educación IT
CODE YOUR FUTURE

Programación Web .NET Core

Módulo 10

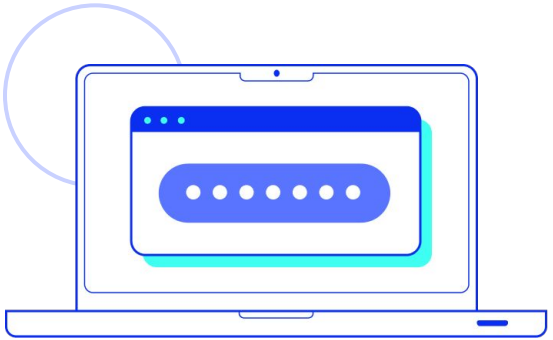
educación IT
CODE YOUR FUTURE

Introducción a la Autenticación en ASP.NET Core

Gestionar la seguridad

Cuando se crea una aplicación, es necesario **reconocer al usuario** para permitirle interactuar y **darle acceso** a la información que le resulte relevante. Para **verificar sus credenciales** y proporcionar acceso a la aplicación se utiliza la ***autenticación***. La autenticación es el proceso para determinar la identidad de un usuario dentro de la aplicación.

Este usuario proporciona un *nombre de usuario* único junto con una *contraseña*.



Utilidad de la autenticación

Identificar al usuario puede ser particularmente útil en plataformas para facilitar interacciones tales como el acceso a las redes sociales, chats, servicios de pago, sitios de compras y otros que brindan servicios personalizados. Además, la autenticación también se puede utilizar para adquirir datos personales de usuarios de un sitio web, y luego utilizarlos para brindar servicios adicionales.

Métodos de autenticación

Un tipo común de autenticación es mediante **nombre de usuario y contraseña**. El usuario deberá conocer tanto el nombre de usuario único que utiliza en la aplicación, como la contraseña relacionada. Sin embargo, esta no es la única opción posible para la autenticación y existen métodos adicionales como **huellas dactilares, confirmación por teléfono o por correo electrónico,** etc. Crear un entorno de autenticación adecuado es crucial, ya que puede agregar una capa importante de seguridad a sus aplicaciones. Las aplicaciones modernas a menudo involucran datos confidenciales. Sin autenticación, los datos son accesibles para todos los usuarios, lo que permite a los usuarios adquirir información sensible.

Autenticación basada en tokens

La mayoría de las formas de autenticación en estos días utilizan lo que se conoce como autenticación basada en ***tokens***. Se realiza la autenticación inicial, el servidor genera un token y lo envía al cliente. El cliente luego envía el token nuevamente en cada solicitud futura, lo que permite al servidor confirmar que es el mismo usuario. Si el token es incorrecto o vencido, el usuario deberá autenticarse nuevamente.

ASP.NET Core Identity

Configuración de la identidad de ASP.NET Core

ASP.NET Core Identity es una solución para manejar la **funcionalidad de inicio de sesión dentro de aplicaciones ASP.NET Core**. Trabaja con **múltiples proveedores** para ayudar a los desarrolladores a crear las infraestructuras de autenticación necesarias.

Como parte de ASP.NET Core Identity, la infraestructura Entity Framework se utiliza para gestionar la configuración e interactuar con los datos del usuario. En casi todos los casos, esto significa que se puede utilizar la misma base de datos que usas en otras partes de tu aplicación, de manera independiente del tipo para manejar el inicio y cierre de sesión. De esta manera, se anula el requisito de configurar una infraestructura adicional.

ASP.NET Core Identity también expone **varios servicios**, que puedes inyectar en toda su aplicación, para administrar los usuarios del sistema y realizar operaciones de inicio y cierre de sesión.

