Link:

https://learn.microsoft.com/en-us/shows/entity-framework-core-for-beginners/

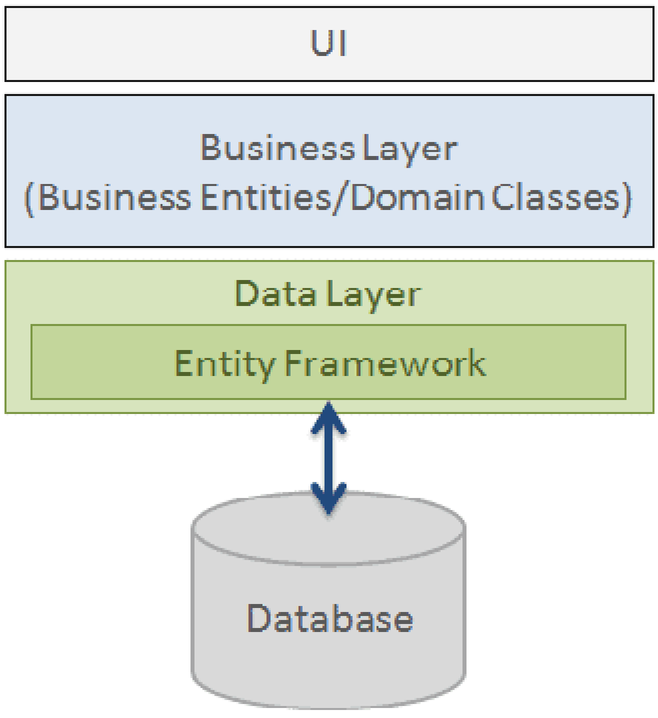
What is Entity Framework?

Prior to .NET 3.5, we (developers) often used to write ADO.NET code or Enterprise Data Access Block to save or retrieve application data from the underlying database. We used to open a connection to the database, create a DataSet to fetch or submit the data to the database, convert data from the DataSet to .NET objects or vice-versa to apply business rules. This was a cumbersome and error prone process. Microsoft has provided a framework called "Entity Framework" to automate all these database related activities for your application.

Entity Framework is an open-source ORM framework for .NET applications supported by Microsoft. It enables developers to work with data using objects of domain specific classes without focusing on the underlying database tables and columns where this data is stored. With the Entity Framework, developers can work at a higher level of abstraction when they deal with data, and can create and maintain data-oriented applications with less code compared with traditional applications.

Official Definition: “Entity Framework is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write.”

The following figure illustrates where the Entity Framework fits into your application.



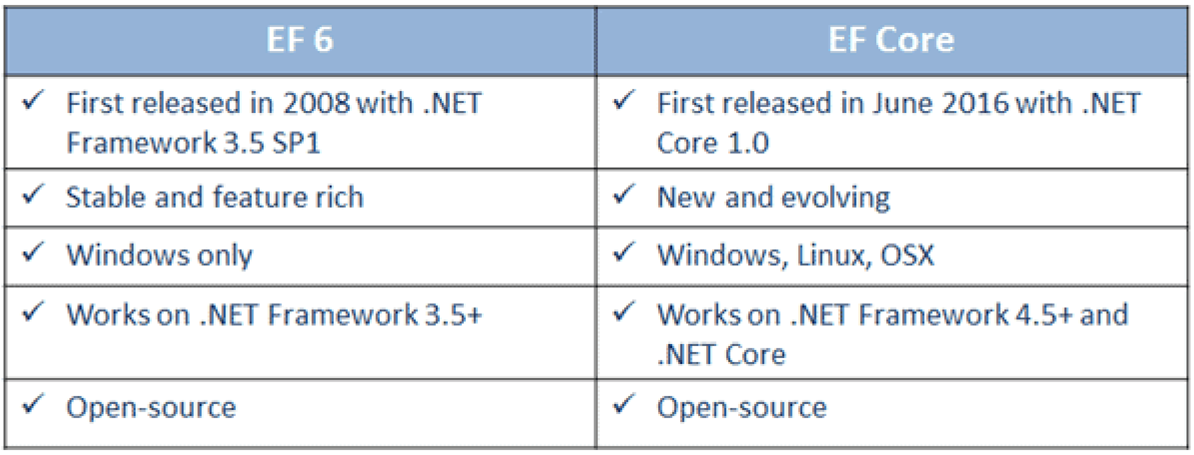
As per the above figure, Entity Framework fits between the business entities (domain classes) and the database. It saves data stored in the properties of business entities and also retrieves data from the database and converts it to business entities objects automatically.

