UD6 Desarrollo de aplicaciones web con ASP.NET Core.
6.3 Autenticación de usuarios.

Ya tenemos una aplicación operativa, pero aún nos quedan tareas que completar antes de considerarla completa. Uno de los aspectos más importantes en cualquier aplicación es el referente a la seguridad. En estos momentos cualquiera puede acceder a nuestra aplicación y utilizarla sin problema, pero supongamos que sólo un usuario autorizado tiene permiso para hacerlo.

En esta parte vamos a implementar un sistema de autenticación y autorización, de modo que sea necesario estar logueado como usuario para tener acceso a nuestra aplicación.

Autenticación y autorización.

Se denomina autenticación a un proceso por el cual un usuario proporciona unas credenciales, que son comparadas con las que están almacenadas en nuestra propia aplicación, una base de datos o un proveedor de autenticación externo. Si estas credenciales coinciden con alguna de las que están almacenadas, el usuario es autenticado con éxito, y podrá realizar las acciones para las que esté autorizado.
La autorización se refiere al proceso que determina, para un usuario autenticado, qué acciones está autorizado para llevar a cabo.

Identity.

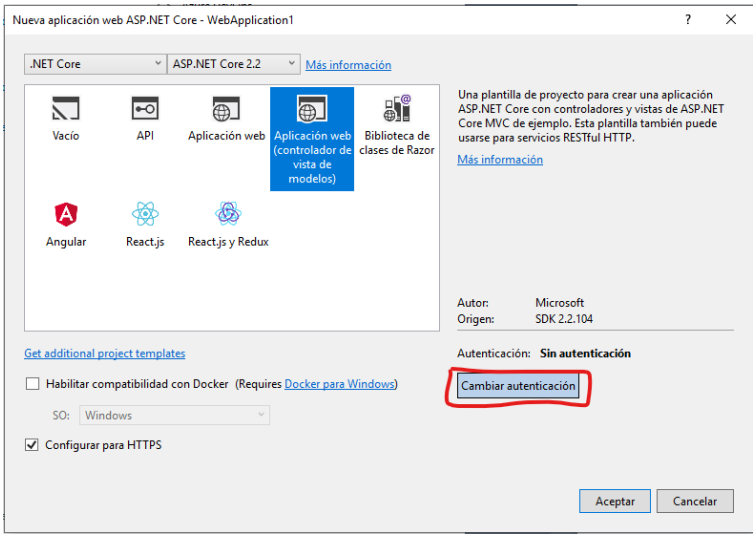
ASP.NET Core incorpora un sistema que añade funcionalidad de login para las aplicaciones web creadas con este framework. Este sistema se denomina ASP.NET Core Identity, y permite utilizar cuentas de usuario almacenadas en una BD o utilizar proveedores de login externos, como Facebook, Google, Microsoft Account o Twitter.
La forma más sencilla de utilizar Identity es utilizando una base de datos SQL Server para almacenar los datos de usuarios y roles. Este será el método que utilicemos en nuestro ejemplo.

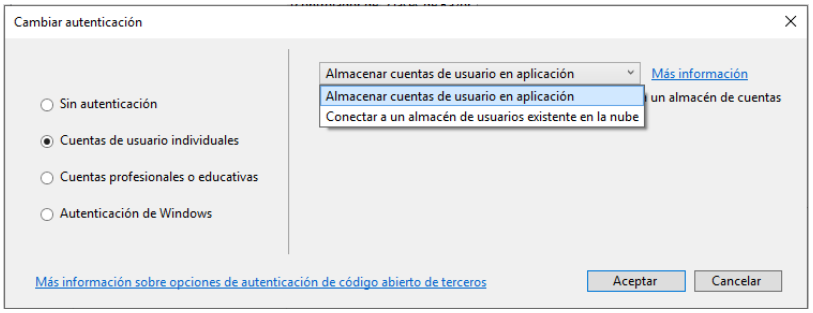
Para consultar más información sobre ASP.NET Core Identity se puede consultar la documentación oficial, en el siguiente enlace:

https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.2&tabs=visual-studio

Añadir soporte para Identity en nuestro proyecto.

La manera más sencilla de utilizar Identity en una aplicación ASP.NET Core es crear una aplicación web y en el asistente, cambiar el método de autenticación. Como tipo de autenticación seleccionaremos cuentas individuales, y en el desplegable podremos escoger si estas cuentas son almacenadas en la aplicación, usando de este modo una BD, o si utilizamos un almacén de datos en la nube.





