educación IT
CODE YOUR FUTURE

Programación Web .NET Core

Módulo 8

educación IT
CODE YOUR FUTURE

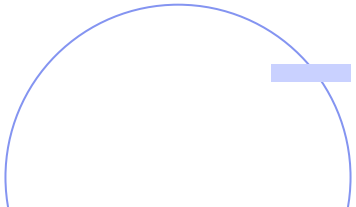
Uso de Entity Framework Core para conectarse a Microsoft SQL Server

Conexión a Microsoft SQL Server

El contexto de Entity Framework se puede utilizar para conectarse a diferentes bases de datos. Ya vimos el caso de una base de datos SQLite. En esta lección, conectaremos a una base de datos SQL Server. El código de la derecha, muestra una clase de entidad:

Una clase de entidad

```
publicclassPerson
{
public int PersonId{ get; set; }
public string FirstName { get; set; }
public string LastName{ get; set; }
}
```



El siguiente ejemplo de código muestra un contexto de Entity Framework:

La claseHrContext

```
public class HrContext :DbContext
{
public HrContext(DbContextOptions<HrContext> options) : base(options)
{
}
public DbSet<Person> Candidates { get; set; }
protected override void OnModelCreating(ModelBuildermodelBuilder)
{
modelBuilder.Entity<Person>().HasData(
new { PersonId = 1, FirstName = "James", LastName = "Smith"},
new { PersonId = 2, FirstName = "Arthur", LastName = "Adams"}
);
}
}
```

El siguiente código muestra cómo configurar **HrContext** para conectarse a una base de datos de un SQL Server:

```
public class Startup
{
public void ConfigureServices(IServiceCollection services)
{
string connectionString =
"Server=(localdb)\\MSSQLLocalDB;Database=HumanResources;Trusted_Connection=True;";
services.AddDbContext<HrContext>(
options =>options.UseSqlServer(connectionString));
services.AddMvc();
}
public void Configure(IApplicationBuilder app, HrContextctx)
{
ctx.Database.EnsureDeleted();
ctx.Database.EnsureCreated();
app.UseMvc();
}
}
```

El método **`UseSqlServer`** configura el contexto de Entity Framework para conectarse a una base de datos de SQL Server.

Cuando se usa la cadena de conexión especificada en el ejemplo anterior, el contexto de Entity Framework se conecta a una base de datos llamada **`HumanResources`** en la instancia **`LocalDB\MSSQLLocalDB`**.

> **Nota: `LocalDB`** es una extensión de SQL Express que ofrece una manera fácil de crear múltiples bases de datos. Dado que se ejecuta en modo de usuario, es fácil de configurar.

Cuando se crea la base de datos denominada `HumanResources`, contiene una tabla llamada **`Candidates`**. La razón de esto es que la clase **`HrContext`** contiene una propiedad denominada **`Candidates`** de tipo **`DbSet<Person>`**. Las propiedades **`DbSet<>`** de un contexto de Entity Framework se asignan a tablas en la base de datos.

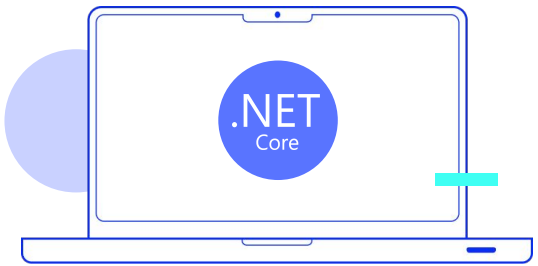
La tabla de candidatos (**`Candidates`**) contiene tres columnas:

- **`PersonId`** (PK, `int`, `no nulo`).
- **`Nombre`** (`nvarchar` (`máx.`), `Nulo`).
- **`Apellido`** (`nvarchar` (`máx.`), `Nulo`).

Esto se debe a que la entidad **`Person`** contiene las propiedades **`PersonId`**, **`FirstName`** y **`LastName`**.

De forma predeterminada, cada propiedad de la entidad **`Person`** se asigna a una columna de la tabla con el mismo nombre y tipo de datos. EF Core determina las restricciones de cada columna mediante un conjunto de convenciones.

> **Nota:** La columna **`PersonId`** es una columna de clave principal en la base de datos porque la propiedad **`PersonId`** es la clave de entidad de la entidad **`Person`**. Por convención, es la clave de la entidad en EF Core.



