
Softland
Lo hacemos fácil

Laboratorio Api rest con net core y JWT

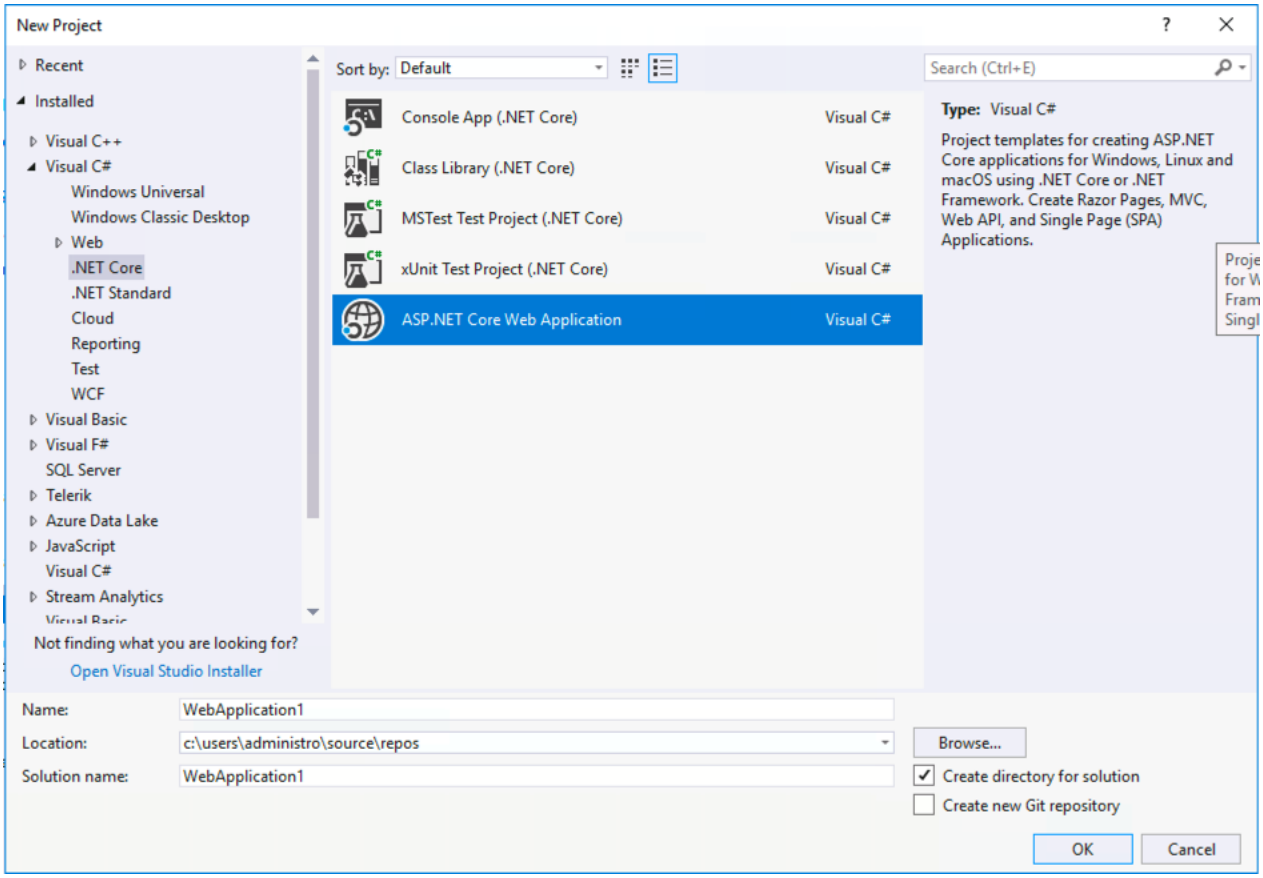
Creación del proyecto

Lo primero que vamos a hacer es crear el proyecto, para ello es necesario tener instalado el sdk de .Net Core 2.0 y en este caso Visual studio 2017.

