

# Renombrar las tablas de ASP .NET Identity para Postgresql

Los modelos definidos en ASP .NET Identity están diseñados para motores de bases de datos que usen los nombres de las tablas y columnas en mayúsculas. Sin embargo, en otros, como por ejemplo Postgresql, los nombres de tablas y columnas se escriben en minúsculas. Cuando ejecutamos una migración, todas las tablas y columnas se crearán con los nombres en mayúsculas. En el caso particular de Postgresql esto puede resultar incómodo cuando queremos escribir consultas SQL a mano, ya que, esto nos obligará a escribir cada nombre de tabla y columna entre comillas.

Para remediar este problema debemos por un lado renombrar todas las tablas de Identity y por otro implementar un mecanismo para que los nombres de las propiedades en las entidades de Identity sean transformadas automáticamente. A continuación vamos a tratar ambos requisitos.

---

## Transformación automática de las propiedades

En las clases de Identity, los nombres de las propiedades siguen la convención *pascal*. Sin embargo en Postgresql siguen la convención *snake*. Una solución para adaptar los nombres sería reescribir todas las propiedades de las clases de Identity, lo cual no es para nada recomendable. Sin embargo, podemos transformar los nombres en tiempo real usando una librería, que transformará el nombre de una propiedad en función de la operación que se esté realizando. Cuando se generen las consultas SQL, la librería transformará los nombres al formato *snake*, para que sean ejecutadas en la base de datos. Cuando se recuperen datos desde la base de datos, la librería transformará los nombres al formato *pascal* para que coincidan con los nombres de las propiedades en las clases de Identity. De este modo la librería actúa como un proxy entre EntityFramework y PostgreSQL para permitir que se sigan las convenciones en cada extremo.

Existen diversas alternativas que permiten realizar esta tarea, pero nosotros nos decantamos por [EFCore.NamingConventions](#), que soporta diferentes tipos de conversiones: snake en minúscula y mayúscula, todo en minúsculas, todo en mayúsculas y camel.

Ten en cuenta que si usas esta librería con una base de datos existente se creará una nueva migración que renombrará todas las tablas, columnas, índices, claves foráneas, etc., cuyos

resultados pueden ser imprevisibles. Úsala con cuidado en estos casos.

Instala el paquete nuget [EFCore.NamingConventions](#) en el proyecto. Modifica el archivo Program.cs para configurar el uso de la librería:

```
//...

builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseNpgsql(connectionString).UseSnakeCaseNamingConvention());

//...
```

Lo que estamos haciendo en el código anterior, es, definir un conversor de formato snake para el contexto de datos `ApplicationDbContext` que usa la librería `Npgsql` como adaptador de datos. De este modo queda configurado el proxy.

## Renombrar las tablas ASP .NET Identity

Pueden haber varias opciones, pero a continuación te mostraremos dos opciones: una usando el esquema por defecto y otra especificando un esquema específico para las tablas de Identity.

### Opción 1: Usar el esquema por defecto de la base de datos

En este primer ejemplo te mostramos como renombrar las tablas de ASP .NET Identity cuando estas se crean el esquema por defecto de una base de datos PostgreSQL, que es *public*.

El modelo de datos de Identity está definido usando instrucciones de construcción de modelos, por lo cual, aun cuando usamos la librería, los nombres de las tablas no se transforman automáticamente. De hecho, los nombres de las tablas creadas no coinciden con los nombres de las entidades. Por lo cual, debemos sobrescribir los nombres de la tabla usando el mismo método:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using pocogen.Entities;

namespace myProject.Data
{
    public class ApplicationDbContext : IdentityDbContext
    {
```

```

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
}

//...

protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);

    builder.Entity<IdentityUser>(). ToTable("identity_users");
    builder.Entity<IdentityRole>(). ToTable("identity_roles");
    builder.Entity<IdentityUserToken<string>>(). ToTable("identity_user_tokens");
    builder.Entity<IdentityUserRole<string>>(). ToTable("identity_user_roles");
    builder.Entity<IdentityRoleClaim<string>>(). ToTable("identity_role_claims");
    builder.Entity<IdentityUserClaim<string>>(). ToTable("identity_user_claims");
    builder.Entity<IdentityUserLogin<string>>(). ToTable("identity_user_logins");
}

//...
}
}

```

Si en tu aplicación has definido un tipo específico para el usuario, el rol o para cualquier otro de Identity, debes especificar este tipo en el constructor en lugar del tipo por defecto, por ejemplo:

```

// ...

