.Net Core Web API

Modules

- Code/Project Architecture
- Swagger
- JWT Authentication
- Repository Pattern
- Entity Framework
- Database Migrations

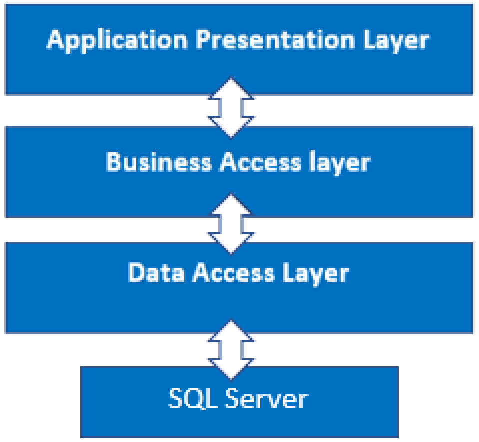
1st Session

- Three Tier Architecture Introduction
- Coding

Three Tier Architecture

Project Structure

We use a three-tier architecture in this project, with a data access layer, a business access layer, and an application presentation layer.



Three Tier Architecture

Presentation Layer

The Presentation layer is the top-most layer of the 3-tier architecture, and its major role is to display the results to the user, or to put it another way, to present the data that we acquire from the business access layer and offer the results to the front-end user.

Business Logic Layer (BLL)

The logic layer interacts with the data access layer and the presentation layer to process the activities that lead to logical decisions and assessments. This layer's primary job is to process data between other layers.

Data Access Layer (DAL)

The main function of this layer is to access and store the data from the database and the process of the data to business access layer data goes to the presentation layer against user request.

Three Tier Architecture

Data Access Layer

In this layer we have the following folders.

- Contracts – In the Contract Folder, we define the interface that performs the desired functionalities with the database
- Data – In the Data folder, we have DB Context Class this class is very important for accessing the data from the database.
- Migrations – The Migration folder contains information on all the migrations we performed during the construction of the project.
- Models – Our application models, which contain entity models of the database.
- Repositories – We add the repositories classes against each model. We write the CRUD function that communicates with the database using the entity framework. We add the repository class that inherits the Interface that is present in Contract folder.

Three Tier Architecture

Business Logic Layer

In this layer we have two folders.

- Interface – In the Inferface folder, we define the interface that performs the desired functionalities of the service.
- Services – We add the services classes against each model. We write the implementation of the interface.

Three Tier Architecture

Application Presentation Layer

ASP.NET Web API is a framework for building HTTP services that can be accessed from any client including browsers and mobile devices. It is an ideal platform for building RESTful applications on the .NET Framework.

Controllers

Web API controller is a class which can be created under the Controllers folder or any other folder under your project's root folder. The name of a controller class must end with "Controller" and it must be derived from System.Web.Http.ApiController class. All the public methods of the controller are called action methods.

Action Methods:
GET/POST/PUT/DELETE

ASP.NET Web API

HTTP Action Methods:

GET - GET is an HTTP method for requesting data from the server. Requests using the HTTP GET method should only fetch data, cannot enclose data in the body of a GET message, and should not have any other effect on data on the server.

POST - The HTTP POST method sends data to the server. The type of the body of the request is indicated.

PUT - The HTTP PUT request method creates a new resource or replaces a representation of the target resource with the request payload.

DELETE – The HTTP Delete request Method deletes the specified resource.

2nd Session

- Entity Framework Core
- Repository Pattern
- DTO
- AutoMapper
- Coding

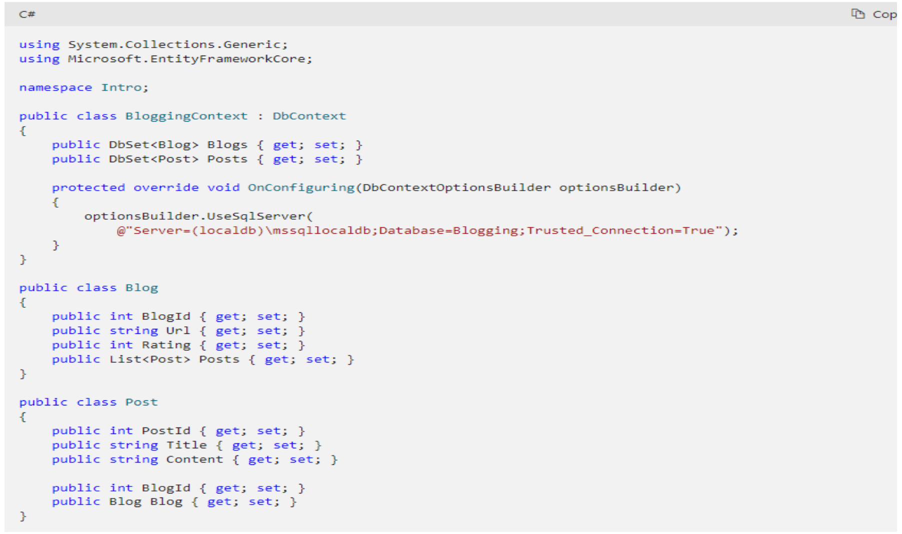
Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.

