Entity Framework

What is Entity Framework?

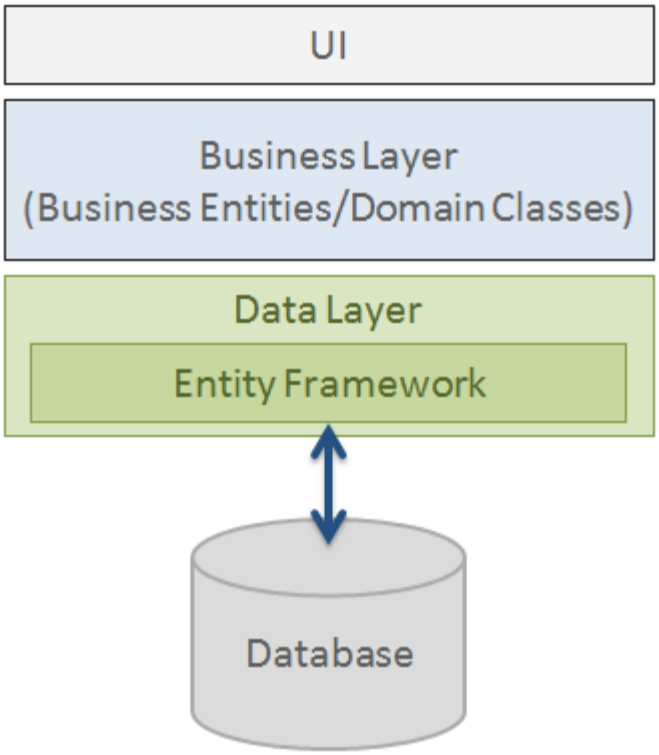
Prior to .NET 3.5, we (developers) often used to write ADO.NET code or Enterprise Data Access Block to save or retrieve application data from the underlying database. We used to open a connection to the database, create a DataSet to fetch or submit the data to the database, convert data from the DataSet 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."**



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.

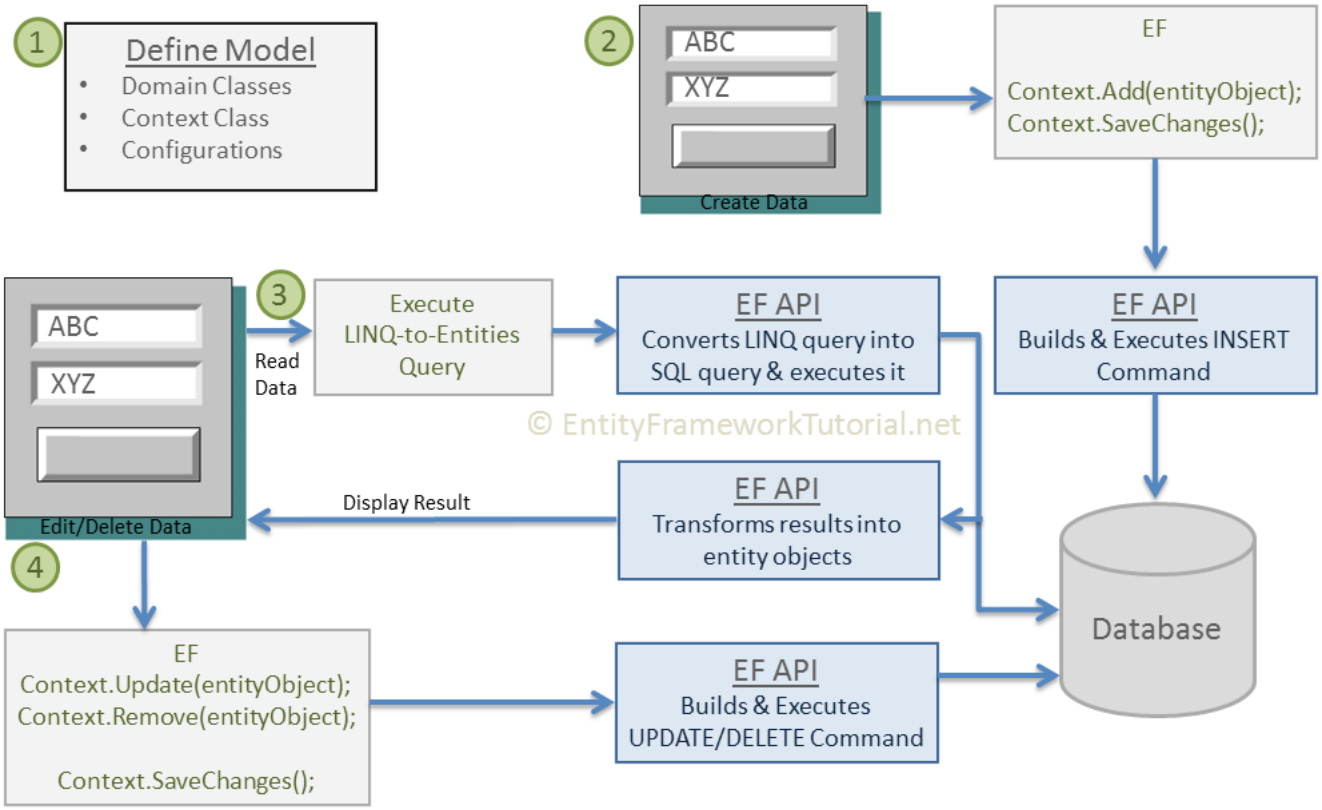
Built-in Conventions: EF follows conventions over the configuration programming pattern, and includes a set of default rules which automatically configure the EF model.