Configurar la clase de usuario

Para configurar ASP.NET Core Identity, primero deberás decidir **qué propiedades sobre los usuarios de tu aplicación deseas almacenar**. De forma predeterminada, propiedades como la dirección de correo electrónico, el nombre de usuario y el hash de contraseña se almacenan como parte de la clase predeterminada llamada **`IdentityUser`** que se expone desde el espacio de nombres **`Microsoft.AspNetCore.Identity`**. Si necesitas **alguna propiedad adicional**, como el nombre y apellido del usuario, la edad o cualquier otra propiedad, deberás crear tu propia clase que hereda de la clase `IdentityUser`. Si no necesitas cualquier propiedad adicional, puedes usar la clase `IdentityUser` directamente en su lugar.

El siguiente código es un ejemplo de herencia de la clase predeterminada **`IdentityUser`**:

```
public class WebsiteUser : IdentityUser
{
public string UserHandle { get; set; }
}
```

En este ejemplo, puedes observar la clase **`WebsiteUser`**, que expone la propiedad llamada **`UserHandle`**. Esta propiedad podría usarse para determinar cómo se muestra el nombre del usuario durante las operaciones en línea, sin revelar el nombre de usuario utilizado para iniciar sesión.

Práctica recomendada: Ten en cuenta que solo se debe almacenar un hash de contraseña para ASP.NET Core.

Configurar el contexto de la base de datos

En el siguiente paso, deberás configurar la clase `DbContext` para admitir la identidad. Para hacer esto, la clase de contexto deberá heredar de la clase **`IdentityDbContext` en lugar de `DbContext`**. También debes proporcionar la clase de identidad de usuario elegida como un tipo genérico para la clase `IdentityDbContext`. Deberás proporcionar la clase que deseas usar; de lo contrario, no podrás acceder a las propiedades. Una ventaja al usar `IdentityDbContext` es que **no necesitarás agregar una instancia de `DbSet`** para administrar las colecciones de usuarios. La lógica para administrar usuarios está integrada como parte del propio `IdentityDbContext`.

El siguiente código es un ejemplo de cómo configurar un **IdentityDbContext**:

```
public class AuthenticationContext : IdentityDbContext<WebsiteUser>
{
publicAuthenticationContext(DbContextOptions<AuthenticationContext>options) :
base(options)
{
}
}
```

En este ejemplo, puedes ver que la clase denominada **IdentityDbContext** y proporciona la clase de usuario personalizada **WebsiteUser** en lugar del **IdentityUser** predeterminado. Por lo demás es idéntico en estructura a **DbContext**.

> **Práctica recomendada:** puedes utilizar el llamado **IdentityDbContext** para administrar conjuntos de datos adicionales, de la misma forma que lo hace con **DbContext**. Se debe tener en cuenta que el uso de **IdentityDbContext** no está exclusivamente destinado a la autenticación.

Configurar la identidad en la clase de inicio

Una vez que se ha establecido el contexto de la base de datos, deberás agregar llamadas a servicios **ASP.NET Core Identity** y **middleware**. Dentro del método **`ConfigureServices`**, en la clase Startup, deberás llamar a **`AddDbContext`** en el parámetro **`IServiceCollection`** con una clase que hereda de **`IdentityDbContext`**. Se debe instanciar la clase como cualquier otro **`DbContext`**, con los mismos parámetros. También deberás agregar una llamada al método llamado **`AddDefaultIdentity<*User*>()`**.

El método **`AddDefaultIdentity`** espera la clase que se utilizará para administrar usuarios en la aplicación. Puede ser una clase personalizada que hereda de **`IdentityUser`** o el propio **`IdentityUser`**.

También debes canalizar una llamada a **`AddEntityFrameworkStores<*DBContext*>()`** inmediatamente después **`AddIdentity`**. Esto se usa para conectar **`IdentityDbContext`** a **`IdentityServices`** agregado por la llamada a **`AddIdentity`**.