Configuración en EF Core

EF Core usa convenciones para construir una base de datos basada en el contenido del contexto de Entity Framework. Una forma de especificar la configuración en EF Core es mediante ***Fluent API***. Cuando usas Fluent API, escribes la configuración en el método **`OnModelCreating`** de la clase de contexto de Entity Framework.

Por ejemplo, en EF Core puedes usar Fluent API para determinar el nombre de una tabla que se crea en una base de datos. Para eso, puedes usar el método **`ToTable`**.

El ejemplo de la siguiente pantalla, muestra cómo crear una tabla denominada **`People`** en la base de datos que será mapeada a la propiedad **`Candidates`** de la clase **`HrContext`**.

Nombrar tablas de bases de datos

```
public class HrContext :DbContext
{
public HrContext(DbContextOptions<HrContext> options) : base(options)
{
}
public DbSet<Person> Candidates { get; set; }
protected override void OnModelCreating(ModelBuildermodelBuilder)
{
modelBuilder.Entity<Person>().ToTable("People");
}
}
```

Además de Fluent API, también puedes especificar la configuración en EF Core mediante otro método llamado ***"anotaciones de datos"***. Sin embargo, el método de anotaciones de datos es bastante diferente del Fluent API. Mientras que en el Fluent API, la configuración se especifica en el método **`OnModelCreating`** de la clase de contexto de Entity Framework, en el método de anotaciones de datos la configuración se especifica usando atributos en las entidades.

El siguiente ejemplo de código demuestra cómo usar la anotación de datos para configurar la entidad **`Person`**:

Usar anotaciones de datos

```
publicclassPerson
{
[Key]
public int MyId{ get; set; }
[Required]
[StringLength(20)]
public string FirstName { get; set; }
[NotMapped]
public string LastName{ get; set; }
}
```

Cuando EF Core usa la entidad **Person** anterior para crear la tabla en la base de datos, la tabla se crea con dos columnas:

- **MyId** (PK, int, no nulo).
- **Nombre** (nvarchar (20), no nulo).

Columna MyId

La columna **MyId**, en la base de datos, es una columna de clave principal, ya que la propiedad MyId es la clave de la entidad **Person** -porque está anotada con el atributo **Key**-.

Columna FirstName

La columna **FirstName**, en la base de datos, no es nula ya que la propiedad FirstName de la entidad **Person** es anotada con el atributo requerido. El tipo de datos de la columna FirstName es **nvarchar (20)** ya que la longitud máxima de
la propiedad FirstName se configura en 20 mediante el atributo **StringLength**.

La propiedad **LastName** está anotada con el atributo **NotMapped**. Por lo tanto, no se asigna a una columna de la base de datos.

Configuración en ASP.NET Core

Por lo general, una aplicación necesita obtener ***parámetros***. Estos parámetros se utilizan para **definir configuraciones y preferencias que se necesitan al ejecutar la solicitud**. Un ejemplo de dicho parámetro es la cadena de conexión que utiliza la aplicación para conectarse a una base de datos. Un enfoque recomendado es almacenar los parámetros en un ***archivo de configuración***. Así, permitirán cambiar la configuración y las preferencias sin recompilar la aplicación.

ASP.NET Core es capaz de leer parámetros de diversas fuentes. Para acceder a una fuente, usa un ***proveedor de configuración***.

Existen proveedores de configuración para varios formatos de archivo de configuración, incluidos **JSON**, **XML** e **INI**.

Adicionalmente, hay proveedores de configuración para otras fuentes, como *argumentos de línea de comando* y *variables de entorno*. Incluso se puede crear un proveedor personalizado si es necesario. Una aplicación ASP.NET Core puede leer parámetros de archivos JSON y archivos XML. Los parámetros de los archivos se almacenarán como **pares de nombre-valor**.

Configuración JSON

JSON es un formato de texto que sirve para varios propósitos, como transmitir datos entre aplicaciones y almacenar datos en un archivo. Una de sus mayores ventajas es que es muy sencillo de leer y escribir. JSON también se puede analizar y generar fácilmente vía aplicaciones.

El código de la derecha muestra el contenido de un archivo JSON llamado ***config.json***:

Un archivo de configuración JSON

```
{
"firstName": "Mark",
"lastName": "Smith",
"address": {
"streetAddress": "35 3rd Street",
"city": "Miami"
}
}
```

Veamos el ejemplo de código de la siguiente pantalla, que demuestra cómo usar el proveedor de configuración JSON para cargar los valores desde el archivo ***config.json***.

