ASP.NET WEB API 2

EF Core

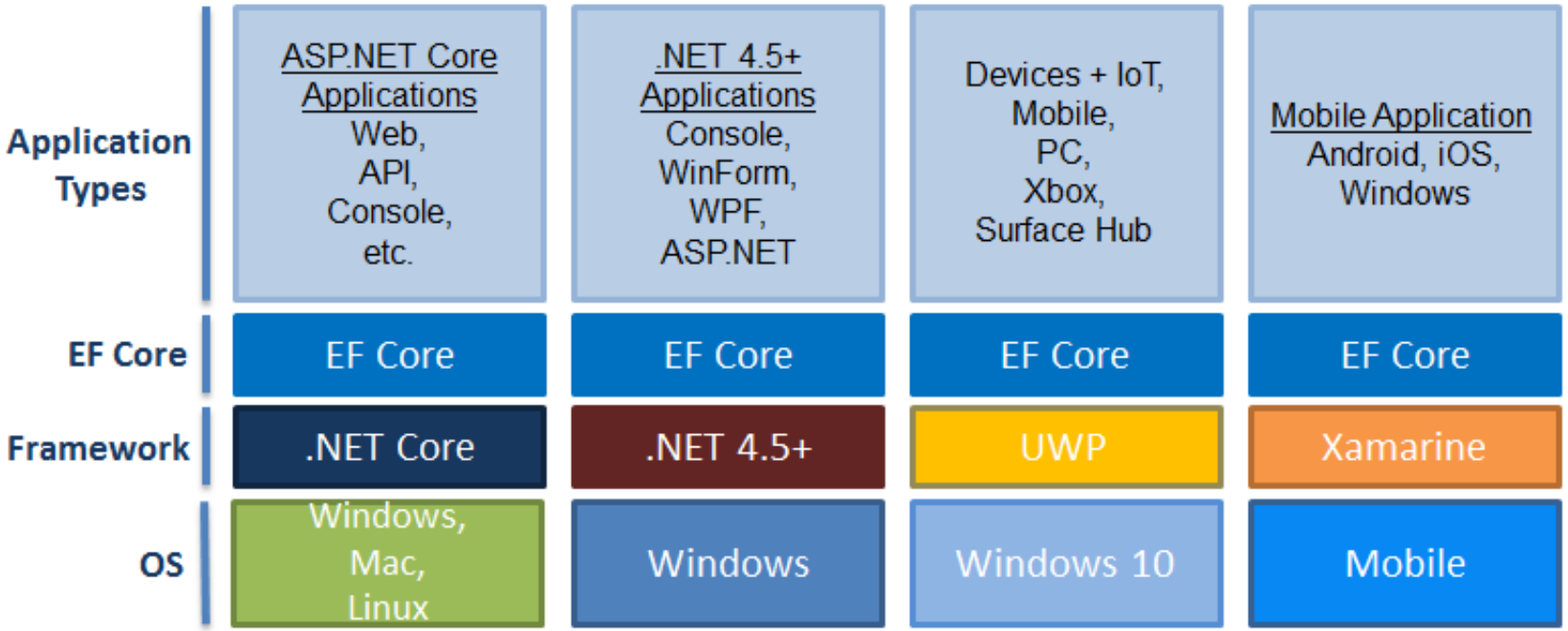
Content

- Introduction
- DbContext
- Domain Class
- Data Annotations Attributes

Introduction

- Entity Framework Core is the new version of Entity Framework after EF 6.x. It is open-source, lightweight, extensible and a cross-platform version of Entity Framework data access technology.
- Entity Framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.
- EF Core is intended to be used with .NET Core applications. However, it can also be used with standard .NET 4.5+ framework based applications.

- The following figure illustrates the supported application types, .NET Frameworks and OSs.

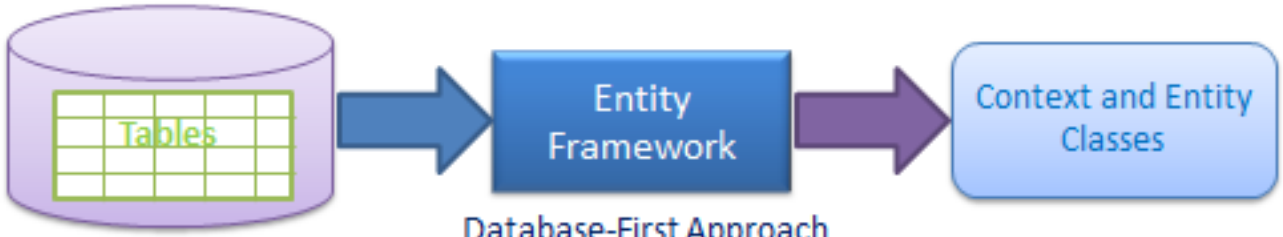


EF Core Version History

EF Core Version	Release Date
EF Core 3.0	
EF Core 2.2	Dec 2018
EF Core 2.1	August 2018
EF Core 2.0	August 2017
EF Core 1.1	November 2016
EF Core 1.0	June 2016