El siguiente es un ejemplo de **ConfigureServices** con el agregado servicios de identidad:

```
public void ConfigureServices(IServiceCollectionservices)
{
services.AddMvc();
services.AddDbContext<AuthenticationContext>(options =>
options.UseSqlite("Data Source=user.db"));
services.AddDefaultIdentity<WebsiteUser>()
.AddEntityFrameworkStores<AuthenticationContext>();
}
```

En el ejemplo de la pantalla anterior, puedes observar que los servicios para **ASP.NET Core Identity** son instanciados por la llamada al método **`AddDefaultIdentity`**. Esta aplicación utiliza la clase **`WebsiteUser`** para administrar usuarios. Entonces conecta los servicios de identidad al **`IdentityDbContext`** proporcionado por **`AuthenticationContext`**.

Finalmente, en el método **`Configure`** de la clase de startup, necesitarás cargar el middleware para manejar el proceso de autenticación para habilitar la identidad. Puedes hacer esto llamando al método **`UseAuthentication`** del objeto **`IApplicationBuilder`**. Es importante que llames a **`UseAuthentication`** antes de llamar a **`UseMvc`**. Esto es para garantizar que la autenticación siempre esté lista para su uso dentro de nuestro ASP.NET Controladores Core MVC.

El siguiente código es un ejemplo de cómo habilitar el **middleware de identidad**:

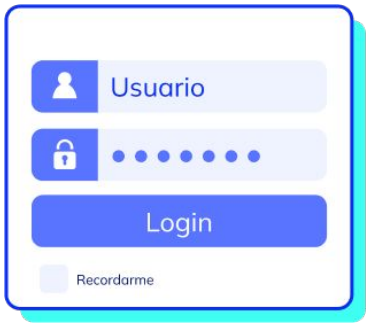
```
public void Configure(IApplicationBuilder app, AuthenticationContextauthContext)
{
authContext.Database.EnsureDeleted();
authContext.Database.EnsureCreated();
app.UseAuthentication();
app.UseMvcWithDefaultRoute();
app.Run(async (context) =>
{
awaitcontext.Response.WriteAsync("");
});
}
```

Interfaz con ASP.NET Core Identity

Una vez que se ha configurado la autenticación dentro de la aplicación **ASP.NET Core MVC**, se puede comenzar a usar.

Realización de inicio de sesión

Uno de los flujos más comunes en el proceso de autenticación es el **inicio de sesión**: el proceso por el que el usuario ingresa sus credenciales de forma específica en una aplicación y el servidor confirma son válidas antes de enviar un **token** de autenticación al cliente. Este token se utilizará para comprobar la identidad del usuario en solicitudes futuras. Para realizar esta conexión inicial, se deberá manejar una **solicitud de inicio de sesión en el servidor**. Para facilitar el inicio de sesión, debes crear un controlador llamado **`AccountController`**. Si bien puedes usar un nombre diferente para él, gran parte de la lógica predeterminada en ASP.NET Core MVC se basa en `AccountController` y usar un nombre separado requerirá trabajo adicional.



Para realizar operaciones de inicio de sesión, deberás **inyectar el servicio** llamado **`SignInManager<T>`**, donde **`T`** es la clase que hereda de **`IdentityUser`** como se vio en el tema anterior. La clase **`SignInManager`** expone métodos para realizar operaciones de autenticación. Por ejemplo, para iniciar sesión, deberás llamar al método **`PasswordSignInAsync(*username*, *password*, *isPersistant*, *lockoutOnFailure*)`**.

El método denominado **`PasswordSignInAsync`** está diseñado para autenticar a un usuario y crear una sesión, mientras que también actualiza el cliente con el token de autenticación requerido para futuras solicitudes.

PasswordSignInAsync acepta los parámetros:

- **`*username*`**: El nombre de usuario utilizado por el usuario. Es una variable de cadena.
- **`*password*`**: La contraseña nunca debe almacenarse en cualquier lugar, ya que es una vulnerabilidad de seguridad. Variable de cadena.
- **`*isPersistant*`**: Determina si la cookie creada por el método de inicio de sesión persiste después de que se cierra el navegador. Es una variable booleana.
- **`*lockoutOnFailure*`**: Determina si se bloqueará al usuario si el intento de inicio de sesión no tiene éxito. El bloqueo se puede utilizar para evitar intentos malintencionados de iniciar sesión. El usuario ha escrito una contraseña incorrecta. Es una variable booleana.

