.MapPath(fileName, Folders.Images);
        //PARA SUBIR EL FICHERO SE UTILIZA Stream CON IFormFile
        using (Stream stream = new FileStream(path, FileMode.Create))
        {
            await fichero.CopyToAsync(stream);
        }
        ViewData["MENSAJE"] = "Fichero subido a " + path;
        return View();
    }
}
```

Gracias a Gabriel

Y si después de subir la imagen, deseamos mostrarla?????

```
using (Stream stream = new FileStream(path, FileMode.Create))
{
    await fichero.CopyToAsync(stream);
}
ViewData["MENSAJE"] = "Fichero subido a " + path;
ViewData["PATH"] = path;
return View();
```

En nuestra página, mostramos la imagen:

```
<h2 style="color: red">@ViewData["MENSAJE"]</h2>

<img src='@ViewData["PATH"]'
style="width: 150px; height: 150px"/>
```

Un sistema de rutas PATH no se puede utilizar en Web.

Necesitamos generar una ruta dinámica a nuestro fichero recién subido.

¿Qué necesitamos?

- URL de nuestro Server

- Folder
- Nombre del fichero

Debemos crear, dentro de nuestro Helper otro método más llamado **MapUrlPath** que nos devuelva el String dinámico del fichero subido.

```
<img src='https://localhost:7239/images/2.jpg'
      style="width: 150px; height: 150px" />
      |
<img src='C:\Users\Profesor MCSD Mañana\Documents\Proyectos\MvcNetCo:
      style="width: 150px; height: 150px" />
```

Para recuperar la información de la petición, tenemos **HttpContextAccesor**

Dependiendo de la petición, el objeto es posible que venga con más cosas, me explico:

Si estamos en una clase limpia, como es el Helper, la petición viene directamente con La URL limpia: <https://localhost:7895>

Si lo utilizamos en algún controller, la petición no vendría limpia y nos puede llevar a error.

<https://localhost:7895/Controller/Action>

Vamos a solucionarlo con **IServer** que es una interface que nos devuelve la dirección de nuestro Server también

No tenemos que inyectar nada en **Program**, viene por defecto con el HOST, así que podemos Utilizarla sin problemas.

HELPERSPATHPROVIDER

```
public enum Folders { Images, Facturas, Uploads, Temporal }

public class HelperPathProvider
{
    private IServer server;
    private IWebHostEnvironment hostEnvironment;

    public HelperPathProvider
        (IWebHostEnvironment hostEnvironment, IServer server)
    {
        this.hostEnvironment = hostEnvironment;
        this.server = server;
    }

    public string MapPath(string fileName, Folders folder)
    {
        string carpeta = "";
        if (folder == Folders.Images)
        {
            carpeta = "images";
        } else if (folder == Folders.Facturas)
        {
            carpeta = "facturas";
        } else if (folder == Folders.Uploads)
        {
            carpeta = "uploads";
        } else if (folder == Folders.Temporal)
        {
            carpeta = "temp";
        }
        string rootPath = this.hostEnvironment.WebRootPath;
        string path = Path.Combine(rootPath, carpeta, fileName);
        return path;
    }

    public string MapUrlPath(string fileName, Folders folder)
    {
        string carpeta = "";
        if (folder == Folders.Images)
        {
            carpeta = "images";
        } else if (folder == Folders.Facturas)
        {
            carpeta = "facturas";
        }
    }
}
```

```

        }
        else if (folder == Folders.Uploads)
        {
            carpeta = "uploads";
        }
        else if (folder == Folders.Temporal)
        {
            carpeta = "temp";
        }
        var addresses =
            this.server.Features.Get<IServerAddressesFeature>().Addresses;
        string serverUrl = addresses.FirstOrDefault();
        string urlPath = serverUrl + "/" + carpeta + "/" + fileName;
        return urlPath;
    }
}

```

UPLOADFILESCONTROLLER

```

public class UploadFilesController : Controller
{
    private HelperPathProvider helperPath;

    public UploadFilesController(HelperPathProvider helperPath)
    {
        this.helperPath = helperPath;
    }

    public IActionResult SubirFichero()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        SubirFichero(IFormFile fichero)
    {
        string fileName = fichero.FileName;
        //LAS RUTAS DE FICHEROS NO DEBO ESCRIBIRLAS, TENGO QUE GENERAR
        //DICHAS RUTAS CON EL SISTEMA DONDE ESTOY TRABAJANDO
        string path =
            this.helperPath.MapPath(fileName, Folders.Images);
        string urlPath =
            this.helperPath.MapPath(fileName, Folders.Images);
        //PARA SUBIR EL FICHERO SE UTILIZA Stream CON IFormFile
        using (Stream stream = new FileStream(path, FileMode.Create))
        {
            await fichero.CopyToAsync(stream);
        }
        ViewData["MENSAJE"] = "Fichero subido a " + path;
        ViewData["URL"] = urlPath;
        return View();
    }
}

```

CACHING NET CORE

jueves, 20 de febrero de 2025 13:50

CACHING NET CORE

Es muy parecido a Session.

Esta funcionalidad se almacena en el cliente de forma temporal y nos permite poder Minimizar la carga de datos dentro de nuestra App.

Se libera el Cache en cuanto cierro el Explorador Web.

Tenemos dos tipos de peticiones:

- 1) **Memoria Cache Personalizada:** Este tipo de memoria necesita programación y es Manual. Se utiliza con inyección y nos permite gestionar peticiones en cualquier Parte de nuestra aplicación.

Utiliza una interface llamada **IMemoryCache**

Necesitamos implementar dicha interface mediante un Nuget e inyección de dependencias En **Program**

PROGRAM

```
builder.Services.AddMemoryCache();
```

```
IMemoryCache.Set(KEY, VALUE)
```

```
var algo = IMemoryCache.Get(KEY)
```

Lo bueno está en que nos permite almacenar Objetos.

También nos permite indicar el tiempo que deseamos que dure el Cache, por si necesitamos Hacerlo de forma explícita.

- 1) **Memoria Cache Distribuida:** Implementamos el Cache sin inyección, viene de serie Dentro del Framework.
Se utiliza mediante decoraciones/atributos
También podemos indicar el tiempo de duración del Cache

Por ejemplo, podemos indicar que un método **IActionResult** no lo leerá durante 15 segundos. Solamente lo leerá la primera vez, el resto lo hará tirando de memoria.

```
[ResponseCache(Duration = 15,
    Location = ResponseCacheLocation.Client)]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult Index()
{
    List<Datos> data =
        this.repo.GetDatosBBDD();
    return View(data);
}
```

Para comprobar diferentes funcionalidades que iremos realizando, vamos a crear un Proyecto llamado **MvcNetCoreUtilidades**

Vamos a comenzar trabajando con la Memoria Distribuida, es decir, la que funciona mediante Decoraciones/Atributos.

Sobre **Program** debemos indicar que utilizaremos Memoria Distribuida

PROGRAM

```
// Add services to the container.  
builder.Services.AddDistributedMemoryCache();
```

Sobre **Controllers** creamos un nuevo controlador llamado **CachingController**

CACHINGCONTROLLER

```
[ResponseCache(Duration = 60,  
    Location = ResponseCacheLocation.Client)]  
0 references  
public IActionResult MemoriaDistribuida()  
{  
    string fecha =  
        DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss")  
        + DateTime.Now.Millisecond;  
    ViewData["FECHA"] = fecha;  
    return View();  
}
```

MEMORIADISTRIBUIDA.CSHTML

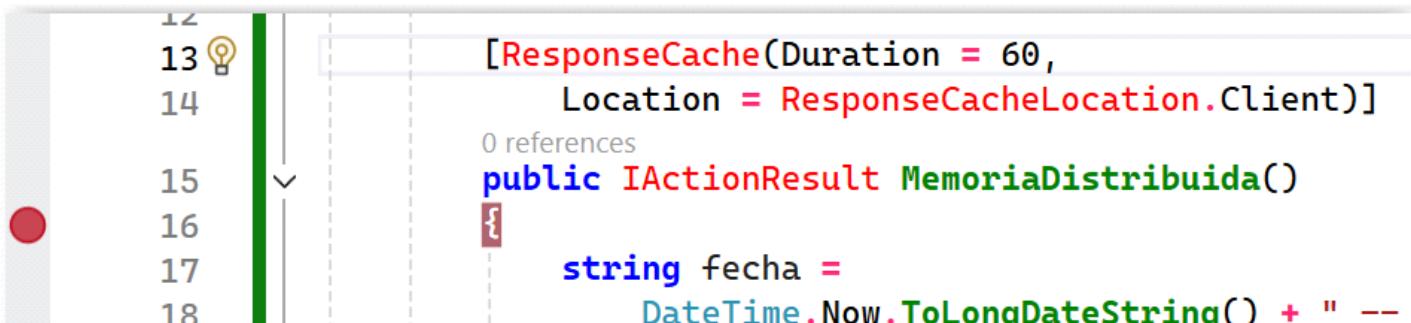
```
<h1>Memoria distribuida</h1>  
<p>  
    <a asp-controller="Caching"  
        asp-action="MemoriaDistribuida">  
        Actualizar página  
    </a>  
</p>  
<h2 style="color:blue">@ViewData["FECHA"]</h2>
```

Memoria distribuida

[Actualizar página](#)

miércoles, 19 de febrero de 2025 -- 13:22:18

Si incluimos un punto de interrupción dentro del método con Caching, veremos que ni siquiera Leerá el método hasta que no han pasado los 60 segundos



```
12  
13 [ResponseCache(Duration = 60,  
    Location = ResponseCacheLocation.Client)]  
0 references  
public IActionResult MemoriaDistribuida()  
{  
    string fecha =  
        DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss")  
        + DateTime.Now.Millisecond;
```

MEMORIA PERSONALIZADA

Esta tecnología implica almacenar, de forma manual, lo que necesitemos dentro De Cache.

Necesitamos un Nuget para trabajar.



Necesitamos recuperar en la inyección **IMemoryCache**.

Para ello, debemos incluir un servicio nuevo dentro de **Program**

PROGRAM

```
// Add services to the container.  
builder.Services.AddMemoryCache();
```

Vamos a crear un método dentro de **CachingController** para almacenar una Fecha y su Hora.

A diferencia del Cache Distribuido, debemos preguntar si tenemos valor en su interior o no.

CACHINGCONTROLLER

```
public class CachingController : Controller  
{  
    private IMemoryCache memoryCache;  
  
    public CachingController(IMemoryCache memoryCache)  
    {  
        this.memoryCache = memoryCache;  
    }  
  
    public IActionResult Index()  
    {  
        return View();  
    }  
  
    public IActionResult MemoriaPersonalizada()  
    {  
        string fecha = DateTime.Now.ToString("yyyy-MM-dd")  
            + " -- "  
            + DateTime.Now.ToString("HH:mm:ss");  
        //DEBEMOS PREGUNTAR SI EXISTE ALGO EN CACHE O NO  
        if (this.memoryCache.Get("FECHA") == null)  
        {  
            //NO EXISTE EN CACHE TODAVIA  
            this.memoryCache.Set("FECHA", fecha);  
            ViewData["MENSAJE"] = "Fecha almacenada en Cache";  
            ViewData["FECHA"] = this.memoryCache.Get("FECHA");  
        }  
        else  
        {  
            fecha = this.memoryCache.Get("FECHA");  
            ViewData["MENSAJE"] = "Fecha recuperada de Cache";  
            ViewData["FECHA"] = fecha;  
        }  
        return View();  
    }  
  
    [ResponseCache(Duration = 60,  
        Location = ResponseCacheLocation.Client)]  
    public IActionResult MemoriaDistribuida()  
    {  
        string fecha =  
            DateTime.Now.ToString("yyyy-MM-dd") + " -- "  
            + DateTime.Now.ToString("HH:mm:ss");  
        ViewData["FECHA"] = fecha;  
        return View();  
    }  
}
```

MEMORIAPERSONALIZADA.CSHTML

```
<h1>Memoria Personalizada</h1>
```

```

<p>
    <a asp-controller="Caching"
        asp-action="MemoriaPersonalizada">
        Actualizar página
    </a>
</p>
<h2 style="color:blue">@ ViewData["MENSAJE"]</h2>
<h2 style="color:red">@ ViewData["FECHA"]</h2>

```

A continuación, vamos a modificar la memoria personalizada indicando el tiempo de Duración del Cache.

Para ello se realiza mediante el objeto **MemoryCacheEntryOptions**

La memoria Cache se establece en Segundos.

Incluimos en la página dos nuevos Links enviando el tiempo de duración

```

<p>
    <a asp-controller="Caching"
        asp-action="MemoriaPersonalizada"
        asp-route-tiempo="15">
        Actualizar página 15 segundos
    </a>
</p>
<p>
    <a asp-controller="Caching"
        asp-action="MemoriaPersonalizada"
        asp-route-tiempo="30">
        Actualizar página 30 segundos
    </a>
</p>

```

CACHINGCONTROLLER

```

public IActionResult MemoriaPersonalizada(int? tiempo)
{
    if (tiempo == null)
    {
        tiempo = 60;
    }
    string fecha = DateTime.Now.ToString("dd/MM/yyyy HH:mm:ss")
        + " -- "
        + DateTime.Now.ToString("dd/MM/yyyy HH:mm:ss");
    //DEBEMOS PREGUNTAR SI EXISTE ALGO EN CACHE O NO
    if (this.memoryCache.Get("FECHA") == null)
    {
        //NO EXISTE EN CACHE TODAVIA
        //CREAMOS EL OBJETO ENTRY OPTIONS CON EL TIEMPO
        MemoryCacheEntryOptions options =
            new MemoryCacheEntryOptions()
                .SetAbsoluteExpiration(TimeSpan.FromSeconds(tiempo.Value));

        this.memoryCache.Set("FECHA", fecha, options);
        ViewData["MENSAJE"] = "Fecha almacenada en Cache";
        ViewData["FECHA"] = this.memoryCache.Get("FECHA");
    }
    else
    {
        fecha = this.memoryCache.Get("FECHA");
        ViewData["MENSAJE"] = "Fecha recuperada de Cache";
        ViewData["FECHA"] = fecha;
    }
    return View();
}

```

Y podremos comprobar el resultado

Memoria Personalizada

[Actualizar página](#)

[Actualizar página 15 segundos](#)

[Actualizar página 30 segundos](#)

Fecha almacenada en Cache

jueves, 20 de febrero de 2025 -- 9:37:47

CSRF

jueves, 20 de febrero de 2025 13:50

CSRF (CROSS SITE REQUEST FORGERY)

Un Hacker está en pijama en casa

Creamos un nuevo proyecto llamado **MvcNetCoreCSRF**

Habilitamos **Session** en **Program**

PROGRAM

```
// Add services to the container.  
builder.Services.AddControllersWithViews();  
builder.Services.AddDistributedMemoryCache();  
builder.Services.AddSession();
```

```
var app = builder.Build();
```

```
app.UseAuthorization();
```

```
app.MapStaticAssets();
```

```
app.UseSession();
```

```
app.MapControllerRoute(
```

Tendremos una página para comprar productos

Para poder comprar, necesito estar dentro del Sistema mediante un Login, es decir, Vamos a emular una validación.

Simplemente, si el usuario hace Login, lo almacenamos en Session y ya podría comprar.

Sobre **Views** creamos una carpeta llamada **Managed** y dentro una vista llamada **Login.cshtml**

LOGIN.CSHTML

```
<h1>Log In Usuario</h1>  
  
<form method="post">  
    <label>Usuario</label>  
    <input type="text" name="usuario" class="form-control"/>  
    <label>Password</label>  
    <input type="password" name="password" class="form-control"/>  
    <button class="btn btn-dark">  
        Log In  
    </button>  
</form>  
<h2 style="color: red">@ViewData["MENSAJE"]</h2>
```

Creamos otra página llamada **Denied.cshtml**

DENIED.CSHTML

```
<h2 style="color: red">Acceso denegado</h2>  
  

```

Sobre **Views** creamos una carpeta llamada **Tienda** y una vista llamada **Productos.cshtml**

PRODUCTOS.CSHTML

```
<h1>Listado de Productos</h1>  
  
<form method="post">  
    <label>Dirección de envío</label>  
    <input type="text" name="direccion" class="form-control"/>  
    <button class="btn btn-info">  
        Realizar pedido  
</button>
```

```

<ul class="list-group">
    <li class="list-group-item">
        <input type="checkbox" name="producto"
               value="Tesla 40.000€" /> Tesla 40.000€
    </li>
    <li class="list-group-item">
        <input type="checkbox" name="producto"
               value="iPhone 19 5.000€" /> iPhone 19 5.000€
    </li>
    <li class="list-group-item">
        <input type="checkbox" name="producto"
               value="Ordenador Gaming 2.500€" /> Ordenador Gaming 2.500€
    </li>
    <li class="list-group-item">
        <input type="checkbox" name="producto"
               value="Arbitros Negreira 900.000€" /> Arbitros Negreira 900.000€
    </li>
</ul>
</form>

```

Creamos otra página dentro de Tienda llamada PedidoFinal.cshtml

PEDIDOFINAL.CSHTML

```

@model string[]
<h1 style="color:blue">
    Pedido realizado correctamente!!!
</h1>
<h2 style="color:green">
    Dirección del pedido: @ViewData["DIRECCION"]
</h2>
<p>Productos del pedido</p>
<ul class="list-group">
    @foreach (string producto in Model){
        <li class="list-group-item list-group-item-info">
            @producto
        </li>
    }
</ul>

```

Sobre _Layout.cshtml incluimos los Links para productos y para Log In

_LAYOUT.CSHTML

```

<li class="nav-item">
    <a class="nav-link text-dark"
       asp-controller="Managed"
       asp-action="Login">Log In</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark"
       asp-controller="Tienda"
       asp-action="Productos">
        Comprar productos
    </a>
</li>

```

Sobre Controllers creamos un nuevo controlador llamado ManagedController

MANAGEDCONTROLLER

```

public class ManagedController : Controller
{
    public IActionResult Login()
    {
        return View();
    }

    [HttpPost]
    public IActionResult Login(string usuario, string password)
    {
        if (usuario.ToLower() == "admin"
            && password.ToLower() == "admin")
        {
            HttpContext.Session.SetString("USUARIO", usuario);
            return RedirectToAction("Productos", "Tienda");
        }
        else
        {
            ViewData["MENSAJE"] = "Usuario/Password incorrectos";
            return View();
        }
    }

    public IActionResult Denied()
    {
        return View();
    }
}

```

Sobre Controllers creamos un nuevo controlador llamado TiendaController

TIENDACONTROLLER

```

public class TiendaController : Controller
{

```

```

public IActionResult Productos()
{
    //SI EL USUARIO NO SE HA VALIDADO TODAVIA
    //LO LLEVAMOS A DENIED
    if (HttpContext.Session.GetString("USUARIO") == null)
    {
        return RedirectToAction("Denied", "Managed");
    }
    return View();
}

[HttpPost]
public IActionResult Productos(string direccion
    , string[] producto)
{
    if (HttpContext.Session.GetString("USUARIO") == null)
    {
        return RedirectToAction("Denied", "Managed");
    }
    else
    {
        //MEDIANTE TEMPDATA SE ALMACENA LA INFORMACION
        //PARA SER ENVIADA A OTROS CONTROLLERS O
        //METODOS IACTIONRESULT EN LAS REDIRECCIONES
        TempData["PRODUCTOS"] = producto;
        TempData["DIRECCION"] = direccion;
        return RedirectToAction("PedidoFinal");
    }
}

public IActionResult PedidoFinal()
{
    //AQUI NECESITO LOS PRODUCTOS Y LA DIRECCION
    //DEL METODO POST DE PRODUCTOS. COMO LOS RECUPERO AQUI??
    string[] productos = TempData["PRODUCTOS"]
        as string[];
    ViewData["DIRECCION"] = TempData["DIRECCION"];
    return View(productos);
}

```

Probamos el funcionamiento de nuestra App.

Log In Usuario

Usuario
admin

Password
admin

Log In

Usuario/Password incorrectos

Listado de Productos

Dirección de envío

Mi casita segura

Realizar pedido

Tesla 40.000€

iPhone 19 5.000€

Ordenador Gaming 2.500€

Arbitros Negreira 900.000€

Pedido realizado correctamente!!!

Dirección del pedido: Mi casita segura

Productos del pedido

iPhone 19 5.000€

Ordenador Gaming 2.500€

El siguiente paso que vamos a realizar será una App rápida, como si la tuvieramos publicada
Que sería el cebo para nuestro Pichón.

Página para los incautos

Juega conmigo...



Sin cerrar el otro proyecto y NO en la misma solución, abrimos otra venta de Visual Studio
Y creamos un proyecto llamado MvcPichon

Trabajamos sobre la vista Home/Index

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="text-center">
    <h1 class="display-4">Página para los incautos</h1>
    /* COPIAMOS AQUI EL FORMULARIO DEL PRODUCTOS */
    <form method="post"
        action="https://localhost:7187/Tienda/Productos">
        <input type="hidden" name="direccion" />
        <button class="btn btn-danger">
            Juega conmigo...
        </button>
        <input type="hidden" name="producto"
            value="Tesla 40.000€" />
        <input type="hidden" name="producto"
            value="iPhone 19 5.000€" />
        <input type="hidden" name="producto"
            value="Ordenador Gaming 2.500€" />
        <input type="hidden" name="producto"
            value="Arbitros Negreira 900.000€" />
    </form>
    
</div>
```

El problema que estamos visualizando está en que NO SABE de dónde viene la petición.
Simplemente nos están enviando un formulario y dicho formulario puede estar dentro
De nuestra página o externo, no importa, es un envío a un Form de forma externa.

Para solucionar esto, debo estar seguro que la petición que se ha realizado es
Desde mi propia página.

¿Cómo se realiza esto? Con un RANDOM.

La solución está en generar un valor aleatorio dentro de la página
Dicho valor lo almacenamos dentro de una Cookie.
Cuando se realice la petición a nuestro FORM, leemos la Cookie y el valor aleatorio.
Si coinciden, la petición viene de mi equipo.

Por suerte, dentro de Net Core esto está automatizado.

Mediante **decoraciones** y código **Razor** podemos hacer que se genere el valor

Aleatorio, la comprobación y la Cookie.

- 1) **AntiForgeryToken**: Debemos incluir, dentro de nuestras etiquetas `<form>` La generación de un Token y el almacenamiento de la Cookie. Se realiza mediante `@Html.AntiForgeryToken()`

```
<form method="post">
    @Html.AntiForgeryToken()
    <label>Dirección de envío</label>
```

- 1) En los métodos POST, para comprobar la validación debemos decorarlos con El atributo `[ValidateAntiForgeryToken]`

```
[ValidateAntiForgeryToken]
[HttpPost]
0 references | serraguti, 53 minutes ago | 1 author, 1 change
public IActionResult Productos(string dirección
    , string[] producto)
```

- 1) Indicar en `Program` que hemos habilitado los servicios para `AntiForgeryToken`

```
// Add services to the container. I
builder.Services.AddAntiforgery();
```

Y ya estaremos protegidos de este determinado ataque.

The screenshot displays two browser windows side-by-side. Both windows have the URL `localhost:7187/Tienda/Productos` in the address bar.

The left window shows a successful page titled "Listado de Productos". It contains a form with a text input field labeled "Dirección de envío" and a blue button labeled "Realizar pedido". Below the button is a list of items with checkboxes:

- Tesla 40.000€
- iPhone 19 5.000€
- Ordenador Gaming 2.500€
- Arbitros Negreira 900.000€

The right window shows a 404 error page with the title "Esta página no funciona". The page includes the message "Si el problema persiste, ponte en contacto con el propietario del sitio web.", the error code "HTTP ERROR 400", and a small icon of a document with a red X.

MAILS NET CORE

lunes, 24 de febrero de 2025 9:11

Para enviar mails necesitamos una cuenta de correo personal.
Para el proyecto, crearos una cuenta diferente a la vuestra, por si acaso

Dependiendo de la cuenta, tendremos una configuración u otra distinta, todo depende
Del proveedor.

Por ejemplo, esta funcionalidad nos puede servir para varias características:

- Si el cliente realiza una compra, enviar un PDF con la factura
- El típico mail de confirmación de cuenta, cuando nos validamos en una página,
Podemos enviar un Token asociado al usuario y, hasta que no pulse en un mail
Y recibamos ese Token, el usuario no estará activo en nuestra página



Host Smtip: smtp.mail.yahoo.com
Puerto: 25
EnableSsl: false
UseDefaultCredentials: false



Host Smtip: smtp-mail.outlook.com
Puerto = 587
EnableSsl: true
UseDefaultCredentials: false



Host Smtip: smtp.gmail.com
Puerto = 587
EnableSsl: true
UseDefaultCredentials: false

GMAIL no funciona mediante credenciales actualmente, debemos habilitar, en la cuenta
La utilización de Tokens por terceras Apps.

Vamos a utilizar nuestro usuario y Password con **Microsoft**

Necesitamos contraseña de App en Outlook (2025)

qdxcusrujjksgdham

alumnotajamar2024@outlook.com

Tajamar123

Necesitamos lo siguiente:

- **Namespace:** System.Net.Mail
- **MailMessage:** Es el propio mensaje, con su asunto y sus características
- **Attachment:** Podríamos tener la posibilidad de utilizar ficheros adjuntos
- **SmtpClient:** Es la clase encargada de enviar los mails.

La información es sensible, los datos nunca debería estar en el código.
Deberían estar dentro de **appsettings.json** y, además, tendríamos que crear un **.gitignore** para el fichero para no subirlo a Git si lo tenemos en modo PUBLIC.