Entity Framework Features

- **Cross-platform:** EF Core is a cross-platform framework which can run on Windows, Linux and Mac.
- **Modelling:** EF (Entity Framework) creates an EDM (Entity Data Model) based on POCO (Plain Old CLR Object) entities with get/set properties of different data types. It uses this model when querying or saving entity data to the underlying database.
- **Querying:** EF allows us to use LINQ queries (C#/VB.NET) to retrieve data from the underlying database. The database provider will translate this LINQ queries to the database-specific query language (e.g. SQL for a relational database). EF also allows us to execute raw SQL queries directly to the database.
- **Change Tracking:** EF keeps track of changes occurred to instances of your entities (Property values) which need to be submitted to the database.
- **Saving:** EF executes INSERT, UPDATE, and DELETE commands to the database based on the changes occurred to your entities when you call the `SaveChanges()` method. EF also provides the asynchronous `SaveChangesAsync()` method.
- **Concurrency:** EF uses Optimistic Concurrency by default to protect overwriting changes made by another user since data was fetched from the database.
- **Transactions:** EF performs automatic transaction management while querying or saving data. It also provides options to customize transaction management.
- **Caching:** EF includes first level of caching out of the box. So, repeated querying will return data from the cache instead of hitting the database.

Entity Framework Latest Versions

Microsoft introduced Entity Framework in 2008 with .NET Framework 3.5. Since then, it released many versions of Entity Framework. Currently, there are two latest versions of Entity Framework: EF 6 and EF Core. The following table lists important difference between EF 6 and EF Core.



EF 6	EF Core
✓ First released in 2008 with .NET Framework 3.5 SP1	✓ First released in June 2016 with .NET Core 1.0
✓ Stable and feature rich	✓ New and evolving
✓ Windows only	✓ Windows, Linux, OSX
✓ Works on .NET Framework 3.5+	✓ Works on .NET Framework 4.5+ and .NET Core
✓ Open-source	✓ Open-source

EF 6 Version History

EF Version	Release Year	.NET Framework
EF 6	2013	.NET 4.0 & .NET 4.5, VS 2012
EF 5	2012	.NET 4.0, VS 2012
EF 4.3	2011	.NET 4.0, VS 2012
EF 4.0	2010	.NET 4.0, VS 2010
EF 1.0 (or 3.5)	2008	.NET 3.5 SP1, VS 2008

Learn more about EF 6 versions history and its features here.

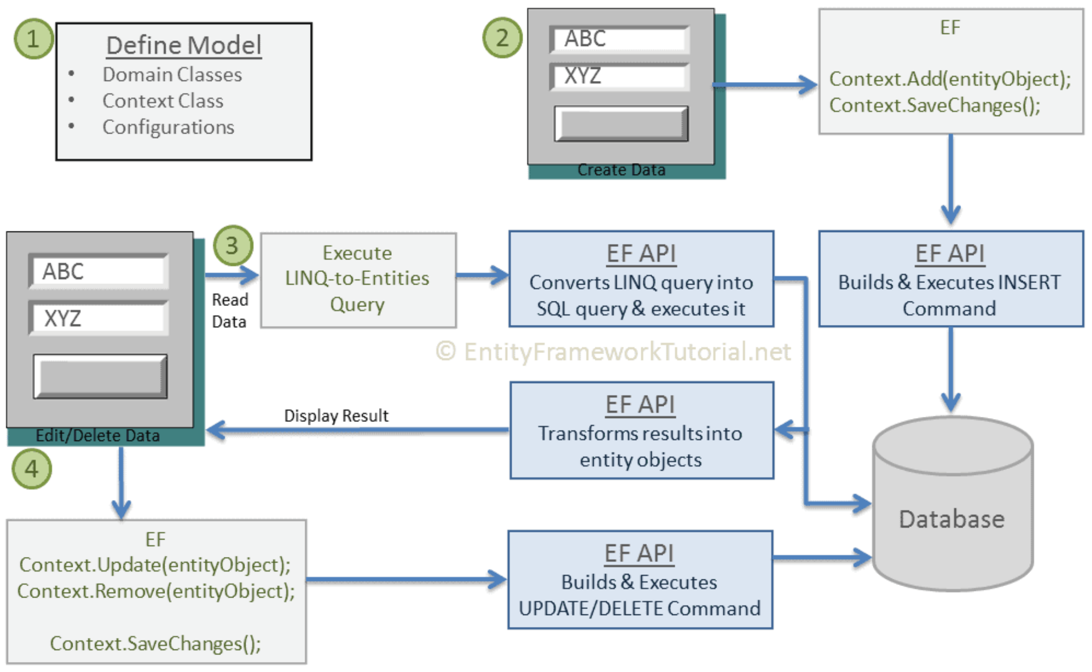
EF Core Version History

EF Core Version	Release Date	.NET Framework
EF Core 2.0	August 2017	.NET Core 2.0, VS 2017
EF Core 1.1	November 2016	.NET Core 1.1
EF Core 1.0	June 2016	.NET Core 1.0

Basic Workflow in Entity Framework

Here you will learn about the basic CRUD workflow using Entity Framework.