Configurations: EF allows us to configure the EF model by using data annotation attributes or Fluent API to override default conventions.

Migrations: EF provides a set of migration commands that can be executed on the NuGet Package Manager Console or the Command Line Interface to create or manage underlying database Schema.

Basic Workflow in Entity Framework

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



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.

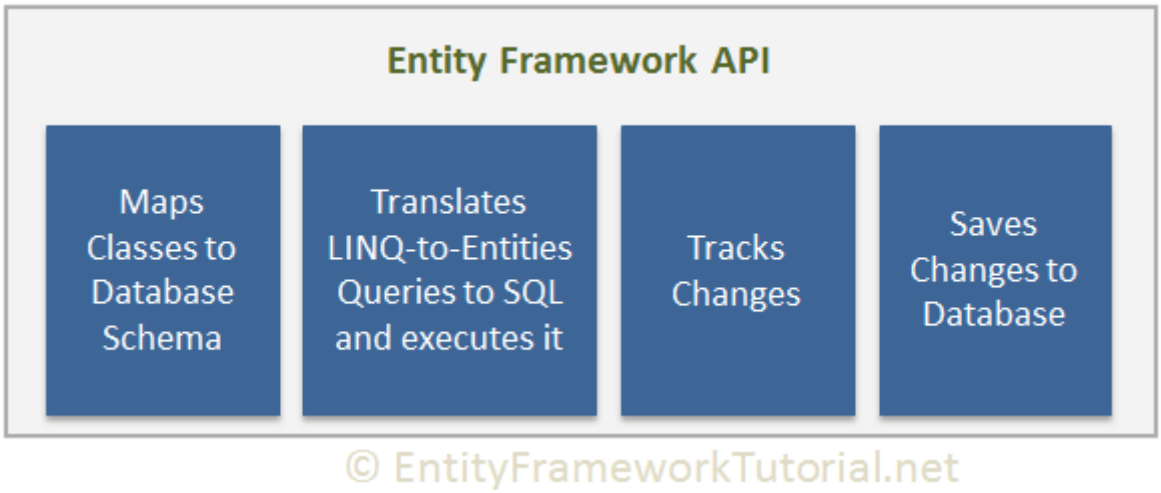
2.To insert data, add a domain object to a context and call the SaveChanges() method. EF API will build an appropriate INSERT command and execute it to the database.

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.