En nuestro caso, como nuestro proyecto ya está creado, vamos a realizar manualmente los pasos necesarios para añadir el soporte para Identity en nuestra aplicación.

1. En primer lugar necesitamos definir un contexto de datos para la BD que contendrá la información de nuestros usuarios. Este contexto de datos debe ser una clase que herede de *IdentityDbContext*.

   Como en nuestra aplicación ya estamos utilizando una BD y tenemos definido un contexto de datos para EF Core, en lugar de uno nuevo, lo que queremos es utilizar esta misma BD para guardar la información relativa a los usuarios.
   Para ello modificamos el código de la clase *CIFPACarballeiraContext*, para que herede de IdentityDbContext, en lugar de DbContext.

   ```
   public class CIFPACarballeiraContext : IdentityDbContext<IdentityUser>
   ```

   En nuestro código indicamos que nuestra base de datos se utilizará como almacén de datos para Identity, y que los usuarios serán del tipo por defecto, *IdentityUser*. Podemos personalizar los datos que se almacenarán en nuestra BD creando una clase que herede de *IdentityUser* y definiendo esta clase al crear el contexto de datos. En nuestro ejemplo, por simplicidad utilizaremos el usuario por defecto. En la documentación oficial se puede encontrar información acerca de cómo hacer esta personalización.

2. Si ejecutamos la aplicación en este momento nos encontraremos con un error, esto es debido a que al definir que nuestro contexto de datos utilizará Identity, hemos cambiado el esquema de

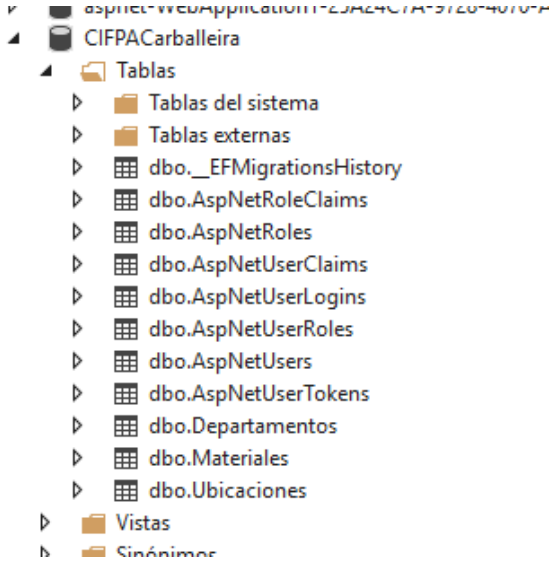
datos de la BD, que ahora debe incluir las tablas para guardar la información relativa a los usuarios. Por esta razón, lo primero que tenemos que hacer es crear una nueva migración, que permita sincronizar la BD con nuestro nuevo esquema.

```
Add-Migration AddIdentity
```

Una vez creada la migración, actualizamos la BD.

```
Update-Database
```

Si observamos ahora la BD, podremos ver cómo ahora contiene una serie de tablas, que se utilizarán para los datos de usuarios y roles.



Si observamos ahora la BD, podremos ver cómo ahora contiene una serie de tablas, que se utilizarán para los datos de usuarios y roles. Si queremos personalizar los nombres de las tablas, podemos especificarlos en la clase que define el contexto de datos, utilizando FluentAPI, como hicimos con las tablas de Ubicaciones y materiales.

Añadir y configurar Identity en los servicios de la aplicación.

Una vez hemos definido el contexto para utilizar Identity, creado las migraciones y actualizado la BD, ésta ya está lista para almacenar la información requerida, pero todavía no hemos configurado nuestra aplicación para trabajar con Identity.

Para ello, añadiremos el siguiente código en el método *ConfigureServices* de la clase *Startup*.

1. Empezamos por añadir Identity a la lista de servicios de la aplicación, justo después de añadir el contexto de datos:

```
services.AddDbContext<CIFPACarballeiraContext>(options => options.UseSqlServer(
    Configuration.GetConnectionString("CIFPACarballeiraContext")));

services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<CIFPACarballeiraContext>();
```

Hemos añadido soporte para usuarios y roles por defecto, y especificamos que los datos de Identity se almacenarán en una BD a la que accederemos a través de EF, utilizando el contexto de datos de la aplicación.