1º Accedemos a Visual studio

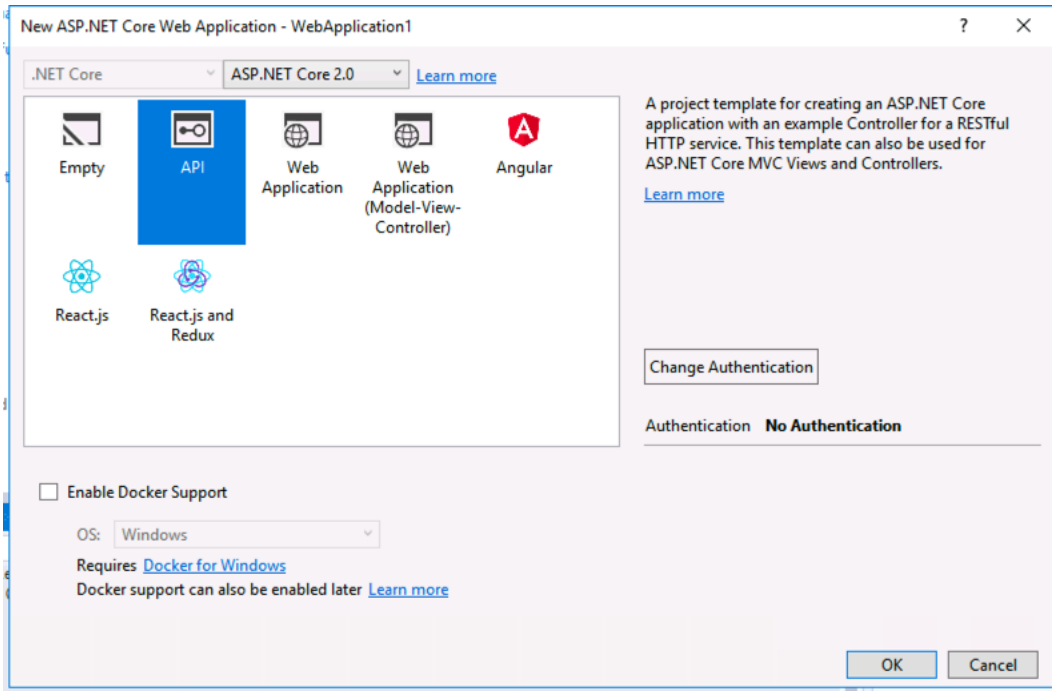
2º Seleccionamos “File/New Project”

3º Buscamos Net Core y elegimos Asp. Net core Web Application



4º Indicamos el nombre, por ejemplo “Camp Tajamar”

5º En la siguiente plantilla, seleccionamos Api y pulsamos sobre Ok, lo que creara el proyecto.



Agregar el Modelo

En esta parte vamos a definir el modelo, este estará compuesto de cuatro clases.

Tienda
Producto
Categoría
TiendaProducto

Antes de continuar necesitaremos los siguientes Nugets en nuestro proyecto:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.SqlServer.Design
- Microsoft.EntityFrameworkCore.Tools

1º Creamos una carpeta llamada Model en el proyecto.
2º Dentro de esta carpeta Agregamos las clases

Tienda:

```
    public class Tienda
{

    [Key]
    public int IdTienda { get; set; }
    public string Nombre { get; set; }
```

```
        public virtual ICollection<ProductoTienda> ProductoTienda { get; set; }

    }
```

Producto:

```
public class Producto
    {

        [Key]
        public int IdProducto { get; set; }
        public string Nombre { get; set; }
        public double Precio { get; set; }
        public int Existencias { get; set; }

        public virtual Categoria Categoria { get; set; }

        public virtual ICollection<ProductoTienda> ProductoTienda { get; set; }

    }
```

Categoria:

```
public class Categoria
    {
        [Key]
        public int IdCategoria { get; set; }
        public string Nombre { get; set; }

        public virtual ICollection<Producto> Producto { get; set; }
    }
```

