3 Onion Architecture Implementation

In this chapter, we are going to talk about the Onion architecture, its layers, and the advantages of using it. We will learn how to create different layers in our application to separate the different application parts and improve the application's maintainability and testability.

That said, we are going to create a database model and transfer it to the MSSQL database by using the code first approach. So, we are going to learn how to create entities (model classes), how to work with the DbContext class, and how to use migrations to transfer our created database model to the real database. Of course, it is not enough to just create a database model and transfer it to the database. We need to use it as well, and for that, we will create a Repository pattern as a data access layer.

With the Repository pattern, we create an abstraction layer between the data access and the business logic layer of an application. By using it, we are promoting a more loosely coupled approach to access our data in the database.

Also, our code becomes cleaner, easier to maintain, and reusable. Data access logic is stored in a separate class, or sets of classes called a repository, with the responsibility of persisting the application’s business model.

Additionally, we are going to create a Service layer to extract all the business logic from our controllers, thus making the presentation layer and the controllers clean and easy to maintain.

So, let’s start with the Onion architecture explanation.

3.1 About Onion Architecture

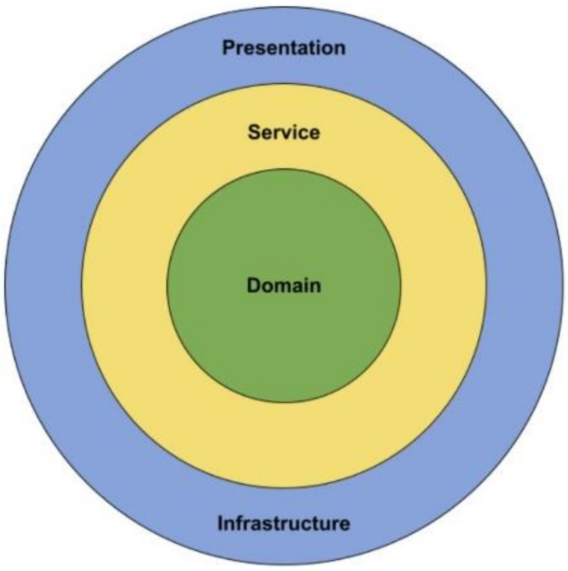
The Onion architecture is a form of layered architecture and we can visualize these layers as concentric circles. Hence the name Onion architecture. The Onion architecture was first introduced by Jeffrey Palermo, to overcome the issues of the traditional N-layered architecture approach.

There are multiple ways that we can split the onion, but we are going to choose the following approach where we are going to split the architecture into 4 layers:

- Domain Layer
- Service Layer
- Infrastructure Layer
- Presentation Layer

Conceptually, we can consider that the Infrastructure and Presentation layers are on the same level of the hierarchy.

Now, let us go ahead and look at each layer with more detail to see why we are introducing it and what we are going to create inside of that layer:



We can see all the different layers that we are going to build in our project.

3.1.1 Advantages of the Onion Architecture

Let us take a look at what are the advantages of Onion architecture, and why we would want to implement it in our projects.

All of the layers interact with each other strictly through the interfaces defined in the layers below. The flow of dependencies is towards the core of the Onion. We will explain why this is important in the next section.

Using dependency inversion throughout the project, depending on abstractions (interfaces) and not the implementations, allows us to switch out the implementation at runtime transparently. We are depending on abstractions at compile-time, which gives us strict contracts to work with, and we are being provided with the implementation at runtime.

Testability is very high with the Onion architecture because everything depends on abstractions. The abstractions can be easily mocked with a mocking library such as Moq. We can write business logic without concern about any of the implementation details. If we need anything from an external system or service, we can just create an interface for it and consume it. We do not have to worry about how it will be implemented. The higher layers of the Onion will take care of implementing that interface transparently.

3.1.2 Flow of Dependencies

The main idea behind the Onion architecture is the flow of dependencies, or rather how the layers interact with each other. The deeper the layer resides inside the Onion, the fewer dependencies it has.

The Domain layer does not have any direct dependencies on the outside layers. It is isolated, in a way, from the outside world. The outer layers are all allowed to reference the layers that are directly below them in the hierarchy.

We can conclude that all the dependencies in the Onion architecture flow inwards. But we should ask ourselves, why is this important?

The flow of dependencies dictates what a certain layer in the Onion architecture can do. Because it depends on the layers below it in the hierarchy, it can only call the methods that are exposed by the lower layers.