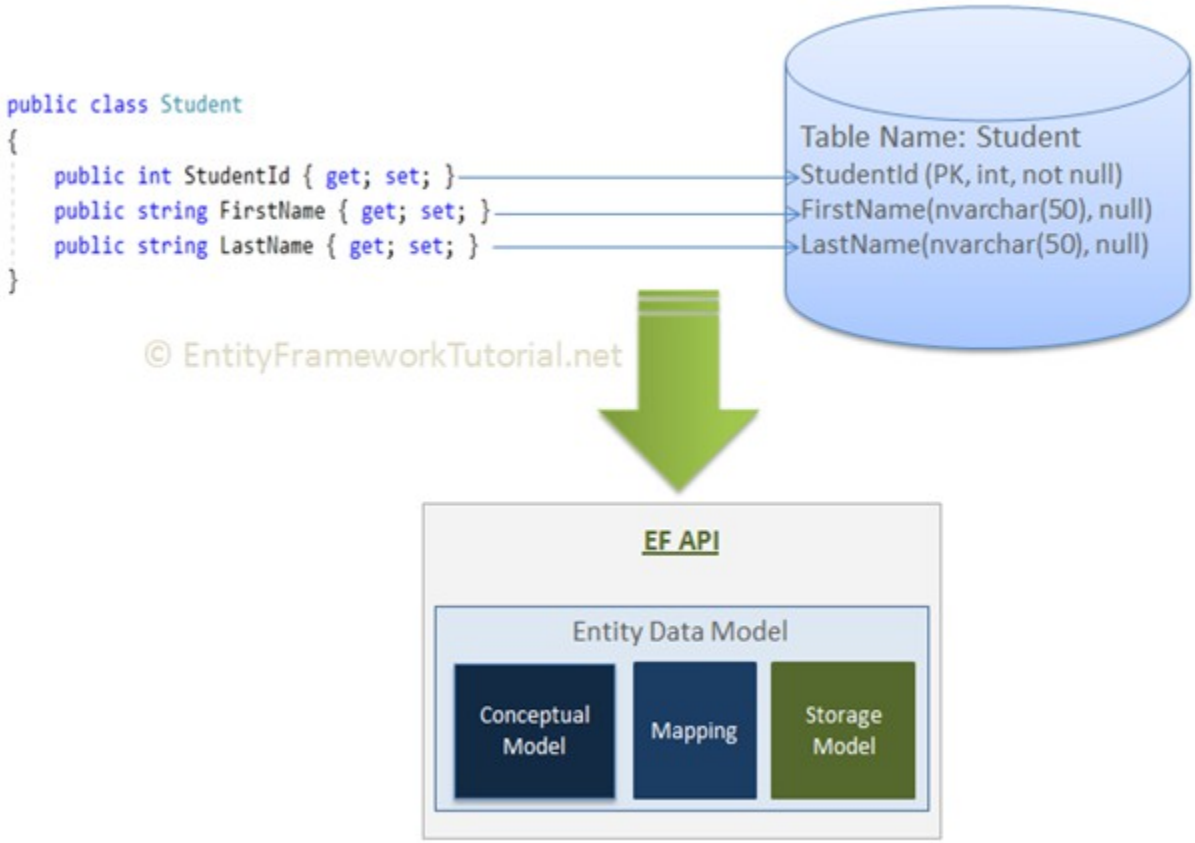
4.To edit or delete data, update or remove entity objects from a context and call the SaveChanges() method. EF API will build the appropriate UPDATE or DELETE command and execute it to the database.

How Entity Framework Works?

Entity Framework API (EF6 & EF Core) includes the ability to map domain (entity) classes to the database schema, translate & execute LINQ queries to SQL, track changes occurred on entities during their lifetime, and save changes to the database.



Entity Data Model



The very first task of EF API is to build an Entity Data Model (EDM). EDM is an in-memory representation of the entire metadata: conceptual model, storage model, and mapping between them.

Conceptual Model: EF builds the conceptual model from your domain classes, context class, default conventions followed in your domain classes, and configurations.

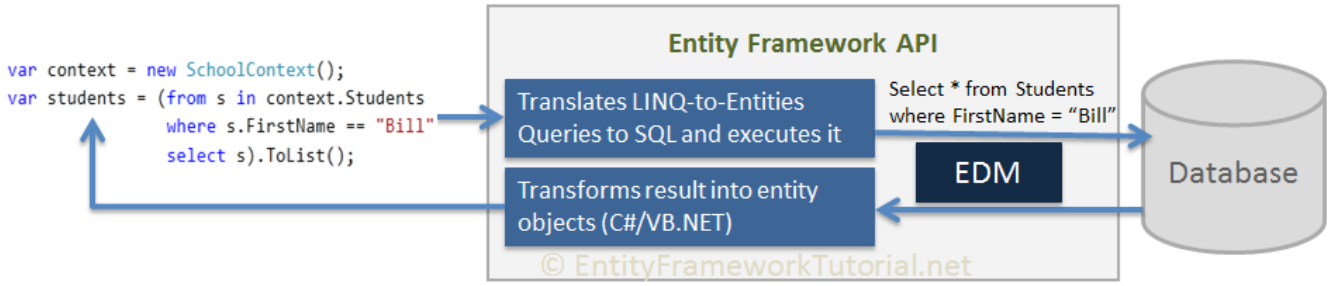
Storage Model: EF builds the storage model for the underlying database schema. In the code-first approach, this will be inferred from the conceptual model. In the database-first approach, this will be inferred from the targeted database.