2. Después de añadir Identity como servicio de la aplicación, debemos configurar diversos parámetros del mismo. Para esto, a continuación añadimos el siguiente código:

```
services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = false;
    options.Password.RequireLowercase = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = false;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});
```

Como se puede observar, especificamos una serie de opciones que definirán diversas políticas referentes a los nombres de usuario, requisitos de contraseña, o tiempo de la sesión.

Datos iniciales.

1. Cuando desplegamos una aplicación web y generamos la BD, es habitual que queramos que esta se inicie con una serie de datos iniciales. En este caso, al utilizar Identity, necesitaremos loguearnos con un usuario para utilizar la aplicación. Por esta razón, es importante asegurar que haya un usuario en la BD para podernos loguear, más concretamente, nuestro usuario administrador. Lo que haremos a continuación es definir una clase que se encargue de inicializar estos datos. Para ello creamos una clase en nuestra carpeta *Data*, que llamaremos *InitializeData*:

```
using Microsoft.AspNetCore.Identity;

namespace CIFPACarballeiraWeb.Data
{
    public static class InitializeData
    {
        public static void SeedData(UserManager<IdentityUser> userManager,
            RoleManager<IdentityRole> roleManager)
        {
            SeedRoles(roleManager);
            SeedUsers(userManager);
        }

        public static void SeedRoles(RoleManager<IdentityRole> roleManager)
        {
            if (!roleManager.RoleExistsAsync("User").Result)
            {
```

```
                IdentityRole role = new IdentityRole();
                role.Name = "User";
                IdentityResult roleResult = roleManager.CreateAsync(role).Result;
            }

            if (!roleManager.RoleExistsAsync("Admin").Result)
            {
                IdentityRole role = new IdentityRole();
                role.Name = "Admin";
                IdentityResult roleResult = roleManager.CreateAsync(role).Result;
            }
        }

        public static void SeedUsers(UserManager<IdentityUser> userManager)
        {
            if (userManager.FindByNameAsync("admin").Result == null)
            {
                IdentityUser user = new IdentityUser
                {
                    UserName = "admin",
                    Email = "cifpcarballeira@edu.xunta.es"
                };

                IdentityResult result =
                    userManager.CreateAsync(user, "admin123").Result;

                if (result.Succeeded)
                {
                    userManager.AddToRoleAsync(user, "Admin").Wait();
                }
            }
        }
    }
}
```

En esta clase comprobamos si existen los roles de usuario Admin y User, y en caso contrario los creamos. Cada usuario pertenecerá a un rol, que definirá las acciones a las que tiene acceso. A continuación comprobará si existe un usuario admin, y de nuevo lo crea en caso contrario. Nuestro usuario pertenecerá al rol Admin, pudiendo de este modo crear nuevos usuarios para la aplicación, mientras que el resto de usuarios pertenecerán al rol User.

Podemos ver que para trabajar con los roles y usuarios utilizaremos los objetos *UserManager* y *RoleManager*.

2. Ahora sólo nos queda modificar el método Configure de Startup para que llame a *InitializeData*, asegurando de esta forma que cuando arranquemos nuestra aplicación, el usuario admin se habrá creado en la BD:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    UserManager<IdentityUser> userManager, RoleManager<IdentityRole> roleManager)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        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.UseCookiePolicy();

        app.UseAuthentication();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
            });

            InitializeData.SeedData(userManager, roleManager);
        }
    }
```

Autorización.

Ahora que hemos configurado lo necesario para almacenar la información de los usuarios para que estos puedan autenticarse, pasamos a la segunda parte, en la que definiremos a cuáles de las acciones de nuestra aplicación puede tener acceso cada usuario.

Para ello, teniendo en cuenta que los controladores son los que reciben las solicitudes de los usuarios, realizando las tareas requeridas y seleccionando las vistas que mostrarán el resultado, será este el lugar donde habrá que especificar los permisos de acceso.

Para especificar que una parte determinada de nuestra aplicación sólo está permitida para un usuario autenticado, el mecanismo es muy sencillo, y consiste simplemente en añadir el atributo *[Authorize]*, bien a un método de acción concreto o a todo un controlador.

En nuestra aplicación, la única parte a la que se puede acceder de forma anónima será a la página inicial de la aplicación, por lo tanto el *HomeController* lo dejaremos tal cual está.
Sin embargo, ningún usuario puede acceder a la información del inventario, ni tampoco a los departamentos o las ubicaciones. De este modo, queda claro que no queremos permitir acceso a ninguna acción en estos controladores, por lo que el modo más sencillo de impedir el acceso a un usuario anónimo será añadir el atributo a cada uno de los controladores, tal y como se indica:

```
[Authorize]
public class DepartamentosController : Controller

[Authorize]
public class InventarioController : Controller

[Authorize]
public class UbicacionesController : Controller
```

Si probamos ahora a ejecutar nuestra aplicación, se mostrará sin ningún problema la página principal, pero en el momento de intentar acceder a cualquier parte de nuestra aplicación, veremos como en la pantalla no se muestra nada.

Esto indica dos cosas. Primero, que efectivamente hay una restricción que nos impide acceder a una parte de la aplicación de forma anónima. Y segundo, que necesitamos un mecanismo para poder loguearnos y así tener acceso al resto de la aplicación.

Login.

Vamos a ver ahora como añadir funcionalidad para poder loguearnos y acceder de este modo a las partes de la aplicación que requieren autorización.

1. Comenzamos creando un nuevo controlador que será el encargado de manejar las cuestiones referentes a las cuentas de usuario, tales como login, logoff, crear nuevos usuarios, etc..

```
[Authorize]
public class AccountController : Controller
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly SignInManager<IdentityUser> _signInManager;

    public AccountController(UserManager<IdentityUser> userManager,
        SignInManager<IdentityUser> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    //
    // GET: /Account/Login
    [HttpGet]
    [AllowAnonymous]
    public async Task<IActionResult> Login(string returnUrl = null)
    {
        // Clear the existing external cookie to ensure a clean login process
        await AuthenticationHttpContextExtensions.SignOutAsync(
            HttpContext, IdentityConstants.ExternalScheme);

        ViewData["ReturnUrl"] = returnUrl;
        return View();
     }
 }
```

   En este controlador almacenamos dos objetos que nos van a hacer falta en diversas ocasiones, que son el *UserManager* y el *SignInManager*. Además, en este controlador nos encontramos con que la mayoría de las acciones que implementemos serán sólo para usuarios autenticados, y por eso añadimos el atributo *[Authorize]*. Sin embargo, debemos permitir que la acción Login pueda ser utilizada por usuarios anónimos, por eso incluimos el atributo *[AllowAnonymous]* para indicar este requerimiento.

2. Antes de crear la vista correspondiente para el login de un usuario, necesitamos un objeto en nuestro modelo de datos que respalde los datos de este formulario. El objeto IdentityUser es un objeto con una gran cantidad de propiedades, mientras que nuestro formulario de login sólo requiere un nombre de usuario y una contraseña. Por esta razón, es mucho más cómodo crear un *ViewModel* específico para esta vista.
   Creamos una carpeta *ViewModels*, y en su interior una clase *LoginViewModel*.

```
using System.ComponentModel.DataAnnotations;
```

```
namespace CIFPACarballeiraWeb.ViewModels
{
    public class LoginViewModel
    {
        [Required]
        [Display(Name = "Usuario")]
        public string User { get; set; }

        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Contraseña")]
        public string Password { get; set; }

       [Display(Name = "¿Recordar cuenta?")]
        public bool RememberMe { get; set; }
    }
 }
```

3. Por último, creamos nuestra vista para el Login. La forma más rápida es hacer clic derecho en el método de acción Login y seleccionar **Agregar vista…**

```
@model LoginViewModel

@{
    ViewBag.Title = "Iniciar sesión";
}

<div class="col-md-6 offset-3">
    <h2>@ViewBag.Title</h2>
    <br />
    <section id="loginForm">
        <form asp-controller="Account" asp-action="Login" asp-antiforgery="true"
              asp-route-returnurl="@ViewData["ReturnUrl"]"
              method="post" class="form-horizontal">
            <h4>Utilice una cuenta local para iniciar sesión.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="User"></label>
                <input asp-for="User" class="form-control" />
                <span asp-validation-for="User" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <div class="checkbox">
                    <label asp-for="RememberMe">
                        <input asp-for="RememberMe" />
                        @Html.DisplayNameFor(m => m.RememberMe)
                    </label>
                </div>
            </div>
            <div class="form-group">
                <button type="submit" class="btn btn-primary">
                    <i class="fas fa-sign-in-alt"></i> Iniciar sesión</button>
            </div>
        </form>
    </section>
</div>
```

```
@section Scripts {
    @{ await Html.RenderPartialAsync("_ValidationScriptsPartial"); }
}
```

4. Ahora ya sólo nos queda el último paso, añadir un método Login que recoja la petición post cuando obteniendo nuestro nombre y contraseña.