Necesitamos recuperar **Iconfiguration** en cualquier lugar de nuestra App.
La clase Configuration ya viene definida e inyectada dentro de **Program**

```
builder.Configuration.GetConnectionString("sql....")
```

Si son claves de otro tipo, se recuperan de la siguiente forma:

Ejemplo:

```
{
    "Logging": {
        "aaa"
    },
    "ConnectionStrings": {
        "SqlHospital": "Data...."
    },
    "ServerMail": {
        "Host": "...",
        "Port": "...",
        "Cosas": "...."
    }
}
```

```
builder.Configuration.GetValue<string>("ServerMail:Host")
```

Sobre **Views** creamos una carpeta llamada **Mails** y una vista llamada **SendMail.cshtml**

SENDMAIL.CSHTML

```
<h1>Send Mail Net Core</h1>

<form method="post">
    <Label>To: </label>
    <input type="text" class="form-control" name="to"/>
    <label>Asunto: </label>
    <input type="text" class="form-control" name="asunto"/>
    <label>Mensaje</label>
    <textarea name="mensaje" class="form-control"></textarea>
    <button class="btn btn-info">
        Send Mail
    </button>
</form>
<h2 style="color:blue">@ViewData["MENSAJE"]</h2>
```

Sobre **APPSETTINGS.JSON** agregamos las claves para nuestro servicio de Mail

APPSETTINGS.JSON

```

},
"AllowedHosts": "*",
"MailSettings": {
    "Credentials": {
        "User": "alumnotajamar2024@outlook.com",
        "Password": "Tajamar123"
    },
    "Server": {
        "Host": "smtp-mail.outlook.com",
        "Port": "587",
        "Ssl": "true",
        "DefaultCredentials": "false"
    }
}

```

Sobre **Controllers** creamos un nuevo controlador llamado **MailsController**

MAILSCONTROLLERS

```

public class MailsController : Controller
{
    //NECESITAMOS EL FICHERO DE CONFIGURATION
    private IConfiguration configuration;

    public MailsController(IConfiguration configuration)
    {
        this.configuration = configuration;
    }

    public IActionResult SendMail()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        SendMail(string to, string asunto
        , string mensaje)
    {
        MailMessage mail = new MailMessage();
        //DEBEMOS INDICAR EL FROM, ES DECIR, DE QUE CUENTA VIENE
        //EL CORREO (LA NUESTRA DE APPSETTINGS.JSON)
        string user = this.configuration.GetValue<string>
            ("MailSettings:Credentials:User");
        mail.From = new MailAddress(user);
        //LOS DESTINATARIOS SON UNA COLECCION
        mail.To.Add(to);
        mail.Subject = asunto;
        mail.Body = mensaje;
        //<h1>holo</h1>
        mail.IsBodyHtml = true;
        mail.Priority = MailPriority.Normal;
        string password = this.configuration.GetValue<string>
            ("MailSettings:Credentials:Password");
        string host = this.configuration.GetValue<string>
            ("MailSettings:Server:Host");
        int port = this.configuration.GetValue<int>
            ("MailSettings:Server:Port");
        bool ssl = this.configuration.GetValue<bool>
            ("MailSettings:Server:Ssl");
        bool defaultCredentials = this.configuration.GetValue<bool>
            ("MailSettings:Server:DefaultCredentials");
    }
}

```

```
//CREAMOS LA CLASE SERVIDOR SMTP
SmtpClient smtpClient = new SmtpClient();
smtpClient.Host = host;
smtpClient.Port = port;
smtpClient.EnableSsl = ssl;
smtpClient.UseDefaultCredentials = defaultCredentials;
//CREAMOS LAS CREDENCIALES PARA EL MAIL
NetworkCredential credentials =
    new NetworkCredential(user, password);
await smtpClient.SendMailAsync(mail);
ViewData["MENSAJE"] = "Mail enviado correctamente??";
return View();
}
}
```

CRIPTOGRAFIA

lunes, 24 de febrero de 2025 10:40

Esta tecnología nos permite cifrar contenidos.
Existen múltiples clases y van cambiando cada X tiempo y, lo más importante, no existe
Nada que no se pueda describir.

Tenemos de cifrado:

- **Cifrado de doble dirección:** Permite cifrar un contenido y, mediante claves, Poder descifrar dicho contenido.
Esto se utiliza para ficheros o mails cifrados o proyectos personales.
- **Cifrado de una sola dirección:** Se utiliza en contenido sensible y no podemos Descifrarlo. Solamente se podrá comparar con el contenido cifrado de nuevo Y utilizando la misma técnica.
Este tipo de cifrado se utiliza, por ejemplo, para las Password de bbdd.

En nuestro proyecto, de todas formas, dejaremos un CAMPO LEGIBLE con nuestro Password, aunque utilicemos cifrado.

Las contraseñas se pueden almacenar de dos formas:

- 1) **String:** Podemos almacenar el string del password del usuario aunque esté cifrado Dependiendo del tipo de cifrado, es posible que nos genere caracteres extraños, Y que luego no sean representados visualmente con el valor real.
- 2) **byte[]:** almacenar el Password con bytes. El byte siempre representará la Información real aunque no se vea de forma nativa

Vamos a comenzar realizando un cifrado simple para visualizar qué necesitamos hacer.
Por ahora, iremos evolucionando, pero trabajaremos con **string**

Cifrado básico

Contenido a cifrar

12345

Resultado cifrado

□ 맞겠네扑打眞日祺媛×

Cifrar contenido **Comparar cifrado**

Para el cifrado, comenzaremos utilizando **Helpers**

Sobre **Helpers** creamos una nueva clase llamada **HelperCryptography**

HELPERCRYPTOGRAPHY

```
public class HelperCryptography
{
    //COMENZAMOS CREANDO UN METODO static PARA CIFRAR UN
    //CONTENIDO. SIMPLEMENTE DEVOLVEMOS EL TEXTO CIFRADO
    public static string EncriptarTextoBasico(string contenido)
    {
        //NECESITAMOS UN ARRAY DE BYTES PARA CONVERTIR
        //EL CONTENIDO DE ENTRADA A byte[]
        byte[] entrada;
        //AL CIFRAR EL CONTENIDO, NOS DEVUELVE BYTES[] DE SALIDA
        byte[] salida;
        //NECESITAMOS UNA CLASE QUE NOS PERMITE CONVERTIR DE
        //STRING A BYTE[] Y VICEVERSA
        UnicodeEncoding encoding = new UnicodeEncoding();
        //NECESITAMOS UN OBJETO PARA CIFRAR EL CONTENIDO
        SHA1 managed = SHA1.Create();
        //CONVERTIMOS EL CONTENIDO DE ENTRADA A byte[]
        entrada = encoding.GetBytes(contenido);
        //LOS OBJETOS PARA CIFRAR CONTIENEN UN METODO LLAMADO
    }
}
```

```

    //ComputedHash QUE RECIBEN UN ARRAY DE BYTES E
    //INTERNAMENTE HACEN COSAS Y DEVUELVE OTRO ARRAY DE BYTES
    salida = managed.ComputeHash(entrada);
    //CONVERTIMOS SALIDA A STRING
    string resultado = encoding.GetString(salida);
    return resultado;
}
}

```

Sobre **Controllers** creamos un nuevo controlador llamado **CifradosController**

CIFRADOSCONTROLLER

```

public class CifradosController : Controller
{
    public IActionResult CifradoBasico()
    {
        return View();
    }

    [HttpPost]
    public IActionResult CifradoBasico(string contenido, string resultado, string accion)
    {
        //CIFRAMOS EL CONTENIDO
        string response =
            HelperCryptography.EncryptarTextoBasico(contenido);
        if (accion.ToLower() == "cifrar")
        {
            ViewData["TEXTOCIFRADO"] = response;
        } else if (accion.ToLower() == "comparar")
        {
            //SI EL USUARIO QUIERE COMPARA, NOS ESTARA ENVIANDO
            //EL RESULTADO PARA COMPARAR
            if (response != resultado)
            {
                ViewData["MENSAJE"] = "Los datos no coinciden";
            }
            else
            {
                ViewData["MENSAJE"] = "Contenidos iguales!!!";
            }
        }
        return View();
    }
}

```

Creamos una nueva vista llamada **CifradoBasico.cshtml**

CIFRAZOBASEICO.CSHTML

```

<h1 style="color:blue">
    Cifrado básico
</h1>

<form method="post">
    <label>Contenido a cifrar</label>
    <input type="text" name="contenido" class="form-control"/>
    <label>Resultado cifrado</label>
    <textarea name="resultado" rows="5" class="form-control">@ViewData["TEXTOCIFRADO"]</textarea>
    <button class="btn btn-info" name="accion" value="cifrar">
        Cifrar contenido
    </button>
    <button class="btn btn-warning" name="accion" value="comparar">
        Comparar cifrado
    </button>
</form>
<h2 style="color:red">@ViewData["MENSAJE"]</h2>

```

Hemos realizado un cifrado donde utilizamos el método standard de **ComputeHash**
 Dependemos de dicho método, si aparece una vulnerabilidad en el método con
 El objeto SHA1, estamos vendidos.

Siempre que trabajemos con cifrados, debemos seguir una serie de patrones para
 Dificultar la posibilidad de vulnerabilidades.

- 1) Utilizar un objeto para cifrar con un mínimo de 128 bytes.
- 2) Realizar el cifrado N iteraciones
- 3) Utilizar un elemento llamado **SALT**. Un salt es un texto generado dinámicamente en
 Nuestro código y que es aleatorio. Este texto se utiliza para incluirlo dentro del
 Contenido a cifrar.

```

Salt = ?&"{$JLFKAS
Password: "12345"
Contenido = salt + password + salt
Contenido = password.insert(2, salt)

```

Creamos un nuevo diseño para el nuevo cifrado

Cifrado Eficiente

Contenido a cifrar

12345

Resultado cifrado

3罫冗登梶鈔〇〇〇妹羨〇塙閃

Cifrar contenido

Comparar cifrado

ú〇R〇ãœ¤↑¥ 1/4f'¤Mttq38〇éH@〇"í€〇

Vamos a crear un método nuevo dentro de **HelperCryptography**

HELPERCRYPTOGRAPHY

```
public class HelperCryptography
{
    //TENDREMOS UNA PROPIEDAD NUEVA PARA ALMACENAR EL
    //SALT QUE HEMOS CREADO DINAMICAMENTE
    public static string Salt { get; set; }

    //CADA VEZ QUE REALICEMOS UN CIFRADO, GENERAMOS UN
    //SALT DISTINTO.
    private static string GenerateSalt()
    {
        Random random = new Random();
        string salt = "";
        for (int i = 1; i <= 30; i++)
        {
            //GENERAMOS UN NUMERO ALEATORIO CON CODIGOS ASCII
            int aleat = random.Next(1, 255);
            char letra = Convert.ToChar(aleat);
            salt += letra;
        }
        return salt;
    }

    //CREAMOS UN METODO PARA CIFRAR DE FORMA EFICIENTE
    public static string CifrarContenido
        (string contenido, bool comparar)
    {
        if (comparar == false)
        {
            //CREAMOS UN NUEVO SALT PARA EL CIFRADO
            //Y LO ALMACENAMOS EN LA PROPIEDAD
            Salt = GenerateSalt();
        }
        //EL SALT LO PODEMOS UTILIZAR EN MULTIPLES LUGARES
        //AL INICIO, FINAL, CON INSERT
        string contenidoSalt = contenido + Salt;
        //CREAMOS UN OBJETO GRANDE PARA CIFRAR
        SHA256 managed = SHA256.Create();
        byte[] salida;
        UnicodeEncoding encoding = new UnicodeEncoding();
        salida = encoding.GetBytes(contenidoSalt);
        //CIFRAMOS EL CONTENIDO CON n ITERACIONES
        for (int i = 1; i <= 22; i++)
        {
            //REALIZAMOS EL CIFRADO SOBRE EL CIFRADO
            salida = managed.ComputeHash(salida);
        }
        //DEBEMOS LIBERAR LA MEMORIA
        managed.Clear();
        string resultado = encoding.GetString(salida);
        return resultado;
    }
}
```

Sobre el controller, creamos un nuevo método para mostrar el resultado del cifrado

CIFRADOSCONTROLLER

```
public IActionResult CifradoEficiente()
```

```

{
    return View();
}

[HttpPost]
public IActionResult
    CifradoEficiente(string contenido, string resultado, string accion)
{
    if (accion.ToLower() == "cifrar")
    {
        string response =
            HelperCryptography.CifrarContenido(contenido, false);
        ViewData["TEXTOCIFRADO"] = response;
        ViewData["SALT"] = HelperCryptography.Salt;
    }else if (accion.ToLower() == "comparar")
    {
        string response =
            HelperCryptography.CifrarContenido(contenido, true);
        if (response != resultado)
        {
            ViewData["MENSAJE"] = "Los datos no son correctos";
        }
        else
        {
            ViewData["MENSAJE"] = "Los datos son CORRECTOS!!!!";
        }
    }
    return View();
}

```

CIFRADOEFICIENTE.CSHTML

```

<h1 style="color:blue">
    Cifrado Eficiente
</h1>

<form method="post">
    <label>Contenido a cifrar</label>
    <input type="text" name="contenido" class="form-control" />
    <label>Resultado cifrado</label>
    <textarea name="resultado" rows="5" class="form-
control">@ViewData["TEXTOCIFRADO"]</textarea>
    <button class="btn btn-info" name="accion" value="cifrar">
        Cifrar contenido
    </button>
    <button class="btn btn-danger" name="accion" value="comparar">
        Comparar cifrado
    </button>
</form>
<h2 style="color:red">@ViewData["SALT"]</h2>
<h2 style="color:green">@ViewData["MENSAJE"]</h2>

```

UTILIZAR CRIPTOGRAFIA CON BBDD

Vamos a realizar una aplicación donde almacenaremos los password cifrados en BBDD

Vamos a realizar las cosas bien hechas, es decir, almacenaremos los datos a nivel de

Byte[]

En la base de datos, también almacenaremos el SALT de forma dinámica.

Abrimos SQL Server y creamos una nueva tabla **USERS** en la base de datos HOSPITAL

```

]create table USERS
(
    IDUSUARIO INT PRIMARY KEY
    , NOMBRE NVARCHAR(60)
    , EMAIL NVARCHAR(150)
    , IMAGEN NVARCHAR(150)
    , SALT NVARCHAR(50)
    , PASS VARBINARY(MAX))

```

Register

Usuario

Nombre

Email

Imagen

Password

[Create](#)[Back to List](#)

Log In

Email

Password

[Log In](#)

Usuario

IdUsuario 1

Nombre Lunes

Email lunes@gmail.com

Creamos una nueva aplicación Net Core llamada **MvcNetCoreCryptography**



Microsoft.EntityFrameworkCore by aspnet, dotnetframework, EntityFram

Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL S...



Microsoft.EntityFrameworkCore.SqlServer by aspnet, dotnetframewor

Microsoft SQL Server database provider for Entity Framework Core.

Sobre **Models** creamos una nueva clase llamada **Usuario**

USUARIO

```
[Table("USERS")]
public class Usuario
{
    [Key]
    [Column("IDUSUARIO")]
    public int IdUsuario { get; set; }
    [Column("NOMBRE")]
    public string Nombre { get; set; }
    [Column("EMAIL")]
    public string Email { get; set; }
    [Column("IMAGEN")]
    public string Imagen { get; set; }
    [Column("SALT")]
    public string Salt { get; set; }
    //UNA VENTAJA QUE TENEMOS ESTA EN QUE LOS
    //VARBINARY O BLOB SON CONVERTIDOS A byte[]
    //AUTOMATICAMENTE CON EF
    [Column("PASS")]
    public byte[] Password { get; set; }
}
```

Sobre el proyecto, creamos una carpeta llamada **Data** y una clase llamada **UsuariosContext**

USUARIOSCONTEXT

```
public class UsuariosContext : DbContext
{
    0 references
    public UsuariosContext(DbContextOptions<UsuariosContext> options)
        :base(options) { }

    0 references
    public DbSet<Usuario> Usuarios { get; set; }
}
```

Creamos una carpeta llamada **Helpers** y una clase llamada **HelperCryptography**

HELPERCRYPTOGRAPHY

```
public class HelperCryptography
{
    //VAMOS A CREAR UN PAR DE METODOS QUE NO
    //TIENEN NADA QUE VER CON CRIPTOGRAFIA
    //SI TENEMOS MUCHOS METODOS QUE SIMPLEMENTE SON
    //HERRAMIENTAS, LO SUYO ES CREAR UNA CLASE
    //HelperToolkit
    public static string GenerateSalt()
    {
        Random random = new Random();
        string salt = "";
        //EL NUMERO DE VUELTAS DEBE COINCIDIR CON
        //EL VALOR DEL CAMPO NVARCHAR
        for (int i = 1; i <= 50; i++)
        {
            int aleat = random.Next(1, 255);
            char letra = Convert.ToChar(aleat);
            salt += letra;
        }
        return salt;
    }

    //NECESITAMOS SABER SI EL PASSWORD QUE HEMOS ALMACENADO
    //EN BBDD ES IGUAL AL PASSWORD QUE NOS HABRAN DADO EN LA APP
    //ESTE ES UN METODO PARA COMPARAR DOS ARRAYS DE BYTES
    public static bool CompararArrays(byte[] a, byte[] b)
    {
        bool iguales = true;
        //COMparamos EL TAMAÑO
        if (a.Length != b.Length)
        {
            iguales = false;
        }
        else
        {
            //RECORREMOS EL ARRAY a
            for (int i = 0; i < a.Length; i++)
            {
                //COMparamos BYTE A BYTE
                if (a[i].Equals(b[i]) == false)
                {
                    iguales = false;
                    break;
                }
            }
        }
    }
}
```

```

        }
    }
    return iguales;
}

//TENDREMOS UN METODO PARA CIFRAR EL PASSWORD
//VAMOS A RECIBIR EL PASSWORD A CIFRAR (string)
//Y TAMBIEN VAMOS A RECIBIR EL SALT (string)
//DEVOLVEREMOS UN ARRAY CON EL RESULTADO
public static byte[] EncryptPassword(string password, string salt)
{
    string contenido = password + salt;
    SHA512 managed = SHA512.Create();
    //CONVERTIMOS EL CONTENIDO A byte[]
    byte[] salida = Encoding.UTF8.GetBytes(contenido);
    //CREAMOS EL BUCLE DE CIFRADO CON ITERACIONES
    for (int i = 1; i <= 15; i++)
    {
        salida = managed.ComputeHash(salida);
    }
    managed.Clear();
    return salida;
}
}

```

Creamos una carpeta llamada **Repositories** y una clase llamada **RepositoryUsuarios**

REPOSITORYUSUARIOS

```

public class RepositoryUsuarios
{
    private UsuariosContext context;

    public RepositoryUsuarios(UsuariosContext context)
    {
        this.context = context;
    }

    private async Task<int> GetMaxIdUser()
    {
        if (this.context.Usuarios.Count() == 0)
        {
            return 1;
        }
        else
        {
            return await this.context.Usuarios.MaxAsync
                (x => x.IdUsuario) + 1;
        }
    }

    public async Task RegisterUserAsync(string nombre,
        string email,
        string password, string imagen)
    {
        Usuario user = new Usuario();
        user.IdUsuario = await this.GetMaxIdUser();
        user.Nombre = nombre;
        user.Email = email;
        user.Imagen = imagen;
        //CADA USUARIO TENDRA UN SALT DIFERENTE
        user.Salt = HelperCryptography.GenerateSalt();
        //ALMACENAMOS EL PASSWORD CIFRADO A byte[]
        user.Password =
            HelperCryptography.EncryptPassword(password, user.Salt);
        this.context.Usuarios.Add(user);
        await this.context.SaveChangesAsync();
    }

    //NECESITAMOS UN METODO PARA HACER UN LOGIN DE USUARIO
    //Y DEVOLVEMOS AL USUARIO SI HEMOS COMPROBADO TODO CORRECTAMENTE
    //COMO TENEMOS CIFRADO, DEBEMOS HACER EL LOGIN PIDIENDO
    //DATOS UNICOS (email, username, nif)
    public async Task<Usuario> LogInUserAsync
        (string email, string password)
    {
        //BUSCAMOS AL USUARIO POR EL DATO UNICO (email)
        var consulta = from datos in this.context.Usuarios
                      where datos.Email == email
                      select datos;
        Usuario user = await consulta.FirstOrDefaultAsync();
        if (user == null)
        {
            return null;
        }
        else
        {
            //RECUPERAMOS EL SALT DEL USUARIO DE BBDD
            string salt = user.Salt;
            //CONVERTIMOS EL PASSWORD QUE NOS HAN DADO Y EL SALT
            //A byte[]
            byte[] temp =
                HelperCryptography.EncryptPassword(password, salt);
            //RECUPERAMOS EL PASSWORD DE BYTE[] DE BBDD
            byte[] passBytes = user.Password;
            //POR ULTIMO, REALIZAMOS LA COMPARACION DE ARRAYS
        }
    }
}

```

```

        bool response =
            HelperCryptography.CompararArrays(temp, passBytes);
        if (response == true)
        {
            return user;
        }
        else
        {
            return null;
        }
    }
}

```

Nos quedan los fuegos artificiales.

Sobre **Controllers** creamos un nuevo controlador llamado **UsuariosController**

USUARIOSCONTROLLER

```

public class UsuariosController : Controller
{
    private RepositoryUsuarios repo;

    public UsuariosController(RepositoryUsuarios repo)
    {
        this.repo = repo;
    }

    public IActionResult Register()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        Register(string nombre, string email
        , string password, string imagen)
    {
        await this.repo.RegisterUserAsync(nombre, email,
            password, imagen);
        ViewData["MENSAJE"] = "Usuario registrado correctamente";
        return View();
    }

    public IActionResult LogIn()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        LogIn(string email, string password)
    {
        Usuario user = await this.repo.LogInUserAsync(email, password);
        if (user == null)
        {
            ViewData["MENSAJE"] = "Credenciales incorrectas";
            return View();
        }
        else
        {
            return View(user);
        }
    }
}

```

Resolvemos la cadena de conexión dentro de **appsettings.json**

APPSETTINGS.JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Initial Catalog=HOSPIT"
  }
}
```

Resolvemos las dependencias dentro de **Program**

PROGRAM

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital");
builder.Services.AddTransient<RepositoryUsuarios>();
builder.Services.AddDbContext<UsuariosContext>
    (options => options.UseSqlServer(connectionString));

var app = builder.Build();
```

Simplemente un par de recomendaciones para vuestro proyecto si utilizamos criptografía:

- 1) Dejar un campo limpio para la contraseña donde se vea
- 2) Como tenemos elementos que forman parte de la App y no son parte del Diseño, no deberíamos mostrar ni salt ni nada en nuestra app.
Necesitamos trabajar con ello, pero en el fondo, es un lastre que no nos sirve

Tener la clase de Usuario para cuando trabajemos con sus modificaciones o alta.
Tener una clase **funcional** para representar datos en nuestra App, es decir,
Una vista de Usuario sin esos datos.

SESSION FUNCIONAL

martes, 25 de febrero de 2025 9:17

Hemos visto la teoría de Session, hemos trabajado con fechas incluyendo extensiones y Objetos para representar los datos almacenados.

En realidad, lo que quiero ver es la funcionalidad de Session como un carrito.

Trabajaremos con **Empleados** como si fuera un **Producto**

Podremos visualizar elementos del carrito, podremos quitar elementos del Carrito o hacer Compras de Producto.

Comenzaremos creando una nueva lógica en la que solamente almacenaremos el Salario de los empleados. Iremos seleccionando y guardando el Salario y lo mostraremos En otra página.

Session salarios Empleados

[Mostrar suma Salarial](#)

SANCHASANCCHA [Almacenar salario: 104000](#)
ARROYOARROYO [Almacenar salario: 208000](#)
SALASALA [Almacenar salario: 162500](#)
JIMENEZJIMENEZ [Almacenar salario: 386750](#)
MARTINEZMARTINEZ [Almacenar salario: 182000](#)

Suma salarial

[Volver a salarios empleados](#)

Suma salarial: 552500

Comenzamos creando una nueva App llamada **MvcNetCoreSessionEmpleados**

 Microsoft.EntityFrameworkCore by aspnet, dotnetframework, EntityFramework, Microsoft.EntityFrameworkCore 9.0.2
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Data...

 Microsoft.EntityFrameworkCore.SqlServer by aspnet, dotnetframework, EntityFramework, Microsoft.EntityFrameworkCore.SqlServer 9.0.2
Microsoft SQL Server database provider for Entity Framework Core.