ProductoTienda:

```
public class ProductoTienda
    {
        public int IdProducto { get; set; }
        public int IdTienda { get; set; }
        public virtual Producto Producto { get; set; }
        public virtual Tienda Tienda { get; set; }

    }
```

3º Una vez hemos agregado las clases, lo siguiente es construir el DbContext, que se va a encargar de manejar la persistencia de los objetos. Dentro de la carpeta Model, creamos la clase ProductosContext, y agregamos el siguiente código.

```
public class ProductosContext: DbContext
    {

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

            public DbSet<Producto> Producto { get; set; }
            public DbSet<Tienda> Tienda { get; set; }
            public DbSet<Categoria> Categoria { get; set; }
            public DbSet<ProductoTienda> ProductoTienda { get; set; }



    }
```

4º Agregar el método OnModelCreating, para manejar las relaciones entre las tablas. Dentro de la clase ProductosContext, agregamos el siguiente código.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
            {
                base.OnModelCreating(modelBuilder);


            modelBuilder.Entity<ProductoTienda>()
                    .HasKey(pc => new { pc.IdProducto, pc.IdTienda });

            modelBuilder.Entity<ProductoTienda>()
                    .HasKey(pc => new { pc.IdProducto, pc.IdTienda });

                modelBuilder.Entity<ProductoTienda>()
                    .HasOne(pc => pc.Producto)
                    .WithMany(p => p.ProductoTienda)
                    .HasForeignKey(pc => pc.IdProducto);

                modelBuilder.Entity<ProductoTienda>()
                    .HasOne(pc => pc.Tienda)
                    .WithMany(c => c.ProductoTienda)
                    .HasForeignKey(pc => pc.IdTienda);




        }
```

5º Agregar un método para inicializar la base de datos. Creamos otra clase mas, llamada DbInitializer, esta clase la usaremos para cargar datos de prueba a nuestra base de datos.

```
public static class DbInitializer
    {

        public static void Initialize(ProductosContext context)
        {



            if (context.Producto.Any())
            {
                return;
            }

            var tienda = new Tienda()
            {
                Nombre = "La tienda de telefonos",

            };
            context.Tienda.Add(tienda);

            var cat = new Categoria()
            {
                Nombre = "Telefonia",

            };
            context.Categoria.Add(cat);

            var productos = new Producto[]{
                new Producto(){

                    Nombre="Iphone",
                    Existencias= 5,
                    Precio=1100,
                    Categoria= cat

                },
                new Producto(){

                    Nombre="Galaxy s9",
                    Existencias= 3,
                    Precio=900,
                    Categoria= cat
                },
            };
            foreach (var compra in productos)
            {
                context.Producto.Add(compra);
                var ti=new ProductoTienda()
                {
                    Producto = compra,
                    Tienda = tienda
                };
                context.ProductoTienda.Add(ti);
            }
            context.SaveChanges();

        }
    }
```

6º Agregar la cadena de conexión a la base de datos. En el fichero appsettings.json, debemos agregar la cadena de conexión a nuestra base de datos.

```
"ConnectionStrings": {
    "DefaultConnection": "Server=tcp:servidor,1433;Initial Catalog=PruebasCamp;Persist
Security Info=False;User
ID=usuario;Password=password;MultipleActiveResultSets=True;Encrypt=True;TrustServerCertificat
e=False;Connection Timeout=30;"
  },
```

7º Configurar el startup de nuestro proyecto, para habilitar el uso de ProductosContext, y preparar la inyección de dependencias.

En el método ConfigureServices del fichero startup.cs, agregamos el siguiente código

```
var conexion = Configuration.GetConnectionString("DefaultConnection");
services.AddDbContext<ProductosContext>(options => options.UseSqlServer(conexion));
```

A continuacion en el metodo Configure, agregamos la llamada al inicializador de la base de datos

```
DbInitializer.Initialize(context);
```

Crear la base de datos usando las migrations

En esta tare crearemos la base de datos desde el modelo.

Para continuar necesitaremos agregar los siguientes nugets a nuestro proyecto.