```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model,
    string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set
           lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(
            model.User, model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            return RedirectToLocal(returnUrl);
        }
        else
        {
            ModelState.AddModelError(string.Empty,
                "Nombre de usuario o contraseña no válidos.");
            return View(model);
        }
    }

    return View(model);
}
```

En este método recogemos el usuario y pass del formulario, e intentamos loguearnos utilizando el SignInManager. Si el resultado es correcto, hemos introducido los datos correctamente y pasaremos a estar autenticados. Como hemos almacenado la dirección desde la que se llamó a la vista de Login, iremos directamente a esa vista, una vez que ya tenemos permiso.

Si ejecutamos nuestra aplicación veremos como se muestra la página principal. Si intentamos por ejemplo acceder al inventario, se mostrará en su lugar la página de login. Si introducimos los datos del administrador que sabemos que están en la BD, comprobaremos como automáticamente se abre la página de inventario que habíamos solicitado, una vez que ya estamos autenticados y tenemos por lo tanto acceso a la misma.

Vista parcial de usuario en la cabecera.

Aunque la aplicación ya nos permite autenticarnos, puede ser confuso para el usuario saber si está iniciada la sesión o no. Por esta razón se hace necesario algún tipo de mecanismo que permita al usuario saber en todo momento quién es el usuario actual, y darle la opción de hacer login / logout. Para esto utilizaremos una vista parcial que incluiremos en la cabecera, tal y como suele ser habitual

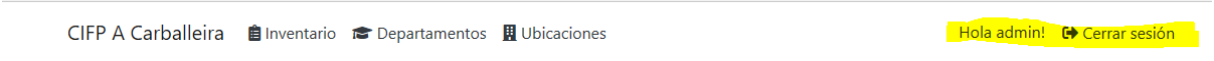
en la mayoría de aplicaciones web. Para eso creamos una vista *_LoginPartial.cshtml* en Shared, o la editamos en caso de que ya la tengamos, para que contenga el siguiente código:

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User))
    {
        <li class="nav-item">
            <a id="manage" class="nav-link text-dark" asp-controller="Account"
                asp-action="ChangePassword" title="Cuenta">
                Hola @UserManager.GetUserName(User)!</a>
        </li>
        <li class="nav-item">
            <form id="logoutForm" class="form-inline" asp-controller="Account"
                asp-action="Logout" asp-route-returnUrl="@Url.Action("Index",
                    "Home", new { area = "" })">
                <button id="logout" type="submit"
                    class="nav-link btn btn-link text-dark">
                    <i class="fas fa-sign-out-alt"></i> Cerrar sesión</button>
            </form>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" id="login" asp-controller="Account"
                asp-action="Login">
                    <i class="fas fa-sign-in-alt"></i> Iniciar sesión</a>
        </li>
    }
</ul>
```

En esta vista parcial utilizamos la directiva @inject para permitir a la vista acceder al UserManager y al SignInManager de Identity. Gracias a esto podemos comprobar si un usuario se ha autenticado en nuestra aplicación. Si esto es así se mostrará un saludo con el nombre de dicho usuario, que será un enlace a una página que le permita cambiar la contraseña, además de una opción para hacer logout.

Si no se detecta ningún usuario, en su lugar se mostrará un enlace a la página de login. De este modo en todo momento podremos saber si hay un usuario activo y de quién se trata, además de cerrar nuestra sesión o iniciar una nueva.



Logout.

Ya que tenemos en nuestra vista parcial un enlace que enlaza con la acción logout, sólo tenemos que implementar dicha acción, así que creamos el método correspondiente en el controlador *Account*:

```
//
// POST: /Account/Logout
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Logout()
```

```
{
    await _signInManager.SignOutAsync();
    return RedirectToAction(nameof(HomeController.Index), "Home");
}
```

Este método es muy sencillo, ya que lo único que hace es cerrar la sesión actual llamando al método adecuado del SignInManager, y nos devuelve a la vista principal, que es la única que se puede ver sin estar autenticado.