Uso del proveedor de configuración JSON en Program.cs

```
public class Program
{
public static void Main(string[] args)
{
CreateWebHostBuilder(args).Build().Run();
}
public static IWebHostBuilderCreateWebHostBuilder(string[] args)
{
var builder = new ConfigurationBuilder()
.SetBasePath(Directory.GetCurrentDirectory())
.AddJsonFile("config.json");
IConfiguration configuration = builder.Build();
return WebHost.CreateDefaultBuilder(args)
.UseConfiguration(configuration)
.UseStartup<Startup>();
}
}
```

Nota: La llamada al método denominado **`WebHost.CreateDefaultBuilder`** inicializa una nueva instancia de la clase llamada **`WebHostBuilder`** con valores predeterminados preconfigurados. Se pueden cargar valores desde un archivo de configuración llamado ***appsettings.json***. Para hacerlo, es suficiente con llamar al método cuyo nombre es **`CreateDefaultBuilder`** y no es necesario llamar al método **`AddJsonFile`** y pasarle ***appsettings.json*** como parámetro.

Para cargar un valor del archivo de configuración, puedes utilizar un indexador de un objeto que implementa el **`IConfiguration`** y pasarle el nombre del par **nombre-valor** como parámetro.

El siguiente ejemplo de código demuestra cómo recuperar un valor del archivo ***config.json***. El valor dela variable de valor será **`Mark`**:

Recuperar un valor de un archivo de configuración

```
public class SomeController : Controller
{
private IConfiguration _configuration;
public SomeController(IConfiguration configuration)
{
_configuration = configuration;
}
public IActionResultIndex()
{
string value = _configuration["firstName"];
return Content(value);
}
}
```

El siguiente ejemplo de código demuestra cómo recuperar un valor de un **nombre jerárquico**. El valor de la variable de valor será **`Miami`**:

Recuperar un valor de un nombre jerárquico

```
public class SomeController : Controller
{
private IConfiguration _configuration;
public SomeController(IConfiguration configuration)
{
_configuration = configuration;
}
public IActionResultIndex()
{
string value = _configuration["address:city"];
return Content(value);
}
}
```

El ejemplo de código de la pantalla siguiente, demuestra cómo leer la cadena de conexión del archivo de configuración mediante el uso de un indexador de **`IConfiguration`**. Luego, la cadena de conexión se usa para conectar la entidad a la base de datos **`HumanResources`** en la instancia **`LocalDB\MSSQLLocalDB`**.

Leer una cadena de conexión desde un archivo de configuración

```
public class Startup
{
private IConfiguration _configuration;
public Startup(IConfiguration configuration)
{
_configuration = configuration;
}
public void ConfigureServices(IServiceCollection services)
{
string connectionString = _configuration["ConnectionStrings:DefaultConnection"];
services.AddDbContext<HrContext>(options =>
options.UseSqlServer(connectionString));
services.AddMvc();
}
public void Configure(IApplicationBuilder app, HrContextctx)
{
ctx.Database.EnsureDeleted();
ctx.Database.EnsureCreated();
app.UseMvc();
}
}
```

Para leer una cadena de conexión de un archivo de configuración, se puede reemplazar la llamada al indexador de **`configuration`** con una llamada al método **`GetConnectionString`**.

El siguiente ejemplo de código demuestra cómo utilizar el método **`GetConnectionString`** para leer la cadena de conexión del archivo de configuración:

Usando el método GetConnectionString

```
public void ConfigureServices(IServiceCollection services)
{
string connectionString = _configuration.GetConnectionString("DefaultConnection");
services.AddDbContext<HrContext>(options =>options.UseSqlServer(connectionString));
services.AddMvc();
}
```

El patrón del repositorio

Las aplicaciones web, normalmente, necesitan acceder a bases de datos y el enfoque más simple es utilizar el controlador para acceder directamente. Sin embargo, esta no es siempre la mejor práctica. Un enfoque más flexible es que el **controlador llame a un repositorio que accederá a la base de datos**.