The following figure illustrates the basic workflow.

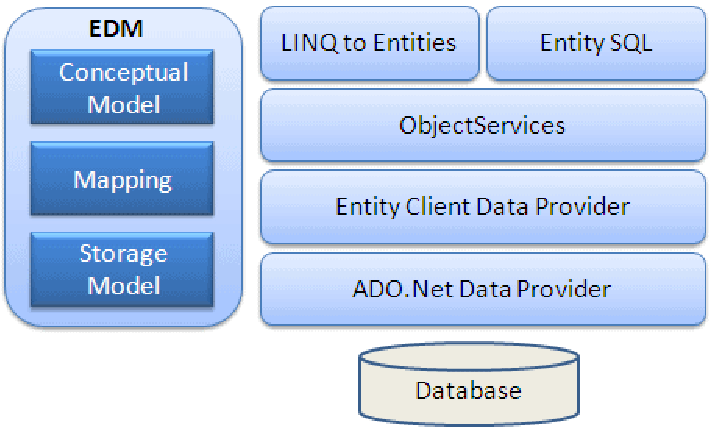

1
Define Model
Domain Classes
Context Class
Configurations
2
ABC
XYZ
Create Data
EF
Context.Add(entityObject);
Context.SaveChanges();
EF API
Builds & Executes INSERT Command
ABC
XYZ
Edit/Delete Data
3
Read Data
Execute LINQ-to-Entities Query
EF API
Converts LINQ query into SQL query & executes it
© EntityFrameworkTutorial.net
Display Result
EF API
Transforms results into entity objects
4
Database
EF
Context.Update(entityObject);
Context.Remove(entityObject);
Context.SaveChanges();
EF API
Builds & Executes UPDATE/DELETE Command

Let's understand the above EF workflow:

1. First of all, you need to define your model. Defining the model includes defining your domain classes, context class derived from DbContext, and configurations (if any). EF will perform CRUD operations based on your model.
2. To insert data, add a domain object to a context and call the `SaveChanges()` method. EF API will build an appropriate INSERT command and execute it to the database.
3. To read data, execute the LINQ-to-Entities query in your preferred language (C#/VB.NET). EF API will convert this query into SQL query for the underlying relational database and execute it. The result will be transformed into domain (entity) objects and displayed on the UI.
4. To edit or delete data, update or remove entity objects from a context and call the `SaveChanges()` method. EF API will build the appropriate UPDATE or DELETE command and execute it to the database.

Entity Framework Architecture

The following figure shows the overall architecture of the Entity Framework.



Let's look at the components of the architecture individually.

EDM (Entity Data Model): EDM consists of three main parts - Conceptual model, Mapping and Storage model.

Conceptual Model: The conceptual model contains the model classes and their relationships. This will be independent from your database table design.

Storage Model: The storage model is the database design model which includes tables, views, stored procedures, and their relationships and keys.

Mapping: Mapping consists of information about how the conceptual model is mapped to the storage model.

LINQ to Entities: LINQ-to-Entities (L2E) is a query language used to write queries against the object model. It returns entities, which are defined in the conceptual model. You can use your LINQ skills here.

Entity SQL: Entity SQL is another query language (For EF 6 only) just like LINQ to Entities. However, it is a little more difficult than L2E and the developer will have to learn it separately.

Object Service: Object service is a main entry point for accessing data from the database and returning it back. Object service is responsible for materialization, which is the process of converting data returned from an entity client data provider (next layer) to an entity object structure.

Entity Client Data Provider: The main responsibility of this layer is to convert LINQ-to-Entities or Entity SQL queries into a SQL query which is understood by the underlying database. It communicates with the ADO.Net data provider which in turn sends or retrieves data from the database.

ADO.Net Data Provider: This layer communicates with the database using standard ADO.Net.

Context Class in Entity Framework

The context class is a most important class while working with EF 6 or EF Core. It represent a session with the underlying database using which you can perform CRUD (Create, Read, Update, Delete) operations.

The context class in Entity Framework is a class which derives from System.Data.Entity.DbContextDbContext in EF 6 and EF Core both. An instance of the context class represents Unit Of Work and Repository patterns wherein it can combine multiple changes under a single database transaction.

The context class is used to query or save data to the database. It is also used to configure domain classes, database related mappings, change tracking settings, caching, transaction etc.