We can use lower layers of the Onion architecture to define contracts or interfaces. The outer layers of the architecture implement these interfaces. This means that in the Domain layer, we are not concerning ourselves with infrastructure details such as the database or external services.

Using this approach, we can encapsulate all of the rich business logic in the Domain and Service layers without ever having to know any implementation details. In the Service layer, we are going to depend only on the interfaces that are defined by the layer below, which is the Domain layer.

So, after all the theory, we can continue with our project implementation. Let's start with the models and the `Entities` project.

3.2 Creating Models

Using the example from the second chapter of this book, we are going to extract a new Class Library project named `Entities`.

Inside it, we are going to create a folder named `Models`, which will contain all the model classes (entities). Entities represent classes that Entity Framework Core uses to map our database model with the tables from the database. The properties from entity classes will be mapped to the database columns.

So, in the Models folder we are going to create two classes and modify them:

```
public class Company
{
    [Column("CompanyId")]
    public Guid Id { get; set; }

    [Required(ErrorMessage = "Company name is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Name is 60 characters.")]
    public string? Name { get; set; }

    [Required(ErrorMessage = "Company address is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Address is 60 characters")]
    public string? Address { get; set; }

    public string? Country { get; set; }

    public ICollection<Employee>? Employees { get; set; }
}

public class Employee
{
    [Column("EmployeeId")]
    public Guid Id { get; set; }

    [Required(ErrorMessage = "Employee name is a required field.")]
    [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
    public string? Name { get; set; }

    [Required(ErrorMessage = "Age is a required field.")]
    public int Age { get; set; }

    [Required(ErrorMessage = "Position is a required field.")]
    [MaxLength(20, ErrorMessage = "Maximum length for the Position is 20 
characters.")]
    public string? Position { get; set; }

    [ForeignKey(nameof(Company))]
    public Guid CompanyId { get; set; }
    public Company? Company { get; set; }
}
```

We have created two classes: the Company and Employee. Those classes contain the properties which Entity Framework Core is going to map to the columns in our tables in the database. But not all the properties will be mapped as columns. The last property of the Company class (Employees) and the last property of the Employee class (Company) are navigational properties; these properties serve the purpose of defining the relationship between our models.

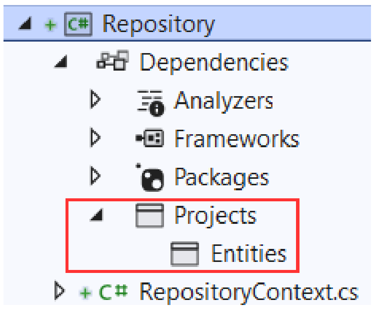
We can see several attributes in our entities. The `[Column]` attribute will specify that the `Id` property is going to be mapped with a different name in the database. The `[Required]` and `[MaxLength]` properties are here for validation purposes. The first one declares the property as mandatory and the second one defines its maximum length.

Once we transfer our database model to the real database, we are going to see how all these validation attributes and navigational properties affect the column definitions.

3.3 Context Class and the Database Connection

Before we start with the context class creation, we have to create another .NET Class Library and name it Repository. We are going to use this project for the database context and repository implementation.

Now, let's create the context class, which will be a middleware component for communication with the database. It must inherit from the Entity Framework Core's `DbContext` class and it consists of `DbSet` properties, which EF Core is going to use for the communication with the database. Because we are working with the DBContext class, we need to install the `Microsoft.EntityFrameworkCore` package in the `Repository` project. Also, we are going to reference the `Entities` project from the `Repository` project:



Then, let's navigate to the root of the `Repository` project and create the `RepositoryContext` class:

```
public class RepositoryContext : DbContext
{
    public RepositoryContext(DbContextOptions options)
        : base(options)
    {
    }
    public DbSet<Company>? Companies { get; set; }
    public DbSet<Employee>? Employees { get; set; }
}
```

After the class modification, let’s open the **appsettings.json** file, in the main project, and add the connection string named **sqlconnection**:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated Security=true"
  },
  "AllowedHosts": "*"
}
```

It is quite important to have the JSON object with the **ConnectionStrings** name in our **appsettings.json** file, and soon you will see why.

But first, we have to add the Repository project’s reference into the main project.

Then, let’s create a new **ContextFactory** folder **in the main project** and inside it a new **RepositoryContextFactory** class. Since our **RepositoryContext** class is in a **Repository** project and not in the main one, this class will help our application create a derived DbContext instance during the design time which will help us with our migrations:

```
public class RepositoryContextFactory : IDesignTimeDbContextFactory<RepositoryContext>
{
        public RepositoryContext CreateDbContext(string[] args)
        {
                var configuration = new ConfigurationBuilder()
                        .SetBasePath(Directory.GetCurrentDirectory())
                        .AddJsonFile("appsettings.json")
                        .Build();

                var builder = new DbContextOptionsBuilder<RepositoryContext>()

        .UseSqlServer(configuration.GetConnectionString("sqlConnection"));

                return new RepositoryContext(builder.Options);
        }
}
```

We are using the `IDesignTimeDbContextFactory<out TContext>` interface that allows design-time services to discover implementations of this interface. Of course, the `TContext` parameter is our `RepositoryContext` class.

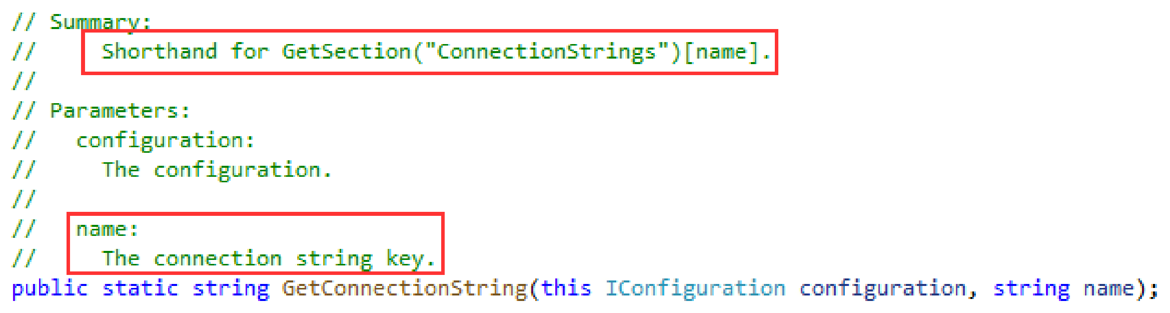
For this, we need to add two using directives:

```
using Microsoft.EntityFrameworkCore.Design;
using Repository;
```

Then, we have to implement this interface with the `CreateDbContext` method. Inside it, we create the `configuration` variable of the `IConfigurationRoot` type and specify the appsettings file, we want to use. With its help, we can use the `GetConnectionString` method to access the connection string from the `appsettings.json` file. Moreover, to be able to use the `UseSqlServer` method, we need to install the `Microsoft.EntityFrameworkCore.SqlServer` package in the main project and add one more using directive:

```
using Microsoft.EntityFrameworkCore;
```

If we navigate to the `GetConnectionString` method definition, we will see that it is an extension method that uses the `ConnectionStrings` name from the `appsettings.json` file to fetch the connection string by the provided key:



```
// Summary:
//     Shorthand for GetSection("ConnectionStrings")[name].
//
// Parameters:
//   configuration:
//     The configuration.
//
//   name:
//     The connection string key.
public static string GetConnectionString(this IConfiguration configuration, string name);
```

Finally, in the `CreateDbContext` method, we return a new instance of our RepositoryContext class with provided options.

3.4 Migration and Initial Data Seed

Migration is a standard process of creating and updating the database from our application. Since we are finished with the database model creation, we can transfer that model to the real database. But we need to modify our `CreateDbContext` method first:

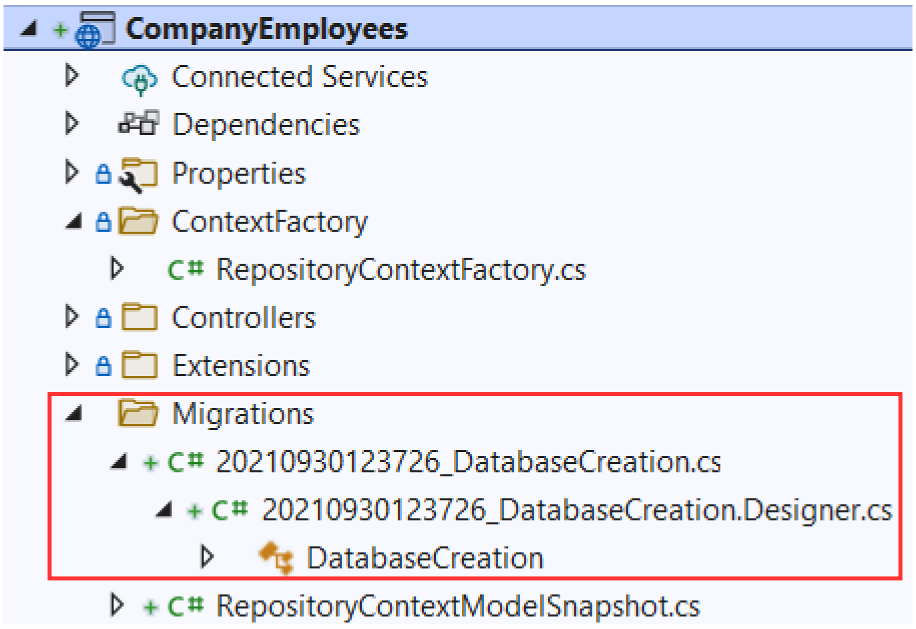
```
var builder = new DbContextOptionsBuilder<RepositoryContext>()
       .UseSqlServer(configuration.GetConnectionString("sqlConnection"),
              b => b.MigrationsAssembly("CompanyEmployees"));
```

We have to make this change because migration assembly is not in our main project, but in the `Repository` project. So, we've just changed the project for the migration assembly.

Before we execute our migration commands, we have to install an additional ef core library: `Microsoft.EntityFrameworkCore.Tools`

Now, let's open the Package Manager Console window and create our first migration: `PM> Add-Migration DatabaseCreation`

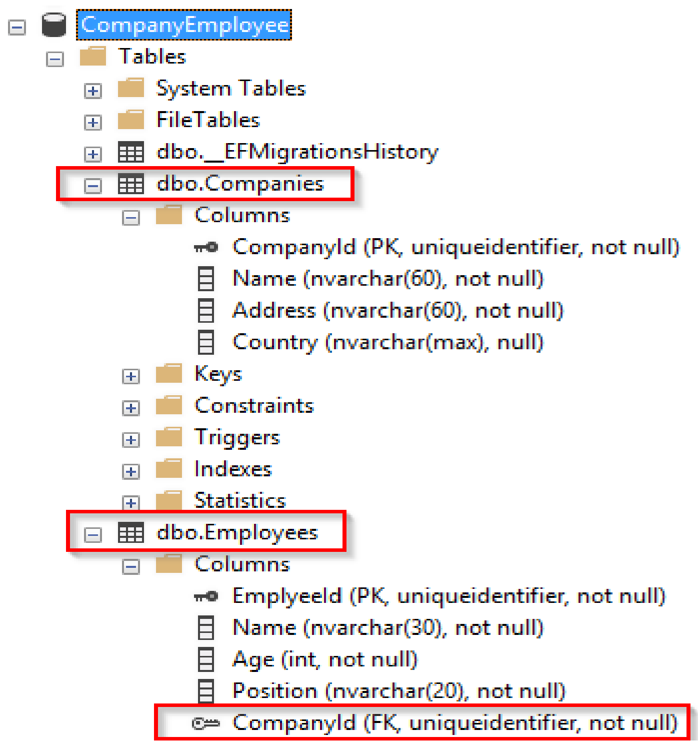
With this command, we are creating migration files and we can find them in the `Migrations` folder in our main project:



```
CompanyEmployees
  Connected Services
  Dependencies
  Properties
  ContextFactory
    RepositoryContextFactory.cs
  Controllers
  Extensions
  Migrations
    20210930123726_DatabaseCreation.cs
      20210930123726_DatabaseCreation.Designer.cs
        DatabaseCreation
    RepositoryContextModelSnapshot.cs
```

With those files in place, we can apply migration: `PM> Update-Database`

Excellent. We can inspect our database now:


```
CompanyEmployee
  Tables
    System Tables
    FileTables
    dbo.__EFMigrationsHistory
    dbo.Companies
      Columns
        CompanyId (PK, uniqueidentifier, not null)
        Name (nvarchar(60), not null)
        Address (nvarchar(60), not null)
        Country (nvarchar(max), null)
      Keys
      Constraints
      Triggers
      Indexes
      Statistics
    dbo.Employees
      Columns
        EmplyeeId (PK, uniqueidentifier, not null)
        Name (nvarchar(30), not null)
        Age (int, not null)
        Position (nvarchar(20), not null)
        CompanyId (FK, uniqueidentifier, not null)
```

Once we have the database and tables created, we should populate them with some initial data. To do that, we are going to create another folder in the `Repository` project called `Configuration` and add the `CompanyConfiguration` class:

```
public class CompanyConfiguration : IEntityTypeConfiguration<Company>
{
    public void Configure(EntityTypeBuilder<Company> builder)
    {
        builder.HasData
        (
            new Company
            {
                Id = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870"),
                Name = "IT_Solutions Ltd",
                Address = "583 Wall Dr. Gwynn Oak, MD 21207",
                Country = "USA"
            },
            new Company
            {
                Id = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3"),
                Name = "Admin_Solutions Ltd",
                Address = "312 Forest Avenue, BF 923",
                Country = "USA"
            }
        );
    }
 }
```

Let's do the same thing for the **EmployeeConfiguration** class:

```
public class EmployeeConfiguration : IEntityTypeConfiguration<Employee>
{
    public void Configure(EntityTypeBuilder<Employee> builder)
    {
        builder.HasData
        (
            new Employee
            {
                Id = new Guid("80abbca8-664d-4b20-b5de-024705497d4a"),
                Name = "Sam Raiden",
                Age = 26,
                Position = "Software developer",
                CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
            },
            new Employee
            {
                Id = new Guid("86dba8c0-d178-41e7-938c-ed49778fb52a"),
                Name = "Jana McLeaf",
                Age = 30,
                Position = "Software developer",
                CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
            },
            new Employee
            {
                 Id = new Guid("021ca3c1-0deb-4afd-ae94-2159a8479811"),
                 Name = "Kane Miller",
                 Age = 35,
                 Position = "Administrator",
                 CompanyId = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3")
             }
        );
    }
}
```

To invoke this configuration, we have to change the **RepositoryContext** class:

```
public class RepositoryContext: DbContext
{
    public RepositoryContext(DbContextOptions options)
    : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new CompanyConfiguration());
        modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
    }

    public DbSet<Company> Companies { get; set; }
    public DbSet<Employee> Employees { get; set; }
}
```

Now, we can create and apply another migration to seed these data to the database:

```
PM> Add-Migration InitialData

PM> Update-Database
```

This will transfer all the data from our configuration files to the respective tables.

3.5 Repository Pattern Logic

After establishing a connection to the database and creating one, it's time to create a generic repository that will provide us with the CRUD methods. As a result, all the methods can be called upon any repository class in our project.

Furthermore, creating the generic repository and repository classes that use that generic repository is not going to be the final step. We will go a step further and create a wrapper class around repository classes and inject it as a service in a dependency injection container.

Consequently, we will be able to instantiate this class once and then call any repository class we need inside any of our controllers.
The advantages of this approach will become clearer once we use it in the project.

That said, let’s start by creating an interface for the repository inside the `Contracts` project:

```
public interface IRepositoryBase<T>
{
    IQueryable<T> FindAll(bool trackChanges);
    IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression,
    bool trackChanges);
    void Create(T entity);
    void Update(T entity);
    void Delete(T entity);
}
```

Right after the interface creation, we are going to reference **Contracts** inside the **Repository** project. Also, in the **Repository** project, we are going to create an abstract class **RepositoryBase** — which is going to implement the **IRepositoryBase** interface:

```
public abstract class RepositoryBase<T> : IRepositoryBase<T> where T : class
{
    protected RepositoryContext RepositoryContext;

    public RepositoryBase(RepositoryContext repositoryContext)
            => RepositoryContext = repositoryContext;

    public IQueryable<T> FindAll(bool trackChanges) =>
        !trackChanges ?
          RepositoryContext.Set<T>()
            .AsNoTracking() :
          RepositoryContext.Set<T>();

    public IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression,
    bool trackChanges) =>
        !trackChanges ?
          RepositoryContext.Set<T>()
            .Where(expression)
            .AsNoTracking() :
          RepositoryContext.Set<T>()
            .Where(expression);

    public void Create(T entity) => RepositoryContext.Set<T>().Add(entity);

    public void Update(T entity) => RepositoryContext.Set<T>().Update(entity);