Cambio de contraseña.

En este apartado vamos a añadir una vista que permita a un usuario autenticado cambiar su contraseña. En la vista parcial de la cabecera ya tenemos un enlace que apunta a la acción ChangePassword, así que implementaremos dicha acción en el controlador.

1. Comenzamos creando un nuevo ViewModel para esta acción. Este ViewModel almacenará la contraseña antigua, y la nueva dos veces, ya que se requerirá que la nueva contraseña que introduzcamos sea la misma, para evitar introducir una contraseña errónea que el usuario luego no pueda recordar. Llamaremos a esta clase *ChangePasswordViewModel*.

```
using System.ComponentModel.DataAnnotations;

namespace CIFPACarballeiraWeb.ViewModels
{
    public class ChangePasswordViewModel
    {
        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Contraseña antigua")]
        public string OldPassword { get; set; }

        [Required]
        [StringLength(100,
            ErrorMessage = "La contraseña debe de tener 6 caracteres como mínimo.",
            MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Nueva contraseña")]
        public string NewPassword { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirmar contraseña")]
        [Compare("NewPassword",
            ErrorMessage = "Las dos contraseñas introducidas no coinciden.")]
        public string ConfirmPassword { get; set; }
    }
}
```

2. Ahora crearemos la vista, que lógicamente se llamará *ChangePassword.cshtml* y estará situada en la carpeta *Account*.

```
@model ChangePasswordViewModel

@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

@{
```

```
    ViewBag.Title = "Cambiar contraseña";
}

<div class="col-md-6 offset-3">
    <h2>@ViewBag.Title.</h2>

    <p class="text-success">@ViewBag.StatusMessage</p>

    <p>Ha iniciado sesión como <strong>@UserManager.GetUserName(User)</strong>.</p>

    <form asp-controller="Account" asp-action="ChangePassword"
        asp-antiforgery="true" asp-route-returnurl="@ViewData["ReturnUrl"]"
        method="post" class="form-horizontal">
        <hr />
        <div class="form-group">
            <label asp-for="OldPassword"></label>
            <input asp-for="OldPassword" class="form-control" />
            <span asp-validation-for="OldPassword" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="NewPassword"></label>
            <input asp-for="NewPassword" class="form-control" />
            <span asp-validation-for="NewPassword" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="ConfirmPassword"></label>
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>

        <div class="form-group">
            <button type="submit" class="btn btn-primary">
                <i class="fas fa-sign-in-alt"></i> Cambiar contraseña</button>
        </div>
    </form>
</div>

@section Scripts {
    @{ await Html.RenderPartialAsync("_ValidationScriptsPartial"); }
}
```

3. Por último, creamos los métodos de acción para las peticiones get y post:

```
//
// GET: /Account/ChangePassword
[HttpGet]
public IActionResult ChangePassword()
{
    return View();
}


//
// POST: /Account/ChangePassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ChangePassword(ChangePasswordViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
```

```
        var user = await GetCurrentUserAsync();
        if (user != null)
        {
            var result = await _userManager.ChangePasswordAsync(user,
                model.OldPassword, model.NewPassword);
            if (result.Succeeded)
            {
                await _signInManager.SignInAsync(user, isPersistent: false);
                return RedirectToAction("Index", nameof(HomeController));
            }
            ModelState.AddModelError("OldPassword",
                "La contraseña introducida no es correcta");
            return View(model);
         }

         return RedirectToAction("Index", nameof(HomeController));
    }
```

De nuevo podemos observar cómo Identity nos proporciona los métodos necesarios para trabajar con nuestros usuarios en los objetos *UserManager* y *SignInManager*. En este ejemplo obtenemos la nueva contraseña, y si es la misma en los dos controles de entrada, intentamos cambiarla. Si se produce un error es porque la contraseña antigua no es correcta, por lo que añadimos ese error al modelo y volvemos al formulario. Si la contraseña se cambia con éxito lo que haremos es ir a la página principal de la aplicación, una vez que nos hemos logueado de nuevo con el mismo usuario, pero con la nueva contraseña.

Gestión de usuarios.