The following `SchoolContext` class is an example of a context class.

```
using System.Data.Entity;

public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }
    // Entities
    public DbSet<Student> Students { get; set; }
    public DbSet<StudentAddress> StudentAddresses { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

In the above example, the `SchoolContext` class is derived from `DbContext`, which makes it a context class. It also includes an entity set for `Student`, `StudentAddress`, and `Grade` entities (learn about it next).

What is an Entity in Entity Framework?

An entity in Entity Framework is a class that maps to a database table. This class must be included as a `DbSet<TEntity>` type property in the `DbContext` class. EF API maps each entity to a table and each property of an entity to a column in the database.

For example, the following `Student`, and `Grade` are domain classes in the school application.

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

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

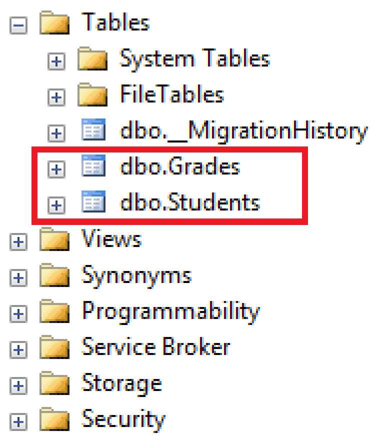
The above classes become entities when they are included as `DbSet<TEntity>` properties in a context class (the class which derives from `DbContext`), as shown below.

```
public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

In the above context class, `Students`, and `Grades` properties of type `DbSet<TEntity>` are called entity sets. The `Student`, and `Grade` are entities. EF API will create the `Students` and `Grades` tables in the database, as shown below.



An Entity can include two types of properties: Scalar Properties and Navigation Properties.

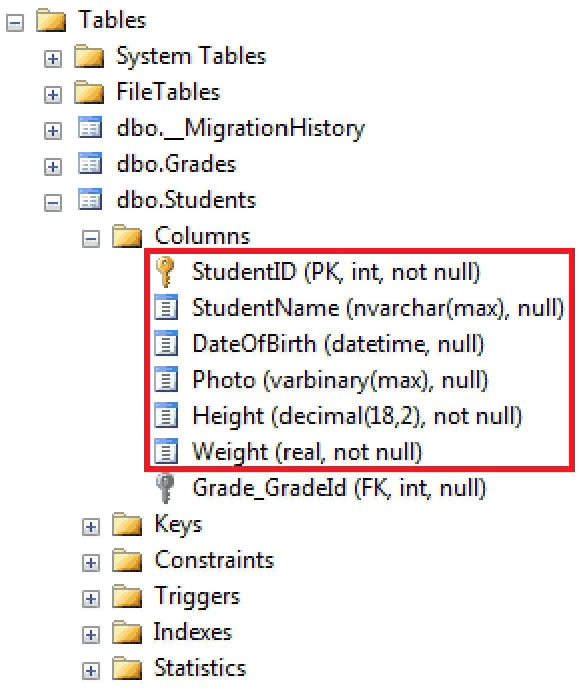
Scalar Property

The primitive type properties are called scalar properties. Each scalar property maps to a column in the database table which stores an actual data. For example, `StudentID`, `StudentName`, `DateOfBirth`, `Photo`, `Height`, `Weight` are the scalar properties in the `Student` entity class.

```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation properties
    public Grade Grade { get; set; }
}
```

EF API will create a column in the database table for each scalar property, as shown below.


Tables
System Tables
FileTables
dbo.__MigrationHistory
dbo.Grades
dbo.Students
Columns
StudentID (PK, int, not null)
StudentName (nvarchar(max), null)
DateOfBirth (datetime, null)
Photo (varbinary(max), null)
Height (decimal(18,2), not null)
Weight (real, not null)
Grade_GradeId (FK, int, null)
Keys
Constraints
Triggers
Indexes
Statistics

Navigation Property

The navigation property represents a relationship to another entity.

There are two types of navigation properties: Reference Navigation and Collection Navigation

Reference Navigation Property

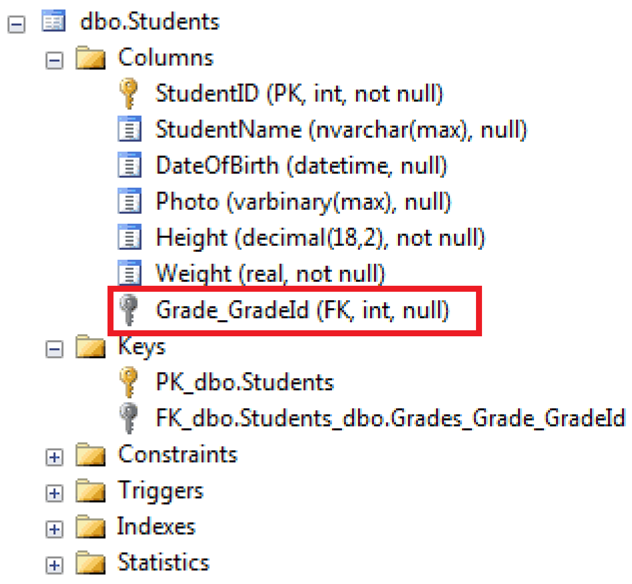
If an entity includes a property of another entity type, it is called a Reference Navigation Property. It points to a single entity and represents multiplicity of one (1) in the entity relationships.