// Las plantillas de Identity crean un usuario llamado ApplicationUser
builder.Entity<ApplicationUser>(). ToTable("users", "identity");

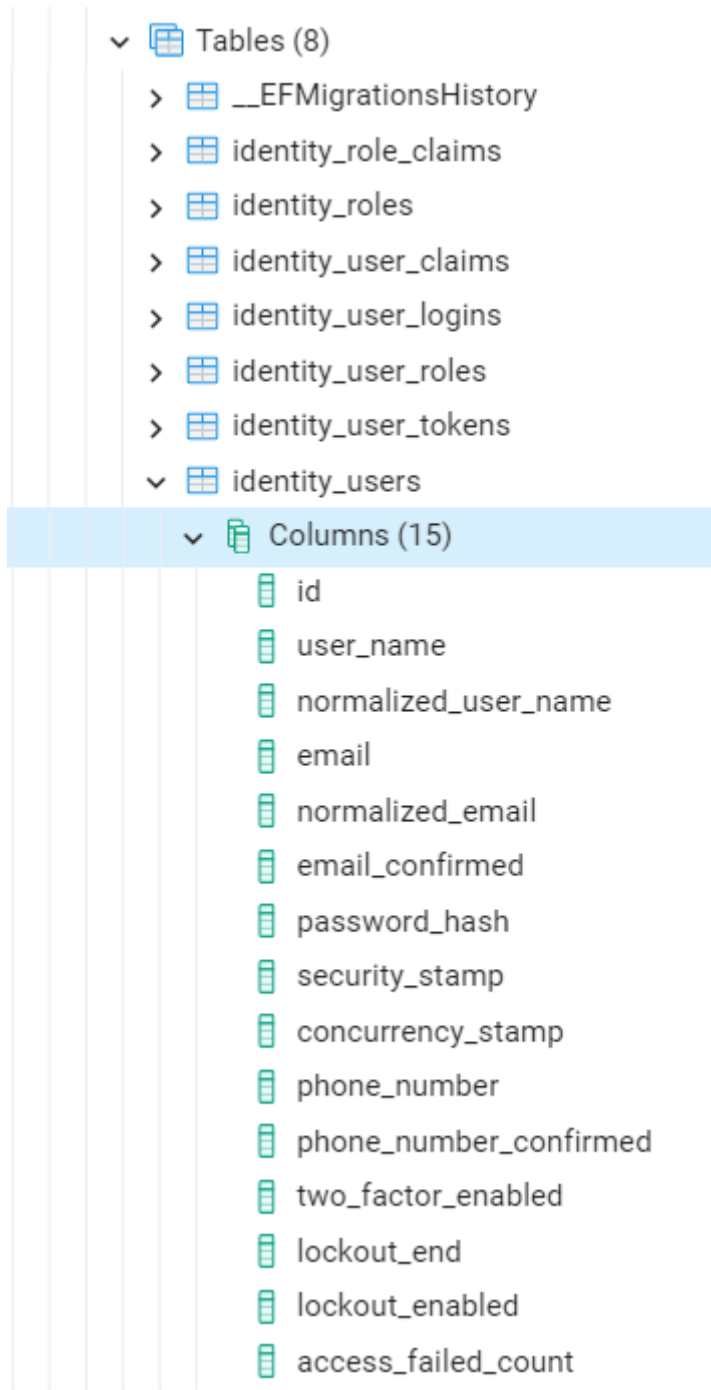
// ...

```

En el código anterior, reescribimos el método `OnModelCreating` que es llamado cada vez que se crea el modelo de datos. Dentro de este método podemos definir todas las características de las entidades. Para renombrar las tablas, hemos definido el nombre de la tabla correspondiente a cada entidad de Identity. En el ejemplo hemos usado el prefijo `identity_` para las tablas, pero, tú puedes nombrarlas como tu quieras, según tus necesidades y convenciones.

Después de crear una migración y de aplicarla a la base de datos, todas las tablas se habrán creado con el nombre definido y todas las columnas estarán en formato snake. En la imagen a

continuación podrás ver el resultado:



The screenshot shows a database schema viewer with a tree view. The 'Tables (8)' folder is expanded, showing a list of tables. The 'identity\_users' table is selected and expanded, showing its 15 columns. The columns are: id, user\_name, normalized\_user\_name, email, normalized\_email, email\_confirmed, password\_hash, security\_stamp, concurrency\_stamp, phone\_number, phone\_number\_confirmed, two\_factor\_enabled, lockout\_end, lockout\_enabled, and access\_failed\_count.

▼	Tables (8)
>	__EFMigrationsHistory
>	identity_role_claims
>	identity_roles
>	identity_user_claims
>	identity_user_logins
>	identity_user_roles
>	identity_user_tokens
▼	identity_users
▼	Columns (15)
	id
	user_name
	normalized_user_name
	email
	normalized_email
	email_confirmed
	password_hash
	security_stamp
	concurrency_stamp
	phone_number
	phone_number_confirmed
	two_factor_enabled
	lockout_end
	lockout_enabled
	access_failed_count

## Opción 2: Definir un esquema específico para las tablas de Identity

En esta segunda opción veremos como definir las tablas de Identity para que se creen en un esquema diferente de *public*. Esto puede ser interesante cuando quieres mantener una separación entre las tablas de Identity y las del resto de la aplicación.