Mappings: EF includes mapping information on how the conceptual model maps to the database schema (storage model).

EF performs CRUD operations using this EDM. It uses EDM in building SQL queries from LINQ queries, building INSERT, UPDATE, and DELETE commands, and transform database result into entity objects.

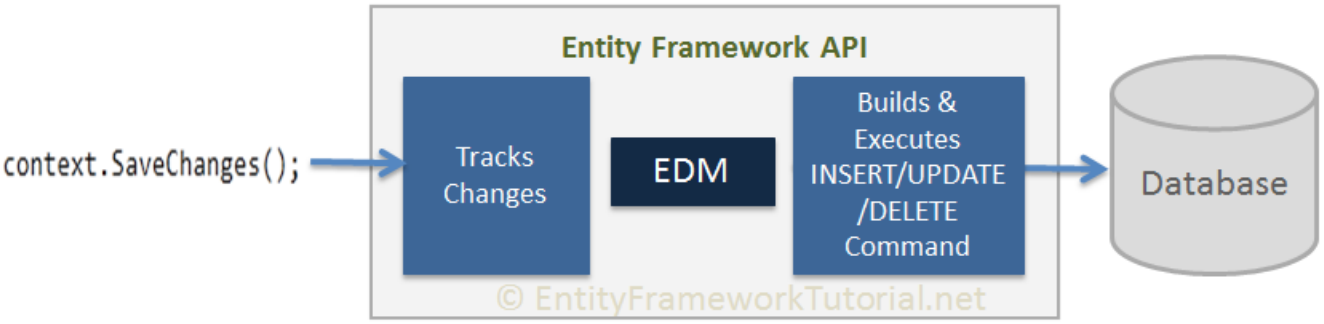
Querying

EF API translates LINQ-to-Entities queries to SQL queries for relational databases using EDM and also converts results back to entity objects.

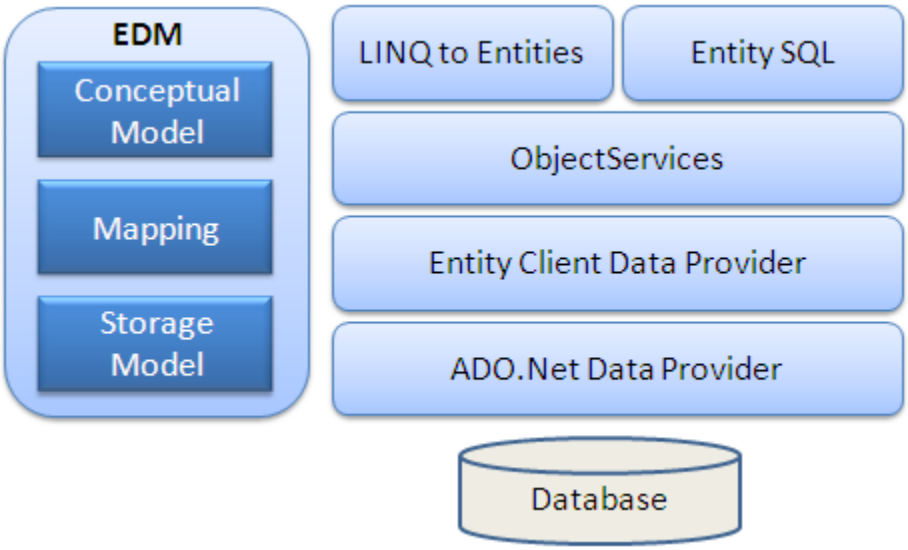


Saving

EF API infers INSERT, UPDATE, and DELETE commands based on the state of entities when the SaveChanges() method is called. The ChangeTrack keeps track of the states of each entity as and when an action is performed.