EF API will create a ForeignKey column in the table for the navigation properties that points to a PrimaryKey of another table in the database. For example, `Grade` are reference navigation properties in the following `Student` entity class.

```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation property
    public Grade Grade { get; set; }
}
```

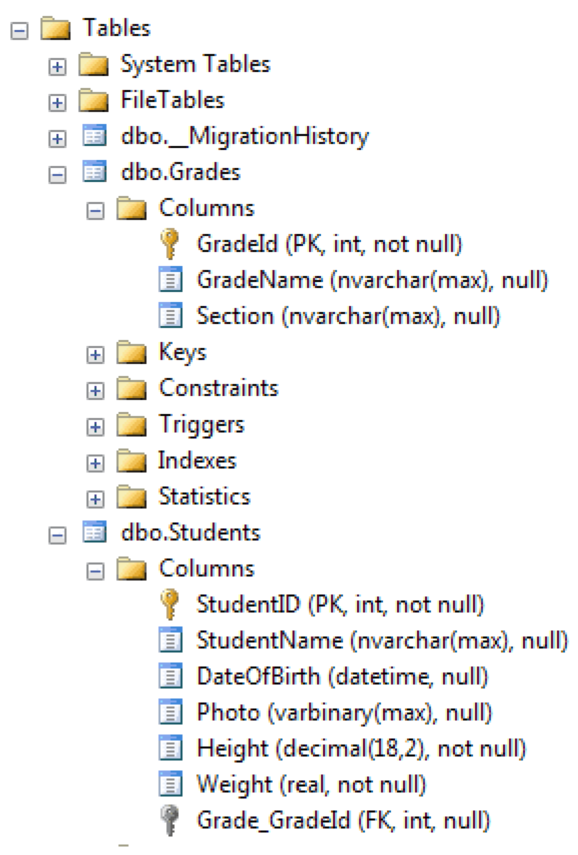
In the database, EF API will create a ForeignKey `Grade_GradeId` in the `Students` table, as shown below.


```
dbo.Students
  Columns
    StudentID (PK, int, not null)
    StudentName (nvarchar(max), null)
    DateOfBirth (datetime, null)
    Photo (varbinary(max), null)
    Height (decimal(18,2), not null)
    Weight (real, not null)
    Grade_GradeId (FK, int, null)
  Keys
    PK_dbo.Students
    FK_dbo.Students_dbo.Grades_Grade_GradeId
  Constraints
  Triggers
  Indexes
  Statistics
```

Collection Navigation Property

If an entity includes a property of generic collection of an entity type, it is called a collection navigation property. It represents multiplicity of many (*).

EF API does not create any column for the collection navigation property in the related table of an entity, but it creates a column in the table of an entity of generic collection. For example, the following `Grade` entity contains a generic collection navigation property `ICollection<Student>`. Here, the `Student` entity is specified as generic type, so EF API will create a column `Grade_GradeId` in the `Students` table in the database.



```
Tables
    System Tables
    FileTables
    dbo.__MigrationHistory
    dbo.Grades
        Columns
            GradeId (PK, int, not null)
            GradeName (nvarchar(max), null)
            Section (nvarchar(max), null)
        Keys
        Constraints
        Triggers
        Indexes
        Statistics
    dbo.Students
        Columns
            StudentID (PK, int, not null)
            StudentName (nvarchar(max), null)
            DateOfBirth (datetime, null)
            Photo (varbinary(max), null)
            Height (decimal(18,2), not null)
            Weight (real, not null)
            Grade_GradeId (FK, int, null)
```

Types of Entities in Entity Framework

There are two types of Entities in Entity Framework: POCO Entities and Dynamic Proxy Entities.

POCO Entities (Plain Old CLR Object)

A POCO entity is a class that doesn't depend on any framework-specific base class. It is like any other normal .NET CLR class, which is why it is called "Plain Old CLR Objects".

POCO entities are supported in both EF 6 and EF Core.

These POCO entities (also known as persistence-ignorant objects) support most of the same query, insert, update, and delete behaviors as entity types that are generated by the Entity Data Model. The following is an example of Student POCO entity.