 Newtonsoft.Json by dotnetfoundation, jamesnk, newtonsoft, 5.8B downloads 13.0.3
Json.NET is a popular high-performance JSON framework for .NET

Sobre **Models** creamos una nueva clase llamada **Empleado**

EMPLEADO

```
[Table("EMP")]
public class Empleado
{
    [Key]
    [Column("EMP_NO")]
    public int IdEmpleado { get; set; }
    [Column("APELLIDO")]
    public string Apellido { get; set; }
    [Column("OFICIO")]
    public string Oficio { get; set; }
    [Column("SALARIO")]
    public int Salario { get; set; }
    [Column("DEPT_NO")]
    public int IdDepartamento { get; set; }
}
```

Creamos una carpeta llamada **Data** y una clase llamada **HospitalContext**

HOSPITALCONTEXT

```

public class HospitalContext: DbContext
{
    0 references
    public HospitalContext(DbContextOptions<HospitalContext> options)
        : base(options) { }

    0 references
    public DbSet<Empleado> Empleados { get; set; }
}

```

Creamos una carpeta llamada **Repositories** y una clase llamada **RepositoryEmpleados**

REPOSITORYEMPLEADOS

```

public class RepositoryEmpleados
{
    private HospitalContext context;

    public RepositoryEmpleados(HospitalContext context)
    {
        this.context = context;
    }

    public async Task<List<Empleado>> GetEmpleadosAsync()
    {
        var consulta = from datos in this.context.Empleados
                      select datos;
        return await consulta.ToListAsync();
    }

    public async Task<Empleado> FindEmpleadoAsync(int idEmpleado)
    {
        var consulta = from datos in this.context.Empleados
                      where datos.IdEmpleado == idEmpleado
                      select datos;
        return await consulta.FirstOrDefaultAsync();
    }
}

```

Creamos una carpeta llamada **Extensions** y una clase llamada **SessionExtension**

SESSIONEXTENSION

```

public static class SessionExtension
{
    public static void SetObject
        (this ISession session, string key, object value)
    {
        string json =
            JsonConvert.SerializeObject(value);
        session.SetString(key, json);
    }

    public static T GetObject<T>
        (this ISession session, string key)
    {
        string data = session.GetString(key);
        if (data == null)
        {
            return default(T);
        }
        else
        {
            return JsonConvert.DeserializeObject<T>(data);
        }
    }
}

```

Sobre **_ViewImports** incluimos el using para la clase **SessionExtension**

_VIEWIMPORTS.CSHMTL

```

@using MvcNetCoreSessionEmpleados
@using MvcNetCoreSessionEmpleados.Models
@using MvcNetCoreSessionEmpleados.Extensions
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

```

Sobre **Controllers** creamos un nuevo controlador llamado **EmpleadosController**

EMPLEADOSCONTROLLER

```

public class EmpleadosController : Controller
{
    private RepositoryEmpleados repo;

    public EmpleadosController(RepositoryEmpleados repo)
    {
        this.repo = repo;
    }

    public async Task<IActionResult> SessionSalarios(int? salario)
    {
        if (salario != null)
        {
            //NECESITAMOS ALMACENAR EL SALARIO DEL EMPLEADO
            //Y LA SUMA TOTAL DE SALARIOS QUE TENGAMOS
            int sumaSalarial = 0;

```

```

//PREGUNTAMOS SI YA TENEMOS LA SUMA ALMACENADA EN SESSION
if (HttpContext.Session.GetString("SUMASALARIAL") != null)
{
    //SI YA EXISTE LA SUMA SALARIAL, RECUPERAMOS
    //SU VALOR
    sumaSalarial =
        HttpContext.Session.GetObject<int>("SUMASALARIAL");
}
//REALIZAMOS LA SUMA
sumaSalarial += salario.Value;
//ALMACENAMOS EL NUEVO VALOR DE LA SUMA SALARIAL
//DENTRO DE SESSION
HttpContext.Session.SetObject("SUMASALARIAL", sumaSalarial);
ViewData["MENSAJE"] = "Salario almacenado: " + salario.Value;
}

List<Empleado> empleados =
    await this.repo.GetEmpleadosAsync();
return View(empleados);
}

public IActionResult SumaSalarial()
{
    return View();
}
}

```

SESSIONSALARIOS.CSHTML

```

@model List<Empleado>

<h1>Session salarios Empleados</h1>

<p>
    <a asp-controller="Empleados"
       asp-action="SumaSalarial">
        Mostrar suma Salarial
    </a>
</p>

<h4>@ViewData["MENSAJE"]</h4>

<ul class="list-group">
    @foreach (Empleado empleado in Model){
        <li class="list-group-item">
            @empleado.Apellido
            <a asp-controller="Empleados"
               asp-action="SessionSalarios"
               asp-route-salario="@empleado.Salario">
                Almacenar salario: @empleado.Salario
            </a>
        </li>
    }
</ul>

```

SUMASALARIAL.CSHTML

```

<h2>Suma salarial</h2>

<p>
    <a asp-controller="Empleados"
       asp-action="SessionSalarios">
        Volver a salarios empleados
    </a>
</p>

@if (Context.Session.GetString("SUMASALARIAL") == null){
    <h2 style="color:red">
        No tiene almacenados salarios en Session
    </h2>
}
else{
    <h2 style="color:blue">
        Suma salarial: @Context.Session.GetString("SUMASALARIAL")
    </h2>
}

```

APPSETTINGS.JSON

```

nastore.org/appsettings.json
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Initial Catalog=HOSP"
    }
}

```

PROGRAM

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();

string connectionString =
    builder.Configuration.GetConnectionString("SqlHospital");
builder.Services.AddTransient<RepositoryEmpleados>();
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseSqlServer(connectionString));

```

```

app.MapStaticAssets();
app.UseSession();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
    .WithStaticAssets();

```

VERSION 2

El siguiente paso es almacenar empleados en Session
En lugar de almacenar el salario, almacenamos un objeto Empleado
Lo que tendremos en Session será un conjunto de empleados, es decir, una colección List

Tendremos un carrito donde visualizaremos los empleados almacenados

Session Empleados List

[Empleados Almacenados](#)

SANCHA [Almacenar empleado](#)

ARROYO [Almacenar empleado](#)

SALA [Almacenar empleado](#) 

Empleados Almacenados Session

[Volver a almacenar empleados](#)

Apellido	Oficio	Salario	Departamento
SANCHA	EMPLEADO	104000	20
ARROYO	VENDEDOR	208000	30

Comenzamos creando un **IActionResult** nuevo sobre **EmpleadosController**

EMPLEADOSCONTROLLER

Acabamos de almacenar dentro de Session objetos Empleado, concretamente una Colección.
Si multiplicamos cada objeto Empleado/Producto de cada usuario con su lista por El número de usuarios que podrían estar almacenando datos en Session del servidor, Es eficiente?

VERSION 3

Deberíamos almacenar el ID de cada Empleado
Dentro de Session tendremos un conjunto de Ids.
¿Cómo recuperar esos Ids para convertirlos en Empleados con EF?

Aquí recuperamos por un valor, pero no tiene lógica ejecutar esta consulta por cada Empleado
En Session.

```
var consulta = from datos in this.context.Empleados  
               where datos.IdEmpleado == idEmpleado  
               select datos;
```

Necesitamos utilizar una consulta IN:

```
Select * from EMP where EMP_NO in (4521, 7839, 7566)
```

Mediante EF, para poder realizar esta consulta se utiliza **Contains** de la clase **List**

Session Empleados OK

[Empleados Almacenados](#)

Empleados almacenados: 3

SANCHА [Almacenar empleado](#)

ARROYO [Almacenar empleado](#)

SALA [Almacenar empleado](#)

JIMENEZ [Almacenar empleado](#)

Empleados Almacenados Session OK

[Volver a almacenar empleados](#)

Apellido	Oficio	Salario	Departamento
SANCHА	EMPLEADO	104000	20
ARROYO	VENDEDOR	208000	30
JIMENEZ	DIRECTOR	386750	20

Comenzamos por **EmpleadosController**

EMPLEADOSCONTROLLER

Necesitamos utilizar una consulta que nos devuelva los empleados de Session.

Dicha consulta la recuperamos de la base de datos.

Necesitamos un método, dentro del Repository que reciba una colección de IDS y
Nos devuelva los Empleados.

REPOSITORYEMPLEADOS

```

public async Task<List<Empleado>>
    0 references | 0 changes | 0 authors, 0 changes
GetEmpleadosSessionAsync(List<int> ids)
{
    var consulta = from datos in this.context.Empleados
                  where ids.Contains(datos.IdEmpleado)
                  select datos;
    if (consulta.Count() == 0)
    {
        return null;
    }
    else
    {
        return await consulta.ToListAsync();
    }
}

```

Implementamos un método **IActionResult** para los empleados de Session dentro de **EmpleadosController**

EMPLEADOSCONTROLLER

```

public async Task<IActionResult> EmpleadosAlmacenadosOK()
{
    //DEBEMOS RECUPERAR LOS IDS DE EMPLEADOS QUE TENGAMOS
    //EN SESSION
    List<int> idsEmpleados =
        HttpContext.Session.GetObject<List<int>>("IDSEMPLEADOS");
    if (idsEmpleados == null)
    {
        ViewData["MENSAJE"] = "No existen empleados almacenados "
            + " en Session.";
        return View();
    }
    else
    {
        List<Empleado> empleados =
            await this.repo.GetEmpleadosSessionAsync(idsEmpleados);
        return View(empleados);
    }
}

```

Creamos la vista **EmpleadosAlmacenadosOK.cshtml**

EMPLEADOSALMACENADOSOK.CSHTML

```

@model List<Empleado>
 @{
    ViewData["Title"] = "EmpleadosAlmacenadosOK";
}

<h1 style="color:blue">Empleados Almacenados Session OK</h1>
<p>
    <a asp-controller="Empleados" asp-action="SessionEmpleadosOK">
        Volver a almacenar empleados
    </a>
</p>
<h2 style="color:red">
    @ViewData["MENSAJE"]
</h2>
@if (Model != null)
{
    <table class="table table-bordered table-warning">
        <thead>
            <tr>
                <th>Apellido</th>
                <th>Oficio</th>
                <th>Salario</th>
                <th>Departamento</th>
            </tr>
        </thead>
        <tbody>
            @foreach (Empleado emp in Model)
            {
                <tr>
                    <td>@emp.Apellido</td>
                    <td>@emp.Oficio</td>
                    <td>@emp.Salario</td>
                    <td>@emp.IdDepartamento</td>
                </tr>
            }
        </tbody>
    </table>
}

```

VERSION 4

No queremos que existan repetidos dentro de la lista de Session
En lugar de dibujar todos los datos de Empleados, podríamos dibujar los datos
De los empleados que NO estén en Session.

Select * from EMP where EMP_NO not in (4521, 7839)

Una vez que tengamos almacenados empleados, no los quiero en el dibujo principal
Para almacenar.

CONTROL + C + CONTROL + V

Creamos un nuevo ActionResult llamado **SessionEmpleadosV4**

EmpleadosAlmacenadosV4

Comenzamos creando un método que nos devuelvan los empleados que NO están en nuestro Session.

REPOSITORYEMPLEADOS

```
public async Task<List<Empleado>>
{
    0 references | 0 changes | 0 authors, 0 changes
    GetEmpleadosNotSessionAsync(List<int> ids)
    {
        var consulta = from datos in this.context.Empleados
                      where ids.Contains(datos.IdEmpleado) == false
                      select datos;
        if (consulta.Count() == 0)
        {
            return null;
        }
        else
        {
            return await consulta.ToListAsync();
        }
    }
}
```

Debemos incluir un código para que nos devuelva los empleados que NO están en Session.
Si Session no existe, devolvemos todos.

EMPLEADOSCONTROLLER

```
public async Task<IActionResult>
    SessionEmpleadosV4(int? idEmpleado)
{
    if (idEmpleado != null)
    {
        //ALMACENAREMOS LO MINIMO QUE PODAMOS (int)
        List<int> idsEmpleados;
        if (HttpContext.Session.GetObject<List<int>>("IDSEMPLEADOS") == null)
        {
            //NO EXISTE Y CREAMOS LA COLECCION
            idsEmpleados = new List<int>();
        }
        else
        {
            //EXISTE Y RECUPERAMOS LA COLECCION
            idsEmpleados =
                HttpContext.Session.GetObject<List<int>>("IDSEMPLEADOS");
        }
        idsEmpleados.Add(idEmpleado.Value);
        //REFRESCAMOS LOS DATOS DE SESSION
        HttpContext.Session.SetObject("IDSEMPLEADOS", idsEmpleados);
        ViewData["MENSAJE"] = "Empleados almacenados: "
            + idsEmpleados.Count;
    }
    //COMPROBAMOS SI TENEMOS IDS EN SESSION
    List<int> ids =
        HttpContext.Session.GetObject<List<int>>("IDSEMPLEADOS");
    if (ids == null)
    {
        List<Empleado> empleados =
            await this.repo.GetEmpleadosAsync();
        return View(empleados);
    }
    else
    {
        List<Empleado> empleados =
            await this.repo.GetEmpleadosNotSessionAsync(ids);
        return View(empleados);
    }
}

public async Task<IActionResult> EmpleadosAlmacenadosV4()
{
    //DEBEMOS RECUPERAR LOS IDS DE EMPLEADOS QUE TENGAMOS
    //EN SESSION
    List<int> idsEmpleados =
        HttpContext.Session.GetObject<List<int>>("IDSEMPLEADOS");
    if (idsEmpleados == null)
    {
        ViewData["MENSAJE"] = "No existen empleados almacenados "
            + " en Session.";
        return View();
    }
    else
    {
        List<Empleado> empleados =
            await this.repo.GetEmpleadosSessionAsync(idsEmpleados);
        return View(empleados);
    }
}
```

VERSION 5

Hemos realizado un planteamiento para mostrar solamente los datos que NO estén Almacenados en Session.

La siguiente idea radica en volver a mostrar TODOS, estén en Session o no estén en Session. No vamos a eliminar al empleado al Pulsar, es decir, se mantendrá en el dibujo. Pero Si que debemos mostrar que está dentro de Session.

Si el empleado está en Session, mostramos una imagen de Check
Si no está, mostramos el Link de **Almacenar Empleado**

Session Empleados V5

[Empleados Almacenados V5](#)

SANCHА	
ARROYO	
SALA	
JIMENEZ	Almacenar empleado
MARTINEZ	Almacenar empleado
NEGRO	Almacenar empleado

```
@model List<Empleado>
@{
    //NECESITAMOS SESSION PARA SABER EL DIBUJO A REALIZAR
    List<int> ids =
        Context.Session.GetObject<List<int>>("IDSEMPLEADOS");
}

<h1 style="color:blue">Session Empleados V5</h1>

<p>
    <a asp-controller="Empleados"
       asp-action="EmpleadosAlmacenadosV5">
        Empleados Almacenados V5
    </a>
</p>

<h4>@ViewData["MENSAJE"]</h4>

<ul class="list-group">
    @foreach (Empleado empleado in Model)
    {
        <li class="list-group-item list-group-item-info">
            @empleado.Apellido
            @* PREGUNTAMOS SI EXISTE SESSION *@
            @if (ids == null) {
                <a asp-controller="Empleados"
                   asp-action="SessionEmpleadosV5"
                   asp-route-idempleado="@empleado.IdEmpleado">
                    Almacenar empleado
                </a>
            } else {
                //PREGUNTAMOS SI EL EMPLEADO QUE ESTAMOS DIBUJANDO
                //ESTA DENTRO DE SESSION
                if (ids.Contains(empleado.IdEmpleado))
                {
                    //DIBUJAMOS LA IMAGEN
                    
                }
            }
        </li>
    }
</ul>
```

VERSION 6

Ya no es necesario crear otro V6. Trabajamos sobre V5 que será el definitivo.

En este caso, debemos trabajar sobre **EmpleadosAlmacenadosV5**

Necesitamos que el usuario pueda Eliminar/Quitar de Session a los Empleados, es decir, Un Click para poder eliminar el ID del empleado en Session.

Empleados Almacenados Session V5

[Volver a almacenar empleados V5](#)

Apellido	Oficio	Salario	Departamento	
SALA	VENDEDOR	162500	30	Eliminar Session
JIMENEZ	DIRECTOR	386750	20	Eliminar Session

Debemos modificar el controlador para poder recibir el ID del empleado a Eliminar de Session.

EMPLEADOSCONTROLLER

```
public async Task<IActionResult>
    EmpleadosAlmacenadosV5(int? idEliminar)
{
    //DEBEMOS RECUPERAR LOS IDS DE EMPLEADOS QUE TENGAMOS
    //EN SESSION
    List<int> idsEmpleados =
        HttpContext.Session.GetObject<List<int>>("IDSEMPLEADOS");
    if (idsEmpleados == null)
    {
        ViewData["MENSAJE"] = "No existen empleados almacenados "
            + " en Session.";
        return View();
    }
    else
    {
        //PREGUNTAMOS SI HEMOS RECIBIDO ALGUN VALOR
        //PARA ELIMINAR
        if (idEliminar != null)
        {
            idsEmpleados.Remove(idEliminar.Value);
            //ES POSIBLE QUE YA NO TENGAMOS EMPLEADOS EN SESSION
            if (idsEmpleados.Count == 0)
            {
                //ELIMINAMOS DE SESSION NUESTRA KEY
                HttpContext.Session.Remove("IDSEMPLEADOS");
            }
            else
            {
                //ACTUALIZAMOS SESSION CON EL EMPLEADO YA ELIMINADO
                HttpContext.Session.SetObject(
                    "IDSEMPLEADOS", idsEmpleados);
            }
        }
        List<Empleado> empleados =
            await this.repo.GetEmpleadosSessionAsync(idsEmpleados);
        return View(empleados);
    }
}
```

Modificamos la vista **EmpleadosAlmacenadosV5** para indicar si existen o no empleados Dentro de nuestra Session

EMPLEADOSALMACENADOSV5.CSHTML

```
@model List<Empleado>

@{
    ViewData["Title"] = "Empleados Almacenados Session V5";
}



# Empleados Almacenados Session V5



@if (Context.Session.GetString("IDSEMPLEADOS") == null)
{
    <h2 style="color:red">
        No existen empleados en Session
    </h2>
}

<p>
    <a asp-controller="Empleados" asp-action="SessionEmpleadosV5">
        Volver a almacenar empleados V5
    </a>
</p>
@if (Model != null)
{
    <table class="table table-bordered table-dark">
        <thead>
            <tr>
                <th>Apellido</th>
                <th>Oficio</th>
                <th>Salario</th>
                <th>Departamento</th>
                <td></td>
            </tr>
        </thead>
        <tbody>
            @foreach (Empleado emp in Model)
            {
                <tr>
                    <td>@emp.Apellido</td>
                    <td>@emp.Oficio</td>
                    <td>@emp.Salario</td>
                    <td>@emp.IdDepartamento</td>
                    <td>
                        <a asp-controller="Empleados"
                            asp-action="EmpleadosAlmacenadosV5"
                            asp-route-ideliminar="@emp.IdEmpleado">

```

```

        Eliminar Session
    </a>
    </td>
}
</tr>
</tbody>
</table>
}

```

VERSION 7

Vamos a visualizar cómo se comporta nuestra aplicación si incluimos características Mezcladas como, por ejemplo, memoria distribuida.

Combinaremos Session con Memoria distribuida

Habilitamos memoria distribuida dentro de **Program** (Ya lo tenemos)

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();

```

Incluimos decoraciones Cache dentro de los dos **Action** que tenemos

```

[ResponseCache(Duration = 80, Location = ResponseCacheLocation.Client)]
public async Task<IActionResult>
SessionEmpleadosV5(int? idEmpleado)

```

CONCLUSIONES

La memoria distribuida solamente la utilizaremos para optimizar las llamadas De Controladores cuando tengamos **Repos o Services** limpios, es decir, que no Utilicen Session por ejemplo.

Vamos a utilizar Memoria Personalizada, es decir, **Cache Personalizado**

Vamos a utilizarlo para almacenar **Favoritos**

Empleados Favoritos



[Volver a almacenar empleados V5](#)

Apellido	Oficio	Salario	Departamento
SANCHAS	EMPLEADO	104000	20
ARROYO	VENDEDOR	208000	30

Comenzamos agregando el siguiente Nuget

.NET Microsoft.Extensions.Caching.Memory by aspnet, dotnetframework, Microsoft, 9.0.2
In-memory cache implementation of Microsoft.Extensions.Caching.Memory.ILMemoryCache.

Habilitamos memoria cache personalizada dentro de Program

PROGRAM

```
// Add services to the container.
builder.Services.AddMemoryCache();
builder.Services.AddControllersWithViews();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();
```

Sobre la vista **SessionEmpleadosV5.cshtml** incluimos un nuevo Link para almacenar Los favoritos y nos enviará un parámetro nuevo llamado **idfavorito**

SESSIONEMPLEADOSV5.CSHTML

```
@foreach (Empleado empleado in Model)
{
    <li class="list-group-item list-group-item-info">
        @empleado.Apellido
        <a asp-controller="Empleados"
            asp-action="SessionEmpleadosV5"
            asp-route-idfavorito="@empleado.IdEmpleado">
            Favorito
        </a>
        @* PREGUNTAMOS SI EXISTE SESSION *@

    </li>
}
```

Debemos realizar la inyección de **IMemoryCache** dentro de **EmpleadosController**

EMPLEADOSCONTROLLER

```
public class EmpleadosController : Controller
{
    private RepositoryEmpleados repo;
    private IMemoryCache memoryCache;

    0 references | 0 changes | 0 authors, 0 changes
    public EmpleadosController
        (RepositoryEmpleados repo,
        IMemoryCache memoryCache)
    {
        this.repo = repo;
        this.memoryCache = memoryCache;
    }
}
```

```
public async Task<IActionResult>
    SessionEmpleadosV5
    (int? idEmpleado, int? idfavorito)
{
    if (idfavorito != null)
    {
        //COMO ESTOY ALMACENANDO EN CACHE DE CLIENTE, VAMOS A
        //UTILIZAR LOS OBJETOS EN LUGAR DE LOS IDS.
        List<Empleado> empleadosFavoritos;
        if (this.memoryCache.Get("FAVORITOS") == null)
        {
            //NO EXISTEN, CREAMOS LA COLECCION DE FAVORITOS
            empleadosFavoritos = new List<Empleado>();
        }
        else
        {
            //RECUPERAMOS LOS EMPLEADOS QUE TENEMOS EN LA
            //COLECCION DE FAVORITOS DE CACHE
            empleadosFavoritos =
                this.memoryCache.Get<List<Empleado>>("FAVORITOS");
        }
        //BUSCAMOS EL OBJETO EMPLEADO A ALMACENAR
        Empleado emp = await this.repo
            .FindEmpleadoAsync(idfavorito.Value);
        empleadosFavoritos.Add(emp);
        this.memoryCache.Set("FAVORITOS", empleadosFavoritos);
    }
}
```

El siguiente paso es crear un nuevo método **IActionResult** para visualizar los Favoritos

```

public IActionResult EmpleadosFavoritos()
{
    //PREGUNTAMOS SI EXISTEN FAVORITOS
    if (this.memoryCache.Get("FAVORITOS") == null)
    {
        ViewData["MENSAJE"] = "No existen favoritos almacenados";
        return View();
    }
    else
    {
        List<Empleado> favoritos =
            this.memoryCache.Get<List<Empleado>>("FAVORITOS");
        return View(favoritos);
    }
}

```

Creamos la vista EmpleadosFavoritos.cshtml

```

EMPLEADOSFAVORITOS.CSHTML

@model List<Empleado>
@{
    ViewData["Title"] = "EmpleadosFavoritos";
}

<h1 style="color:fuchsia">Empleados Favoritos</h1>
<img src("~/images/love.png" style="width: 100px; height: 100px" />

<p>
    <a asp-controller="Empleados" asp-action="SessionEmpleadosV5">
        Volver a almacenar empleados V5
    </a>
</p>
<h2 style="color:red">
    @ViewData["MENSAJE"]
</h2>

@if (Model != null)
{
    <table class="table table-bordered table-danger">
        <thead>
            <tr>
                <th>Apellido</th>
                <th>Oficio</th>
                <th>Salario</th>
                <th>Departamento</th>
            </tr>
        </thead>
        <tbody>
            @foreach (Empleado emp in Model)
            {
                <tr>
                    <td>@emp.Apellido</td>
                    <td>@emp.Oficio</td>
                    <td>@emp.Salario</td>
                    <td>@emp.IdDepartamento</td>
                </tr>
            }
        </tbody>
    </table>
}

```

Me gustaría tener un contador que indique el número de empleados que tenemos en Session

Session Empleados V5

Empleados Session: 3

[Empleados Almacenados V5](#)

Empleados almacenados: 3

SANCHÁ	Favorito	<input checked="" type="checkbox"/>
ARROYO	Favorito	<input checked="" type="checkbox"/>
SALA	Favorito	<input checked="" type="checkbox"/>

```
@model List<Empleado>

{@
    //NECESITAMOS SESSION PARA SABER EL DIBUJO A REALIZAR
    List<int> ids =
        Context.Session.GetObject<List<int>>("IDSEMPLEADOS");
}

<h1 style="color: #blue">Session Empleados V5</h1>

@if (ids != null){
    <h3 style="background-color: #blue; color: #white">
        Empleados Session: @ids.Count
    </h3>
}
}
```

Me gustaría realizar lo mismo visualizando el número de Favoritos que tengo actualmente En la misma página.

El problema está en que no existe **IMemoryCache** de forma nativa dentro de Ninguna clase de MVC Net Core, incluidas nuestras vistas.
Para poder acceder a MemoryCache, es necesario utilizar la inyección y estamos en Un triste dibujo.
Para poder acceder a objetos que tengamos inyectados dentro de nuestra App Necesitamos utilizar y recuperar dicho objeto con una sintaxis en la vista.
Esto nos permitirá acceder a cualquier objeto que hayamos incluido como Transient o Singleton o Service en **Program**

Se realiza mediante **@inject**

@inject CLASE variableObjeto

```
@using MvcNetCoreSessionEmpleados.Repositories
@model List<Empleado>
@inject RepositoryEmpleados repo

{@
    var data = await this.repo.FindEmpleadoAsync(7839);
    //NECESITAMOS SESSION PARA SABER EL DIBUJO A REALIZAR
    List<int> ids =
```

```
dos
@using Microsoft.Extensions.Caching.Memory
@inject IMemoryCache memoryCache
@model List<Empleado>

{@
    //NECESITAMOS SESSION PARA SABER EL DIBUJO A REALIZAR
    List<int> ids =
        Context.Session.GetObject<List<int>>("IDSEMPLEADOS");

    //NECESITAMOS MEMORYCACHE PARA LOS DIBUJOS
    List<Empleado> favoritos =
        memoryCache.Get<List<Empleado>>("FAVORITOS");
}
```

```

@if (favoritos != null){
    <h3 style="background-color: lightgreen">
        Favoritos almacenados: @favoritos.Count
    </h3>
}

```

Y podemos comprobar la funcionalidad

Session Empleados V5

Favoritos almacenados: 2

Empleados Session: 1

[Empleados Almacenados V5](#)

SANCHAS	Favorito	<input checked="" type="checkbox"/>
ARROYO	Favorito	Almacenar empleado
SALA	Favorito	Almacenar empleado

El objeto que hemos declarado dentro de nuestra vista es solamente para nuestra Vista, podríamos tener la posibilidad de utilizar dicho objeto de forma **global** en cualquier Vista de nuestra aplicación.

Si declaramos el `@inject` dentro de `_ViewImports`, dicho objeto será accesible Desde cualquier Vista.

`_VIEWIMPORTS.CSHTML`

```

dos
@using Microsoft.Extensions.Caching.Memory
@using MvcNetCoreSessionEmpleados
@using MvcNetCoreSessionEmpleados.Models
@using MvcNetCoreSessionEmpleados.Extensions
@inject IMemoryCache memoryCache
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

```

El último paso que nos quedaría es dibujar los favoritos SIN tener que recuperarlos desde el Controller.

Recuperar el dibujo de favoritos directamente desde las vistas.

`EMPLEADOSCONTROLLER`

```

public IActionResult EmpleadosFavoritos()
{
    return View();
}

```

`EMPLEADOSFAVORITOS.CSHTML`

```

@{
    List<Empleado> favoritos =
        memoryCache.Get<List<Empleado>>("FAVORITOS");
}

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

<h1 style="color:fuchsia">Empleados Favoritos</h1>


<p>
    <a asp-controller="Empleados" asp-action="SessionEmpleadosV5">
        Volver a almacenar empleados V5
    </a>
</p>

@if (favoritos == null){
    <h2 style="color:red">
        No existen Empleados favoritos
    </h2>
} else{

```

```

<table class="table table-bordered table-danger">
  <thead>
    <tr>
      <th>Apellido</th>
      <th>Oficio</th>
      <th>Salario</th>
      <th>Departamento</th>
    </tr>
  </thead>
  <tbody>
    @foreach (Empleado emp in favoritos)
    {
      <tr>
        <td>@emp.Apellido</td>
        <td>@emp.Oficio</td>
        <td>@emp.Salario</td>
        <td>@emp.IdDepartamento</td>
      </tr>
    }
  </tbody>
</table>
}

```

El siguiente paso es implementar la funcionalidad de Eliminar Favoritos.