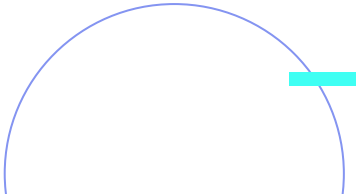
Una vez que el método de autenticación tiene éxito, puedes redirigir al usuario a una **URL** adecuada. También necesitarás crear una vista que llamará al método de inicio de sesión en el controlador. Considera que la vista debe usar un modelo diferente al que usa **`DatabaseContext`**.

Práctica recomendada: es posible que necesites modelos separados para los datos del servidor y los del cliente. Un ejemplo común de esto es cuando se usa una contraseña. Si bien el usuario deberá ingresar su contraseña en el cliente, esta no debe almacenarse en la base de datos. Necesitarás un modelo del lado del cliente que almacene una contraseña y un modelo del lado del servidor que no conserve la contraseña. Para diferenciar esto, puedes nombrar modelos destinados a ser utilizados en el lado del cliente con **`ViewModel`** al final. Esto los hará fácilmente visibles como modelos.

El siguiente código es un ejemplo de **`ViewModel`** que se puede utilizar para iniciar sesión:

```
public class LoginViewModel
{
public string Username { get; set; }
public string Password { get; set; }
}
```

Tener en cuenta que la contraseña en este modelo no tendrá *hash* y no debe guardarse.



El siguiente código es un ejemplo de **una vista para iniciar sesión**:

```
@modelAuthentication.Models.LoginViewModel
<h2>Login</h2>
<div>
<formmethod="post" asp-action="Login">
<div>
<labelasp-for="Username">Username</label>
<input asp-for="Username" />
</div>
<div>
<labelasp-for="Password">Password</label>
<input asp-for="Password" />
</div>
<input type="submit" value="Login" />
</form>
</div>
```

El siguiente es un ejemplo de un **AccountController** que puede manejar el inicio de sesión:

```
public class AccountController : Controller
{
private readonlySignInManager<WebsiteUser> _signInManager;
public AccountController(SignInManager<WebsiteUser>signInManager)
{
_signInManager = signInManager;
}
[HttpPost]
public asyncTask<IActionResult>Login(LoginViewModelmodel)
{
if (ModelState.IsValid)
{
var result = await _signInManager.PasswordSignInAsync(model.Username,
model.Password, true, false);
if (result.Succeeded)
{
if (Request.Query.Keys.Contains("ReturnUrl"))
{
returnRedirect(Request.Query["ReturnUrl"].First());
}
```

...

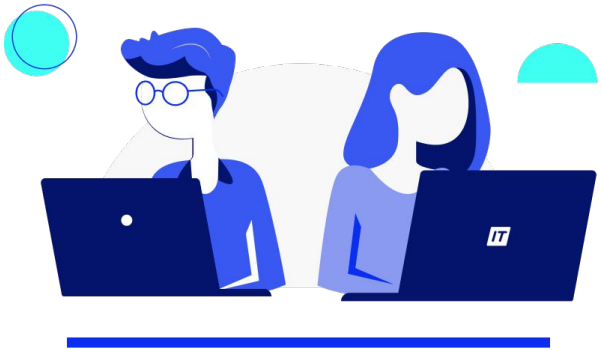
```
...

else
{
return RedirectToAction("Index", "Home");
}
}
}
return View();
}
}
```

Como puedes ver, en el ejemplo anterior, hay una clase **`AccountController`** que recibe una instancia de **`SignInManager`** con el tipo **`WebsiteUser`** mediante inyección de dependencia. Siempre que se realice una solicitud **`POST`** para el método de inicio de sesión, el estado del modelo se valida y, si es válido, el nombre de usuario y la contraseña se utilizan para realizar el inicio de sesión. Si el inicio de sesión se realiza de manera correcta, se comprobará si hay una **`ReturnUrl`** disponible. Si existe, redirigirá a la página originalmente solicitada. De lo contrario, navegará a la página de inicio. En todos los demás casos, será recargada la vista de inicio de sesión.