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public StudentAddress StudentAddress { get; set; }
    public Grade Grade { get; set; }
}
```

Dynamic Proxy Entities (POCO Proxy)

Dynamic Proxy is a runtime proxy class which wraps POCO entity. Dynamic proxy entities allow **lazy loading**.

Note: Dynamic proxy entities are only supported in EF 6. EF Core 2.0 does not support them yet.

A POCO entity should meet the following requirements to become a POCO proxy:

1. A POCO class must be declared with public access.
2. A POCO class must not be sealed (NotInheritable in Visual Basic).
3. A POCO class must not be abstract (MustInherit in Visual Basic).
4. Each navigation property must be declared as public, virtual.
5. Each collection property must be ICollection<T>.
6. The `ProxyCreationEnabled` option must **NOT** be false (default is true) in context class.

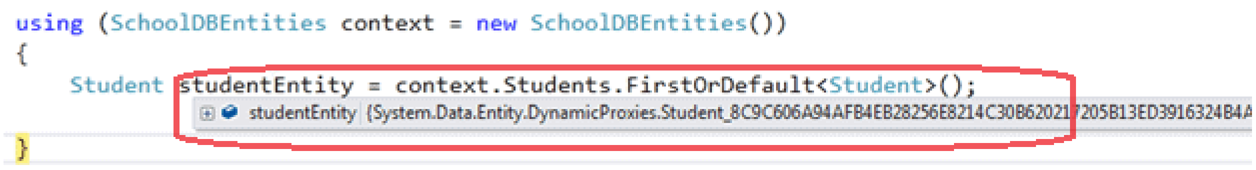
The following POCO entity meets all of the above requirements to become a dynamic proxy entity at runtime.

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public virtual StudentAddress StudentAddress { get; set; }
    public virtual Grade Grade { get; set; }
}
```

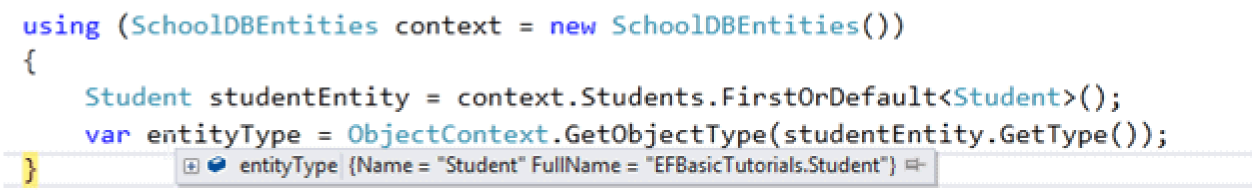
Note: By default, dynamic proxy is enabled for every entity. However, you can disable dynamic proxy by setting `context.Configuration.ProxyCreationEnabled = false;` in the context class.

At runtime, EF API will create an instance of dynamic proxy for the above `Student` entity. The type of dynamic proxy for `Student` will be `System.Data.Entity.DynamicProxies.Student`, as shown below:



```
using (SchoolDBEntities context = new SchoolDBEntities())
{
    Student studentEntity = context.Students.FirstOrDefault<Student>();
              studentEntity {System.Data.Entity.DynamicProxies.Student_8C9C606A94AFB4EB28256E8214C30B620217205B13ED3916324B4A
}
```

Use `ObjectContext.GetObjectType()` to find the underlying wrapped type by the dynamic proxy as shown below:



```
using (SchoolDBEntities context = new SchoolDBEntities())
{
    Student studentEntity = context.Students.FirstOrDefault<Student>();
    var entityType = ObjectContext.GetObjectType(studentEntity.GetType());
}              entityType {Name = "Student" FullName = "EFBasicTutorials.Student"}
```

EntityState in Entity Framework

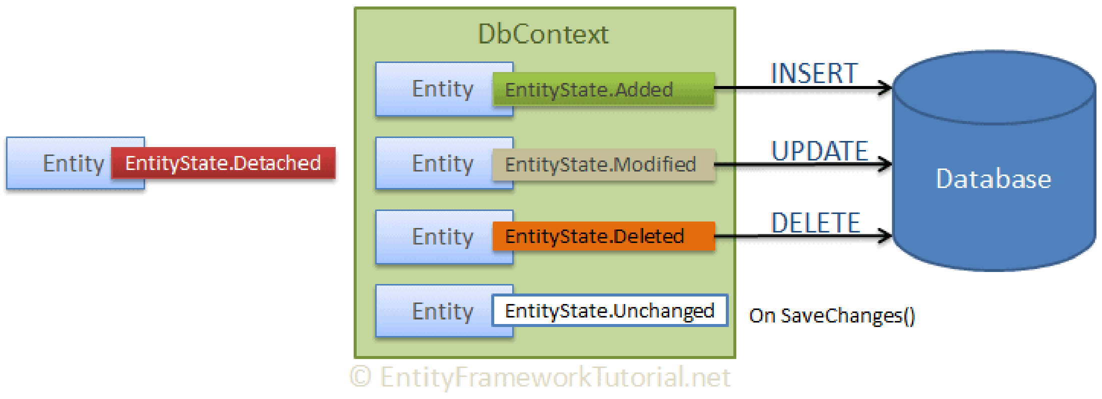
EF API maintains the state of each entity during its lifetime. Each entity has a state based on the operation performed on it via the context class. The entity state represented by an enum `System.Data.Entity.EntityState` in EF 6 and `Microsoft.EntityFrameworkCore.EntityState` in EF Core with the following values:

1. Added
2. Modified
3. Deleted
4. Unchanged
5. Detached

The Context not only holds the reference to all the entity objects as soon as retrieved from the database, but also keeps track of entity states and maintains modifications made to the properties of the entity. This feature is known as *Change Tracking*.

The change in entity state from the Unchanged to the Modified state is the only state that's automatically handled by the context. All other changes must be made explicitly using proper methods of `DbContext` or `DbSet`. (You will learn about these methods in EF 6 and EF Core sections.)

EF API builds and executes the INSERT, UPDATE, and DELETE commands based on the state of an entity when the `context.SaveChanges()` method is called. It executes the INSERT command for the entities with Added state, the UPDATE command for the entities with Modified state and the DELETE command for the entities in Deleted state. The context does not track entities in the Detached state. The following figure illustrates the significance of entity states:



Thus, entity states play an important role in Entity Framework.

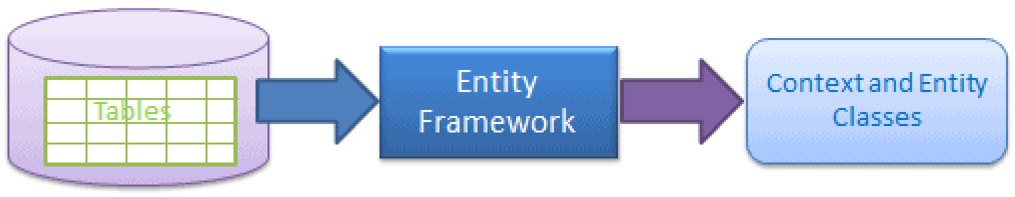
Development Approaches with Entity Framework

There are three different approaches you can use while developing your application using Entity Framework:

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

Database-First Approach

In the database-first development approach, you generate the context and entities for the existing database using EDM wizard integrated in Visual Studio or executing EF commands.



Database-First Approach

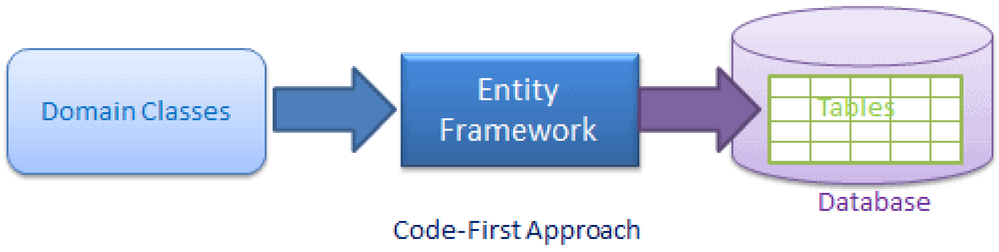
EF 6 supports the database-first approach extensively. Visit EF 6 DB-First section to learn about the database-first approach using EF 6.

EF Core includes limited support for this approach.

Code-First Approach

Use this approach when you do not have an existing database for your application. In the code-first approach, you start writing your entities (domain classes) and context class first and then create the database from these classes using migration commands.

Developers who follow the Domain-Driven Design (DDD) principles, prefer to begin with coding their domain classes first and then generate the database required to persist their data.