Cualquier aplicación web que requiera la autenticación de sus usuarios necesita, evidentemente, un mecanismo para dar de alta dichos usuarios. Existen multitud de opciones para gestionar esto, desde permitir el registro automático de cualquier usuario que lo desee, a registrarse utilizando servicios de terceros, que se gestionen solicitudes que luego ha de validad el usuario, etc.. En esta aplicación estamos intentando simplificar al máximo la gestión de autenticación y autorización, pero si se quiere ampliar la información se puede recurrir a la documentación oficial de ASP.NET Core.

Siguiendo con esa idea, vamos a implementar un sencillo mecanismo, que permita únicamente al administrador la creación de nuevos usuarios, cuyos datos tendría que comunicar luego a cada uno de estos usuarios, que una vez logueados ya podrían acceder a la aplicación y cambiar su contraseña si así lo desean.

1. Comenzamos editando nuestra plantilla de la aplicación o Layout. Lo que queremos es añadir una nueva opción junto al resto, en este caso para acceder a la gestión de usuarios. La diferencia es que en este caso, esta opción debe aparecer únicamente en caso de que el usuario actual sea un administrador. Para esto lo primero que necesitaremos es inyectar el objeto *SignInManager* en nuestra vista:

   ```
   @using Microsoft.AspNetCore.Identity
   @inject SignInManager<IdentityUser> SignInManager
   ```

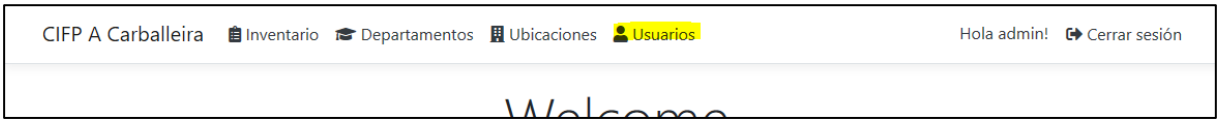
   Ahora utilizamos este objeto para comprobar si hay un usuario autenticado y si este pertenece al rol “Admin”. Si esto es así se mostrará una opción para acceder a la gestión de usuarios. Para esto añadimos el siguiente código a continuación del elemento Ubicaciones de la cabecera:

   ```
   @if (SignInManager.IsSignedIn(User) && User.IsInRole("Admin"))
   ```

```
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Account"
            asp-action="Users"><i class="fas fa-user"></i> Usuarios</a>
    </li>
}
```

Así si no estamos autenticados, o nuestro usuario no es un administrador, no se mostrará el enlace.

Pero si estará visible cuando el usuario actual es el administrador.



2. Ahora necesitamos un método de acción que reciba la petición para la página de usuarios en el *AccountController*.

```
// GET: /Manage/Users
public ActionResult Users()
{
    return View(_userManager.Users.ToList());
}
```

El método es muy sencillo, ya que lo único que hace es obtener todos nuestros usuarios gracias al *UserManager* y pasarlos a la vista correspondiente.

3. Y a continuación la vista correspondiente, *Users.cshtml*.

```
@using Microsoft.AspNetCore.Identity
@model IEnumerable<IdentityUser>

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

<h2>Usuarios</h2>

<p>
    <a asp-action="NewUser"><i class="fa fa-user-plus"></i> Nuevo usuario</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                Usuario
            </th>
            <th></th>
        </tr>
    </thead>
```

```
    <tbody>
        @foreach (var item in Model)
        {
            @if (item.UserName != "admin")
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem => item.UserName)
                    </td>
                    <td>
                        <a asp-action="Delete" asp-route-id="@item.Id"
title="Eliminar"><i class="fa fa-user-times"></i></a>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

En esta vista, simplemente mostramos todos los usuarios disponibles excepto el usuario admin, que no será en ningún caso eliminable. También tenemos la opción de crear un nuevo usuario.

Eliminación de usuarios.

En la vista de usuarios hemos añadido un enlace a una opción que permite eliminar un usuario, así que ahora debemos implementar las acciones que lo manejen, y la vista correspondiente. Como en el resto de la aplicación tendremos dos métodos, uno que atienda la petición get, y que muestre los datos del usuario que queremos eliminar, que obtenemos por su id, y otra que atiende la petición de confirmación, cuando confirmamos que queremos eliminar el usuario.

1. A continuación se muestra el código de los dos métodos, del *AccountController*:

```
// GET: Personal/Delete/5
public async Task<ActionResult> Delete(string id, bool? saveChangesError = false)
{
    if (string.IsNullOrEmpty(id))
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewBag.ErrorMessage = "Se ha producido algún error al eliminar el usuario";
    }

