Creating an Entity Framework in C# involves several steps to set up and use the ORM (Object-Relational Mapping) framework to interact with a database.

Let's create a simple Entity Framework project that interacts with a SQL Server database.

Step 1: Create a New Project

1. Open Visual Studio.
2. Click on "Create a new project."
3. Select "Web API (ASP.NET Core)" template under "Create a new project."
4. Give your project a name and choose a location to save it.
5. OpenAPI & Controllers should be checked
6. Click "Create."
7. Remove unwanted files from project (eg. WeatherForecastController).

Step 2: Install Entity Framework Core

1. In the Solution Explorer, right-click on your project name and select "Manage NuGet Packages."
2. Search for "Microsoft.EntityFrameworkCore" and install the latest version.
3. Install the appropriate database provider package. For SQL Server, search for "Microsoft.EntityFrameworkCore.SqlServer" and install it.
4. Also, install "Microsoft.EntityFrameworkCore.Tools".

Step 3: Create Model Classes

1. Right-click on your project name in Solution Explorer.
2. Add a new folder called "Models."
3. Inside the "Models" folder, add your C# classes that represent your database tables. For example, create a class named "Product" with properties like "Id," "Name," and "Description."

Step 4: Create the DbContext Class

1. Right-click on your project name in Solution Explorer.
2. Add a new folder called "DbContext."
3. Add a new class to your project (inside "DbContext" folder). Name it something like "ProductDbContext.cs."
4. Inherit from `DbContext` class and define your DbSet properties for each model class. Here's an example:

```csharp
using Microsoft.EntityFrameworkCore;

namespace YourNamespace
{
    public class ProductDbContext : DbContext
    {
        public DbSet<Product> Products { get; set; }

        public ProductDbContext(DbContextOptions options) : base(options)
        {

        }
    }
}
```

Step 5: Configure Connection String
1. Open "appsettings.json" in your project.
2. Add your connection string under `"ConnectionStrings"`:

```json
{
  "ConnectionStrings": {
    "ProductDbConnectionString":
"server=localdbname;database=DatabaseName;Trusted_Connection=true"
  }
}
```

Step 6: Configure your database connection string in Program.cs
```csharp
builder.Services.AddDbContext<ProductDbContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("ProductDb
ConnectionString")));
```

Step 7: Open package manager console (Tools -> Nuget package manager -> Package manager console) and run following CLI commands

1. add-migration 'migration-name' (it will create two files - c# file & snapshot) (c# file will contain all details about database) (snapshot will contain migration history)
2. update-database (it will first find the latest migration file & convert all code to sql)

Step 8: Create repository folder

1. Create a interface(eg. IProductRepository)

```csharp
 public interface IProductRepository
    {
        Task<IReadOnlyList<Product>> GetAllProductsAsyc();
    }
```

2. Create a class that implements the interface(eg. ProductRepository which implements IProductRepository)

```csharp
public class ProductRepository : IProductRepository
    {
        private readonly ProductDbContext _context;

        public ProductRepository(ProductDbContext context)
        {
            _context = context;
        }
        public async Task<IReadOnlyList<Product>> GetAllProductsAsyc()
        {
            return await _context.Products.ToListAsync();
        }
    }
```

Step 9: Use dependency injection to inject the necessary repository.

```csharp
public class ProductController : ControllerBase
    {
        private readonly IProductRepository _productRepository;

        public ProductController(IProductRepository productRepository)
        {
            _productRepository = productRepository;
        }
     }
```

Step 10: Add necessary CRUD operations to the controller.

```csharp
    [HttpGet]
    public async Task<ActionResult<IReadOnlyList<Product>>> Get()
    {
        var products = await _productRepository.GetAllProductsAsyc();

        return products.Count == 0 ? NotFound("No products to display.")
: Ok(products);
    }
```

Step 11: Add the Service to the Program.cs

```csharp
builder.Services.AddScoped<IProductRepository, ProductRepository>();
````

Step 12: To add your own Configuration

1.Right-click on the DBContext folder, inside this create a Configuration folder

2.Inside Configuration folder, add the class (eg: ProductConfiguration) that implements IEntityTypeConfiguration

```csharp
public class ProductConfiguration : IEntityTypeConfiguration<Product>
{
      public void Configure(EntityTypeBuilder<Product> builder)
      {
            builder.HasKey(x => x.Id);
            builder.Property(x=>x.Id).UseIdentityColumn();

            builder.Property(x=>x.ProductName)
                  .IsRequired()
                  .HasMaxLength(20);

            builder.Property(x=>x.ProductDescription)
                  .IsRequired()
                  .HasMaxLength(20);
      }
}
```

3.Inside DBContext class (eg. ProductDBContext) override the OnModelCreating method

```csharp
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
     modelBuilder.ApplyConfiguration(new ProductConfiguration());
}
```

Step 13: Run the Application

1. Press F5 or click the "Start" button to run your application.
2. Check the console output to see the results of your database interactions.

Congratulations! You've created a basic Entity Framework setup in C# to interact with a SQL Server database.