    public void Delete(T entity) => RepositoryContext.Set<T>().Remove(entity);
}
```

This abstract class as well as the **IRepositoryBase** interface work with the generic type **T**. This type **T** gives even more reusability to the **RepositoryBase** class. That means we don't have to specify the exact model (class) right now for the **RepositoryBase** to work with. We can do that later on.

Moreover, we can see the **trackChanges** parameter. We are going to use it to improve our read-only query performance. When it's set to false, we attach the **AsNoTracking** method to our query to inform EF Core that it doesn't need to track changes for the required entities. This greatly improves the speed of a query.

3.6 Repository User Interfaces and Classes

Now that we have the **RepositoryBase** class, let’s create the user classes that will inherit this abstract class.

By inheriting from the **RepositoryBase** class, they will have access to all the methods from it. Furthermore, every user class will have its interface for additional model-specific methods.

This way, we are separating the logic that is common for all our repository user classes and also specific for every user class itself.

Let’s create the interfaces in the **Contracts** project for the **Company** and **Employee** classes:

```
namespace Contracts
{
    public interface ICompanyRepository
    {
    }
}

namespace Contracts
{
    public interface IEmployeeRepository
    {
    }
}
```

After this, we can create repository user classes in the **Repository** project.

The first thing we are going to do is to create the **CompanyRepository** class:

```
public class CompanyRepository : RepositoryBase<Company>, ICompanyRepository
{
    public CompanyRepository(RepositoryContext repositoryContext)
        : base(repositoryContext)
    {
    }
}
```

And then, the **EmployeeRepository** class:

```
public class EmployeeRepository : RepositoryBase<Employee>, IEmployeeRepository
```

```
{
    public EmployeeRepository(RepositoryContext repositoryContext)
        : base(repositoryContext)
    {
    }
}
```

After these steps, we are finished creating the repository and repository-user classes. But there are still more things to do.

3.7 Creating a Repository Manager

It is quite common for the API to return a response that consists of data from multiple resources; for example, all the companies and just some employees older than 30. In such a case, we would have to instantiate both of our repository classes and fetch data from their resources.

Maybe it's not a problem when we have only two classes, but what if we need the combined logic of five or even more different classes? It would just be too complicated to pull that off.

With that in mind, we are going to create a repository manager class, which will create instances of repository user classes for us and then register them inside the dependency injection container. After that, we can inject it inside our services with constructor injection (supported by ASP.NET Core). With the repository manager class in place, we may call any repository user class we need.

But we are also missing one important part. We have the `Create`, `Update`, and `Delete` methods in the `RepositoryBase` class, but they won't make any change in the database until we call the `SaveChanges` method. Our repository manager class will handle that as well.

That said, let's get to it and create a new interface in the `Contract` project:

```
public interface IRepositoryManager
{
    ICompanyRepository Company { get; }
    IEmployeeRepository Employee { get; }
    void Save();
}
```

And add a new class to the **Repository** project:

```
public sealed class RepositoryManager : IRepositoryManager
{
       private readonly RepositoryContext _repositoryContext;
       private readonly Lazy<ICompanyRepository> _companyRepository;
       private readonly Lazy<IEmployeeRepository> _employeeRepository;

       public RepositoryManager(RepositoryContext repositoryContext)
       {
              _repositoryContext = repositoryContext;
              _companyRepository = new Lazy<ICompanyRepository>(() => new
CompanyRepository(repositoryContext));
              _employeeRepository = new Lazy<IEmployeeRepository>(() => new
EmployeeRepository(repositoryContext));
       }

       public ICompanyRepository Company => _companyRepository.Value;
       public IEmployeeRepository Employee => _employeeRepository.Value;

       public void Save() => _repositoryContext.SaveChanges();
}
```

As you can see, we are creating properties that will expose the concrete repositories and also we have the **Save()** method to be used after all the modifications are finished on a certain object. This is a good practice because now we can, for example, add two companies, modify two employees, and delete one company — all in one action — and then just call the **Save** method once. All the changes will be applied or if something fails, all the changes will be reverted:

```
_repository.Company.Create(company);
_repository.Company.Create(anotherCompany);
_repository.Employee.Update(employee);
_repository.Employee.Update(anotherEmployee);
_repository.Company.Delete(oldCompany);