EF Core can serve as an object-relational mapper (O/RM), which:
- Enables .NET developers to work with a database using .NET objects.
- Eliminates the need for most of the data-access code that typically needs to be written.

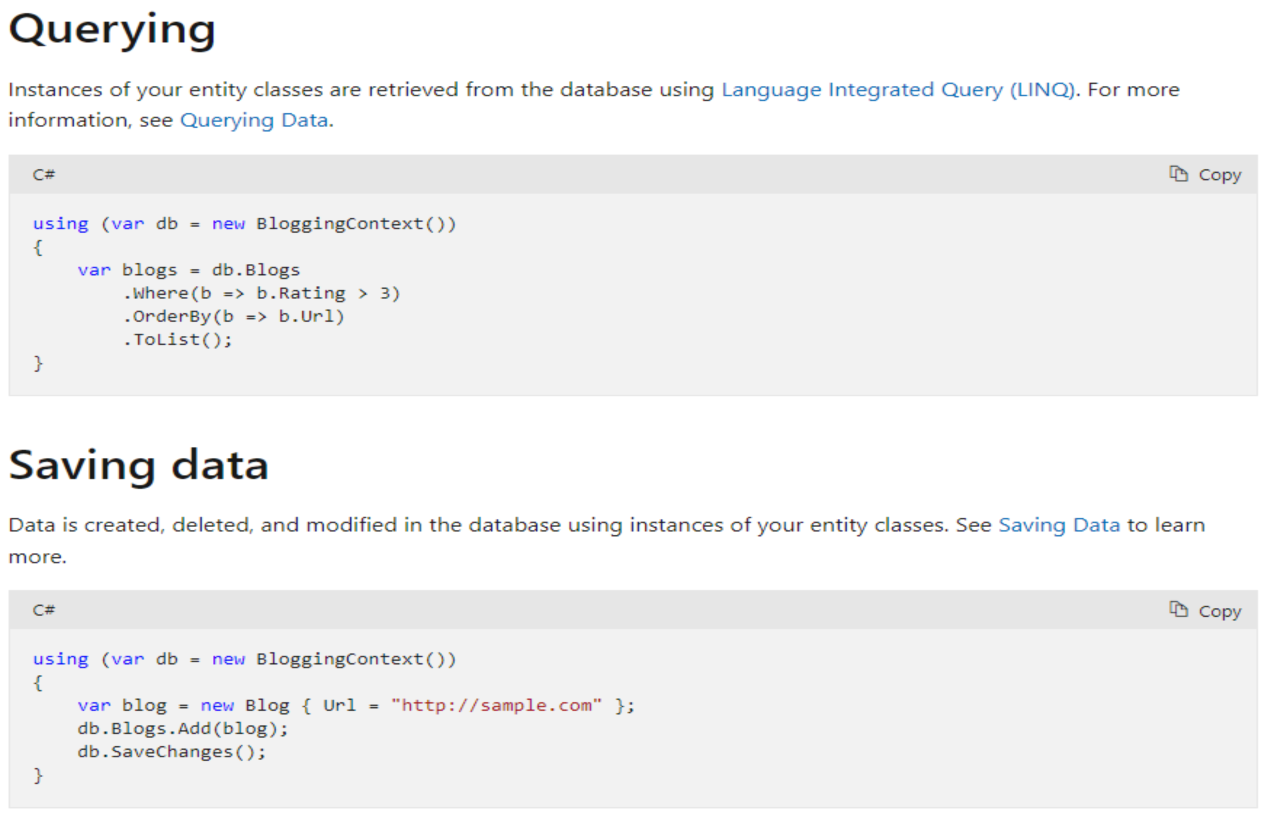
Entity Framework Core

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data.



```csharp
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace Intro;

public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Entity Framework Core



Querying

Instances of your entity classes are retrieved from the database using Language Integrated Query (LINQ). For more information, see Querying Data.

C# Copy