Empleados Favoritos



[Volver a almacenar empleados V5](#)

Apellido	Oficio	Salario	Departamento	
SANCHAS	EMPLEADO	104000	20	
ARROYO	VENDEDOR	208000	30	

Comenzamos incluyendo un Link para eliminar dentro de **EMPLEADOSFAVORITOS.CSHTML**

```

<td>
  <a asp-controller="Empleados"
     asp-action="EmpleadosFavoritos"
     asp-route-ideliminar="@emp.IdEmpleado">
    Delete
  </a>
</td>

```

Dentro del Controller recibimos el Id para eliminar de Favoritos (Cache)

```

EMPLEADOSCONTROLLER

public IActionResult EmpleadosFavoritos(int? ideliminar)
{
  if (ideliminar != null)
  {
    List<Empleado> favoritos =
      this.memoryCache.Get<List<Empleado>>("FAVORITOS");
    //BUSCAMOS AL EMPLEADO A ELIMINAR DENTRO DE LA
    //COLECCION DE FAVORITOS
    Empleado empDelete =
      favoritos.Find(z => z.IdEmpleado == ideliminar.Value);
    favoritos.Remove(empDelete);
    if (favoritos.Count == 0)
    {
      this.memoryCache.Remove("FAVORITOS");
    }
    else
    {
      this.memoryCache.Set("FAVORITOS", favoritos);
    }
  }
  return View();
}

```

El siguiente paso es poder seleccionar un favorito solamente UNA vez.

Lo mismo que hemos hecho con Session.

Dibujamos un corazón o el botón para seleccionar favoritos

Como Dani



SESSIONEMPLEADOSV5.CSHMTL

```
@model List<Empleado>

@{
    //NECESITAMOS SESSION PARA SABER EL DIBUJO A REALIZAR
    List<int> ids =
        Context.Session.GetObject<List<int>>("IDSEMPLEADOS");

    //NECESITAMOS MEMORYCACHE PARA LOS DIBUJOS
    List<Empleado> favoritos =
        memoryCache.Get<List<Empleado>>("FAVORITOS");
}

<h1 style="color:blue">Session Empleados V5</h1>

@if (favoritos != null){
    <h3 style="background-color:lightgreen">
        Favoritos almacenados: @favoritos.Count
    </h3>
}

@if (ids != null){
    <h3 style="background-color:blue; color:white">
        Empleados Session: @ids.Count
    </h3>
}

<p>
    <a asp-controller="Empleados"
       asp-action="EmpleadosAlmacenadosV5">
        Empleados Almacenados V5
    </a>
</p>

<h4>@ViewData["MENSAJE"]</h4>

<ul class="list-group">
    @foreach (Empleado empleado in Model)
    {
        <li class="list-group-item list-group-item-info">
            @empleado.Apellido
            @if (favoritos == null){
                <a asp-controller="Empleados"
                   asp-action="SessionEmpleadosV5"
                   asp-route-idfavorito="@empleado.IdEmpleado">
                    
                </a>
            }else{
                //DEBEMOS PREGUNTAR SI EXISTE NUESTRO EMPLEADO
                //CON EL ID QUE ESTAMOS RECORRIENDO DENTRO DE LA COLECCION
                Empleado empFavorito =
                    favoritos.Find(x => x.IdEmpleado == empleado.IdEmpleado);
                if (empFavorito != null){
                    
                }else{
                    <a asp-controller="Empleados"
                       asp-action="SessionEmpleadosV5"
                       asp-route-idfavorito="@empleado.IdEmpleado">
                        
                    </a>
                }
            }

            @* PREGUNTAMOS SI EXISTE SESSION *@
            @if (ids == null) {
                <a asp-controller="Empleados"
                   asp-action="SessionEmpleadosV5"
                   asp-route-idempleado="@empleado.IdEmpleado"
                   class="btn btn-warning">
                    Almacenar empleado
                </a>
            }else{
                //PREGUNTAMOS SI EL EMPLEADO QUE ESTAMOS DIBUJANDO
                //ESTA DENTRO DE SESSION
                if (ids.Contains(empleado.IdEmpleado))
                {
                    //DIBUJAMOS LA IMAGEN
                    
                }else{
                    <a asp-controller="Empleados"
                       asp-action="SessionEmpleadosV5"
                       asp-route-idempleado="@empleado.IdEmpleado"
                       class="btn btn-warning">
                        Almacenar empleado
                    </a>
                }
            }
        </li>
    }
</ul>
```

Y si quisieramos visualizar el número de Favoritos y el número de Compras en Nuestro **NavBar**??

The screenshot shows a navigation bar with items: Home, Session Empleados V5, Favoritos, Ejemplos Session ▾, Privacy. To the right of the navigation bar, there are two status indicators: 'Compras: 3' in a blue box and 'Favoritos: 3' in a green box. Below the navigation bar, the main content area has a title 'Session Empleados V5' and two large, colored sections: a green section containing 'Favoritos almacenados: 3' and a blue section containing 'Empleados Session: 3'.

Simplemente tendríamos que incluir, dentro de nuestro **_Layout** el mismo código que Hemos utilizado en las vistas.

_LAYOUT.CSHTML

```
@{
    //NECESITAMOS SESSION PARA SABER EL DIBUJO A REALIZAR
    List<int> ids =
        Context.Session.GetObject<List<int>>("IDSEMPLEADOS");

    //NECESITAMOS MEMORYCACHE PARA LOS DIBUJOS
    List<Empleado> favoritos =
        memoryCache.Get<List<Empleado>>("FAVORITOS");
}

<li>
    @if (ids != null){
        <h6 style="background-color:blue;color:white">
            Compras: @ids.Count
        </h6>
    }
    @if (favoritos != null){
        <h6 style="background-color:lightgreen">
            Favoritos: @favoritos.Count
        </h6>
    }
</li>
```

Y si quisieramos comprar más de un Empleado/Producto???

Yo lo que haría es crear una clase con dos INT

Class Compras
 Idproducto
 Cantidad

Almacenar las Compras (IdProducto en Session)

PRACTICA CARRITO DE LA COMPRA

Necesito una página para mostrar una serie de Productos, Cubos de Rubik

Tendremos la posibilidad de mostrar detalles, crear nuevos cubos y modificar Cubos

Podremos almacenar cubos dentro de nuestro carrito de la compra (Session).
Podremos seleccionar cubos Favoritos (Cache). (Creamos un Helper para Cache para Hacerlo más sencillo SI LO VEMOS NECESARIO)

Podremos visualizar, en el Carrito los cubos que tenemos y el precio total
Podremos quitar tanto cubos favoritos como productos del carrito.

Tendremos la posibilidad de visualizar los favoritos y las compras desde la página principal

Podremos finalizar la compra e insertar dentro de una tabla que tenemos llamada **COMPRAS**

**Implementamos la posibilidad de CANTIDAD de producto en el carrito.
Si cambiamos la cantidad, cambiará el PRECIO TOTAL EN CLIENTE**

Al finalizar, quiero visualizar la compra realizada, mostrando
NOMBRE DEL CUBO, PRECIO INDIVIDUAL, CANTIDAD COMPRADA Y PRECIO FINAL Y FECHA

Los cubos tienen imágenes LOCALES, implementamos SUBIR IMAGEN

Lo hacemos en MYSQL

VISTAS PARCIALES

miércoles, 5 de marzo de 2025 9:10

Todas las vistas son parciales, es decir, van incluidas dentro de un Layout, pero podemos Tener algún tipo de vista que, por ejemplo, NO tenga controller.

VISTAS CODIGO RAZOR

Este tipo de vistas son muy útiles para dibujar fragmentos que podríamos utilizar en Múltiples páginas.

Como su nombre indica, **no tienen controller**, solamente código Razor para su Funcionalidad.

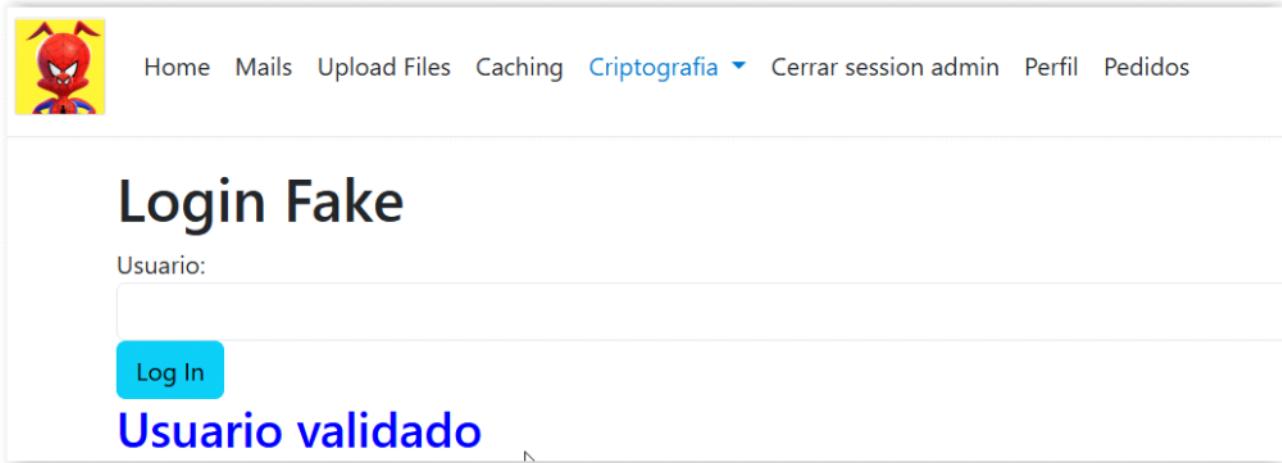
Se utilizan para dibujar algún código "complicado" y repetitivo en algunas vistas.

Por ejemplo, pongamos que tenemos en múltiples Layout la comprobación de si Un User está validado o no.
Tendremos que hacer un código para dibujar un Link de LogIn o dibujar un círculo y Un saludo con su imagen y nombre...

CODIGO RAZOR

```
@if (Session.Get("USER") == null)
{
    <a asp-action="LogIn">Log In User</a>
}
else{
    DIBUJO DEL USER DADO DE ALTA
}
```

Las vistas parciales se representan mediante el Guión bajo.
Para utilizar este código se utiliza un componente llamado **<partial>**



The screenshot shows a web application interface. At the top, there's a navigation bar with links: Home, Mails, Upload Files, Caching, Criptografia (with a dropdown arrow), Cerrar session admin, Perfil, and Pedidos. To the left of the navigation is a yellow square icon featuring a cartoon character. Below the navigation, the main content area has a title 'Login Fake'. Underneath the title is a form field labeled 'Usuario:' with a placeholder input field. Below the input field is a blue 'Log In' button. At the bottom of the form, the text 'Usuario valido' is displayed in blue, indicating a successful login attempt.

En cualquier vista que deseemos debemos realizar lo siguiente:

1) En la vista dónde deseemos dibujar (_Layout)

```
<partial name="_PartialView"/>
```

2) Las vistas parciales deben estar situadas dentro de la carpeta **Shared**

Vamos a realizar algo muy sencillo, una validación de broma para visualizar qué Tendríamos que hacer para utilizar este tipo de vistas.

Vamos a crear una nueva vista dentro de **Shared** llamada **_MenuUsuario.cshtml**

```
_MENUUSUARIO.CSHTML

@{
    string usuario = Context.Session.GetString("USUARIO");
}

@if (usuario == null){
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Home"
            asp-action="LogIn">
            Log In
        </a>
    </li>
} else{
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Home"
            asp-action="LogOut">
            Cerrar session @usuario
        </a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Home"
```

```

        asp-action="Perfil">
            Perfil
        </a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Home"
            asp-action="Pedidos">
            Pedidos
        </a>
    </li>
}

```

Sobre **HomeController** creamos un par de métodos para entrar y salir el usuario.

HOMECONTROLLER

```

public IActionResult LogIn()
{
    return View();
}
[HttpPost]
public IActionResult LogIn(string usuario)
{
    HttpContext.Session.SetString("USUARIO", usuario);
    ViewData["MENSAJE"] = "Usuario validado";
    return View();
}

public IActionResult LogOut()
{
    HttpContext.Session.Remove("USUARIO");
    return RedirectToAction("Index");
}

```

LOGIN.CSHTML

```

<h1>Login Fake</h1>

<form method="post">
    <label>Usuario: </label>
    <input type="text" name="usuario"
        class="form-control"/>
    <button class="btn btn-info">
        Log In
    </button>
</form>

<h2 style="color:blue">@ViewData["MENSAJE"]</h2>

```

Por último, sobre la vista que deseemos: **_Layout.cshtml** incluimos nuestro código

De Partial View

```

        <li><a class="dropdown-item" href="#">So
            </ul>
        </li>
        <partial name="_MenuUsuario"/>
    </ul>
</div>
...

```

VISTAS ASÍNCRONAS

Se utilizan para llamar a código de un **IActionResult** de forma asíncrona y mediante La utilización de **AJAX**.

Es como hacer un Front dentro de un Back.

Para utilizar esta funcionalidad debemos hacerlo mediante dos características conocidas:

- **JQUERY**
- **RENDERSECTION**: Secciones dentro del Layout para sustituir códigos.

Las vistas parciales siguen con la denominación del guión bajo.

Podemos integrar las vistas en dos lugares:

- 1) Dentro de la zona de Views y en un Controller.
- 2) Dentro de Shared

Como estamos hablando de código Cliente integrado en el Server, si tenemos algún Error no nos sirven los puntos de interrupción, debemos hacerlo con **F12**.

Truco: Probar primero que funciona correcto todo el back y luego llevarlo a esta Tecnología.

Necesitamos alguna ZONA dónde dibujar el Ajax.

```
<div id="container"></div>
```

Para poder dibujar las vistas parciales llamando a un **CONTROLLER/IActionResult** Se realiza mediante el siguiente código:

```
$("#container").load("/Controller/IActionResult")
```

Vamos a cargar una serie de coches en nuestra página. Crearemos un Model de coches y Mediante la tecnología Ajax, cargaremos los coches de forma asíncrona

Sobre **Models** creamos una nueva clase llamada **Coche**

COCHE

```
public class Coche
{
    0 references | 0 changes | 0 authors, 0 changes
    public int IdCoche { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Marca { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Modelo { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Imagen { get; set; }
}
```

Sobre **Controllers** creamos un nuevo controlador llamado **CochesController**

CODIGO EJEMPLO

```
this.Cars = new List<Coche>
{
    new Coche { IdCoche = 1, Marca = "Pontiac"
        , Modelo = "Firebird", Imagen = "https://mudfeed.com/wp-
content/uploads/2021/08/KITT-1200x640.jpg",
        new Coche { IdCoche = 2, Marca = "Volkswagen"
            , Modelo = "Escarabajo", Imagen =
"https://www.quadis.es/documents/80345/95274/herbie-el-volkswagen-beetle-
mas.jpg",
            new Coche { IdCoche = 3, Marca = "Ferrari"
                , Modelo = "Testarrosa", Imagen =
"https://www.lavanguardia.com/files/article_main_microformat/uploads/2017/01/03/5_
f15f8b7c1229.png",
                new Coche { IdCoche = 4, Marca = "Ford"
                    , Modelo = "Mustang GT", Imagen =
"https://cdn.autobild.es/sites/navi.axelspringer.es/public/styles/1200/public/med_
ia/image/2018/03/prueba-wolf-racing-mustang-gt.jpg"
                };
}
```

COCHESCONTROLLER

```
public class CochesController : Controller
{
    private List<Coche> Cars;

    public CochesController()
    {
        this.Cars = new List<Coche>
        {
            new Coche { IdCoche = 1, Marca = "Pontiac"
                , Modelo = "Firebird", Imagen = "https://mudfeed.com/wp-
content/uploads/2021/08/KITT-1200x640.jpg",
                new Coche { IdCoche = 2, Marca = "Volkswagen"
                    , Modelo = "Escarabajo", Imagen = "https://www.quadis.es/documents/80_
345/95274/herbie-el-volkswagen-beetle-mas.jpg",
                    new Coche { IdCoche = 3, Marca = "Ferrari"
                        , Modelo = "Testarrosa", Imagen =
"https://www.lavanguardia.com/files/article_main_microformat/uploads/2017/01/03/5f15f8b7c1229.png",
                        new Coche { IdCoche = 4, Marca = "Ford"
                            , Modelo = "Mustang GT", Imagen =
"https://cdn.autobild.es/sites/navi.axelspringer.es/public/styles/1200/public/media/
image/2018/03/prueba-wolf-racing-mustang-gt.jpg"
                        };
}
}

//ESTA ES LA VISTA QUE VAMOS A VISUALIZAR COMO PRINCIPAL
public IActionResult Index()
{
    return View();
}

//TENDREMOS UN IACTIONRESULT PARA INTEGRAR DENTRO DE OTRA
//VISTA, EN NUESTRO EJEMPLO, DENTRO DE INDEX
public IActionResult _CochesPartial()
```

```

    {
        //SI SON VISTAS PARCIALES CON AJAX, DEBEMOS
        //DEVOLVER PartialView
        //EN EL MOMENTO DE DEVOLVER, INDICAMOS A QUE VISUALIZACION
        //TIENE QUE HACERLO
        return PartialView("_CochesPartial", this.Cars);
    }
}

```

El siguiente paso es dibujar la vista `_CochesPartial.cshtml`
La vista puede estar en **Shared** o en el **Controller**

Sobre **Views/Coches**, creamos una nueva vista llamada `_CochesPartial.cshtml`

_COCHESPARTIAL.CSHTML

```

@model List<Coche>
 @{
     ViewData["Title"] = "_CochesPartial";
 }

<h1>Coches con AJAX</h1>

<table class="table table-bordered">
 <thead>
 <tr>
 <th>IDCOCHE</th>
 <th>TIPO</th>
 <th>IMAGEN</th>
 </tr>
 </thead>
 <tbody>
 @foreach (var item in Model) {
 <tr>
 <td>@item.IdCoche<td>
 </td>
 <td>
 @item.Marca @item.Modelo
 </td>
 <td>
 
 </td>
 </tr>
 }
 </tbody>
</table>

```

El siguiente paso que debemos realizar es agregar un **RenderSection** dentro de **Layout** para poder integrar código JQUERY en las vistas.

_LAYOUT.CSHTML

```

<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("Scripts", required: false)

```

Sobre la vista **Coches/Index** es dónde agregamos la funcionalidad para las Peticiones Ajax y cargar nuestra vista parcial asíncrona.

INDEX.CSHTML

```

@section Scripts {
<script>
$(document).ready(function() {
    $("#botonloadcoches").click(function() {
        $("#containercoches").load("/Coches/_CochesPartial");
    })
})
</script>
}

<h1>Index de AJAX COCHES</h1>
<button id="botonloadcoches" class="btn btn-warning">
    Load Coches Ajax
</button>
<div id="containercoches" style="background-color:lightgreen"></div>

```

Y podremos comprobar el resultado

Index de AJAX COCHES

Load Coches Ajax

Coches con AJAX

IDCOCHE	TIPO	IMAGEN
1	Pontiac Firebird	
2	Volkswagen Escarabajo	

El siguiente paso será utilizar un método con parámetros y cargarlo con Ajax.

Vamos a implementar **Details** para buscar un Coche por su ID.

Lo haremos mediante Ajax llamando a un método.

Esta vez, el método del Controller lo llamaré distinto a la vista.

Nota: si estamos utilizando Ajax no podemos enviar/recibir **Models**, es decir, solamente Primitivos.

```
public IActionResult _CochesPost(Coche car)
{
```

COCHESCONTROLLER

```
public IActionResult _CochesPost(int idcoche)
{
    Coche car =
        this.Cars.FirstOrDefault(x => x.IdCoche == idcoche);
    return PartialView("_DetailsCoche", car);
}
```

Sobre **Views/Coches** creamos una nueva vista llamada **_DetailsCoche.cshtml**

```
_DETAILSCOCHE.CSHTML

@model Coche
<h1 style="color:blue">Coche Details</h1>

@if (Model == null){
    <h2 style="color:red">
        No existe el Coche buscado
    </h2>
} else{
    <h2 style="color:fuchsia">
        @Model.Marca @Model.Modelo
    </h2>
    
}
```

Implementamos la funcionalidad dentro de **Index.cshtml**

INDEX.CSHTML

```
@section Scripts {
    <script>
        $(document).ready(function() {
            $("#botonloadcoches").click(function() {
                $("#containercoches").load("/Coches/_CochesPartial");
            })
            $("#botonbuscarcoche").click(function() {
                //PARA ENVIAR PARAMETROS, SI NUESTRO PARAMETRO
                //SE LLAMA id, SIMPLEMENTE PONEMOS UNA BARRA Y
                //EL VALOR DEL PARAMETRO EN LA LLAMADA
                // controller/action/ID
                //SI EL PARAMETRO SE LLAMA DE FORMA DIFERENTE
                //DEBEMOS HACERLO CON LAS LLAMADAS TIPICAS DE HTML GET
                // controller/action?param1=valor
                //SI EXISTE MAS DE UN PARAMETRO EN LA PETICION
                // controller/action?param1=valor&param2=valor2
                var id = $("#cajaidcoche").val();
                console.log(id);
            })
        })
    </script>
}
```

```

        var request = "/Coches/_CochesPost?idcoche=" + id;
        $("#detallescoche").load(request);
    })
</script>
}

<h1>Index de AJAX COCHES</h1>
<label>Id coche a buscar</label>
<input type="text" id="cajaidcoche" class="form-control"/>
<button id="botonbuscarcoche" class="btn btn-danger">
    Buscar coche
</button>
<div id="detallescoche" style="background-color: lightgoldenrodyellow"></div>
<button id="botonloadcoches" class="btn btn-warning">
    Load Coches Ajax
</button>
<div id="containercoches" style="background-color: lightgreen"></div>

```

Index de AJAX COCHES

Id coche a buscar

Buscar coche

Coche Details

Ferrari Testarrosa

VIEWCOMPONENTS

Un View Component es una funcionalidad que nos permite utilizar código del servidor
Dentro del Layout

Como todos sabemos, un Layout no tiene Controller.
Y si necesito cargar un menú con datos de la base de datos en la vista principal??

Se utiliza para cuando necesitamos Models dentro del Layout y un Controller para Poder recuperar dichos Models.

Un View Component está formado por un **Action** y por un código **Razor** que será el que Administre el Model

Tenemos que seguir una serie de normas para poder utilizar un View component

- Las peticiones deben ser asíncronas aunque no hagamos algo asíncrono (**Task**)
- Las vistas deben estar, de forma obligatoria, dentro de **Shared** y además, seguir Un sistema de carpetas específico. Debemos situar las vistas dentro de una Carpeta llamada **Components**
- Las clases deben heredar de **ViewComponent**

Tenemos dos formas de declarar una clase **ViewComponent**

- 1) Con la decoración de **[ViewComponent]**
- 2) Con la herencia :**ViewComponent**

Nota: Las clases deben finalizar con la palabra **ViewComponent**

Ejemplo:

```

[ViewComponent(name="menucategorias")]
public class MiClaseViewComponent : ViewComponent
{
    public async Task InvokeAsync()
    {
        return View(MODEL);
    }
}

```

Posteriormente, para dibujar el Model dentro del Layout

```
@await Component.InvokeAsync("menucategorias")
```

Como en este proyecto no tenemos un Repo real, vamos a hacer un Repo como
Si llamásemos a una base de datos.
Tendremos un menú dinámico dentro del Layout con los coches.

Comenzamos creando una nueva carpeta llamada **Repositories** y una clase llamada **RepositoryCoches**

REPOSITORYCOCHES

```
public class RepositoryCoches
{
    private List<Coche> Cars;

    public RepositoryCoches()
    {
        this.Cars = new List<Coche>
        {
            new Coche { IdCoche = 1, Marca = "Pontiac"
                , Modelo = "Firebird", Imagen = "https://espirituracer.com/archivos/2018/03/Pontiac-Firebird-KITT.jpg"},

            new Coche { IdCoche = 2, Marca = "Volkswagen"
                , Modelo = "Escarabajo", Imagen = "https://www.quadis.es/documents/80345/95274/herbie-el-volkswagen-beetle-mas.jpg"},

            new Coche { IdCoche = 3, Marca = "Ferrari"
                , Modelo = "Testarrosa", Imagen = "https://www.lavanguardia.com/files/article_main_microformat/uploads/2017/01/03/5f15f8b7c1229.png"},

            new Coche { IdCoche = 4, Marca = "Ford"
                , Modelo = "Mustang GT", Imagen = "https://cdn.autobild.es/sites/navi.axelspringer.es/public/styles/1200/public/media/image/2018/03/prueba-wolf-racing-mustang-gt.jpg"},

            new Coche
            {
                IdCoche = 5,
                Marca = "DMG",
                Modelo = "Delorean",
                Imagen = "https://www.topgear.es/sites/topgear.es/public/delorean-dmc-12-lateral.jpg"
            }
        };
    }

    public List<Coche> GetCoches()
    {
        return this.Cars;
    }

    public Coche FindCoche(int idCoche)
    {
        return this.Cars.FirstOrDefault(x => x.IdCoche == idCoche);
    }
}
```

Las clases **ViewComponents** pueden estar ubicadas en cualquier lugar de nuestro proyecto
Las VISTAS **ViewComponents** deben estar a la fuerza dentro de **Shared**

Sobre el proyecto, creamos una nueva carpeta llamada **ViewComponents** y dentro
Una clase llamada **MenuCochesViewComponent**

MENUCOCHESVIEWCOMPONENT

```
public class MenuCochesViewComponent: ViewComponent
{
    private RepositoryCoches repo;

    public MenuCochesViewComponent(RepositoryCoches repo)
    {
        this.repo = repo;
    }

    //PODRÍAMOS TENER TODOS LOS METODOS QUE DESEEMOS.
    //ES OBLIGATORIO TENER EL METODO InvokeAsync con Task
    //Y SERÁ EL METODO QUE DEVOLVERÁ EL MODELO A LA VISTA
    public async Task<IViewComponentResult> InvokeAsync()
    {
        List<Coche> coches = this.repo.GetCoches();
        return View(coches);
    }
}
```

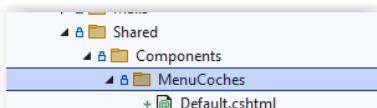
Necesitamos una vista para representar el dibujo que deseamos pintar en el
Layout

Normas:

- 1) Debemos crear una carpeta llamada **Components** dentro de **Shared**
- 2) Debemos crear una subcarpeta por cada **ViewComponent** que tengamos con
El nombre de la clase sin el final de **ViewComponent**
- 3) Dentro de la carpeta necesitamos una vista que se llamará **Default.cshtml**

La ruta de las vistas será:

Views/Shared/Components/Clase/Default.cshtml



Por último, realizamos el diseño que deseemos integrar dentro de nuestro Layout para dibujar los Coches.