Para llamar a un repositorio de un controlador:

1. **Definir una interfaz para la clase de repositorio**. Esta interfaz declara los métodos que utiliza la clase de repositorio para leer y escribir datos desde y hacia la base de datos.
2. **Crear y escribir código para la clase de repositorio**. Esta clase debe implementar todos los métodos de acceso a datos implementando la interfaz.
3. **Utilizar la inyección de dependencia** para inyectar la clase de repositorio a un controlador.
4. **Modificar la clase de controlador** para usar la clase de repositorio.

Una interfaz de repositorio

```
PublicinterfaceIRepository
{
IEnumerable<Person>GetPeople();
void InsertPerson();
voidDeletePerson();
voidUpdatePerson();
}
```

El siguiente código muestra una clase de repositorio:

Una clase de repositorio

```
publicclassMyRepository :IRepository
{
private HrContext _context;
public MyRepository(HrContext context)
{
_context = context;
}
public IEnumerable<Person>GetPeople()
{
return _context.Candidates;
}
public void InsertPerson()
{
_context.Candidates.Add(new Person() { FirstName = "John", LastName = "Smith" });
_context.SaveChanges();
}
```

...

```
public void DeletePerson()
{
var person = _context.Candidates.FirstOrDefault(c => c. LastName == "Smith");
_context.Candidates.Remove(person);
_context.SaveChanges();
}
public void UpdatePerson()
{
var person = (from c in _context.Candidates where c.FirstName == "James" select
c).Single();
person.FirstName = "Mike";
_context.SaveChanges();
}
}
```

El siguiente código muestra cómo registrar el repositorio en el método **`ConfigureServices`**, en el archivo ***Startup.cs***:

Registro del repositorio

```
public void ConfigureServices(IServiceCollection services)
{
services.AddTransient<IRepository, MyRepository>();
string connectionString = _configuration.GetConnectionString("DefaultConnection");
services.AddDbContext<HrContext>(options =>options.UseSqlServer(connectionString));
services.AddMvc();
}
```

El siguiente ejemplo de código demuestra cómo un controlador puede usar el repositorio:

Un controlador que usa el repositorio

```
public class HrController : Controller
{
IRepository _repository;
public HrController(IRepository repository)
{
_repository = repository;
}
[Route("Hr/Index")]
public IActionResultIndex()
{
var list = _repository.GetPeople();
return View(list);
}
```

```
[Route("Hr/Insert")]
public IActionResultInsert()
{
_repository.InsertPerson();
return RedirectToAction("Index");
}
[Route("Hr/Delete")]
public IActionResultDelete()
{
_repository.DeletePerson();
return RedirectToAction("Index");
}
[Route("Hr/Update")]
public IActionResultUpdate()
{
_repository.UpdatePerson();
return RedirectToAction("Index");
}
}
```

Herramientas de la consola de Entity Framework Core Package Manager

Puedes trabajar con **migraciones** usando ***Entity Framework Core Package Manager Console (PMC)***. Es posible utilizarar la consola del administrador de paquetes de **NuGet** para ejecutar las herramientas de PMC en Visual Studio.

Para **crear una base de datos mediante migraciones**, primero debes agregar una ***migración inicial***. Entonces, necesitas aplicar la migración a la base de datos. Como resultado, se creará el esquema de la base de datos.

El siguiente ejemplo de código demuestra cómo agregar una migración inicial:

Agregar una migración inicial

```
Add-MigrationInitDatabase
```

El siguiente ejemplo muestra qué comando debes ejecutar para aplicar la migración para crear el esquema de la base de datos:

Crea el esquema de la base de datos

```
Update-Database
```

Cuando cambia una entidad, la base de datos y el código ya no están sincronizados. Para **sincronizar el código y la base de datos**, **debes agregar otra migración**.

El siguiente ejemplo de código muestra un cambio en la entidad **`Person`**. Se agrega, a la entidad `Person`, una nueva propiedad llamada **`Edad`**.

Por lo tanto, la base de datos y el código ya no están sincronizados:

Actualización de la entidad Persona

```
publicclassPerson
{
public int PersonId{ get; set; }
public string FirstName { get; set; }
public string LastName{ get; set; }
public int? Age { get; set; }
}
```

El siguiente código muestra cómo agregar una nueva migración:

Agregar una nueva migración

```
Add-MigrationAddAge
```

Para aplicar la migración de **AddAge** a la base de datos, usa el comando de la derecha.

Aplicar la migración de AddAge a la base de datos

```
Update-Database
```

Después de aplicar la migración **AddAge**, se agrega una nueva columna denominada **Edad** a la base de datos.

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!