```csharp
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Saving data

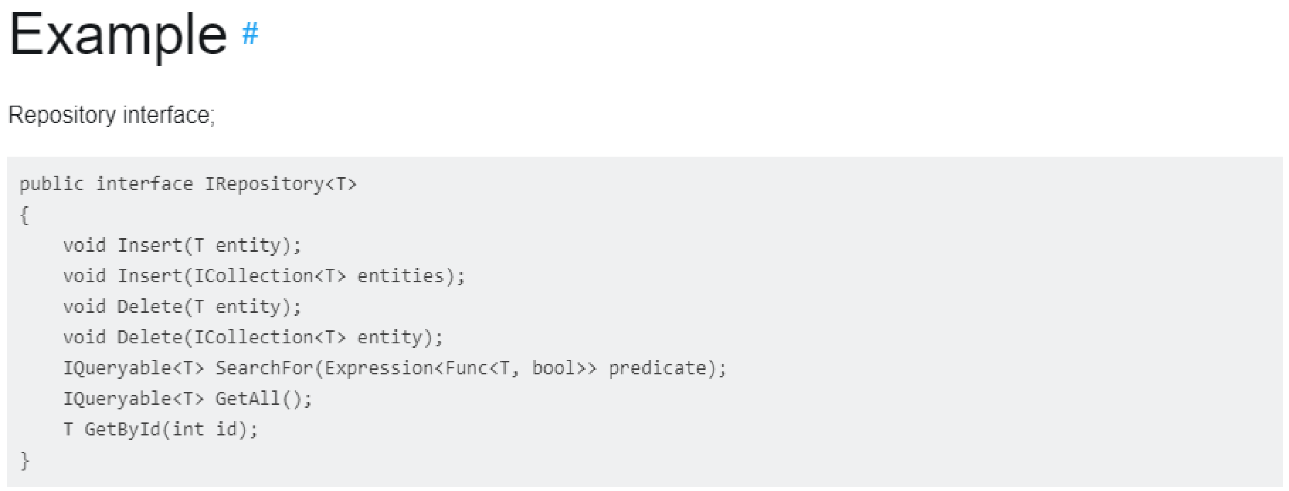
Data is created, deleted, and modified in the database using instances of your entity classes. See Saving Data to learn more.

C# Copy

```csharp
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

Repository Pattern

A repository pattern can be used to encapsulate data storage specific code in designated components. The part of your application, that needs the data will only work with the repositories.

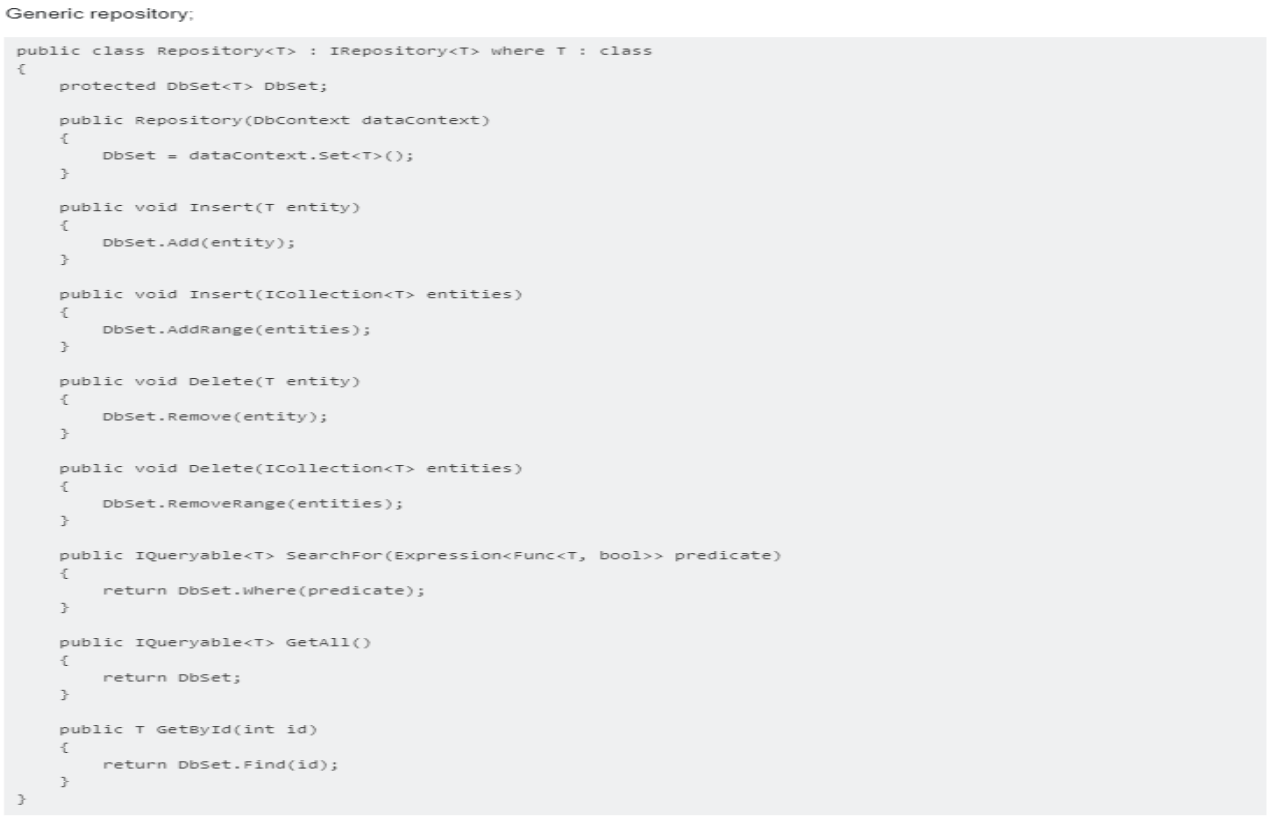


Example

Repository interface;