```
Microsoft.EntityFrameworkCore.Tools.DotNet,
```

En caso de que nos de un error al agregarlo, se deberá editar el fichero csproj y agregar lo siguiente:

```
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1"
/>
```

Dentro de un ItemGroup

1º Agregar la primera migration. Desde el interfaz de comandos, nos situamos a la misma altura que el fichero csproj y ejecutamos el comando

```
Dotnet restore
```

2º Agregar la migration. En el mismo interfaz de comandos, ejecutamos lo siguiente

```
Dotnet ef migrations add Inicializar
```

3º Crear la base de datos

```
Dotnet ef database update
```

Agregar los controladores

Una vez creada la base de datos, ya podemos terminar nuestro api, para ello crearemos los controladores.

1º En la carpeta Controllers, vamos a agregar un controlador para productos, para ello pulsamos con el botón derecho encima y seleccionamos “add/controller” y elegimos Api Controller – Empty

2º Sobre el controlador, agregamos una referencia al context, y la inicializamos en el constructor.

```
      public ProductosContext Context { get; private set; }

       public ProductosController(ProductosContext context)
       {
           Context = context;
       }
```

3º Agregamos el siguiente método para obtener la lista de todos los productos

```
public IEnumerable<Producto> Get()
       {
           var data = Context.Producto.ToList();

           return data;
       }
```

4º Agregamos el siguiente código para obtener un producto por su código.

```
[HttpGet("{id}")]
 public IActionResult Get(int id)
 {
     var data = Context.Producto.FirstOrDefault(o=>o.IdProducto==id);

     if (data == null)
         return NotFound();
     return Ok(data);
 }
```

5º Agregamos el siguiente Método para crear un nuevo producto

```
[HttpPost]
 public IActionResult Post([FromBody]Producto value)
 {
     Context.Producto.Add(value);
     try
     {
         Context.SaveChanges();
     }
     catch (Exception e)
     {
         return BadRequest(e.Message);
     }

     return Created("",value);
 }
```

6º Agregamos el siguiente método para modificar un producto

```
[HttpPut("{id}")]
 public IActionResult Put(int id, [FromBody]Producto value)
 {
     var data = Context.Producto.FirstOrDefault(o => o.IdProducto == id);
     if (data == null)
         return NotFound();
     data.Existencias = value.Existencias;
     data.Nombre = value.Nombre;
     data.Precio = value.Precio;

     try
     {
         Context.SaveChanges();
     }
     catch (Exception e)
     {
         return BadRequest(e.Message);
     }

     return Ok(data);

 }
```

7º Agregamos el siguiente método para Borrar un producto.

```
      [HttpDelete("{id}")]
       public IActionResult Delete(int id)
       {

           var data = Context.Producto.Find(id);

           if (data == null)
               return NotFound();

           Context.Producto.Remove(data);

           try
           {
               Context.SaveChanges();
           }
           catch (Exception e)
           {
               return BadRequest(e.Message);
           }

           return Ok();
       }
```

8º Podemos probar el controlador, y repetir esto para el resto de las tablas. Crearíamos un controlador por tabla.

Agregar seguridad con JWT

Con esta tarea, vamos a agregar autenticación basada en Jwt a nuestro api. Para ello lo primero será agregar los siguientes nugets.

```
Microsoft.AspNetCore.Authentication.JwtBearer
```

1º Crear el modelo de usuarios. Creamos una clase Usuario, que será la encargada de persistir los datos de los usuarios.

```
  public class Usuario
    {
        [Key]
        public int IdUsuario { get; set; }
        public string Login { get; set; }
        public string Password { get; set; }

    }
```

2º Agregamos la clase usuario al DbContext. Agregando el siguiente código.

```
  public DbSet<Usuario> Usuario { get; set; }
```

3º Creamos un Helper para transformer datos a Sha1, para ello nos creamos una clase llamada Sha1, con el siguiente código.

