There are 3 approaches through which Entity framework is implemented.

1. Database First
2. Code First
3. Model First

Of these Database first and Code First  are the most used ones. In this article I will not discuss model first approach.

First let us understand what are code first and database first

**Code First Approach**In code first approach we will first create entity classes with properties defined in it. Entity framework will create the database and tables based on the entity classes defined. So database is generated from the code. When the dot net code is run database will get created.

**Advantages**

1. You can create  the database and tables from your business objects.
2. You can specify which related collections are to be eager loaded, or not be serialized at all.
3. Database version control.
4. Good for small applications.

**Disadvantages**

1. You have to write everything related to database in the visual studio code.
2. For stored procedures you have to map stored procedure using Fluent API and write Stored Procedure inside the code.
3. If you want to change anything in the database tables you to make changes in the entity classes in the code file and run the update-database from the  package manager console.
4. Not preferred for Data intensive applications.

**Database First Approach**In this approach Database and tables are created first. Then you create entity Data Model using the created database.

**Advantages**

1. Simple to create the data model
2. Graphical  user interface.
3. Mapping and creation of keys and relationships are easy as you need not have to write any code .
4. Preferred for data intense and large applications

**Disadvantages**

1. Using an existing database to generate a .edmx model file and the associated code models results in a giant pile of auto generated code.
2. When you need to add any functionality to generated model you have to extend the model class generated.

Choosing the appropriate approach is purely based on the applications you are developing.

**Entity Framework 6**

Entity Framework 6 (EF6) is a tried and tested data access technology. It was first released in 2008, as part of .NET Framework 3.5 SP1 and Visual Studio 2008 SP1. Starting with the 4.1 release it has shipped as the [EntityFramework](https://www.nuget.org/packages/EntityFramework/) NuGet package. EF6 runs on the .NET Framework 4.x, which means it runs only on Windows.

EF6 continues to be a supported product, and will continue to see bug fixes and minor improvements.

**Entity Framework Core**

Entity Framework Core (EF Core) is a complete rewrite of EF6 that was first released in 2016. It ships in Nuget packages, the main one being [Microsoft.EntityFrameworkCore](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore/). EF Core is a cross-platform product that can run on .NET Core or .NET Framework.

EF Core was designed to provide a developer experience similar to EF6. Most of the top-level APIs remain the same, so EF Core will feel familiar to developers who have used EF6.

Database : 125.62.198.183

Uid: IT-TekDB

Pwd: u$t@y0ut

Change the connection string in the appsettings.

Now select the DataAccess proeject as a default project

Open the Nuget package Manager Concole

>Add-Migration filename [-ignoreChanges]

>Update-Database

More than one DbContext was found. Specify which one to use. Use the '-Context' parameter for PowerShell commands and the '--context' parameter for dotnet commands.

PM> Update-Database -c SafeHarborDBContext

public class Student

{

public int StudentId;

public string StudentName;

public int CourseId;

public virtual Course Courses { get; set; }

}

public class Course

{

public int CourseId;

public string CourseName;

public string Description;

public ICollection<Student> Students {get;set;}

public ICollection<Lecture> Lectures { get; set; }

}

public class Lecture

{

public int LectureId;

public string LectureName;

public int CourseId;

public virtual Course Courses { get; set; }

}

1) why lecture and student class has common property called CourseId ?

2) see below code

public ICollection<Student> Students {get;set;}

public ICollection<Lecture> Lectures { get; set; }

why they use ICollection ? they can use here list etc ?

3) why Courses property has been declared with virtual keyword in lecture and student class ?

please help me to understand the real use of virtual keyword

public virtual Course Courses { get; set; }

in what kind of scenario we use virtual keyword ? what is advantage ?

virtual keyword is used when people use scaffolding to generate UI from model ?

virtual keyword has any good use when people develop apps with winform ?

1) "why lecture and student class has common property called CourseId ?"

That is a 1:n relationship. A lecture can have a Course and a course can have multiple lectures. A Student can have a course and a course can have multiple students. (The last one does not make much sense, because I would have expected a n:m relationshio here where a student can have multiple courses, too.)

2) "why they use ICollection ? they can use here list etc ?"

When writing code, you should be as generic as possible. Of course you could use a concrete class, too. But then your implementation would be fixed to that. The ICollection just defines the interface so you can use any implementation of ICollection.

3) "why Courses property has been declared with virtual keyword in lecture and student class ?"

The [virtual keyword](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) is used if you want to offer the option that derived classes can override the current implementation. With just the currently given code, it does not make a difference. The difference will come when it is overridden.

0

For your current model design, it will create migration below:

migrationBuilder.AddForeignKey(

name: "FK\_UserRoleRelationship\_UserRole\_ChildUserRoleId",

table: "UserRoleRelationship",

column: "ChildUserRoleId",

principalTable: "UserRole",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

migrationBuilder.AddForeignKey(

name: "FK\_UserRoleRelationship\_UserRole\_ParentUserRoleId",

table: "UserRoleRelationship",

column: "ParentUserRoleId",

principalTable: "UserRole",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

FK\_UserRoleRelationship\_UserRole\_ChildUserRoleId and FK\_UserRoleRelationship\_UserRole\_ParentUserRoleId both will delete the records in UserRole when deleting UserRoleRelationship which will cause multiple cascade delete.

For a workaround, try to make int as int? like below:

public int? ParentUserRoleId { get; set; }

Which will create

migrationBuilder.AddForeignKey(

name: "FK\_UserRoleRelationship\_UserRole\_ParentUserRoleId",

table: "UserRoleRelationship",

column: "ParentUserRoleId",

principalTable: "UserRole",

principalColumn: "Id",

onDelete: ReferentialAction.Restrict);

**TO DO:**

1. Create a new branch from trace-restructure
2. Change the DefaultConnection under
3. Shm.MemberPortal.DataAccess/appsettings.json

to

"Data Source=10.181.1.30,1500;Initial Catalog=SafeHarborMarinasTIR;User ID=sa;Password=<password from passportal>;MultipleActiveResultSets=true"

1. Delete all files under Shm.MemberPortal.DataAccess.Migrations (including the Sql folder)
2. Review the SafeHarborMarinasDev database schema.
3. Attribute all Models under Shm.MemberPortal.Domain such that the Migration script will create the appropriate code to re-create the scema just as it exists now in SafeHarborMarinasDev
4. From the Shm.MemberPortal.DataAccess project folder run the following
   * Powershell:
     + Add-Migration Baseline -c SafeHarborDBContext
   * Console:
     + dotnet ef migrations add Baseline -c SafeHarborDbContext
5. Review the generated migration file and check for appropriate data types being specified, Primary Keys, Indexes, Foreign Keys, Nullable, and Auto Increment
6. If problem discovered above, run
   * Powershell:
     + Remove-Migration
   * Console:
     + dotnet ef migrations remove
7. Then make your changes to the Domain and re-run the Add Migration
8. If you're not able to attribute the models such as they generate the migration file perfectly that's ok. You can manually tweak the migration file some to get the required results.
9. Here's an MSDN doc that's a useful reference for some of this. <https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/> Note that if you need to search around for other references that we're using, and are limited to at present, EF Core 2.1
10. Once you think you've got it pretty close commit and push to your new branch name and then let me know. I'll try to touch base with you tomorrow (Tuesday) morning.