Utilizamos un menú desplegable de Bootstrap

DEFAULT.CSHTML

```
@model List<Coche>



- Coches

@foreach (Coche car in Model){
    <li>
        <a class="dropdown-item" href="#">@car.Marca</a>
    </li>
}

```

Por último, realizamos el dibujo dentro del Layout

_LAYOUT.CSHTML

```
<li class="nav-item">
    <a class="nav-link text-dark"
       asp-controller="Coches"
       asp-action="Index">
        Coches Ajax
    </a>
</li>
@await Component.InvokeAsync("MenuCoches")
```

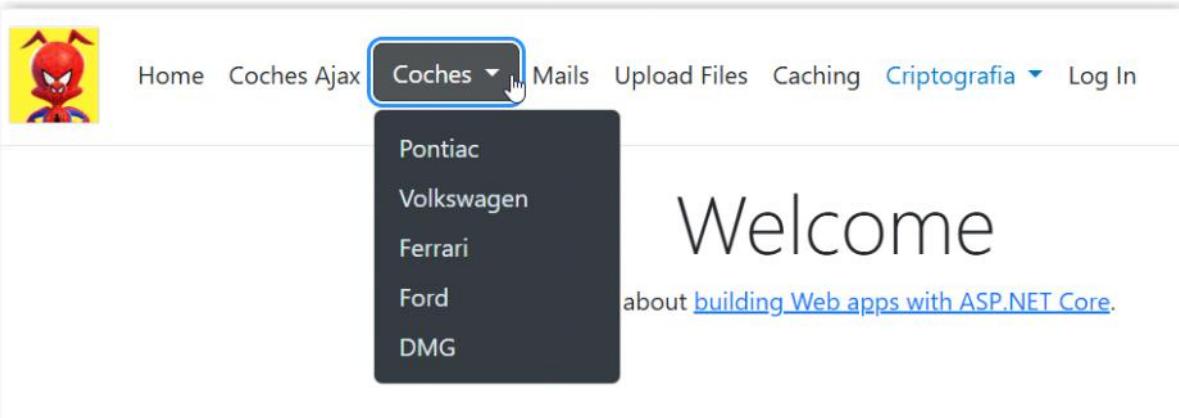
Por último, resolvemos la inyección del Repo dentro de Program

PROGRAM

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddTransient<RepositoryCoches>();
builder.Services.AddSingleton<HelperPathProvider>();
```

Y podremos comprobar la funcionalidad



A continuación vamos a implementar la funcionalidad de mostrar los detalles de un coche desde el menú del Layout.

Escribimos un nuevo método dentro de CochesController

COCHESCONTROLLER

```
0 references | 0 changes | 0 authors, 0 changes
public IActionResult Details(int idcoche)
{
    Coche car =
        this.Cars.FirstOrDefault(z => z.IdCoche == idcoche);
    return View(car);
}
```

Modificamos nuestra vista ViewComponents llamada **Default.cshtml**

DEFAULT.CSHTML

```
<ul class="dropdown-menu dropdown-menu-dark">
@foreach (Coche car in Model){
    <li>
        <a class="dropdown-item"
            asp-controller="Coches"
            asp-action="Details"
            asp-route-idcoche="@car.IdCoche">@car.Marca</a>
    </li>
}
</ul>
```

Sobre **Views/Coches** creamos una vista llamada **Details.cshtml**

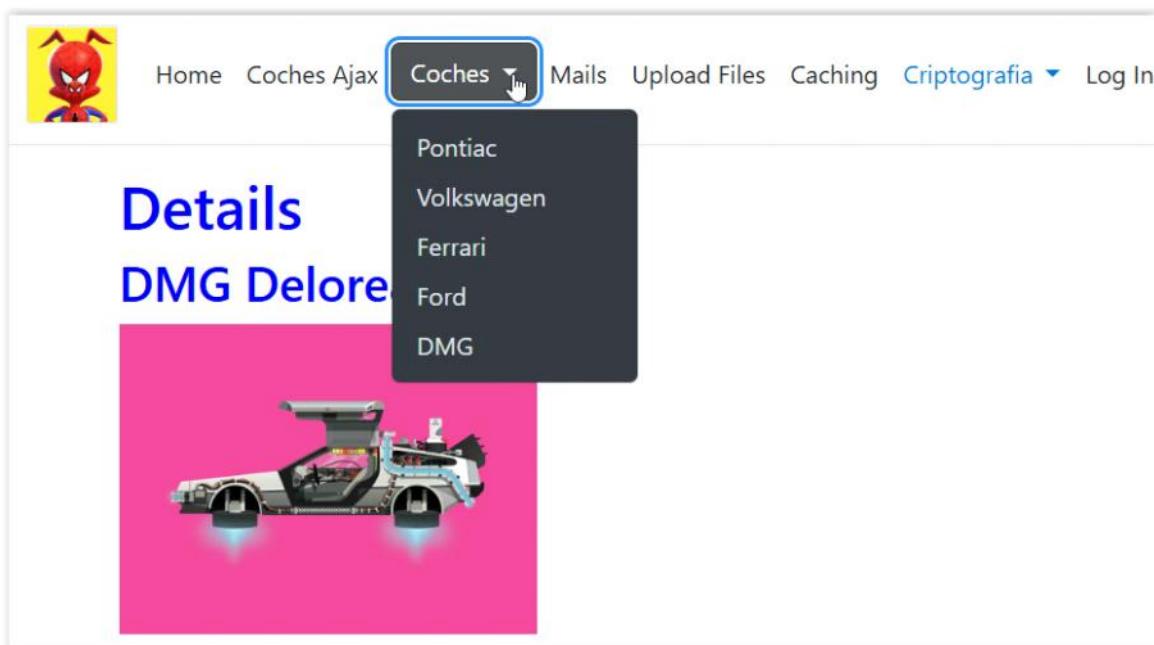
DETAILS.CSHTML

```
@model Coche
<h1 style="color:blue">Details</h1>

<h2 style="color:blue">
    @Model.Marca @Model.Modelo
</h2>

```

Y visualizaremos la funcionalidad



PAGINACION DE REGISTROS

miércoles, 5 de marzo de 2025 12:58

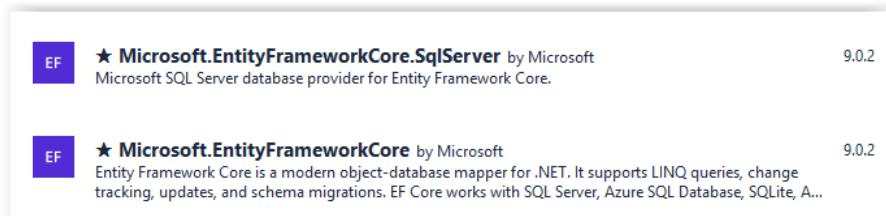
Cuando paginamos, tenemos varias formas de realizarlo.

- 1) Utilizando **EF** con sus métodos para la paginación
- 2) Utilizando la base de datos para la paginación.

La base de datos es más eficiente para paginar.

Creamos un nuevo proyecto llamado **MvcNetCorePaginacionRegistros**

Agregamos los NuGet para EF con SQL Server



Sobre **Models** trabajamos con **Departamento**

```
[Table("DEPT")]
0 references
public class Departamento
{
    [Key]
    [Column("DEPT_NO")]
    0 references
    public int IdDepartamento { get; set; }
    [Column("DNOMBRE")]
    0 references
    public string Nombre { get; set; }
    [Column("LOC")]
    0 references
    public string Localidad { get; set; }
}
```

EMPLEADO

```
public class Empleado
{
    [Key]
    [Column("EMP_NO")]
    0 references
    public int IdEmpleado { get; set; }
    [Column("APELLIDO")]
    0 references
    public string Apellido { get; set; }
    [Column("OFICIO")]
    0 references
    public string Oficio { get; set; }
    [Column("SALARIO")]
    0 references
    public int Salario { get; set; }
    [Column("DEPT_NO")]
    0 references
    public int IdDepartamento { get; set; }
}
```

Sobre la carpeta **Data** creamos una nueva clase llamada **HospitalContext**

HOSPITALCONTEXT

```
public class HospitalContext: DbContext
{
    0 references
    public HospitalContext(DbContextOptions<HospitalContext> options)
        :base(options) { }

    0 references
    public DbSet<Empleado> Empleados { get; set; }
    0 references
    public DbSet<Departamento> Departamentos { get; set; }
}
```

Creamos una carpeta llamada **Repositories** y una clase llamada **RepositoryHospital**

REPOSITORYHOSPITAL

```
public class RepositoryHospital
{
    private HospitalContext context;

    public RepositoryHospital(HospitalContext context)
    {
        this.context = context;
    }

    public async Task<List<Departamento>> GetDepartamentosAsync()
    {
        var departamentos = await
            this.context.Departamentos.ToListAsync();
        return departamentos;
    }

    public async Task<List<Empleado>> GetEmpleadosDepartamentoAsync
        (int idDepartamento)
    {
        var empleados = this.context.Empleados
            .Where(x => x.IdDepartamento == idDepartamento);
        if (empleados.Count() == 0)
        {
            return null;
        }
        else
        {
            return await empleados.ToListAsync();
        }
    }
}
```

Resolvemos las dependencias dentro de la App

APPSETTINGS.JSON

```
nastore.org/appsettings.json
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;In
    }
}
```

PROGRAM

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
string connectionString = builder.Configuration.GetConnectionString
    ("SqlHospital");
builder.Services.AddTransient<RepositoryHospital>();
builder.Services.AddDbContext<HospitalContext>
    (options => options.UseSqlServer(connectionString));

builder.Services.AddControllersWithViews();

```

Tenemos dos formas de realizar la paginación:

- 1) **Registro a registro:** Solamente dibujamos un registro a la vez, los típicos botones de Primero, siguiente, anterior y último
- 2) **Paginación en Grupo:** Paginamos un conjunto de N registros, mostramos los típicos Números de página

Nota: Si paginamos por base de datos es más eficiente, pero dependemos de la base de Datos.
Las paginaciones se realizan mediante Procedimientos almacenados, si migramos de Base de datos por alguna razón, tendremos que migrar dichos procedimientos y Adaptarnos a las funciones de bases de datos de paginación

Por ejemplo, en SQL Server se utiliza la función **Row_number()** y en MySql se utiliza **limit**

Vamos a comenzar paginando los registros de uno en uno. Departamentos

The screenshot shows a web page titled "Página de Registro Individual". The main content displays a single department record with the following details:
 - Número: 10
 - Nombre: CONTABILIDAD
 - Localidad: ELCHE
 Below the record, there are four navigation buttons: "Primero", "Anterior", "Siguiente", and "Último".

Vamos a crear una vista en SQL Server que nos devolverá los registros paginados.

```

select * FROM
(select ROW_NUMBER() over (order by DEPT_NO) AS POSICION
, DEPT_NO, DNOMBRE, LOC from DEPT) QUERY
where QUERY.POSICION=1

```

VISTA SQL SERVER

```

create view V_DEPARTAMENTOS_INDIVIDUAL
as
    select ROW_NUMBER() over (order by DEPT_NO) AS POSICION
    , DEPT_NO, DNOMBRE, LOC from DEPT
go
select * from V_DEPARTAMENTOS_INDIVIDUAL where POSICION=1

```

Sobre **Models** creamos un nuevo model llamada **VistaDepartamento**

VISTADEPARTAMENTO

```

[Table("V_DEPARTAMENTOS_INDIVIDUAL")]
0 references | 0 changes | 0 authors, 0 changes
public class VistaDepartamento
{
    [Key]
    [Column("DEPT_NO")]
    0 references | 0 changes | 0 authors, 0 changes
    public int IdDepartamento { get; set; }
    [Column("DNOMBRE")]
    0 references | 0 changes | 0 authors, 0 changes
    public string Nombre { get; set; }
    [Column("LOC")]
    0 references | 0 changes | 0 authors, 0 changes
    public string Localidad { get; set; }
    [Column("POSICION")]
    0 references | 0 changes | 0 authors, 0 changes
    public int Posicion { get; set; }
}

```

Agregamos la vista a **HospitalContext**

HOSPITALCONTEXT

```

public class HospitalContext: DbContext
{
    0 references | serraguti, 20 hours ago | 1 author, 1 change
    public HospitalContext(DbContextOptions<HospitalContext> options)
        :base(options) { }

    0 references | 0 changes | 0 authors, 0 changes
    public DbSet<VistaDepartamento> VistaDepartamentos { get; set; }
    1 reference | serraguti, 20 hours ago | 1 author, 1 change
    public DbSet<VistaDepartamento> VistaDepartamentos { get; set; }
}

```

El siguiente paso es implementar la funcionalidad dentro de nuestro repo.

¿Qué necesitamos para poder paginar?

Necesitamos la **POSICION** y el **NUMERO DE REGISTROS**

REPOSITORYHOSPITAL

```

public async Task<int> GetNumeroRegistrosVistaDepartamentosAsync()
{
    return await this.context.VistaDepartamentos.CountAsync();
}

public async Task<VistaDepartamento>
    GetVistaDepartamentoAsync(int posicion)
{
    VistaDepartamento departamento = await
        this.context.VistaDepartamentos
        .Where(z => z.Posicion == posicion).FirstOrDefaultAsync();
    return departamento;
}

```

Creamos un nuevo Controller llamado PaginacionController

PAGINACIONCONTROLLER

```
public class PaginacionController : Controller
{
    private RepositoryHospital repo;

    public PaginacionController(RepositoryHospital repo)
    {
        this.repo = repo;
    }

    public async Task<IActionResult>
        PaginarRegistroVistaDepartamento(int? posicion)
    {
        if (posicion == null)
        {
            posicion = 1;
        }
        int numRegistros = await
            this.repo.GetNumeroRegistrosVistaDepartamentosAsync();
        //PRIMERO = 1
        //ULTIMO = 4
        //SIGUIENTE = 5
        //ANTERIOR = 4
        int siguiente = posicion.Value + 1;
        if (siguiente > numRegistros)
        {
            siguiente = numRegistros;
        }
        int anterior = posicion.Value - 1;
        if (anterior < 1)
        {
            anterior = 1;
        }
        ViewData["ULTIMO"] = numRegistros;
        ViewData["SIGUIENTE"] = siguiente;
        ViewData["ANTERIOR"] = anterior;
        VistaDepartamento departamento =
            await this.repo.GetVistaDepartamentoAsync(posicion.Value);
        return View(departamento);
    }
}
```

PAGINARREGISTROVISTADEPARTAMENTO.CSHTML

```
@model VistaDepartamento
<h1>Paginación Registro Vista</h1>
<h2 style="color:red">
    Registro @Model.Posicion de @ViewData["ULTIMO"]
</h2>
<div>
    <ul class="list-group list-group-horizontal">
        <li class="list-group-item">
            <a asp-controller="Paginacion"
                asp-action="PaginarRegistroVistaDepartamento"
                asp-route-posicion="1">
                Primero
            </a>
        </li>
        <li class="list-group-item">
            <a asp-controller="Paginacion"
                asp-action="PaginarRegistroVistaDepartamento"
                asp-route-posicion="@ViewData["ANTERIOR"]">
                Anterior
            </a>
        </li>
        <li class="list-group-item">
            <a asp-controller="Paginacion"
                asp-action="PaginarRegistroVistaDepartamento"
                asp-route-posicion="@ViewData["SIGUIENTE"]">
                Siguiente
            </a>
        </li>
        <li class="list-group-item">
            <a asp-controller="Paginacion"
                asp-action="PaginarRegistroVistaDepartamento"
                asp-route-posicion="@ViewData["ULTIMO"]">
                Ultimo
            </a>
        </li>
    </ul>
</div>
<hr/>
<ul class="list-group">
    <li class="list-group-item list-group-item-info">
        Número: @Model.IdDepartamento
    </li>
    <li class="list-group-item list-group-item-info">
        Nombre: @Model.Nombre
    </li>
    <li class="list-group-item list-group-item-info">
        Localidad: @Model.Localidad
    </li>
</ul>
```

El siguiente paso es paginar registros en GRUPO.



Paginar Grupo VistaDepartamento

[Página 1](#) | [Página 2](#) |

POSICION	ID	NOMBRE	LOCALIDAD
1	10	CONTABILIDAD	ELCHE
2	20	INVESTIGACION	MADRID

Vamos a continuar mediante Departamentos.

Aprovechamos la vista que ya tenemos para realizar la paginación.

```
|select * from V_DEPARTAMENTOS_INDIVIDUAL  
where POSICION >=1 and posicion < (1 + 2)
```

Creamos un nuevo método dentro del Repository

REPOSITORYHOSPITAL

```
public async Task<List<VistaDepartamento>>  
GetGrupoVistaDepartamentoAsync(int? posicion)  
{  
    //select* from V_DEPARTAMENTOS_INDIVIDUAL  
    //where POSICION >= 1 and posicion< (1 + 2)  
    var consulta = from datos in this.context.VistaDepartamentos  
        where datos.Posicion >= posicion  
        && datos.Posicion < (posicion + 2)  
        select datos;  
    return await consulta.ToListAsync();  
}
```

Creamos un nuevo método dentro de PaginacionController

PAGINACIONCONTROLLER

```
public async Task<IActionResult>  
PaginarGrupoVistaDepartamento(int? posicion)  
{  
    if (posicion == null)  
    {  
        posicion = 1;  
    }  
    //AHORA DEBEMOS DIBUJAR NUMEROS DE PAGINA  
    //QUE DEPENDEN DEL NUMERO DE REGISTROS  
    //<a href='paginacion?posicion=1'>Página 1</a>  
    //<a href='paginacion?posicion=3'>Página 2</a>  
    //<a href='paginacion?posicion=5'>Página 3</a>  
    int numeroPagina = 1;  
    int numRegistros =  
        await this.repo.GetNumeroRegistrosVistaDepartamentosAsync();  
    //VAMOS A REALIZAR UN DIBUJO DINAMICO CON  
    //HTML Y SE LO ENVIAMOS A LA VISTA  
    string html = "<div>";  
    for (int i = 1; i <= numRegistros; i += 2)  
    {  
        html += "<a href='PaginarGrupoVistaDepartamento?posicion=" +  
            i + "'>Página " + numeroPagina + "</a> | ";  
        numeroPagina += 1;  
    }  
    html += "</div>";  
    ViewData["LINKS"] = html;  
    List<VistaDepartamento> departamentos =  
        await this.repo.GetGrupoVistaDepartamentoAsync(posicion.Value);  
    return View(departamentos);  
}
```

PAGINARGRUPOVISTADEPARTAMENTO.CSHTML

```
@model List<VistaDepartamento>  
@{  
    ViewData["Title"] = "PaginarGrupoVistaDepartamento";  
}  
<h1>Paginar Grupo VistaDepartamento</h1>
```

```

<hr/>
@Html.Raw(@ViewData["LINKS"])
<hr/>

<table class="table table-warning">
  <thead>
    <tr>
      <th>POSICION</th>
      <th>ID</th>
      <th>NOMBRE</th>
      <th>LOCALIDAD</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model) {
      <tr>
        <td>@item.Posicion</td>
        <td>@item.IdDepartamento</td>
        <td>@item.Nombre</td>
        <td>@item.Localidad</td>
      </tr>
    }
  </tbody>
</table>

```

Si deseo modificar el diseño de los Links, no me gustan nada, quiero ponerlos "bonitos",
Qué es lo que haríamos normalmente???

Lo suyo es hacerlo en la vista

¿Qué necesitamos en la vista para generar el dibujo?

- Número de registros
- Número de página (contador declarado en la vista)

Vamos a modificar el código de nuestra vista y tendremos que hacerlo con código Razor.

PAGINACIONCONTROLLER

```

public async Task<IActionResult>
  PaginarGrupoVistaDepartamento(int? posicion)
{
  if (posicion == null)
  {
    posicion = 1;
  }
  int numRegistros =
    await this.repo.GetNumeroRegistrosVistaDepartamentosAsync();
  ViewData["REGISTROS"] = numRegistros;
  List<VistaDepartamento> departamentos =
    await this.repo.GetGrupoVistaDepartamentoAsync(posicion.Value);
  return View(departamentos);
}

```

PAGINARGRUPOVISTADEPARTAMENTO.CSHMTL

```

@model List<VistaDepartamento>

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

@{
  int numPagina = 1;
  int numRegistros = (int)ViewData["REGISTROS"];
}

<h1>Paginar Grupo VistaDepartamento</h1>

<ul class="list-group list-group-horizontal">
  @for (int i = 1; i <= numRegistros; i+=2){
    <li class="list-group-item list-group-item-success">
      <a asp-controller="Paginacion"
         asp-action="PaginarGrupoVistaDepartamento"
         asp-route-posicion="@i">
        Página @numPagina
      </a>
    </li>
    numPagina += 1;
  }
</ul>

<table class="table table-warning">
  <thead>
    <tr>
      <th>POSICION</th>
      <th>ID</th>
      <th>NOMBRE</th>
      <th>LOCALIDAD</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model) {
      <tr>
        <td>@item.Posicion</td>
        <td>@item.IdDepartamento</td>
        <td>@item.Nombre</td>
        <td>@item.Localidad</td>
      </tr>
    }
  </tbody>
</table>

```

```
</tbody>
</table>
```



Home Privacy Ejemplos paginación ▾

Paginar Grupo VistaDepartamento

POSICION	ID	NOMBRE	LOCALIDAD
3	30	VENTAS	BARCELONA
4	40	PRODUCCION	SALAMANCA

Ya tenemos todos los conceptos.

Ahora es el momento de pulir detalles.

Actualmente, si ahora me pongo a leer el marca y os digo de hacer lo mismo con Empleados.

¿Qué pasos tenéis que hacer?

- 1) Crear la vista de SQL Server
- 2) Crear una clase Model de VistaEmpleado
- 3) Crear una clase Model de Plantilla???
- 4) Fuegos artificiales

No tendríamos que estar creando un Model determinado para la paginación.

Nosotros creamos la vista en Base de datos y para recuperar los Models que ya tengamos

Lo vamos a realizar mediante un Procedimiento almacenado.

El Procedimiento nos devolverá el Model que ya tengamos, es decir, Empleado o Departamento
El procedimiento recibirá la Posición.

Vamos a realizar esto sobre **Grupo de departamentos** y mapearemos nuestro Model de Departamento

STORED PROCEDURE

```
create procedure SP_GRUPO_DEPARTAMENTOS
(@posicion int)
as
select DEPT_NO, DNOMBRE, LOC from V_DEPARTAMENTOS_INDIVIDUAL
where POSICION >= @posicion AND POSICION < (@posicion + 2)
go
exec SP_GRUPO_DEPARTAMENTOS 1
```

Ya tenemos el Model

```
[Table("DEPT")]
2 references | serraguti, 21 hours ago | 1 author, 1 change
public class Departamento
{
    [Key]
    [Column("DEPT_NO")]
    0 references | serraguti, 21 hours ago | 1 author, 1 change
    public int IdDepartamento { get; set; }
    [Column("DNOMBRE")]
    0 references | serraguti, 21 hours ago | 1 author, 1 change
    public string Nombre { get; set; }
    [Column("LOC")]
    0 references | serraguti, 21 hours ago | 1 author, 1 change
    public string Localidad { get; set; }
}
```

```

public async Task<List<Departamento>>
    GetGrupoDepartamentosAsync(int posicion)
{
    string sql = "SP_GRUPO_DEPARTAMENTOS @posicion";
    SqlParameter pamPosicion =
        new SqlParameter("@posicion", posicion);
    var consulta =
        this.context.Departamentos.FromSqlRaw(sql, pamPosicion);
    return await consulta.ToListAsync();
}

```

Sobre el Controller de paginación realizamos un método igual al que hemos hecho de Grupo.