```
public class Sha1
    {
        public static string GetSha1(string data)
        {
            if(string.IsNullOrWhiteSpace(data))
                throw new ApplicationException("La cadena no puede ser nula");

            var buffer = Encoding.UTF8.GetBytes(data);

            var sha1 = System.Security.Cryptography.SHA1.Create();

            var hash = sha1.ComputeHash(buffer);

            return BitConverter.ToString(hash).Replace("-","").ToLower();
        }
    }
```

4º Ejecutar las migrations. Desde la consola agregamos una nueva migration y actualizamos la base de datos.

```
Dotnet ef migrations add usuarios
Dotnet ef database update
```

5º Creamos una clase para manejar las opciones de emisión de los tokens, para ello agregamos la clase JwtIssuerOptions, y agregamos el siguiente código.

```
public class JwtIssuerOptions
    {

            public string Issuer { get; set; }
            public string Subject { get; set; }
            public string Audience { get; set; }
            public DateTime NotBefore { get; set; } = DateTime.UtcNow;
            public DateTime IssuedAt { get; set; } = DateTime.UtcNow;
            public TimeSpan ValidFor { get; set; } = TimeSpan.FromMinutes(5);
            public DateTime Expiration { get; set; }
            public Func<Task<string>> JtiGenerator =>
                () => Task.FromResult(Guid.NewGuid().ToString());
            public SigningCredentials SigningCredentials { get; set; }
```

```
            public void UpdateToken()
            {
                IssuedAt = DateTime.UtcNow;
                NotBefore = IssuedAt;
                Expiration = IssuedAt.Add(ValidFor);
            }
        }
```

6º Agregamos un nuevo controlador para autenticar a los usuarios y devolver los tokens que serán validados.

```
[Produces("application/json")]
    [Route("api/Jwt")]
    public class JwtController : Controller
    {
    }
```

7º Creamos en el controller un objeto de Tipo JwtIssuerOptions y la referencia al context. Estos serán inicializados en el controlador. Adicionalmente vamos a agregar un método para lanzar una excepción en caso de que las opciones no se hayan inicializado.

```
  private readonly JwtIssuerOptions _jwtOptions;
        private readonly JsonSerializerSettings _serializerSettings;
        public ProductosContext Context { get; set; }

        public JwtController(IOptions<JwtIssuerOptions> jwtOptions, ProductosContext
_context)
        {
            _jwtOptions = jwtOptions.Value;
            ThrowIfInvalidOptions(_jwtOptions);


            _serializerSettings = new JsonSerializerSettings
            {
                Formatting = Formatting.Indented
            };
            Context = _context;
        }
private static void ThrowIfInvalidOptions(JwtIssuerOptions options)
        {
            if (options == null) throw new ArgumentNullException(nameof(options));

            if (options.ValidFor <= TimeSpan.Zero)
            {
                throw new ArgumentException("No puede ser cero.",
nameof(JwtIssuerOptions.ValidFor));
            }
```

```
            if (options.SigningCredentials == null)
            {
                throw new ArgumentNullException(nameof(JwtIssuerOptions.SigningCredentials));
            }
```

```
            if (options.JtiGenerator == null)
            {
                throw new ArgumentNullException(nameof(JwtIssuerOptions.JtiGenerator));
            }
        }
```

8º Creamos un método que valide los datos recibidos y cree un claim simple con la información del usuario.

```
  private Task<ClaimsIdentity> GetClaimsIdentity(Usuario user)
        {

            var us = Context.Usuario.FirstOrDefault(o => o.Login == user.Login && o.Password
== Sha1.GetSha1(user.Password));

            if (us != null)
            {
                return Task.FromResult(new ClaimsIdentity(
                  new GenericIdentity(user.Login, "Token"),
                  new Claim[] { }));
            }

            // Credentials are invalid, or account doesn't exist
            return Task.FromResult<ClaimsIdentity>(null);
        }
```

9º Creamos en el controlador un método para recibir las peticiones de autenticar, y que devuelva el token una vez autenticado.