        IdentityUser user = await _userManager.FindByIdAsync(id);
        if (user == null)
        {
            return NotFound();
        }
        return View(user);
    }

    // POST: Personal/Delete/5
    [HttpPost, ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> DeleteConfirmed(string id)
    {
        try
```

```
            {
                IdentityUser user = await _userManager.FindByIdAsync(id);
                await _userManager.DeleteAsync(user);
            }
            catch (RetryLimitExceededException dex)
            {
                 //Log the error (uncomment dex variable name and
                 add a line here to write a log.
                 return RedirectToAction("Delete",
                     new { id = id, saveChangesError = true });
            }

            return RedirectToAction("Users");
        }
    }
```

2. Ahora sólo nos queda la vista, que será bastante sencilla y similar a otras vistas de eliminación de la aplicación:

```
@using Microsoft.AspNetCore.Identity
@model IdentityUser

@{
    ViewBag.Title = "Eliminar usuario";
}

<h3>Eliminar usuario</h3>
<p class="error">@ViewBag.ErrorMessage</p>

<h4>Estás seguro de eliminar el usuario?</h4>
<div>
    <hr />
    <dl class="row">
        <dt class="col-sm-1">
            Usuario
        </dt>

        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.UserName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="Id" />
        <input type="submit" value="Eliminar" class="btn btn-danger" /> |
        <a asp-action="Users"><i class="fa fa-backward"></i> Volver</a>
    </form>
</div>
```

Creación de usuarios.

Estamos ya apunto de terminar con esta parte. Lo único que nos queda ya es añadir una acción y su vista correspondiente, que permitan al administrador crear un nuevo usuario.

1. Lo primero será crear un NewUserViewModel que almacene los datos del formulario de creación de usuarios. La información contenida será simplemente el nombre de usuario, y la contraseña, repetida de nuevo para evitar errores en la introducción de la misma.

```
using System.ComponentModel.DataAnnotations;

namespace CIFPACarballeiraWeb.ViewModels
{
    public class NewUserViewModel
    {
        [Required]
        [Display(Name = "Usuario")]
        public string User { get; set; }

        [Required]
        [StringLength(100,
            ErrorMessage = "La contraseña debe de tener 6 caracteres como mínimo.",
            MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Contraseña")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirmar contraseña")]
        [Compare("Password",
            ErrorMessage = "Las dos contraseñas introducidas no coinciden.")]
        public string ConfirmPassword { get; set; }
    }
}
```

2. A continuación creamos los métodos de acción:

```
@using Microsoft.AspNetCore.Identity
//
// GET: /Account/NewUser
[HttpGet]
public ActionResult NewUser()
{
    return View();
}

//
// POST: /Account/NewUser
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> NewUser(NewUserViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = model.User };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await _userManager.AddToRoleAsync(user, "User");

            return RedirectToAction("Users");
        }
         ModelState.AddModelError("User", "Ya existe un usuario con ese nombre");
     }

     return View(model);
}
```

Como se puede observar, el método se limita a crear un nuevo usuario con los datos recogidos del formulario e intenta crearlo utilizando el *UserManager*. Si se produce un error será debido a que ya tenemos un usuario con el mismo nombre.

3. Y finalmente, añadimos la vista correspondiente para completar así esta parte. de nuevo se trata de un formulario sencillo, que contiene únicamente tres campos:

```
@model NewUserViewModel
@{
    ViewData["Title"] = "Registrar usuario";
}

<div class="col-md-6 offset-3">
    <h2>@ViewBag.Title</h2>
    <br />
    <section id="loginForm">
        <form asp-controller="Account" asp-action="NewUser" asp-antiforgery="true"
              asp-route-returnurl="@ViewData["ReturnUrl"]"
              method="post" class="form-horizontal">
            <h4>Crea un nuevo usuario.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="User"></label>
                <input asp-for="User" class="form-control" />
                <span asp-validation-for="User" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <button type="submit" class="btn btn-primary">
                    <i class="fas fa-user-plus"></i> Crear usuario</button>
            </div>
        </form>
    </section>
</div>

@section Scripts {
    @{ await Html.RenderPartialAsync("_ValidationScriptsPartial"); }
}
```

Y con esto habremos terminado con el mecanismo de autenticación y autorización, además de la gestión de usuarios de nuestra aplicación.