PAGINACIONCONTROLLER

```

public async Task<IActionResult>
    PaginarGrupoDepartamentos(int? posicion)
{
    if (posicion == null)
    {
        posicion = 1;
    }
    int numRegistros =
        await this.repo.GetNumeroRegistrosVistaDepartamentosAsync();
    ViewData["REGISTROS"] = numRegistros;
    List<Departamento> departamentos =
        await this.repo.GetGrupoDepartamentosAsync(posicion.Value);
    return View(departamentos);
}

```

Por último, creamos una nueva vista llamada **PaginarGrupoDepartamentos.cshtml**

PAGINARGRUPODEPARTAMENTOS.CSHTML

```

@model List<Departamento>

@{
    ViewData["Title"] = "Paginar Grupo Departamento";
}

@{
    int numPagina = 1;
    int numRegistros = (int)ViewData["REGISTROS"];
}

<h1 style="color:blue">Paginar Grupo Departamento</h1>

<ul class="list-group list-group-horizontal">
    @for (int i = 1; i <= numRegistros; i += 2)
    {
        <li class="list-group-item list-group-item-danger">
            <a asp-controller="Paginacion"
                asp-action="PaginarGrupoDepartamento"
                asp-route-posicion="@i">
                Página @numPagina
            </a>
        </li>
        numPagina += 1;
    }
</ul>

<table class="table table-bordered table-warning">
    <thead>
        <tr>
            <th>ID</th>
            <th>NOMBRE</th>
            <th>LOCALIDAD</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.IdDepartamento</td>
                <td>@item.Nombre</td>
                <td>@item.Localidad</td>
            </tr>
        }
    </tbody>
</table>

```

Para la siguiente versión vamos a paginar Empleados
Paginaremos de 3 en 3. Ya tenemos el Model



Home Privacy Ejemplos paginación ▾

Paginar Grupo Empleados

1 2 3 4 5 6 7

ID	APELLIDO	OFICIO	SALARIO	DEPARTAMENTO
7919	SERRA	DIRECTOR	390000	20
7777	TORMO	VENDEDOR	165900	30
7844	TOVAR	VENDEDOR	195000	30

Lo vamos a realizar con una vista y un procedimiento

SQL SERVER

```
create view V_GRUPO_EMPLEADOS
as
    select CAST(ROW_NUMBER() over (order by APELLIDO) AS INT) as POSICION
    , EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM EMP
go
create procedure SP_GRUPO_EMPLEADOS
(@posicion int)
as
    select EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM V_GRUPO_EMPLEADOS
    where POSICION >= @posicion AND POSICION < (@posicion + 3)
go
```

REPOSITORYHOSPITAL

```
public async Task<int> GetEmpleadosCountAsync()
{
    return await this.context.Empleados.CountAsync();
}

public async Task<List<Empleado>> GetGrupoEmpleadosAsync
(int posicion)
{
    string sql = "SP_GRUPO_EMPLEADOS @posicion";
    SqlParameter pamPosicion = new SqlParameter("@posicion", posicion);
    var consulta =
        this.context.Empleados.FromSqlRaw(sql, pamPosicion);
    return await consulta.ToListAsync();
}
```

PAGINACIONCONTROLLER

```
@model List<Empleado>
@{
    ViewData["Title"] = "Paginar Grupo Empleados";
}

@{
    int numRegistros = (int)ViewData["REGISTROS"];
    int numPagina = 1;
}



# Paginar Grupo Empleados




@for (int i = 1; i <= numRegistros; i+= 3){
    <li class="list-group-item list-group-item-warning">
        <a asp-controller="Paginacion"
            asp-action="PaginarGrupoEmpleados"
            asp-route-posicion="@i">
            @numPagina
        </a>
    </li>
    numPagina += 1;
}



---



| ID | APELLIDO | OFICIO | SALARIO | DEPARTAMENTO |
|----|----------|--------|---------|--------------|
|----|----------|--------|---------|--------------|


```

```

        <td>@item.Salario</td>
        <td>@item.IdDepartamento</td>
    </tr>
}
</tbody>
</table>

```

Vamos a realizar la paginación, pero filtrando por un campo, por ejemplo, con el OFICIO

Pongamos que necesitamos crear un procedimiento almacenado que reciba el oficio

```

select POSICION
,EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM V_GRUPO_EMPLEADOS
where OFICIO = 'EMPLEADO'
and POSICION >= 1 AND POSICION < (1 + 3)

```

	POSICION	EMP_NO	APELLIDO	OFICIO	SALARIO	DEPT_NO
1	1	7618	ALCALA	EMPLEADO	119000	10
2	2	7876	ALONSO	EMPLEADO	143000	20
3	4	7589	CASALES	EMPLEADO	179000	10
4	11	7900	JIMENO	EMPLEADO	123500	30
5	13	7934	MUÑOZ	EMPLEADO	169000	10
6	17	7369	SANCHAS	EMPLEADO	104000	20

```

select POSICION
,EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM V_GRUPO_EMPLEADOS
where OFICIO = 'ANALISTA'
and POSICION >= 1 AND POSICION < (1 + 3)

```

	POSICION	EMP_NO	APELLIDO	OFICIO	SALARIO	DEPT_NO
1	6	7902	FERNANDEZ	ANALISTA	390000	20
2	8	7788	GIL	ANALISTA	390000	20
3	9	7614	GUTIERREZ	ANALISTA	219000	20
4	18	7988	SANTIUSTE	ANALISTA	225000	20

Esto sucede porque ROW_NUMBER genera un campo numérico para la consulta
En la que se ha realizado.

Si, posteriormente, aplicamos otro filtro, no genera un nuevo ROW_NUMBER, utiliza
El de todos los empleados, es decir, el de la vista

Conclusión: No utilizar vistas si necesitamos aplicar filtros.

Necesitamos hacer el ROW_NUMBER() sobre el filtro, la VISTA no lleva el ROW_NUMBER.

El ROW_NUMBER irá sobre la consulta que hagamos.

```

select
ROW_NUMBER() over (order by APELLIDO) as POSICION,
EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM EMP
where OFICIO = 'ANALISTA'

```

La consulta debemos hacerla sobre el procedimiento y en el origen, es decir, nuestra VISTA o
La TABLA

En el procedimiento necesitamos filtrar por la columna POSICION que es un campo calculado, así
que
Tenemos que volver a SELECT TO SELECT

```

select * from
(select
ROW_NUMBER() over (order by APELLIDO) as POSICION,
EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM EMP
where OFICIO = 'ANALISTA' ) query
where POSICION >= 1 AND POSICION < (1 + 3)

```



Paginar Empleados Oficio

Introduzca oficio:

EMPLEADO

Buscar empleados

1 2 3



ID	APELLIDO	OFICIO	SALARIO	DEPARTAMENTO
7589	CASALES	EMPLEADO	179000	10
7900	JIMENO	EMPLEADO	123500	30

STORED PROCEDURE

```
create procedure SP_GRUPO_EMPLEADOS_OFICIO
(@posicion int, @oficio nvarchar(50))
as
select EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO from
(select
    ROW_NUMBER() over (order by APELLIDO) as POSICION,
    EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM EMP
    where OFICIO = @oficio) query
    where POSICION >= @posicion AND POSICION < (@posicion + 2)
go
exec SP_GRUPO_EMPLEADOS_OFICIO 1, 'ANALISTA'
```

REPOSITORYHOSPITAL

```
public async Task<int>
GetEmpleadosOficioCountAsync(string oficio)
{
    return await this.context.Empleados
        .Where(x => x.Oficio == oficio).CountAsync();
}

public async Task<List<Empleado>>
GetEmpleadosOficioAsync(int posicion, string oficio)
{
    string sql = "SP_GRUPO_EMPLEADOS_OFICIO @posicion, @oficio";
    SqlParameter pamPosicion = new SqlParameter("@posicion", posicion);
    SqlParameter pamOficio = new SqlParameter("@oficio", oficio);
    var consulta =
        this.context.Empleados.FromSqlRaw(sql, pamPosicion, pamOficio);
    return await consulta.ToListAsync();
}
```

PAGINACIONCONTROLLER

```
public async Task<IActionResult>
EmpleadosOficio(int? posicion, string oficio)
{
    if (posicion == null)
    {
        posicion = 1;
        return View();
    }
    else
    {
        List<Empleado> empleados = await
            this.repo.GetEmpleadosOficioAsync(posicion.Value, oficio);
        int registros =
            await this.repo.GetEmpleadosOficioCountAsync(oficio);
        ViewData["REGISTROS"] = registros;
        ViewData["OFICIO"] = oficio;
        return View(empleados);
    }
}

[HttpPost]
public async Task<IActionResult> EmpleadosOficio(string oficio)
{
    List<Empleado> empleados = await
        this.repo.GetEmpleadosOficioAsync(1, oficio);
    int registros =
        await this.repo.GetEmpleadosOficioCountAsync(oficio);
    ViewData["REGISTROS"] = registros;
    ViewData["OFICIO"] = oficio;
```

```

        return View(empleados);
    }

EMPLEADOSOFICIO.CSHTML

@model List<Empleado>

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

 @{
     int numRegistros = 0;
     int numPagina = 1;
     string oficio = "";
     if (ViewData["REGISTROS"] != null){
         numRegistros = (int)ViewData["REGISTROS"];
         oficio = ViewData["OFICIO"].ToString();
     }
 }

<h1 style="color:blue">Paginar Empleados Oficio</h1>
<form method="post">
    <label>Introduzca oficio: </label>
    <input type="text" name="oficio" class="form-control"/>
    <button class="btn btn-info">
        Buscar empleados
    </button>
</form>

@if (Model != null){
    <ul class="list-group list-group-horizontal">
        @for (int i = 1; i <= numRegistros; i += 2)
        {
            <li class="list-group-item list-group-item-info">
                <a asp-controller="Paginacion"
                   asp-action="EmpleadosOficio"
                   asp-route-posicion="@i"
                   asp-route-oficio="@oficio">
                    @numPagina
                </a>
            </li>
            numPagina += 1;
        }
    </ul>
    <hr />
    <table class="table table-bordered table-light">
        <thead>
            <tr>
                <th>ID</th>
                <th>APELLIDO</th>
                <th>OFICIO</th>
                <th>SALARIO</th>
                <th>DEPARTAMENTO</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.IdEmpleado</td>
                    <td>@item.Apellido</td>
                    <td>@item.Oficio</td>
                    <td>@item.Salario</td>
                    <td>@item.IdDepartamento</td>
                </tr>
            }
        </tbody>
    </table>
}

```

Los procedimientos deberían ser autónomos, es decir, que ejecuten la consulta y nos Ofrezcan todo a la vez.
Actualmente, no estamos haciendo esa acción.
Por cada filtro, estamos teniendo que utilizar un método para recuperar los datos De registros del filtro.

Parámetros de salida en procedimientos de paginación.

SQL SERVER

```

create procedure SP_GRUPO_EMPLEADOS_OFICIO_OUT
(@posicion int, @oficio nvarchar(50)
, @registros int out)
as
--ALMACENAMOS EL NUMERO DE REGISTROS DEL FILTRO
select @registros = count(EMP_NO) from EMP
where OFICIO=@oficio
select EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO from
(select
ROW_NUMBER() over (order by APELLIDO) as POSICION,
EMP_NO, APELLIDO, OFICIO, SALARIO, DEPT_NO FROM EMP
where OFICIO = @oficio) query
where POSICION >= @posicion AND POSICION < (@posicion + 2)
go

```

Ahora mismo, desde nuestro Repository tenemos que devolver dos elementos:

- List<Empleado>
- int registros

Tenemos dos opciones para hacer esta acción, es decir, devolver dos elementos o más dentro de un método de una clase:

- 1) Hemeroteca: Variables de referencia. Una variable de referencia dentro de C# nos permite enviar una variable y recuperarla con esa referencia

```
public async Task<List<Empleado>>
GetEmpleadosOficioAsync(int posicion, string oficio, ref int registros)
{
    return null;
}
```

- 1) Crear un Model con la lista y los registros

Sobre **Models** creamos un nuevo modelo llamado **ModelEmpleadosOficio**

```
public class ModelEmpleadosOficio
{
    public List<Empleado> Empleados { get; set; }
    public int NumeroRegistros { get; set; }
}
```

REPOSITORYHOSPITAL

```
public async Task<ModelEmpleadosOficio>
GetEmpleadosOficioOutAsync(int posicion, string oficio)
{
    string sql = "SP_GRUPO_EMPLEADOS_OFICIO_OUT @posicion, @oficio, @registros
out";
    SqlParameter pamPosicion = new SqlParameter("@posicion", posicion);
    SqlParameter pamOficio = new SqlParameter("@oficio", oficio);
    SqlParameter pamRegistros = new SqlParameter("@registros", 0);
    pamRegistros.Direction = ParameterDirection.Output;
    var consulta =
        this.context.Empleados.FromSqlRaw
        (sql, pamPosicion, pamOficio, pamRegistros);
    //PRIMERO DEBEMOS EJECUTAR LA CONSULTA PARA PODER
    //RECUPERAR LOS PARAMETROS DE SALIDA
    List<Empleado> empleados = await consulta.ToListAsync();
    int registros = (int)pamRegistros.Value;
    return new ModelEmpleadosOficio
    {
        NumeroRegistros = registros,
        Empleados = empleados
    };
}
```

PAGINACIONCONTROLLER

```
public async Task<IActionResult>
EmpleadosOficioOut(int? posicion, string oficio)
{
    if (posicion == null)
    {
        posicion = 1;
        return View();
    }
    else
    {
        ModelEmpleadosOficio model = await
            this.repo.GetEmpleadosOficioOutAsync(posicion.Value, oficio);
        ViewData["REGISTROS"] = model.NumeroRegistros;
        ViewData["OFICIO"] = oficio;
        return View(model);
    }
}

[HttpPost]
public async Task<IActionResult> EmpleadosOficioOut(string oficio)
{
    ModelEmpleadosOficio model = await
        this.repo.GetEmpleadosOficioOutAsync(1, oficio);
    ViewData["REGISTROS"] = model.NumeroRegistros;
    ViewData["OFICIO"] = oficio;
    return View(model);
}
```

EMPLEADOSOFICIOOUT.CSHTML

```
@model ModelEmpleadosOficio

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

@{
    int numRegistros = 0;
    int numPagina = 1;
    string oficio = "";
    if (ViewData["REGISTROS"] != null)
    {
        numRegistros = (int)ViewData["REGISTROS"];
        oficio = ViewData["OFICIO"].ToString();
    }
}

<h1 style="color:blue">Paginar Empleados Oficio Out</h1>

<form method="post">
    <label>Introduzca oficio: </label>
    <input type="text" name="oficio" class="form-control" />
    <button class="btn btn-info">
        Buscar empleados
    </button>
</form>

@if (Model != null)
{
    <ul class="list-group list-group-horizontal">
        @for (int i = 1; i <= numRegistros; i += 2)
        {
            <li class="list-group-item list-group-item-info">
                <a asp-controller="Paginacion"
                    asp-action="EmpleadosOficioOut"
                    asp-route-posicion="@i"
                    asp-route-oficio="@oficio">
                    @numPagina
                </a>
            </li>
            numPagina += 1;
        }
    </ul>

    <hr />
    <table class="table table-bordered table-dark">
        <thead>
            <tr>
                <th>ID</th>
                <th>APELLIDO</th>
                <th>OFICIO</th>
                <th>SALARIO</th>
                <th>DEPARTAMENTO</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model.Empleados)
            {
                <tr>
                    <td>@item.IdEmpleado</td>
                    <td>@item.Apellido</td>
                    <td>@item.Oficio</td>
                    <td>@item.Salario</td>
                    <td>@item.IdDepartamento</td>
                </tr>
            }
        </tbody>
    </table>
}
}
```

Tendremos una página inicial dónde veremos enlaces con los Departamentos.

Al pulsar sobre un determinado departamento:

- Veremos los detalles de dicho departamento
- Veremos el número de empleados que tiene el departamento
- Podremos paginar los empleados del departamento de uno en uno con Siguiente y Anterior.

Empleados Registro Departamento

[Volver a departamentos](#)

Id Departamento: 10

Nombre: CONTABILIDAD

Localidad: ELCHE

Registro 1 de 4

[Primero](#) [Anterior](#) [Siguiente](#) [Ultimo](#)

Id Empleado: 7589

Apellido: CASALES

Oficio: EMPLEADO

Salario: 179000

Departamento: 10

SEGURIDAD APPLICACIONES MVC NET CORE

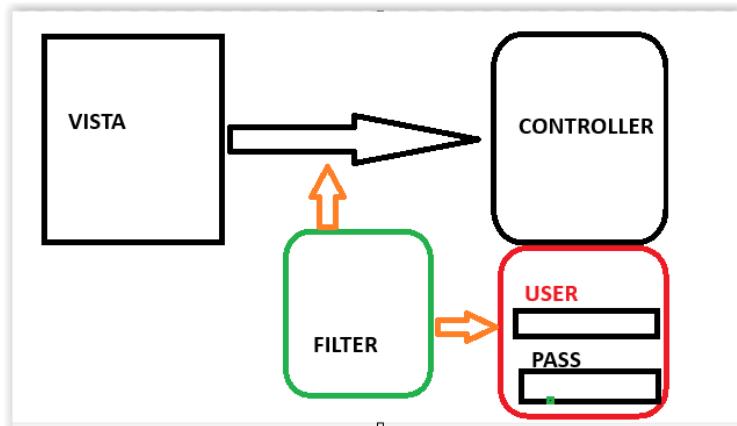
viernes, 7 de marzo de 2025 9:17

La seguridad de aplicaciones en Net Core funciona como el resto de módulos, es decir, Un módulo se acopla o desacopla y no interfiere con el resto de módulos.

El hecho de tener seguridad no implica utilizar otras características, simplemente, se acopla O no a nuestros Controllers

Nosotros NUNCA llamamos al LOGIN de forma explícita

La seguridad hace que el Login sea implementado automáticamente



Tenemos varios tipos de implementación de la seguridad:

- 1) **Seguridad Administrada:** Funciona de forma automática dentro de nuestra App.
No tenemos que hacer absolutamente NADA, solamente tener una base de datos SQL Server.
Es un cuello de botella porque mi App depende completamente de Microsoft.
Si algún día me va bien y me empiezan a cobrar una barbaridad, no puedo irme.
- 2) **Seguridad personalizada:** La lógica es igual a la seguridad administrada, pero nosotros Nos encargamos de todo, de los select, de los insert y todo el control de acceso.
Podemos utilizar cualquier base de datos y origen de datos

Vamos a comenzar con la seguridad Administrada.

Lo primero será crearnos una base de datos necesaria para este tipo de seguridad.
Dicha base de datos contendrá unas tablas que se generan internamente para la App.
Debemos, mediante herramientas de Net Core, crear la base de datos y algunas tablas.

Necesitamos el comando `aspnet_regsql.exe`

Asistente para la instalación de SQL Server de ASP.NET

Selección de una opción de configuración

¿Qué tarea de base de datos desea realizar?

Configurar SQL Server para los servicios de aplicaciones

Esta opción ejecuta un script que crea una nueva base de datos o configura una existente para almacenar información sobre la pertenencia, los perfiles, la administración de roles y la personalización de ASP.NET, y sobre el proveedor de eventos Web de SQL. Nota: este proceso no puede deshacerse.

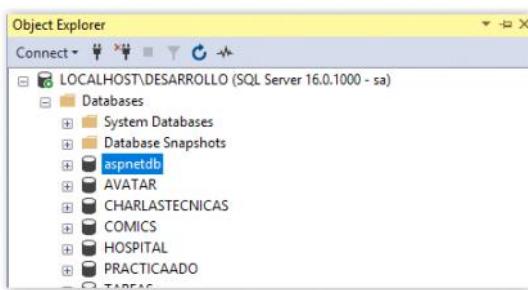
Eliminar la información sobre los servicios de aplicaciones de una base de datos existente

Esta opción elimina de la base de datos la información acerca de la pertenencia, los perfiles, la administración de roles y la personalización de ASP.NET, y acerca del proveedor de eventos Web de SQL. Nota: este proceso no puede deshacerse.

Nota:
Para configurar la base de datos para características adicionales como el estado de sesión o la dependencia de la caché de SQL, ejecute `aspnet_regsql` en la línea de comandos. Para obtener ayuda acerca de las opciones de la línea de comandos, utilice el modificador `-?`.

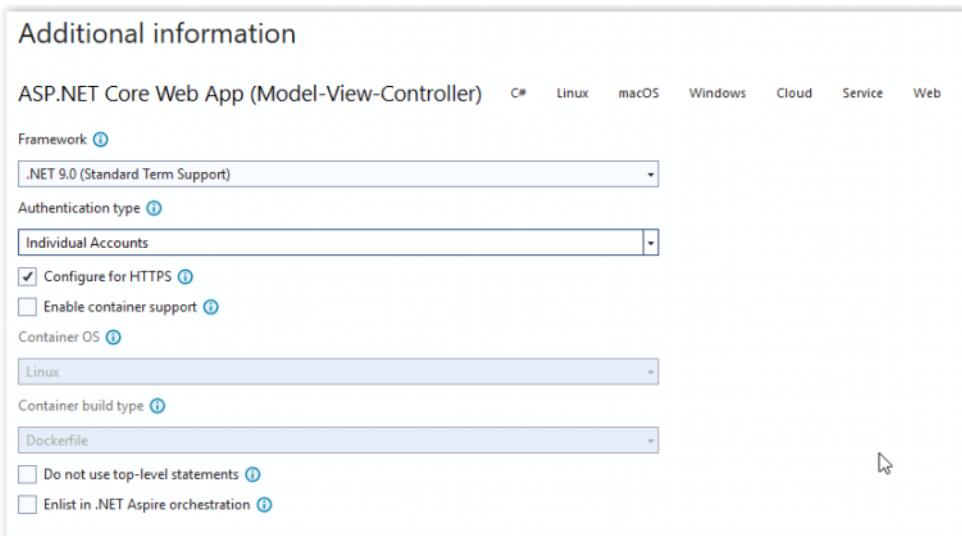


Nos ha creado una base de datos nueva llamada **aspnetdb**



Creamos un nuevo proyecto llamado **NetCoreSeguridadAdministrada**

Seleccionamos **Individual Accounts** en Authentication

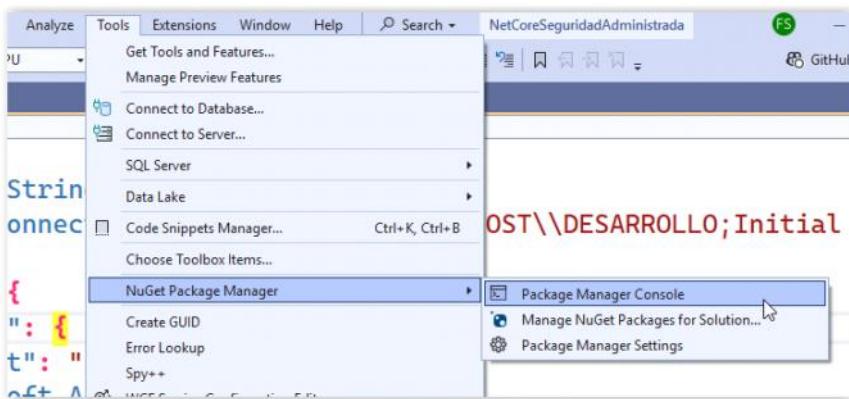


Nos ha creado un proyecto con seguridad montada apuntando a una base de datos fake.
Entramos dentro de **appsettings.json** y apuntamos a nuestra base de datos recién creada

La base de datos que hemos generado es una base de datos que está incompleta, se utiliza para algunas herramientas de Net, por ejemplo, Cache.

Debemos generar unas tablas en dicha base de datos para que funcione la seguridad Correctamente.

Necesitamos ejecutar el siguiente comando dentro de **Nuget Package Console**



Entramos en la ruta absoluta de nuestro proyecto

```
PM> cd .\NetCoreSeguridadAdministrada
PM> ls
```

Abrimos CMDER y escribimos el siguiente comando

```
C:\Users\Profesor MCSD Mañana
\ dotnet tool install --global dotnet-ef --version 9.*
```

Lo que hemos realizado es instalar una herramienta para utilizar una funcionalidad de Net Core Entity Framework llamada **First Code**

Lo que hace esta funcionalidad es primero crear el código y, después, mediante comandos Crear elementos en la base de datos.

Ejecutamos el siguiente comando dentro de Nuget Package Console

`dotnet ef database update`

```
PM> dotnet ef database update|
```

SEGURIDAD ADMINISTRADA DESDE CERO

Vamos a visualizar cómo podemos utilizar la seguridad administrada desde CERO, es decir, En un proyecto que ya tuviéramos creado.

No seleccionamos **Individual Accounts**

Creamos un nuevo proyecto llamado **NetCoreSeguridadAdministradaCero**

Aunque no vamos a utilizar un usuario de base de datos sí que es necesario indicar Que tendremos una identidad que será administrada por una base de datos, es decir, Un contexto de Identity

Category	Package Name	Version
EF	Microsoft.EntityFrameworkCore.SqlServer by Microsoft	9.0.2
EF	Microsoft.EntityFrameworkCore by Microsoft	9.0.2
.NET	Microsoft.EntityFrameworkCore.Tools by aspnet, dotnetframework, EntityFramework, Microsoft	373M downloads 9.0.2
.NET	Microsoft.AspNetCore.Identity.EntityFrameworkCore by aspnet, dotnetframework, Microsoft	185M download 9.0.2
.NET	Microsoft.AspNetCore.Identity.UI by aspnet, dotnetframework, Microsoft	92.2M downloads 9.0.2

Para que funcione de forma automática, necesitamos la cadena de conexión en **appsettings.json**
Apuntando a **ASPNETDB**
La cadena de conexión la llamaremos **DefaultConnection**

```

1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          }
7      },
8      "AllowedHosts": "*",
9      "ConnectionStrings": {
10         "DefaultConnection": "Data Source=LOCALHOST\\DESARROLLO;Initial Catalog=HOSPITAL;Integrated Security=True"
11     }
12 }

```

Por supuesto, en nuestro proyecto podríamos tener nuestros DbContext propios.

También no sería necesario tener una base de datos **aspnetdb**, simplemente podríamos haber utilizado el asistente para generar las tablas en nuestra base de datos **HOSPITAL**.

Necesitamos un Context que administre los usuarios de forma automática
Dicha clase context debe heredar de **IdentityDbContext**

Creamos una carpeta llamada **Data** y una clase llamada **ApplicationDbContext**

```

public class ApplicationDbContext : IdentityDbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        :base(options)
    {}
}

```

Nuestro nuevo Context de seguridad es igual al resto de Context

Debemos incluirlo en la inyección dentro de **Program**

También tenemos que dar de alta un usuario llamado **IdentityUser**

PROGRAM

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
string connectionString =
    builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>
    (options => options.UseSqlServer(connectionString));
//NECESITAMOS INCLUIR UN USUARIO IdentityUser ASOCIADO
//A NUESTRO CONTEXT
builder.Services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddControllersWithViews();

var app = builder.Build();

```

SUPER IMPORTANTE

No importa el tipo de seguridad, tenemos que indicar al servidor que utilizaremos seguridad
Dentro del Middleware.

El orden es básico

```
app.UseHttpsRedirection();
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapStaticAssets();

✓ app.MapControllerRoute(
    name: "default"
```

Como las redirecciones no las controlamos nosotros y genera dibujos que NO EXISTEN en La aplicación, debemos indicar que nuestro server podrá mapear páginas de Razor.