Nota: Se debe tener en cuenta que, dado que **`PasswordSignInAsync`** es un método asíncrono, deberás ajustar el método en una **`Task`** y usar **`await`** para poder determinar si el inicio de sesión es exitoso.

Realización de cierre de sesión

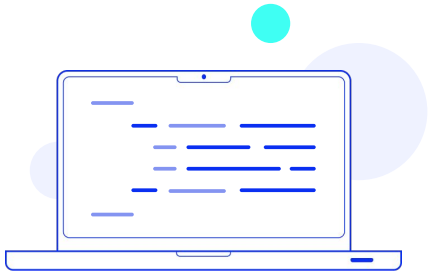
Ocasionalmente, los usuarios que han iniciado sesión en tu aplicación desearán cerrarla. Esto puede deberse a una variedad de razones, como cambiar a otro usuario o impedir el acceso a su cuenta de fuentes cercanas potencialmente maliciosas. Para facilitar esto, deberás agregar lógica para manejar las solicitudes de cierre de sesión. De manera similar al inicio de sesión, el cierre también debe estar en el mismo controlador, aunque es posible separarlos si eso tiene más sentido para tu aplicación. También utiliza el servicio **`SignInManager`** para cerrar sesión.



Antes de realizar un cierre de sesión debes **verificar si hay un usuario autenticado que inició sesión en la sesión actual**. Esto se puede hacer con la propiedad **`User`** de la clase de controlador. La propiedad de `User` expone varias propiedades relacionadas con el usuario actual en la sesión.

Para comprobar si un usuario se ha autenticado correctamente, puedes comprobar la propiedad **`IsAuthenticated`** en la propiedad **`Identity`**, en el **`User`**. Esto devolverá el estado de autenticación actual que te servirá para tomar decisiones sobre cómo manejar al usuario en toda su aplicación.

Para realizar realmente un cierre de sesión, deberás llamar al método **`SignOutAsync()`** desde el **`SignInManager`**. Este método borra los tokens de identidad almacenados para el usuario, lo que garantiza que las llamadas futuras no estarán en el usuario actual, a menos que se realice un nuevo **inicio de sesión**.



El siguiente es un ejemplo de una vista de cierre de sesión:

```
<a asp-action="Logout" asp-controller="Account">Log out</a>
```

El siguiente código es un ejemplo de un método de cierre de sesión en el controlador:

```
public IActionResultLogout()
{
if (User.Identity.IsAuthenticated)
{
_signInManager.SignOutAsync();
}
return RedirectToAction("Index", "Home");
}
```

Se debe tener en cuenta que, en este ejemplo, se realiza una comprobación para ver si el usuario actual está autenticado antes de realizar el cierre de sesión. Esto se hace marcando **`User.Identity.IsAuthenticated`**.

Agregar nuevos usuarios

Otra ocurrencia común en las aplicaciones web es agregar nuevos usuarios a la base de datos. Para realizar esto, se debe añadir la opción de registrar nuevos usuarios. A diferencia del inicio y cierre de sesión, la adición de nuevos usuarios no depende del servicio **`SignInManager`**. En cambio, necesitarás inyectar el servicio **`UserManager<T>`**, siendo la **`T`** el tipo que hereda de **`IdentityUser`**, que se utiliza para modificar usuarios existentes. Puedes utilizar el método cuyo nombre es **`CreateAsync(*user*,*password*)`** para crear un nuevo usuario en el sistema.

El método **`CreateAsync`** creará un nuevo usuario en la base de datos utilizando las distintas propiedades del usuario almacenado en la clase elegida, mientras convierte la contraseña en una cadena hash más segura. Esto ayuda a asegurar que la contraseña real no se almacena en ningún lugar, manteniendo de esta manera la seguridad de la aplicación. Una vez que el usuario se ha creado correctamente, el usuario podrá iniciar sesión con las nuevas credenciales y a continuación, puede utilizar los datos del usuario para iniciar sesión.