```
public async Task<IActionResult> Post([FromBody] Usuario usuario)
        {
            var identity = await GetClaimsIdentity(usuario);
            if (identity == null)
            {
                return BadRequest("Credenciales incorrectas");
            }

            var claims = new List<Claim>()
      {
        new Claim(JwtRegisteredClaimNames.Sub, usuario.Login),
        new Claim(JwtRegisteredClaimNames.Jti, await _jwtOptions.JtiGenerator()),
        new Claim(JwtRegisteredClaimNames.Iat,
                  ToUnixEpochDate(_jwtOptions.IssuedAt).ToString(),
                  ClaimValueTypes.Integer64),

      };
            _jwtOptions.UpdateToken();
            // Create the JWT security token and encode it.
            var jwt = new JwtSecurityToken(
```

```
                issuer: _jwtOptions.Issuer,
                audience: _jwtOptions.Audience,
                claims: claims,
                notBefore: DateTime.UtcNow,
                expires: _jwtOptions.Expiration,
                signingCredentials: _jwtOptions.SigningCredentials);
```

```
            var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

            // Serialize and return the response
            var response = new
            {
                access_token = encodedJwt,
                expires_in = (int)_jwtOptions.ValidFor.TotalSeconds
            };

            var json = JsonConvert.SerializeObject(response, _serializerSettings);
            return new OkObjectResult(json);
        }


private static long ToUnixEpochDate(DateTime date)
             => (long)Math.Round((date.ToUniversalTime() -
                                  new DateTimeOffset(1970, 1, 1, 0, 0, 0, TimeSpan.Zero))
                                 .TotalSeconds);
```

10º Al appsettings.json vamos a agregar la configuración del emisor del token.

```
"JwtIssuerOptions": {
    "Issuer": "ApiDeLuis",
    "Audience": "http://localhost:5000"
  },
```

11º En el fichero startup.cs debemos definir cual es la clave para cifrar el token.

```
  private const string SecretKey = "ClaveParaGenerarLosTokens";
        private readonly SymmetricSecurityKey _signingKey = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(SecretKey));
```

12º En el método ConfigureServices, vamos a configurar la autenticación usando jwt. El método completo quedaría de la siguiente manera.

```
public void ConfigureServices(IServiceCollection services)
        {
            var jwtAppSettingOptions = Configuration.GetSection(nameof(JwtIssuerOptions));
            var tokenValidationParameters = new TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidIssuer = jwtAppSettingOptions[nameof(JwtIssuerOptions.Issuer)],

                ValidateAudience = true,
                ValidAudience = jwtAppSettingOptions[nameof(JwtIssuerOptions.Audience)],
```

```
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = _signingKey,

                RequireExpirationTime = true,
                ValidateLifetime = true,

                ClockSkew = TimeSpan.Zero
            };
```

```
            var conexion = Configuration.GetConnectionString("DefaultConnection");
            // Add framework services.
            services.AddMvc(c =>
            {
                var policy = new AuthorizationPolicyBuilder()
                    .RequireAuthenticatedUser()
                    .Build();
                c.Filters.Add(new AuthorizeFilter(policy));

            });
            services.AddAuthorization();

            services.Configure<JwtIssuerOptions>(options =>
            {
                options.Issuer = jwtAppSettingOptions[nameof(JwtIssuerOptions.Issuer)];
                options.Audience = jwtAppSettingOptions[nameof(JwtIssuerOptions.Audience)];
                options.SigningCredentials = new SigningCredentials(_signingKey,
SecurityAlgorithms.HmacSha256);
            });



            services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                .AddJwtBearer(options => {


                    options.TokenValidationParameters = tokenValidationParameters;
                    options.RequireHttpsMetadata = false;


                });

            services.AddCors();


            services.AddDbContext<ProductosContext>(options =>
options.UseSqlServer(conexion));
        }
```

13º En el método Configure definimos como va a funcionar cors en nuestro api.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,ProductosContext
context)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseCors(builder =>
```

```
                builder.WithOrigins("*").AllowAnyHeader().AllowAnyMethod());
            app.UseAuthentication();

            app.UseMvc();
            DbInitializer.Initialize(context);
        }
```

14º Con esto el api estaría lista para probarla y publicarla.