Entity Framework Architecture



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.

```
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

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

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.

EF API will create a column in the database table for each scalar property .

Navigation Property

The navigation property represents a relationship to another entity.

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

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 does 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.
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.

Development Approaches with Entity Framework

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

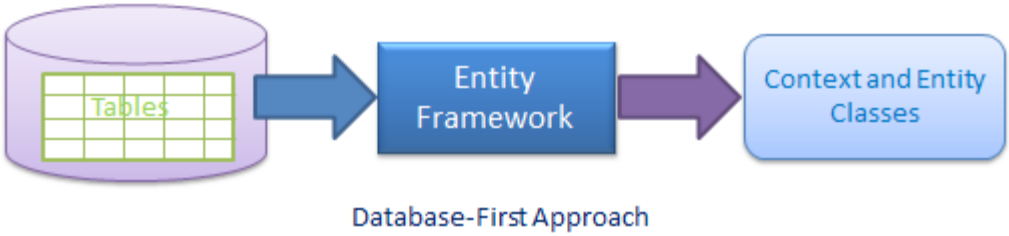
1.Database-First

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. EF 6 supports the database-first approach extensively. EF Core includes limited support for this approach.

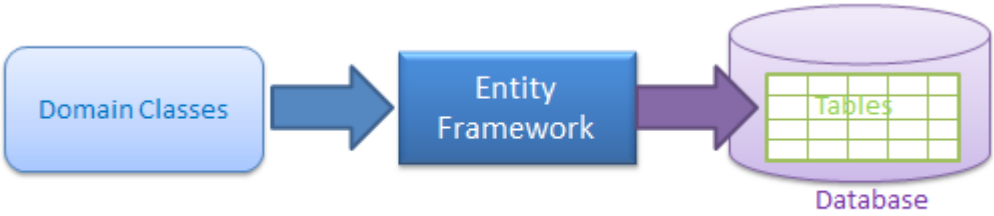


Database-First Approach

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.

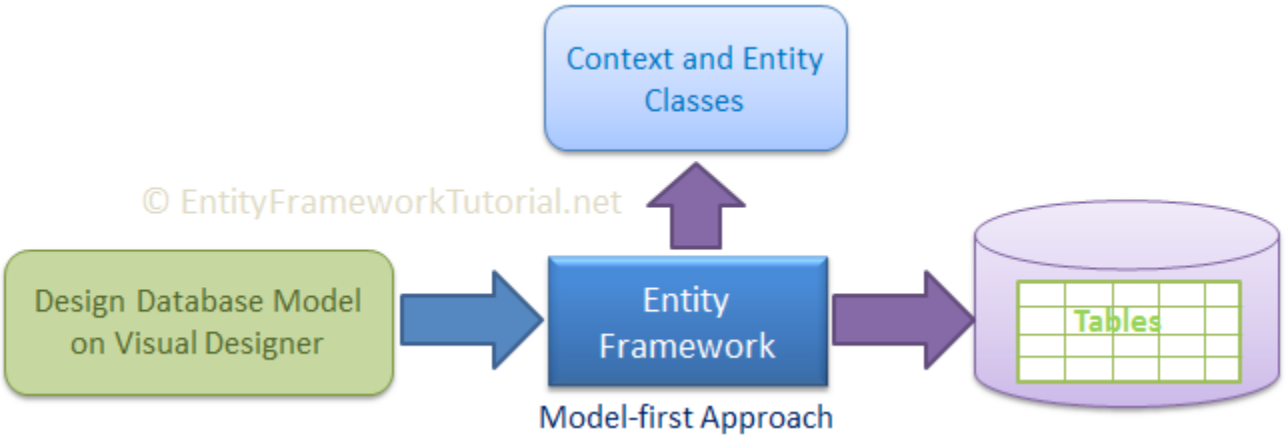
Developers who follow the Domain-Driven Design (DDD) principles, prefer to begin with coding their domain classes first and then generate the database required to persist their data.



Code-First Approach

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.



Model-first Approach

EF 6 includes limited support for this approach.

EF Core does not support this approach.