```
public interface IRepository<T>
{
    void Insert(T entity);
    void Insert(ICollection<T> entities);
    void Delete(T entity);
    void Delete(ICollection<T> entity);
    IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate);
    IQueryable<T> GetAll();
    T GetById(int id);
}
```

Repository Pattern

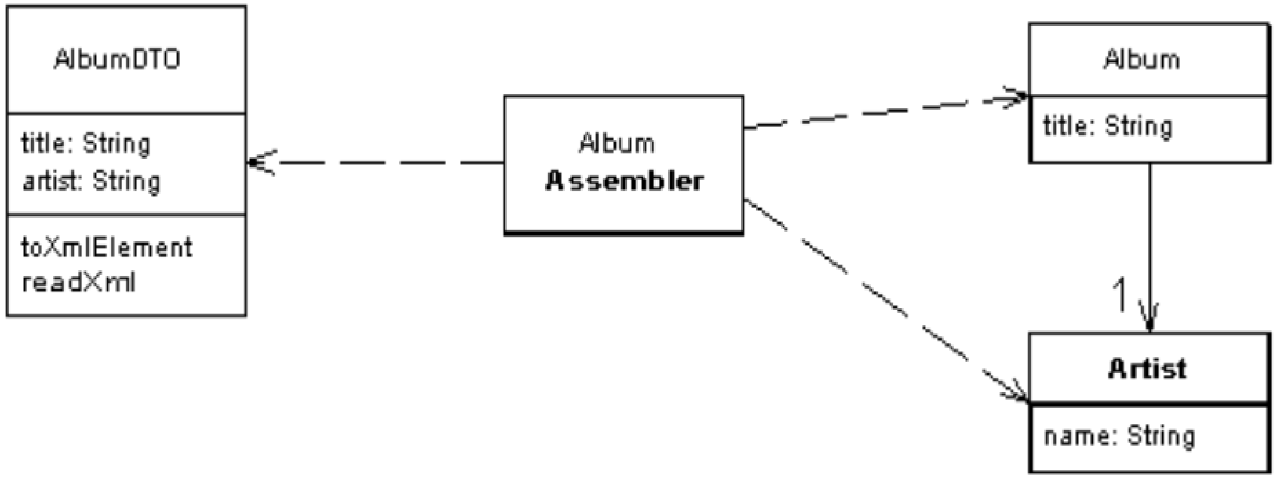


Generic repository;

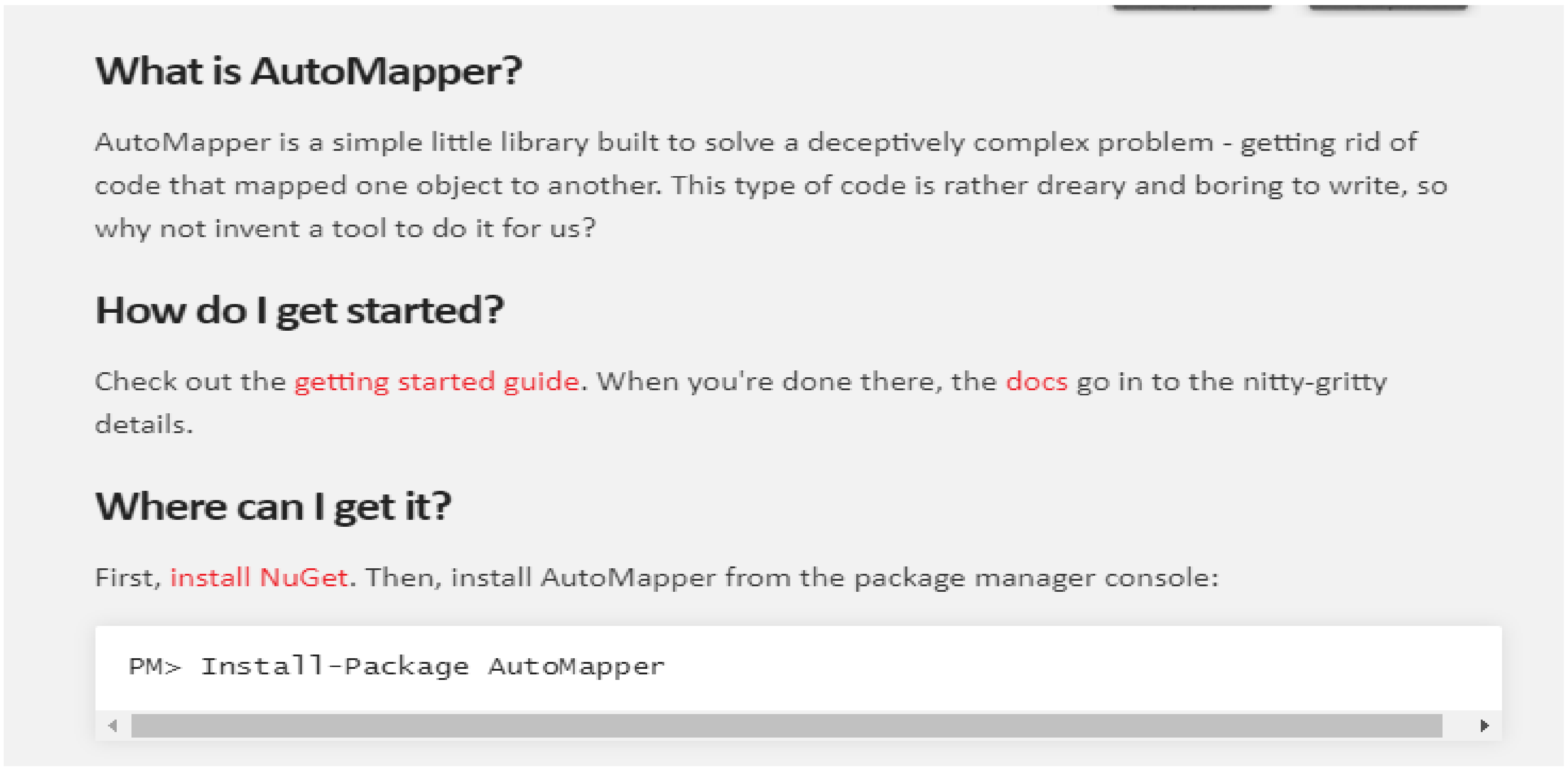
```
public class Repository<T> : IRepository<T> where T : class
{
    protected DbSet<T> DbSet;