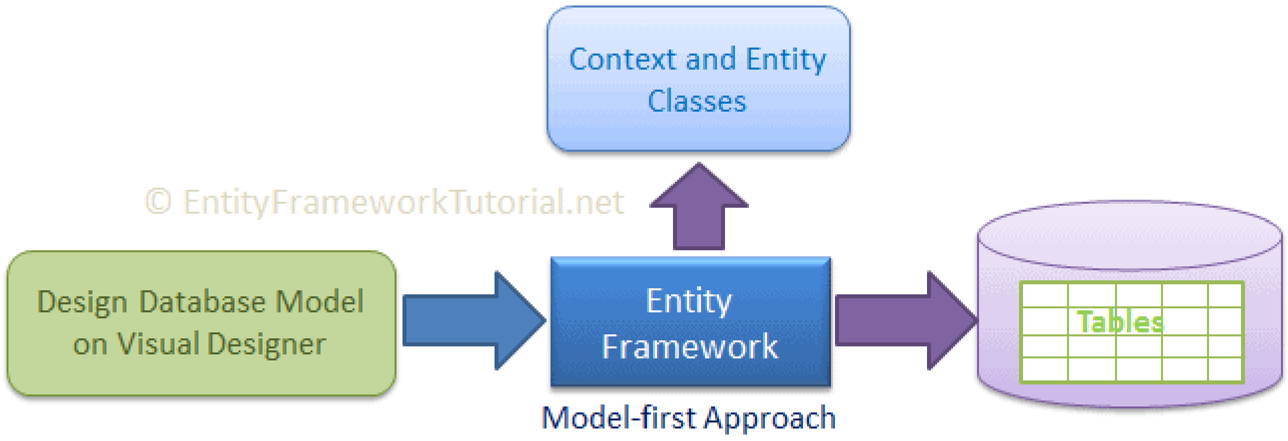


Visit the EF 6 Code-First Tutorials section to learn EF 6 code-first development from scratch.

Visit EF Core section to learn about the code-first approach in EF Core.

Model-First Approach

In the model-first approach, you create entities, relationships, and inheritance hierarchies directly on the visual designer integrated in Visual Studio and then generate entities, the context class, and the database script from your visual model.

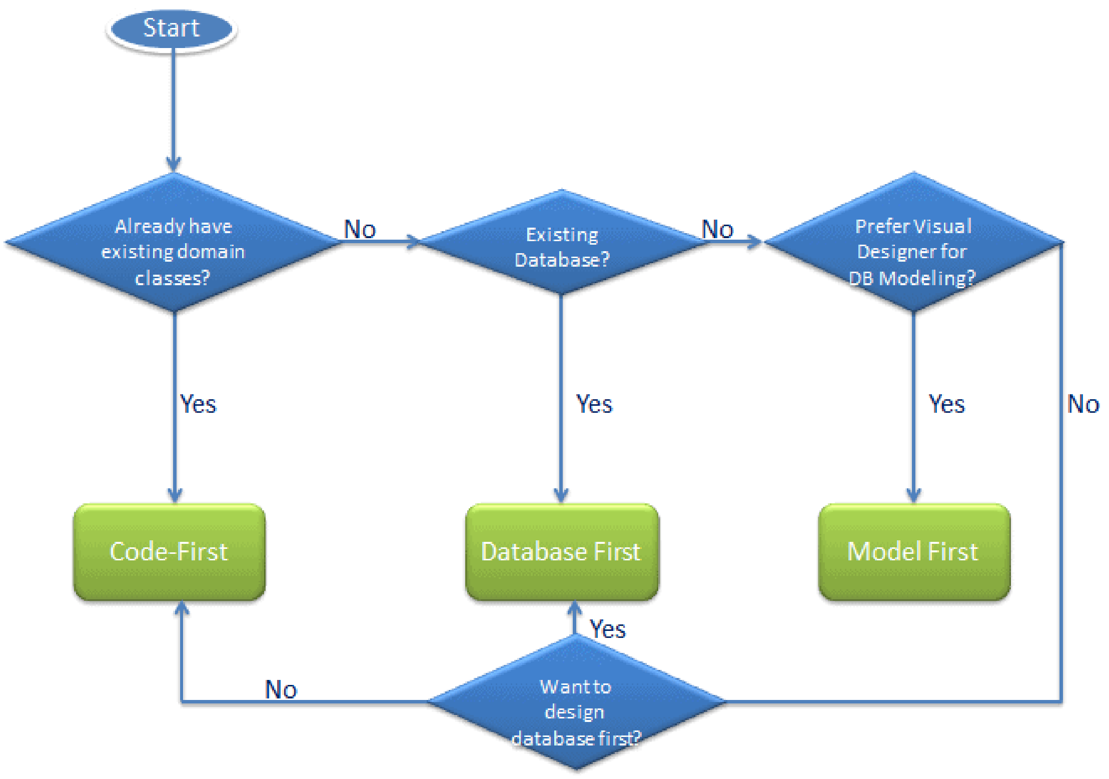

Context and Entity Classes
© EntityFrameworkTutorial.net
Design Database Model on Visual Designer
Entity Framework
Tables
Model-first Approach

EF 6 includes limited support for this approach.

EF Core does not support this approach.

Choosing the Development Approach for Your Application

Use the following flow chart to decide which is the right approach to develop your application using Entity Framework:


Start
Already have existing domain classes?
No
Existing Database?
No
Prefer Visual Designer for DB Modeling?
Yes
Yes
Yes
No
Code-First
Database First
Model First
Yes
No
Want to design database first?

As per the above figure, if you already have an existing application with domain classes, then you can use the code-first approach because you can create a database from your existing classes. If you have an existing database, then you can create an EDM from an existing database in the database-first approach. If you do not have an existing database or domain classes, and you prefer to design your DB model on the visual designer, then go for the Model-first approach.