El siguiente código es un ejemplo de un modelo de vista para registrar nuevos usuarios:

```
public class RegisterViewModel : LoginViewModel
{
public string UserHandle { get; set; }
}
```

Se debe tener en cuenta que dado que el proceso de registro también requiere las propiedades Nombre de usuario y Contraseña, es lógico ampliar la clase **`LoginViewModel`**.

El siguiente código es un ejemplo de una vista para registrar nuevos usuarios:

```
@modelAuthentication.Models.RegisterViewModel
<div>
<formmethod="post" asp-action="Register">
<div>
<labelasp-for="Username">Username</label>
<input asp-for="Username" />
</div>
<div>
<labelasp-for="Password">Password</label>
<input asp-for="Password" />
</div>
<div>
<labelasp-for="UserHandle">DisplayName</label>
<input asp-for="UserHandle" />
</div>
<input type="submit" value="Register" />
</form>
</div>
```

El siguiente es un ejemplo de un método de registro en un controlador:

```
public classAccountController : Controller
{
privatereadonlySignInManager<WebsiteUser> _signInManager;
privatereadonlyUserManager<WebsiteUser> _userManager;

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

[HttpPost]
public asyncTask<IActionResult>Register(RegisterViewModelmodel)
{
if (ModelState.IsValid)
{
WebsiteUser user = new WebsiteUser
{
UserHandle = model.UserHandle,
UserName = model.Username,
};
```

...

...

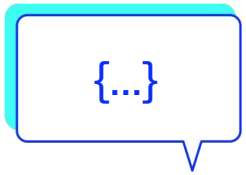
```
var result = await _userManager.CreateAsync(user, model.Password);
if (result.Succeeded)
{
return awaitLogin(model);
}
}
return View();
}
}
```

Acceder a las propiedades del usuario

Finalmente, una vez completado el flujo de inicio de sesión, el último paso es acceder a las propiedades de su usuario. Esto se puede hacer utilizando la propiedad **`User`** junto con el servicio cuyo nombre es **`UserManager`**. Al usar el método denominado **`FindByNameAsync(*name*)`** en el objeto llamado **`UserManager`** y proporcionándole el valor de la propiedad **`Name`** en **`User.Identity`**, **`UserManager`** recuperará al usuario, si lo encuentra. Luego, puede extraer propiedades de la clase de usuario que seleccionó y se mostrará en la página con el modelo.

El siguiente código es un ejemplo de un modelo de vista para mostrar al usuario:

```
public class UserDisplayViewModel
{
public string UserHandle { get; set; }
public string UserName { get; set; }
}
```



El siguiente código es un ejemplo de una vista que muestra propiedades personalizadas de **`IdentityUser`**:

```
@modelIdentityAgain.Models.UserDisplayViewModel
@if (Model != null)
{
<div>Hello @Model.UserHandle</div>
}
```

El siguiente código es un ejemplo de un controlador que recupera las propiedades de **`IdentityUser`**:

```
public asyncTask<IActionResult>Index()
{
if (!User.Identity.IsAuthenticated)
{
return RedirectToAction("Login", "Account");
}
WebsiteUser user = await _userManager.FindByNameAsync(User.Identity.Name);
if (user != null)
{
UserDisplayViewModelmodel = new UserDisplayViewModel
{
UserHandle = user.UserHandle,
UserName = user.UserName
};
return View(model);
}
return View();
}
```

En el ejemplo anterior, puedes ver que esta acción solo se mostrará para los usuarios autenticados, los que actualmente están conectados con la clase personalizada **`WebsiteUser`** que hereda de **`IdentityUser`**.

Luego, el controlador usa al usuario para crear una nueva vista que contiene las propiedades de la clase **`WebsiteUser`**, incluido **`UserHandle`** que no están presentes en la clase **`UserIdentity`** predeterminada.

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!