_repository.Save();
```

The interesting part with the **RepositoryManager** implementation is that we are leveraging the power of the **Lazy** class to ensure the lazy initialization of our repositories. This means that our repository instances are only going to be created when we access them for the first time, and not before that.

After these changes, we need to register our manager class in the main project. So, let's first modify the `ServiceExtensions` class by adding this code:

```
public static void ConfigureRepositoryManager(this IServiceCollection services) =>
    services.AddScoped<IRepositoryManager, RepositoryManager>();
```

And in the `Program` class above the `AddController()` method, we have to add this code:

```
builder.Services.ConfigureRepositoryManager();
```

Excellent.

As soon as we add some methods to the specific repository classes, and add our service layer, we are going to be able to test this logic.

So, we did an excellent job here. The repository layer is prepared and ready to be used to fetch data from the database.

Now, we can continue towards creating a service layer in our application.

3.8 Adding a Service Layer

The Service layer sits right above the Domain layer (the Contracts project is the part of the Domain layer), which means that it has a reference to the Domain layer. The Service layer will be split into two projects, Service.Contracts and Service.

So, let's start with the `Service.Contracts` project creation (.NET Core Class Library) where we will hold the definitions for the service interfaces that are going to encapsulate the main business logic. In the next section, we are going to create a presentation layer and then, we will see the full use of this project.

Once the project is created, we are going to add three interfaces inside it.

`ICompanyService`:

```
public interface ICompanyService
```

```
{
}
```

IEmployeeService:

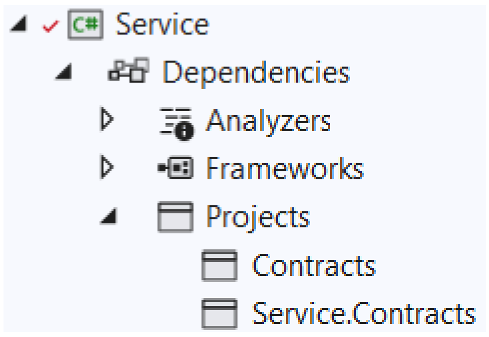
```
public interface IEmployeeService
{
}
```

And IServiceManager:

```
public interface IServiceManager
{
	ICompanyService CompanyService { get; }
	IEmployeeService EmployeeService { get; }
}
```

As you can see, we are following the same pattern as with the repository contracts implementation.

Now, we can create another project, name it **Service**, and reference the **Service.Contracts** and **Contracts** projects inside it:



After that, we are going to create classes that will inherit from the interfaces that reside in the **Service.Contracts** project.

So, let's start with the **CompanyService** class:

```
using Contracts;
using Service.Contracts;

namespace Service
{
	internal sealed class CompanyService : ICompanyService
	{
		private readonly IRepositoryManager _repository;
		private readonly ILoggerManager _logger;

		public CompanyService(IRepositoryManager repository, ILoggerManager
logger)
```

```
		{
			_repository = repository;
			_logger = logger;
		}
	}
}
```

As you can see, our class inherits from the `ICompanyService` interface, and we are injecting the `IRepositoryManager` and `ILoggerManager` interfaces. We are going to use `IRepositoryManager` to access the repository methods from each user repository class (CompanyRepository or EmployeeRepository), and `ILoggerManager` to access the logging methods we've created in the second section of this book.

To continue, let's create a new `EmployeeService` class:

```
using Contracts;
using Service.Contracts;

namespace Service
{
	internal sealed class EmployeeService : IEmployeeService
	{
		private readonly IRepositoryManager _repository;
		private readonly ILoggerManager _logger;

		public EmployeeService(IRepositoryManager repository, ILoggerManager 
logger)
		{
			_repository = repository;
			_logger = logger;
		}
	}
}
```

Finally, we are going to create the `ServiceManager` class:

```
public sealed class ServiceManager : IServiceManager
{
	private readonly Lazy<ICompanyService> _companyService;
	private readonly Lazy<IEmployeeService> _employeeService;

	public ServiceManager(IRepositoryManager repositoryManager, ILoggerManager 
logger)
	{
		_companyService = new Lazy<ICompanyService>(() => new 
CompanyService(repositoryManager, logger));
		_employeeService = new Lazy<IEmployeeService>(() => new 
EmployeeService(repositoryManager, logger));
	}

	public ICompanyService CompanyService => _companyService.Value;
	public IEmployeeService EmployeeService => _employeeService.Value;
```