    public Repository(DbContext dataContext)
    {
        DbSet = dataContext.Set<T>();
    }

    public void Insert(T entity)
    {
        DbSet.Add(entity);
    }

    public void Insert(ICollection<T> entities)
    {
        DbSet.AddRange(entities);
    }

    public void Delete(T entity)
    {
        DbSet.Remove(entity);
    }

    public void Delete(ICollection<T> entities)
    {
        DbSet.RemoveRange(entities);
    }

    public IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate)
    {
        return DbSet.Where(predicate);
    }

    public IQueryable<T> GetAll()
    {
        return DbSet;
    }

    public T GetById(int id)
    {
        return DbSet.Find(id);
    }
}
```

DTO (Data Transfer Object)

A data transfer object (DTO) is an object that carries data between processes. You can use this technique to facilitate communication between two systems (like an API and your server) without potentially exposing sensitive information.



AutoMapper



What is AutoMapper?

AutoMapper is a simple little library built to solve a deceptively complex problem - getting rid of code that mapped one object to another. This type of code is rather dreary and boring to write, so why not invent a tool to do it for us?

How do I get started?

Check out the getting started guide. When you're done there, the docs go in to the nitty-gritty details.

Where can I get it?

First, install NuGet. Then, install AutoMapper from the package manager console:

```
PM> Install-Package AutoMapper
```