```
✓ app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
    .WithStaticAssets();
app.MapRazorPages();

app.Run();
```

Ya tenemos la seguridad montada.
Si ejecutamos, no veremos nada extraño, todo debería funcionar pero, no tenemos nada para Hacer Login ni nada de nada. Tampoco tenemos que hacerlo, las propias páginas se van a generar Automáticamente.

Acoplar/Desacoplar seguridad.
En cada **Controller** o en cada **IActionResult** que deseemos tener seguridad con un Log In
Tenemos que incluir la decoración **[Authorize]**

En **HomeController** incluimos un nuevo **IActionResult**

```
[Authorize]
0 references
public IActionResult ZonaProtegida()
{
    return View();
}
```

```
<h1 style="color:blue">
    Has entrado a la zona protegida con tu Login!!!!
</h1>
```

Incluimos un Link dentro de **_Layout.cshtml**

```
<li class="nav-item">
    <a class="nav-link text-danger"
        asp-controller="Home"
        asp-action="ZonaProtegida">
        Zona protegida
    </a>
</li>
```

Al ejecutar la aplicación, está buscando una vista de Login llamada **_LoginPartial**

Dicha vista no existe en nuestro proyecto, está generada por la seguridad administrada de **Identity UI**

An unhandled exception occurred while processing the request.

InvalidOperationException: The default Identity UI layout requires a partial view '_LoginPartial' usually located at '/Pages/_LoginPartial' or at '/Views/Shared/_LoginPartial' to work. Based on your configuration we have looked at it in the following locations:

/Areas/Identity/Pages/Account/_LoginPartial.cshtml
/Areas/Identity/Pages/_LoginPartial.cshtml

Tenemos que crear la vista de forma física para que **Identity UI** pueda realizar el dibujo

Sobre **Shared** creamos una nueva vista llamada **_LoginPartial.cshtml**

Y lo tenemos!!!

SEGURIDAD PERSONALIZADA

El concepto es el mismo que con seguridad administrada, es decir, nosotros NUNCA llamamos A nuestra vista **Login**, debe aparecer sola si se detecta que no estamos validados.

MvcNetCoreSeguridadPersonalizada Home Privacy Perfil usuario

Log In Usuarios

Username
admin

Password:
admin

Log In

Tendremos una serie de elementos nuevos, empezando por nuestro **Filter**

- **ActionFilter**: Es un filtro que intercepta peticiones entre Vistas y Controllers. Por ejemplo, se puede utilizar para cifrar los datos que vemos dentro del GET.
- **AuthorizeFilter**: Es un filtro creado para la seguridad dentro de Net Core, también Intercepta peticiones entre Views y Controllers y tiene implementado USERS para Saber si se han validado en mi página.

Necesitamos montar las siguientes características:

- Controllers con sus Actions
- Filter de autorizaciones para controlar las peticiones y validar si los usuarios están Activos en mis páginas.
- Controller para gestionar las peticiones de seguridad (Login, Logout, Register)

Creamos un nuevo proyecto llamado **MvcNetCoreSeguridadPersonalizada**

Sobre **Controllers** creamos un nuevo controlador llamado **ManagedController**

MANAGEDCONTROLLER

```
public class ManagedController : Controller
{
    0 references
    public IActionResult Login()
    {
        return View();
    }

    [HttpPost]
    0 references
    public IActionResult Login
        (string username, string password)
    {
        return View();
    }
}
```

Creamos una nueva vista llamada **Login.cshtml**

LOGIN.CSHTML

```
<h1 style="color:blue">
    Log In Usuarios
</h1>
<form method="post">
    <label>Username:</label>
    <input type="text" name="username"
        class="form-control"/>
    <label>Password:</label>
    <input type="password" name="password"
        class="form-control"/>
    <button class="btn btn-info">
        Log In
    </button>
</form>
<h2 style="color:red">
    @ViewData["MENSAJE"]
</h2>
```

No vamos a utilizar bases de datos ni nada por ahora.
Tendremos un Controller dónde protegeremos los datos del usuario.
Por ejemplo, una página de Perfil del usuario.

Sobre Controllers creamos un nuevo controlador llamado **UsuariosController**

USUARIOSCONTROLLER

```
public class UsuariosController : Controller
{
    public IActionResult Perfil()
    {
        return View();
    }
}
```

PERFIL.CSHTML

```
<h2 style="color:fuchsia">
    Bienvenido a tu perfil!!!
</h2>

```

El siguiente paso es crear nuestro Filter

El filtro de autorización debe seguir una serie de pautas:

- 1) Heredar de dos clases/interface
 - **IAuthorizationFilter**: Parar redireccionar el filtro con seguridad
 - **AuthorizeAttribute**: Dicha clase sirve para poder incluir el filtro como Decoraciones para los controllers o para los IActionResult
- 2) La clase del filtro debe terminar con la palabra **Attribute**

Por ejemplo, si mi clase se llama **MiFiltroAutorizacionAttribute**:

```
[MiFiltroAutorizacion]
public IActionResult SitioSeguro() {
}
```

Dentro de nuestro sistema, debemos crear a nuestro usuario que se ha validado.

El usuario no es algo que nosotros administremos, lo almacenamos y, posteriormente, Simplemente vamos preguntando por él.

Cuando tenemos un User administrado se almacena en una mezcla de Session y Cookies.

Sobre el proyecto, creamos una nueva carpeta llamada **Filters** y dentro una clase llamada **AuthorizeUsersAttribute**

AUTHORIZEUSERSATTRIBUTE

```
public class AuthorizeUsersAttribute : AuthorizeAttribute,
    IAuthorizationFilter
{
    //ESTE METODO ES EL QUE NOS PERMITIRA IMPEDIR EL ACCESO
    //A LOS CONTROLLERS/IACTIONRESULT QUE ESTEN DECORADOS CON
    //ESTA CLASE
    //EL FILTER ES EL ENCARGADO DE INTERCEPTAR LAS PETICIONES Y
    //DECIDIR QUE HACER.
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        //EL USUARIO QUE SE HA VALIDADO EN NUESTRA APP
        //ESTARA DENTRO DE Context Y EN UNA PROPIEDAD
        //LLAMADA User
        //UN USUARIO ESTA COMPUESTO POR DOS CARACTERISTICAS:
        //1) IDENTITY: EL NOMBRE DEL USUARIO Y SI ESTA ACTIVO
        //2) PRINCIPAL: SIRVE PARA PREGUNTAR SI EL USER ESTA EN ALGUN ROLE
    }
}
```

```

var user = context.HttpContext.User;
//PREGUNTAMOS SI EL USUARIO ESTA AUTENTIFICADO
if (user.Identity.IsAuthenticated == false)
{
    //SI EL USUARIO NO SE HA VALIDADO LO DEBEMOS LLEVAR A
    //NUESTRA PAGINA DE LOGIN
    //CREAMOS UNA RUTA PARA IR A LA DIRECCION DE LOGIN
    RouteValueDictionary rutaLogin =
        new RouteValueDictionary(
            new { controller = "Managed", action = "Login" });
    //DIRECCIONAMOS EL FILTRO HACIA LA RUTA DE LOGIN
    context.Result =
        new RedirectToRouteResult(rutaLogin);
}
}
}

```

El siguiente paso es validar a nuestro usuario.
Para ello, vamos a realizarlo dentro del Controller Managed y en el Post de **Login**

Una vez que el usuario ha escrito sus credenciales correctas debemos crear una clase
Llamada **ClaimsPrincipal**
Dicha clase Principal está compuesta por una Array de características llamadas **Claim**

Un **Claim** son propiedades que podemos asociar al usuario como, por ejemplo, Teléfono,
Email, nombre o demás.

Un Claim Principal tendrá por defecto dos Claims en su interior: **Name, Role**

MANAGEDCONTROLLER

```

public class ManagedController : Controller
{
    public IActionResult Login()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult> Login
        (string username, string password)
    {
        if (username.ToLower() == "admin"
            && password.ToLower() == "admin")
        {
            //AUNQUE NOSOTROS NO LO VEAMOS, SE GENERA UNA COOKIE
            //CIFRADA QUE ES PARA SABER SI EL USUARIO ESTA
            //VALIDADO EN SESION CON ESTE EQUIPO
            ClaimsIdentity identity =
                new ClaimsIdentity(
                    CookieAuthenticationDefaults.AuthenticationScheme,
                    ClaimTypes.Name, ClaimTypes.Role
                );
            //UN CLAIM ES INFORMACION DEL USUARIO
            Claim claimUserName =
                new Claim(ClaimTypes.Name, username);
            Claim claimRole =
                new Claim(ClaimTypes.Role, "USUARIO");
            identity.AddClaim(claimUserName);
            identity.AddClaim(claimRole);
            //CREAMOS AL USUARIO PRINCIPAL CON ESTA IDENTIDAD
            ClaimsPrincipal userPrincipal =
                new ClaimsPrincipal(identity);
            await HttpContext.SignInAsync
            (
                CookieAuthenticationDefaults.AuthenticationScheme,
                userPrincipal,
                new AuthenticationProperties
                {
                    ExpiresUtc = DateTime.Now.AddMinutes(15)
                });
            //LLEVAMOS AL USUARIO A PERFIL UNA VEZ QUE SE HA VALIDADO
            return RedirectToAction("Perfil", "Usuarios");
        }
        else
        {
            ViewData["MENSAJE"] = "Credenciales incorrectas";
            return View();
        }
    }
}

```

Sobre **Program** debemos habilitar la seguridad, Session e incluir el tipo de esquema
(Cookies) que vamos a utilizar en Authentication

PROGRAM

Vamos al Layout e incluimos un Link para ir a nuestro Perfil

```

<li class="nav-item">
    <a class="nav-link text-info"
        asp-controller="Usuarios"
        asp-action="Perfil">
        Perfil usuario
    </a>
</li>

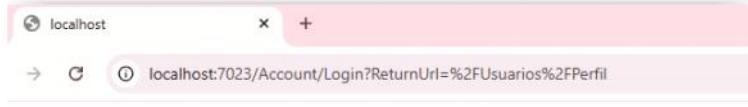
```

Por último, incluimos la decoración dentro del IActionResult que deseemos proteger.

USUARIOSCONTROLLER

```
[AuthorizeUsers]
public IActionResult Perfil()
{
    return View();
}
```

Y ya veremos que nuestro método tiene seguridad hacia un Login



Como hemos podido comprobar, la seguridad ya está activa.
El problema está en que la App intenta utilizar la seguridad administrada y debemos indicar
Que es la seguridad personalizada.

Debemos indicar, dentro de **Program** que utilizaremos otro tipo de redirección

PROGRAM

```
using Microsoft.AspNetCore.Authentication.Cookies;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();
builder.Services.AddAuthentication(
    options =>
{
    options.DefaultSignInScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultAuthenticateScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
}) .AddCookie();

builder.Services
    .AddControllersWithViews(options => options.EnableEndpointRouting = false)
;

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
//app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

//app.MapStaticAssets();

app.UseSession();
//SI TENEMOS SEGURIDAD PERSONALIZADA SE UTILIZA UseMvc EN LUGAR DE
//MapControllerRoute
//MapControllerRoute
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "Default",
        template: "{controller=Home}/{action=Index}/{id?}")
});
//app.MapControllerRoute(
//    name: "default",
//    pattern: "{controller=Home}/{action=Index}/{id?}")
//    .WithStaticAssets();

app.Run();
```

El siguiente paso es poder cerrar la sesión del usuario.

MANAGEDCONTROLLER

```
public async Task<IActionResult> Logout()
{
    await HttpContext.SignOutAsync
        (CookieAuthenticationDefaults.AuthenticationScheme);
    return RedirectToAction("Index", "Home");
}
```

Vamos a crear una vista parcial que nos llevará a Login si no estamos validados o nos
Dejará cerrar sesión si estamos validados.

Sobre Shared creamos una nueva vista llamada _MenuLogin.cshtml

_MENULOGIN.CSHTML

```
@if (Context.User.Identity.IsAuthenticated == false){  
    <li class="nav-item">  
        <a class="nav-link text-info"  
            asp-controller="Usuarios"  
            asp-action="Perfil">  
            Log In  
        </a>  
    </li>  
} else{  
    <li class="nav-item">  
        <a class="nav-link text-danger"  
            asp-controller="Managed"  
            asp-action="Logout">  
            Log out  
            <span style="color:blue">  
                @Context.User.Identity.Name  
            </span>  
        </a>  
    </li>  
}
```

Por último, dibujamos la vista parcial dentro del Layout.cshtml

LAYOUT.CSHTML

```
<li class="nav-item">  
    <a class="nav-link text-info"  
        asp-controller="Usuarios"  
        asp-action="Perfil">  
        Perfil usuario  
    </a>  
</li>  
<partial name="_MenuLogin"/>
```

Y ya tendremos nuestra App funcional



El siguiente paso será combinar la seguridad con acceso a datos de modo "real". Vamos a validar empleados con su username (APELLIDO) y su password (EMP_NO)

Una vez que tengamos todo montado, intentaremos la validación mediante ROLES.

Creamos un nuevo proyecto llamado NetCoreSeguridadEmpleados

- Tendremos una vista con todos los Empleados
- Tendremos una vista con los detalles de un empleado
- Tendremos una vista para subir el salario de los empleados por un departamento

 Microsoft.EntityFrameworkCore [0](#) by aspnet, dotnetframework, EntityFramework 9.0.2
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ, queries, change tracking, updates, and schema migrations. EF Core works with SQL Server...

 Microsoft.EntityFrameworkCore.SqlServer [0](#) by aspnet, dotnetframework, EntityFramework 9.0.2
Microsoft SQL Server database provider for Entity Framework Core.

Sobre Models creamos una nueva clase llamada Empleado

EMPLEADO

```
[Table("EMP")]  
public class Empleado  
{  
    [Key]  
    [Column("EMP_NO")]  
    public int IdEmpleado { get; set; }
```

```

[Column("APELLIDO")]
public string Apellido { get; set; }
[Column("OFICIO")]
public string Oficio { get; set; }
[Column("SALARIO")]
public int Salario { get; set; }
[Column("DEPT_NO")]
public int Departamento { get; set; }
}

```

Sobre Data creamos una clase llamada **HospitalContext**

HOSPITALCONTEXT

```

public class HospitalContext : DbContext
{
    public HospitalContext(DbContextOptions<HospitalContext> options)
        :base(options) { }

    public DbSet<Empleado> Empleados { get; set; }
}

```

Sobre Repositories creamos una nueva clase llamada **RepositoryHospital**

REPOSITORYHOSPITAL

```

public class RepositoryHospital
{
    private HospitalContext context;

    public RepositoryHospital(HospitalContext context)
    {
        this.context = context;
    }

    public async Task<List<Empleado>> GetEmpleadosAsync()
    {
        return await this.context.Empleados.ToListAsync();
    }

    public async Task<Empleado> FindEmpleadoAsync(int idEmpleado)
    {
        return await this.context.Empleados
            .FirstOrDefaultAsync(x => x.IdEmpleado == idEmpleado);
    }

    public async Task<List<Empleado>> GetEmpleadosDepartamentoAsync
        (int idDepartamento)
    {
        return await this.context.Empleados
            .Where(x => x.Departamento == idDepartamento).ToListAsync();
    }

    public async Task UpdateSalarioEmpleadosAsync
        (int idDepartamento, int incremento)
    {
        List<Empleado> empleados = await
            this.GetEmpleadosDepartamentoAsync(idDepartamento);
        foreach (Empleado emp in empleados)
        {
            emp.Salario += incremento;
        }
        await this.context.SaveChangesAsync();
    }

    public async Task<Empleado> LogInEmpleadoAsync
        (string apellido, int idEmpleado)
    {
        Empleado empleado = await this.context.Empleados
            .Where(z => z.Apellido == apellido &&
            z.IdEmpleado == idEmpleado).FirstOrDefaultAsync();
        return empleado;
    }
}

```

Sobre Controllers creamos un nuevo controlador llamado **EmpleadosController**

EMPLEADOSCONTROLLER

```

public class EmpleadosController : Controller
{
    private RepositoryHospital repo;

    public EmpleadosController(RepositoryHospital repo)
    {
        this.repo = repo;
    }

    public async Task<IActionResult> Index()
    {
        List<Empleado> empleados = await this.repo.GetEmpleadosAsync();
        return View(empleados);
    }

    public async Task<IActionResult> Details(int idempleado)
    {
        Empleado empleado = await this.repo.FindEmpleadoAsync(idempleado);
        return View(empleado);
    }
}

```

Montamos las vistas y resolvemos dependencias

APPSETTINGS.JSON

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft.AspNetCore": "Warning"  
        }  
    },  
    "AllowedHosts": "*",  
    "ConnectionStrings": {  
        "SqlHospital": "Data Source=LOCALHOST\\DESARROLLO;Ini  
    }  
}
```

PROGRAM

```
var builder = WebApplication.CreateBuilder(args);  
  
// Add services to the container.  
string connectionString =  
    builder.Configuration.GetConnectionString("SqlHospital");  
builder.Services.AddTransient<RepositoryHospital>();  
builder.Services.AddDbContext<HospitalContext>  
    (options => options.UseSqlServer(connectionString));
```

A continuación es el momento de montar la seguridad

Lo primero será tener un método dentro de **RepositoryHospital** para validar al usuario
Y devolver el propio User.

REPOSITORYHOSPITAL

```
public async Task<Empleado> LogInEmpleadoAsync  
    (string apellido, int idEmpleado)  
{  
    Empleado empleado = await this.context.Empleados  
        .Where(z => z.Apellido == apellido &&  
        z.IdEmpleado == idEmpleado).FirstOrDefaultAsync();  
    return empleado;  
}
```

Habilitamos la seguridad dentro de **Program** y quitamos lo de app.Routing y MapStatics

Gracias AARON!!!!

```
builder.Services.AddDistributedMemoryCache();  
builder.Services.AddSession();  
builder.Services.AddAuthentication(options =>  
{  
    options.DefaultSignInScheme =  
        CookieAuthenticationDefaults.AuthenticationScheme;  
    options.DefaultAuthenticateScheme =  
        CookieAuthenticationDefaults.AuthenticationScheme;  
    options.DefaultChallengeScheme =  
        CookieAuthenticationDefaults.AuthenticationScheme;  
}).AddCookie();  
  
builder.Services  
    .AddControllersWithViews(options => options.EnableEndpointRouting = false);
```

```

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();
app.UseSession();
app.UseMvc(routes =>
{
    routes.MapRoute(name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

```

A continuación, creamos un controlador para las validaciones

Sobre **Controllers** creamos un nuevo controlador llamado **ManagedController**

MANAGEDCONTROLLER

```

public class ManagedController : Controller
{
    private RepositoryHospital repo;

    public ManagedController(RepositoryHospital repo)
    {
        this.repo = repo;
    }

    public IActionResult Login()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        Login(string username, string password)
    {
        int idEmpleado = int.Parse(password);
        Empleado empleado = await
            this.repo.LogInEmpleadoAsync(username, idEmpleado);
        if (empleado != null)
        {
            ClaimsIdentity identity =
                new ClaimsIdentity(
                    CookieAuthenticationDefaults.AuthenticationScheme,
                    ClaimTypes.Name, ClaimTypes.Role);
            Claim claimName =
                new Claim(ClaimTypes.Name, empleado.Apellido);
            identity.AddClaim(claimName);
            //COMO POR AHORA, NO VAMOS A UTILIZAR ROLES
            //NO LO HACEMOS
            ClaimsPrincipal userPrincipal =
                new ClaimsPrincipal(identity);
            await HttpContext.SignInAsync(
                CookieAuthenticationDefaults.AuthenticationScheme,
                userPrincipal);
            //LLEVAMOS AL USER A UNA VISTA QUE TODAVIA NO TENEMOS
            //PERO QUE HAREMOS EN BREVE
            return RedirectToAction("PerfilEmpleado", "Empleados");
        }
        else
        {
            ViewData["MENSAJE"] = "Usuario/Password incorrectos";
            return View();
        }
    }

    public async Task<IActionResult> Logout()
    {
        await HttpContext.SignOutAsync
            (CookieAuthenticationDefaults.AuthenticationScheme);
        return RedirectToAction("Index", "Home");
    }
}

```

LOGIN.CSHTML

```

<h1 style="color:blue">
    Log In Empleados
</h1>

<form method="post">
    <label>User</label>
    <input type="text" name="username"
        class="form-control"/>
    <label>Password</label>
    <input type="password" name="password"
        class="form-control"/>
    <button class="btn btn-info">
        Log In
    </button>
</form>
<h2 style="color:red">@ViewData["MENSAJE"]</h2>

```

Creamos una carpeta llamada **Filters** y una clase llamada **AuthorizeEmpleadosAttribute**

```

AUTHORIZEEMPLEADOSATTRIBUTE

public class AuthorizeEmpleadosAttribute : AuthorizeAttribute,
    IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        //POR AHORA, SOLAMENTE NOS VA A INTERESAR SI
        //EXISTE EL EMPLEADO
        var user = context.HttpContext.User;
        if (user.Identity.IsAuthenticated == false)
        {
            context.Result = this.GetRoute("Managed", "Login");
        }
    }

    //TENDREMOS MULTIPLES REDIRECCIONES, POR LO QUE NOS CREAMOS UN
    //METODO PARA FACILITAR EL CODIGO
    private RedirectToRouteResult GetRoute(
        string controller, string action)
    {
        RouteValueDictionary ruta =
            new RouteValueDictionary(
                new { controller = controller, action = action });
        RedirectToRouteResult result =
            new RedirectToRouteResult(ruta);
        return result;
    }
}

```

Sobre **Shared** creamos una nueva vista llamada **_MenuEmpleado.cshtml**

```

_MENUEMPLEADO.CSHTML

@if (Context.User.Identity.IsAuthenticated == false){
    //EFECTO OPTICO DEL LOGIN
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Empleados"
            asp-action="PerfilEmpleado">
            Log In
        </a>
    </li>
} else{
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Managed"
            asp-action="Logout">
            Log Out <span style="color:blue">@Context.User.Identity.Name</span>
        </a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-controller="Empleados"
            asp-action="PerfilEmpleado">
            Mi perfil
        </a>
    </li>
}

```

Incluimos el menú dentro de nuestro **_Layout**

```

        <ul>
            <li>
                <a href="#">Empleados
                </a>
            </li>
            <partial name="_MenuEmpleado"/>
        </ul>
</div>

```

Creamos un método dentro de **EmpleadosController** llamado **PerfilEmpleado** y lo Protegemos mediante la decoración de seguridad

EMPLEADOSCONTROLLER

```
[AuthorizeEmpleados]  
  
public IActionResult PerfilEmpleado()  
{  
    return View();  
}
```

Por último, creamos una vista para visualizar la funcionalidad.

El siguiente paso es poder dibujar los datos del usuario que ha realizado la validación.

Para empezar, no tenemos esos datos. ¿Qué es lo que tenemos actualmente guardado?

Tenemos el apellido como Name del usuario para ponerlo bonito, por ejemplo, en la Vista parcial del Layout.

Si necesitamos almacenar más datos, debemos hacerlo en **Claim**

Un **Claim** representa datos propios del usuario que ha realizado el Login

```
Claim claimId =  
    new Claim("IDUSER", "ID USUARIO");  
identity.AddClaim(claimId);
```

De esta forma, podemos recuperar cualquier dato del usuario a lo largo de toda Nuestra aplicación. Para validar o para dibujar.

Vamos a almacenar algunos datos del Empleado en diferentes Claims.

MANAGEDCONTROLLER

```
[HttpPost]  
public async Task<IActionResult>  
    Login(string username, string password)  
{  
    int idEmpleado = int.Parse(password);  
    Empleado empleado = await  
        this.repo.LogInEmpleadoAsync(username, idEmpleado);  
    if (empleado != null)  
    {  
        ClaimsIdentity identity =  
            new ClaimsIdentity(  
                CookieAuthenticationDefaults.AuthenticationScheme,  
                ClaimTypes.Name, ClaimTypes.Role);  
        Claim claimName =  
            new Claim(ClaimTypes.Name, empleado.Apellido);  
        identity.AddClaim(claimName);  
        //ALMACENAMOS EL ID  
        Claim claimId =  
            new Claim(ClaimTypes.NameIdentifier  
                , empleado.IdEmpleado.ToString());  
        identity.AddClaim(claimId);  
        //COMO ROLE, VOY A UTILIZAR EL DATO DEL OFICIO  
        Claim claimOficio =  
            new Claim(ClaimTypes.Role, empleado.Oficio);  
        identity.AddClaim(claimOficio);  
        Claim claimSalario =  
            new Claim("Salario", empleado.Salario.ToString());  
        identity.AddClaim(claimSalario);  
        Claim claimDept =  
            new Claim("Departamento", empleado.Departamento.ToString());  
        identity.AddClaim(claimDept);  
  
        ClaimsPrincipal userPrincipal =  
            new ClaimsPrincipal(identity);  
        await HttpContext.SignInAsync(  
            CookieAuthenticationDefaults.AuthenticationScheme,  
            userPrincipal);  
        //LLEVAMOS AL USER A UNA VISTA QUE TODAVIA NO TENEMOS  
        //PERO QUE HAREMOS EN BREVE  
        return RedirectToAction("PerfilEmpleado", "Empleados");  
    }  
    else  
    {  
        ViewData["MENSAJE"] = "Usuario/Password incorrectos";  
        return View();  
    }  
}
```

Vamos a dibujar los datos de los Claims del usuario directamente en la Vista del Perfil del Empleado.

Podemos tener múltiples Claim para un mismo **ClaimType**.

Por ejemplo, acabamos de poner un Role al usuario con su oficio.

Un usuario podría tener múltiples Roles???

Para recuperar un Role se utiliza a modo de colección.

Ahora mismo solamente tenemos uno, porque utilizaremos **FirstOrDefault** para recuperar El único Role que tenemos...

PROFILEMPLEADO.CSHTML

```
@using System.Security.Claims

<h1>
    Perfil del Sr/Sra @Context.User.Identity.Name
</h1>

<ul class="list-group">
    <li class="list-group-item">
        Oficio: @Context.User.FindFirst(ClaimTypes.Role).Value
    </li>
    <li class="list-group-item">
        Salario: @Context.User.FindFirst("Salario").Value
    </li>
    <li class="list-group-item">
        Departamento: @Context.User.FindFirst("Departamento").Value
    </li>
</ul>
<h2>Todos los claims del usuario</h2>
<ul>
    @foreach (Claim claim in Context.User.Claims){
        <li>@claim.Type: <b>@claim.Value</b></li>
    }
</ul>
```

Y podremos visualizar el resultado

Home Privacy Empleados Log Out **REY** Mi perfil

Perfil del Sr/Sra REY

Oficio: PRESIDENTE

Salario: 650000

Departamento: 10

Todos los claims del usuario

- http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: **REY**
- http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier: **7839**
- http://schemas.microsoft.com/ws/2008/06/identity/claims/role: **PRESIDENTE**
- Salario: **650000**
- Departamento: **10**

El siguiente paso es implementar seguridad mediante Roles.