El procedimiento sigue siendo el mismo, simplemente vamos a especificar un esquema para cada tabla de ASP .NET Identity como se muestra a continuación.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using pocogen.Entities;

namespace myProject.Data
{
    public class ApplicationDbContext : IdentityDbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        //...

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);

            builder.Entity<IdentityUser>().ToTable("users", "identity");
            builder.Entity<IdentityRole>().ToTable("roles", "identity");
            builder.Entity<IdentityUserToken<string>>().ToTable("user_tokens", "identity");
            builder.Entity<IdentityUserRole<string>>().ToTable("user_roles", "identity");
            builder.Entity<IdentityRoleClaim<string>>().ToTable("role_claims", "identity");
            builder.Entity<IdentityUserClaim<string>>().ToTable("user_claims", "identity");
            builder.Entity<IdentityUserLogin<string>>().ToTable("user_logins", "identity");
        }

        //...
    }
}
```

Si en tu aplicación has definido un tipo específico para el usuario, el rol o para cualquier otro de Identity, debes especificar este tipo en el constructor en lugar del tipo por defecto, por ejemplo:

```
// ...
```

```
// Las plantillas de Identity crean un usuario llamado ApplicationUser  
builder.Entity<ApplicationUser>().ToTable("users", "identity");
```

```
// ...
```

Como puedes observar en este segundo ejemplo, estamos usando el segundo parámetro del método `ToTable()`, que permite definir el esquema para cada tabla por separado. Esta opción nos da un control muy preciso de la ubicación de las tablas dentro de la base de datos.

En función del usuario que hayas definido en la cadena de conexión y de qué privilegios tenga, es posible que Entity Framework no pueda crear el esquema. En ese caso deberás crear el esquema manualmente antes de aplicar una migración.

---

Revisión #5

Creado 27 junio 2023 08:30:50 por Guillermo

Actualizado 1 febrero 2024 11:11:02 por Guillermo