Aunque el usuario esté en nuestra base de datos, vamos a filtrar los permisos de lo que puede visualizar dependiendo de su OFICIO.

Tendremos una vista llamada **ErrorAcceso.cshtml** que será lo que verá nuestro usuario Sin permisos suficientes.

En nuestra aplicación con Login, tendrán acceso a zonas determinadas solamente los Siguientes Roles: **ANALISTA, DIRECTOR Y PRESIDENTE**

Nota: También podríamos validar con el nombre del usuario.

La validación mediante roles ya hemos comenzado a aplicarla con los Claims.

```
//COMO ROLE, VOY A UTILIZAR EL DATO DEL OFICIO
Claim claimOficio =
    new Claim(ClaimTypes.Role, empleado.Oficio);
```

Dentro de **User** tenemos un método llamado **IsInRole("ROLE")** para preguntar si un usuario Perteneces a un determinado grupo.

Comenzamos creando un nuevo método dentro de **ManagedController** para el Acceso.

MANAGEDCONTROLLER

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

```

<h1 style="color:red">
    No tiene permisos para acceder a esta zona
</h1>
<img src "~/images/erroracceso.jpg"/>

```

Vamos a realizar la validación en el filter mediante una condición.

AUTHORIZEEMPLEADOSATTRIBUTE

```

public void OnAuthorization(AuthorizationFilterContext context)
{
    //POR AHORA, SOLAMENTE NOS VA A INTERESAR SI
    //EXISTE EL EMPLEADO
    var user = context.HttpContext.User;
    if (user.Identity.IsAuthenticated == false)
    {
        context.Result = this.GetRoute("Managed", "Login");
    }
    else
    {
        //COMPROBAMOS LOS ROLES DE ACCESO
        if (user.IsInRole("PRESIDENTE") == false
            && user.IsInRole("DIRECTOR") == false
            && user.IsInRole("ANALISTA") == false)
        {
            context.Result = this.GetRoute("Managed", "ErrorAcceso");
        }
    }
}

```

Y podemos comprobar el resultado actual

The screenshot shows a web browser displaying a 403 Forbidden error page. The page has a red header with the text "No tiene permisos para acceder a esta zona". Below the header is a video player showing a scene from a TV show. The video shows two men in black shirts, one with a mustache and the other bald. The video has a caption at the bottom that reads "NO LO SE RICK, PARECE FALSO". The browser's navigation bar is visible at the top, showing links for Home, Privacy, Empleados, Log Out, SANCHA, and Mi perfil.

Lo siguiente que vamos a realizar será interactuar con el usuario que se ha validado.

Por ejemplo, tenemos un método que nos devuelve los Empleados por departamento

REPOSITORYHOSPITAL

```

public async Task<List<Empleado>> GetEmpleadosDepartamentoAsync
    (int idDepartamento)
{
    return await this.context.Empleados
        .Where(x => x.Departamento == idDepartamento).ToListAsync();
}

```

Vamos a crear una vista que mostrará los compis de trabajo de un empleado validado, es decir, Nos mostrará los compañeros del mismo departamento que el usuario validado.

Ya tenemos también el Claim del usuario por su Departamento

```

Claim claimDept =
    new Claim("Departamento", empleado.Departamento.ToString());
identity.AddClaim(claimDept);

```

Sobre **EmpleadosController** creamos un nuevo método que recuperará el Claim del Departamento y devolverá los Empleados de dicho departamento.

EMPLEADOSCONTROLLER

```

[AuthorizeEmpleados]
0 references
public async Task<IActionResult> Compis()
{
    //RECUPERAMOS EL DATO DEL CLAIM DE Departamento
    string dato =
        HttpContext.User.FindFirst("Departamento").Value;
    int idDepartamento = int.Parse(dato);
    List<Empleado> empleados = await
        this.repo.GetEmpleadosDepartamentoAsync(idDepartamento);
    return View(empleados);
}

```

Dibujamos nuestra vista **Compis.cshtml**

Ponemos un Link dentro de la página de **Layout.cshtml**

_LAYOUT.CSHTML

```

<li class="nav-item">
    <a class="nav-link text-danger"
       asp-controller="Empleados"
       asp-action="Compis">
        Compis
    </a>
</li>

```

Y podremos visualizar el resultado

Apellido	Oficio	Departamento
CEREZO	DIRECTOR	10
REY	PRESIDENTE	10
MUÑOZ	EMPLEADO	10
CASALES	EMPLEADO	10
ALCALÁ	EMPLEADO	10

Podemos visualizar que nuestra aplicación tiene un comportamiento "extraño"

Si pulsamos en Log In, nos lleva al Perfil después del login.

Si pulsamos en Compis, nos lleva a Perfil después del Login

Esta acción sucede porque estamos utilizando una devolución de ruta estática dentro del Login.

```

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    userPrincipal);
//LLEVAMOS AL USER A UNA VISTA QUE TODAVIA NO TENEMOS
//PERO QUE HAREMOS EN BREVE
return RedirectToAction("PerfilEmpleado", "Empleados");

```

Deberíamos devolver a la vista donde hemos pulsado.

¿Qué clase puede saber en qué vista hemos pulsado?

La única clase que sabe de donde venimos para hacer la navegación dinámica es la Clase **FILTER**

Debemos enviar la información desde **Filter** hasta **ManagedController** para poder Hacer RedirectToAction dinámico.

Para ello, se utiliza **TempData**

Para utilizar **TempData** necesitamos tres características:

- 1) Memoria distribuida en Cache
- 2) Session
- 3) AddControllersWithViews() debemos indicar añada un proveedor de rutas en TempData

PROGRAM

```

builder.Services
    .AddControllersWithViews
    (options => options.EnableEndpointRouting = false)
    .AddSessionStateTempDataProvider();

```

Dentro de cualquier **Context** tenemos la información de la navegación de los controllers y Los action, además de posibles parámetros que tuviera el método mediante **RouteData**

```
template: "{controller=Home}/{action=Index}/{id?}");
```

```
Context.RouteData.Values["controller"]
Context.RouteData.Values["action"]
Context.RouteData.Values["id"]
```

La información de los parámetros es opcional, puede que no venga el ID, pero siempre Tendremos acceso al Controller y al Action

Vamos a capturar la información de **dónde** ha pulsado el usuario para la validación antes De llegar al **Filter**

Necesitamos recuperar **TempData** dentro de **Filter**, pero no cualquier tempdata sino El que está dentro de nuestra aplicación.
Para ello, dentro de la inyección tenemos **ITempDataProvider** que es la clase que utiliza TempData dentro de nuestra App.

Para recuperar el **ITempDataProvider** necesitamos hacerlo de forma explícita, es decir, Recuperar directamente el Objeto que está en los servicios

Cuando realizamos la inyección de objetos, es posible que tengamos que recuperar Un objeto que está dentro de los Servicios de Program y que no podamos recuperarlo Desde un constructor, por ejemplo, imaginemos que estamos en un método **static**, ahí No existe clase ni constructor
Para estos casos, si deseamos recuperar un objeto inyectado, debemos hacerlo directamente Desde el container (Program)

En nuestro ejemplo, debemos recuperar utilizando el **Context**, que tiene una propiedad llamada **RequestServices** que nos permite recuperar cualquier objeto de la inyección, por ejemplo:

```

RepositoryHospital repo =
    context.HttpContext.RequestServices
    .GetService<RepositoryHospital>();

```

AUTHORIZEEMPLEADOSATTRIBUTE

```

public void OnAuthorization(AuthorizationFilterContext context)
{
    //POR AHORA, SOLAMENTE NOS VA A INTERESAR SI
    //EXISTE EL EMPLEADO
    var user = context.HttpContext.User;
    //NECESITAMOS EL CONTROLLER Y EL ACTION DE DONDE
    //HA PULSADO EL USUARIO ANTES DE ENTRAR AQUÍ
    string controller =
        context.RouteData.Values["controller"].ToString();
    string action =
        context.RouteData.Values["action"].ToString();
    ITempDataProvider provider =
        context.HttpContext.RequestServices
        .GetService<ITempDataProvider>();
}

```

```

//ESTA CLASE TIENE EL TEMPDATA DE NUESTRA APP
var TempData =
    provider.LoadTempData(context.HttpContext);
TempData["controller"] = controller;
TempData["action"] = action;
//GUARDAMOS EL TEMPDATA QUE ACABAMOS DE RECUPERAR
//DENTRO DE LA APLICACION
provider.SaveTempData(context.HttpContext, TempData);
if (User.Identity.IsAuthenticated == false)
{
    context.Result = this.GetRoute("Managed", "Login");
}
else
{
    //COMPROBAMOS LOS ROLES DE ACCESO
    if (user.IsInRole("PRESIDENTE") == false
        && user.IsInRole("DIRECTOR") == false
        && user.IsInRole("ANALISTA") == false)
    {
        context.Result = this.GetRoute("Managed", "ErrorAcceso");
    }
}

```

A continuación, dentro de **Login** en **ManagedController** hacemos la ruta de Redirect Dinámica.

MANAGEDCONTROLLER

```

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    userPrincipal);
string controller =
    TempData["controller"].ToString();
string action =
    TempData["action"].ToString();
return RedirectToAction(action, controller);

```

A continuación, vamos a realizar un dibujo dinámico dependiendo quién ha entrado en nuestra Página de Compis, es decir, dependiendo del Role del usuario.

Por ejemplo, lo que vamos a realizar es la funcionalidad para incrementar Salarios de los Empleados por departamento dentro de **Compis**

COMPIS.CSHTML

```

<form method="post">
    <label>Incremento salarial</label>
    <input type="text" name="incremento" class="form-control"/>
    <button class="btn btn-info">
        A subir salarios!!!!
    </button>
</form>

```

EMPLEADOSCONTROLLER

```

[AuthorizeEmpleados]
[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public async Task<IActionResult> Compis(int incremento)
{
    //DEBEMOS RECUPERAR EL ID DEL DEPARTAMENTO
    //DEL USUARIO QUE SE HA VALIDADO
    string dato =
        HttpContext.User.FindFirst("Departamento").Value;
    int idDepartamento = int.Parse(dato);
    await this.repo.UpdateSalarioEmpleadosAsync
        (idDepartamento, incremento);
    List<Empleado> empleados = await
        this.repo.GetEmpleadosDepartamentoAsync(idDepartamento);
    return View(empleados);
}

```

Una vez que hemos visto que es funcional, podemos preguntar perfectamente en la vista Quién ha entrado para nuestro dibujo, por ejemplo, solamente podrán subir salarios PRESIDENTE y DIRECTOR

```

@if (Context.User.IsInRole("PRESIDENTE") == true
|| Context.User.IsInRole("DIRECTOR") == true){
    <form method="post">
        <label>Incremento salarial</label>
        <input type="text" name="incremento" class="form-control"/>
        <button class="btn btn-info">
            A subir salarios!!!!
        </button>
    </form>
}

```

SEGURIDAD CON POLITICAS Y ROLES

Actualmente, tenemos varias posibilidades, hemos creado una vista donde solamente Deseamos que puedan modificar unos determinados Roles (PRESIDENTE Y DIRECTOR)

En nuestra App solamente pueden entrar PRESIDENTE, DIRECTOR Y ANALISTA

Lo que hemos visto es seguridad para USUARIOS, pero **no** hemos visto seguridad con Roles, Ha sido simplemente un ejemplo.

Para poder tener seguridad por Roles o por "ideas" (mayor de edad) se utilizan **Políticas**

Una clase **Filter** de autorización solamente debe encargarse de validar si el usuario EXISTE o NO.

Con políticas podemos crear este escenario:

- **[AuthorizeEmpleados("PRESIDENTE", "DIRECTOR")]**: Claim Role
- **[AuthorizeEmpleados(Política = "MayorEdad")]**: Esto es personalizar el filtro y Realizar algún código para comprobar si es menor de edad o no.

Las políticas de seguridad de Claims o de Roles simplemente se declaran, no necesitamos nada

Más, es decir, que el Usuario tenga un determinado Role.

Las políticas de lógica (MayorEdad) necesitan una clase **Policy** asociada para poder realizar el Código.

Lo que deseamos hacer ahora mismo es:

- Cualquier Empleado puede entrar en nuestra App y visualizar su PERFIL
- Solamente DIRECTOR, PRESIDENTE Y ANALISTA pueden ver sus Compis
- Solamente DIRECTOR y PRESIDENTE pueden subir salarios. (View)

Lo primero que vamos a realizar para visualizar esta funcionalidad es dejar en la clase **Filter** solamente el código de validar USER a Login.

```

//GUARDAMOS EL TEMPDATA QUE ACABAMOS DE RECUPERAR
//DENTRO DE LA APLICACION
provider.SaveTempData(context.HttpContext, TempData);
if (user.Identity.IsAuthenticated == false)
{
    context.Result = this.GetRoute("Managed", "Login");
}

```

Por supuesto, debemos tener un **Role** para el usuario validado

MANAGEDCONTROLLER

```

//COMO ROLE, VOY A UTILIZAR EL DATO DEL OFICIO
Claim claimOficio =
    new Claim(ClaimTypes.Role, empleado.Oficio);
identity.AddClaim(claimOficio);

```

Nota: Los Roles los escribimos a mano dentro de nuestro código C#, deben coincidir mayúsculas Con minúsculas.

Dentro de nuestro **Program** debemos habilitar la Autorización mediante política de Roles Indicando los Roles que deseamos incluir y un nombre para dicha política

PROGRAM

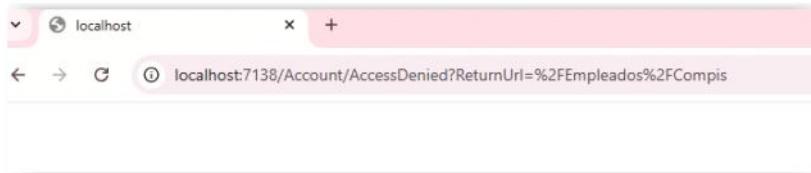
```
//LAS POLITICAS SE AGREGAN A Authorization
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("SOLOJEFES",
        policy => policy.RequireRole("PRESIDENTE", "DIRECTOR", "ANALISTA"));
});
```

Para comprobar el funcionamiento, implementamos la política dentro de **Compis** e incluimos SOLOJEFES dentro de **[AuthorizeEmpleados]**

EMPLEADOSCONTROLLER

```
[AuthorizeEmpleados(Policy = "SOLOJEFES")]
0 references | serraguti, 1 day ago | 1 author, 1 change
public async Task<IActionResult> Compis()
{
    //RECUPERAMOS EL DATO DEL CLAIM DE Departamento
    string dato =
        HttpContext.User.FindFirst("Departamento").Value;
    int idDepartamento = int.Parse(dato);
    List<Empleado> empleados = await
        this.repo.GetEmpleadosDepartamentoAsync(idDepartamento);
    return View(empleados);
}
```

Si entramos con un Role que NO es cualquiera de SOLOJEFES veremos que la zona ya está Protegida y que no tenemos acceso, ofreciendo una URL extraña...



También podemos personalizar que nos lleve a una página personalizada para cuando no tengamos Acceso con permisos suficientes.

No podemos personalizar la página de acceso denegado dependiendo quién ha entrado.

Vamos a utilizar nuestro IActionResult de ErrorAcceso dentro de **ManagedController**

```
0 references | serraguti, 1 day ago | 1 author, 1 change
public IActionResult ErrorAcceso()
{
    return View();
}
```

```
<h1 style="color:red">
    No tiene permisos para acceder a esta zona
</h1>

```

Sobre **Program** tenemos que agregar **AddCookie()** al método **AddAuthentication()**
E incluir la página y su ubicación.

PROGRAM

```

builder.Services.AddAuthentication(options =>
{
    options.DefaultSignInScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultAuthenticateScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
}).AddCookie(
    CookieAuthenticationDefaults.AuthenticationScheme,
    config =>
{
    config.AccessDeniedPath = "/Managed/ErrorAcceso";
});

```

Y ya tendremos nuestra página personalizada

No tiene permisos para acceder a esta zona



POLITICAS DE CLAIMS

Podemos personalizar el acceso a un determinado Controller/IActionResult dependiendo De los Claims de un usuario, pero NO de su valor, sino de si tiene un Claim o no.

Pongamos, por ejemplo, que no tenemos múltiples Roles dentro de nuestra App, Solamente tenemos un Admin y Usuarios.

Si un usuario tiene el Claim de **Admin** ya tendríamos el acceso a los sitios.

Vamos a crear una zona no real dónde solamente entrará un Empleado que sea Administrador, es decir, que tenga un Claim de **Admin**.

Sobre **EmpleadosController** creamos un nuevo **IActionResult**

EMPLEADOSCONTROLLER

```

0 references | 0 changes | 0 authors, 0 changes
public IActionResult AdminEmpleados()
{
    return View();
}

```

Creamos nuestra vista

```

<h1 style="color:blue">
    Zona de administración de empleados
</h1>


```

Sobre **ManagedController** vamos a seleccionar cualquier empleado por su ID y le

Incluimos un Claim de Admin

MANAGEDCONTROLLER

```
//INCLUIMOS UN CLAIM DE ADMIN A CUALQUIER EMPLEADO
//AL AZAR (ARROYO)
if (empleado.IdEmpleado == 7499)
{
    //CREAMOS UN CLAIM
    Claim claimAdmin =
        new Claim("Admin", "Soy el super jefe de la empresa");
    identity.AddClaim(claimAdmin);
}
```

Incluimos una nueva política dentro de Program y dentro de AddAuthorization

PROGRAM

```
//LAS POLITICAS SE AGREGAN A Authorization
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("SOLOJEFES",
        policy => policy.RequireRole("PRESIDENTE", "DIRECTOR", "ANALISTA"));
    options.AddPolicy("AdminOnly",
        policy => policy.RequireClaim("Admin"));
});
```

Dentro de EmpleadosController incluimos la seguridad sobre el método que hemos creado
Llamado AdminEmpleados

EMPLEADOSCONTROLLER

```
[AuthorizeEmpleados(Policy ="AdminOnly")]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult AdminEmpleados()
{
    return View();
}
```

POLITICA HANDLE PERSONALIZADA

El último concepto nos permite personalizar la política de acceso dependiendo de una Lógica cualquiera.

Necesitamos un código para aplicar dicha lógica de entrada al método, por ejemplo, Controlar si alguien es Mayor de Edad para entrar a una determinada zona.

Para este tipo de política necesitamos de una clase llamada Requirements que hereda de AuthorizationHandler

Podemos preguntar por cualquier característica, es decir, algo del Claim del usuario, Del Repo o de lo que sea.

Vamos a crear un política en la que solamente tendremos una zona donde entrarán Los Empleados RICOS

Necesitamos el Claim del Salario dentro del User.

MANAGEDCONTROLLER

```
Claim claimSalario =
    new Claim("Salario", empleado.Salario.ToString());
identity.AddClaim(claimSalario);
```

Sobre el proyecto, creamos una nueva carpeta llamada Policies y dentro una clase llamada OverSalarioRequirement

OVERSALARIOREQUIREMENT

```
public class OverSalarioRequirement :
    AuthorizationHandler<OverSalarioRequirement>,
    IAuthorizationRequirement
{
    protected override Task HandleRequirementAsync
        (AuthorizationHandlerContext context,
        OverSalarioRequirement requirement)
    {
        ...
    }
}
```

```

//PODEMOS PREGUNTAR SI EXISTE O NO UN CLAIM
if (context.User.HasClaim(x => x.Type == "Salario") == false)
{
    context.Fail();
}
else
{
    string data =
        context.User.FindFirst("Salario").Value;
    int salario = int.Parse(data);
    if (salario >= 250000)
    {
        context.Succeed(requirement);
    }
    else
    {
        context.Fail();
    }
}
return Task.CompletedTask;
}
}

```

Incluimos dentro de Program nuestra nueva política creada.

PROGRAM

```

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("SOLOJEFES",
        policy => policy.RequireRole("PRESIDENTE", "DIRECTOR", "ANALISTA"));
    options.AddPolicy("AdminOnly",
        policy => policy.RequireClaim("Admin"));
    options.AddPolicy("SoloRicos",
        policy => policy.Requirements.Add(new OverSalarioRequirement()));
});

```

Sobre **EmpleadosController** creamos una nueva zona y una página para probar esta nueva Funcionalidad.

EMPLEADOSCONTROLLER

```

[AuthorizeEmpleados(Policy = "SoloRicos")]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult ZonaNoble()
{
    return View();
}

```

CONTROL DE VARIABLES DINAMICAS DENTRO DE LAS RUTAS DEL LOGIN

Como hemos visto, si necesitamos entrar en una zona protegida que tiene algún parámetro La redirección desde el Login no lleva dichos parámetros, por lo que no funciona
Por ejemplo, si estamos protegiendo un **Delete**

```

[AuthorizeEmpleados]
0 references | serraguti, 1 day ago | 1 author, 1 change
public async Task<IActionResult> Details(int idempleado)
{
    Empleado empleado = await this.repo.FindEmpleadoAsync(idempleado);
    return View(empleado);
}

```

Dentro de **ManagedController** solamente estamos enviando el Controller y el action

MANAGEDCONTROLLER

```

string controller =
    TempData["controller"].ToString();
string action =
    TempData["action"].ToString();
return RedirectToAction(action, controller);

```

El problema lo tenemos en las RUTAS (Routes) de nuestra aplicación.

Actualmente estamos queriendo recuperar **idempleado** y debemos recuperarlo
Para poder enviarlo del Filter al Managed

En nuestras rutas definidas, debemos indicar los posibles valores de parámetros que

Tendremos cuando realizamos esta acción con Filter, si no lo indicamos solamente Recuperará los parámetros declarados por defecto. ({id})

```
routes.MapRoute(name: "default",
template: "{controller=Home}/{action=Index}/{id?}");
```

Tenemos dos opciones:

- 1) Mapeamos una o varias rutas con los parámetros dinámicos que tengamos que Capturar en los Filter.

```
app.UseMvc(routes =>
{
    routes.MapRoute(name: "defaultIdEmpleado",
        template: "{controller=Home}/{action=Index}/{idempleado?}");
    routes.MapRoute(name: "otraRuta",
        template: "{controller=Home}/{action=Index}/{apellido?}/{oficio?}");
    routes.MapRoute(name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

- 2) Como solo tenemos un parámetros, llamar a nuestro parámetro de recepción en En el método id sin crear más rutas en Program

```
[AuthorizeEmpleados]
public async Task<IActionResult> Details(int id)
{
    Empleado empleado = await this.repo.FindEmpleadoAsync(id);
    return View(empleado);
}
```

Posteriormente, dentro del Filter recuperamos el ID que es un parámetro opcional y Que tendremos que preguntar si existe para mandarlo o no mandarlo.

AUTHORIZEEMPLEADOSATTRIBUTE

```
public void OnAuthorization(AuthorizationFilterContext context)
{
    //POR AHORA, SOLAMENTE NOS VA A INTERESAR SI
    //EXISTE EL EMPLEADO
    var user = context.HttpContext.User;
    //NECESITAMOS EL CONTROLLER Y EL ACTION DE DONDE
    //HA PULSADO EL USUARIO ANTES DE ENTRAR AQUI
    string controller =
        context.RouteData.Values["controller"].ToString();
    string action =
        context.RouteData.Values["action"].ToString();
    var id = context.RouteData.Values["id"];
    ITempDataProvider provider =
        context.HttpContext.RequestServices
            .GetService<ITempDataProvider>();
    //ESTA CLASE TIENE EL TEMPDATA DE NUESTRA APP
    var TempData =
        provider.LoadTempData(context.HttpContext);
    TempData["controller"] = controller;
    TempData["action"] = action;
    if (id != null)
    {
        TempData["id"] = id.ToString();
    }
    else
    {
        //ELIMINAMOS LA KEY DEL ID SI NO VIENE NADA
        TempData.Remove("id");
    }
    //GUARDAMOS EL TEMPDATA QUE ACABAMOS DE RECUPERAR
    //DENTRO DE LA APLICACION
    provider.SaveTempData(context.HttpContext, TempData);
    if (user.Identity.IsAuthenticated == false)
    {
        context.Result = this.GetRoute("Managed", "Login");
    }
}
```

Por último, sobre **ManagedController**, capturamos el id y lo enviamos dentro de RedirectToAction.

Si el id no existe, no lo enviamos.

MANAGEDCONTROLLER

```
[HttpPost]
public async Task<IActionResult>
    Login(string username, string password)
{
    int idEmpleado = int.Parse(password);
    Empleado empleado = await
        this.repo.LogInEmpleadoAsync(username, idEmpleado);
    if (empleado != null)
    {
```

```

ClaimsIdentity identity =
    new ClaimsIdentity(
        CookieAuthenticationDefaults.AuthenticationScheme,
        ClaimTypes.Name, ClaimTypes.Role);
Claim claimName =
    new Claim(ClaimTypes.Name, empleado.Apellido);
identity.AddClaim(claimName);
//ALMACENAMOS EL ID
Claim claimId =
    new Claim(ClaimTypes.NameIdentifier
        , empleado.IdEmpleado.ToString());
identity.AddClaim(claimId);
//COMO ROLE, VOY A UTILIZAR EL DATO DEL OFICIO
Claim claimOficio =
    new Claim(ClaimTypes.Role, empleado.Oficio);
identity.AddClaim(claimOficio);
Claim claimSalario =
    new Claim("Salario", empleado.Salario.ToString());
identity.AddClaim(claimSalario);
Claim claimDept =
    new Claim("Departamento", empleado.Departamento.ToString());
identity.AddClaim(claimDept);
//INCLUIREMOS UN CLAIM DE ADMIN A CUALQUIER EMPLEADO
//AL AZAR (ARROYO)
if (empleado.IdEmpleado == 7499)
{
    //CREAMOS UN CLAIM
    Claim claimAdmin =
        new Claim("Admin", "Soy el super jefe de la empresa");
    identity.AddClaim(claimAdmin);
}

ClaimsPrincipal userPrincipal =
    new ClaimsPrincipal(identity);
await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    userPrincipal);
string controller =
    TempData["controller"].ToString();
string action =
    TempData["action"].ToString();
if (TempData["id"] != null)
{
    string id =
        TempData["id"].ToString();
    return RedirectToAction(action, controller
        , new { id = id });
}
else
{
    return RedirectToAction(action, controller);
}
}
else
{
    ViewData["MENSAJE"] = "Usuario/Password incorrectos";
    return View();
}
}

```