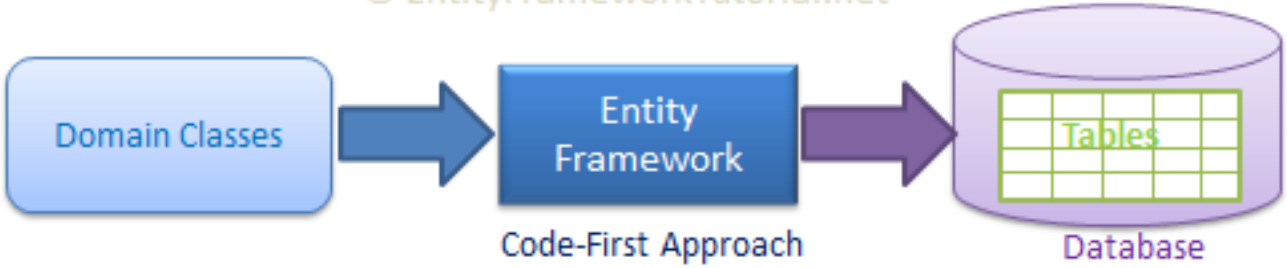
EF Core Development Approaches

- EF Core supports two development approaches
  - Code-First
  - Database-First
- In the code-first approach, EF Core API creates the database and tables using migration based on the conventions and configuration provided in your domain classes. This approach is useful in Domain Driven Design (DDD).
- In the database-first approach, EF Core API creates the domain and context classes based on your existing database using EF Core commands. This has limited support in EF Core as it does not support visual designer or wizard.



Database-First Approach

Generate Data Access Classes for Existing Database



Code-First Approach

Create Database from the Domain Classes

EF Core Database Providers

- Entity Framework Core uses a provider model to access many different databases. EF Core includes providers as NuGet packages which you need to install.

Database	NuGet Package
SQL Server	Microsoft.EntityFrameworkCore.SqlServer
MySQL	MySql.Data.EntityFrameworkCore
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL
SQLite	Microsoft.EntityFrameworkCore.SQLite
SQL Compact	EntityFrameworkCore.SqlServerCompact40
In-memory	Microsoft.EntityFrameworkCore.InMemory
DB2 (IBM)	IBM.EntityFrameworkCore

DbContext

- The DbContext class is an integral part of Entity Framework. An instance of DbContext represents a session with the database which can be used to query and save instances of your entities to a database.
- DbContext is a combination of the Unit Of Work and Repository patterns.

- DbContext in EF Core allows us to perform following tasks:
  - Manage database connection
  - Configure model & relationship
  - Querying database
  - Saving data to the database
  - Configure change tracking
  - Caching
  - Transaction management

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

    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }
    //entities
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
}
```

DbContext Methods

Method	Usage
Add	Adds a new entity to DbContext with Added state and starts tracking it. This new entity data will be inserted into the database when SaveChanges() is called.
AddAsync	Asynchronous method for adding a new entity to DbContext with Added state and starts tracking it. This new entity data will be inserted into the database when SaveChangesAsync() is called.
AddRange	Adds a collection of new entities to DbContext with Added state and starts tracking it. This new entity data will be inserted into the database when SaveChanges() is called.
AddRangeAsync	Asynchronous method for adding a collection of new entities which will be saved on SaveChangesAsync().

Method	Usage
Attach	Attaches a new or existing entity to DbContext with Unchanged state and starts tracking it.
AttachRange	Attaches a collection of new or existing entities to DbContext with Unchanged state and starts tracking it.
Entry	Gets an EntityEntry for the given entity. The entry provides access to change tracking information and operations for the entity.
Find	Finds an entity with the given primary key values.
FindAsync	Asynchronous method for finding an entity with the given primary key values.

Method	Usage
Remove	Sets Deleted state to the specified entity which will delete the data when SaveChanges() is called.
RemoveRange	Sets Deleted state to a collection of entities which will delete the data in a single DB round trip when SaveChanges() is called.
SaveChanges	Execute INSERT, UPDATE or DELETE command to the database for the entities with Added, Modified or Deleted state.
SaveChangesAsync	Asynchronous method of SaveChanges()
Set	Creates a DbSet<TEntity> that can be used to query and save instances of TEntity.

Method	Usage
Update	Attaches disconnected entity with Modified state and start tracking it. The data will be saved when SaveChagnes() is called.
UpdateRange	Attaches a collection of disconnected entities with Modified state and start tracking it. The data will be saved when SaveChagnes() is called.
OnConfiguring	Override this method to configure the database (and other options) to be used for this context. This method is called for each instance of the context that is created.
OnModelCreating	Override this method to further configure the model that was discovered by convention from the entity types exposed in DbSet<TEntity> properties on your derived context.

DbContext Properties

Method	Usage
ChangeTracker	Provides access to information and operations for entity instances this context is tracking.
Database	Provides access to database related information and operations for this context.
Model	Returns the metadata about the shape of entities, the relationships between them, and how they map to the database.

Domain Class

```
public class Student
{
    public int StudentId { get; set; }
    public string Code { get; set; }
    public string Name { get; set; }
    public int MajorId { get; set; }
}

public class Major
{
    public int MajorId { get; set; }
    public string Name { get; set; }
}
```

```
[Table("Students")]
public class Student
{
    [Key]
    public int StudentId { get; set; }
    public string Code { get; set; }
    [Column("StudentName", TypeName = "ntext")]
    [MaxLength(100)]
    public string Name { get; set; }
    public int MajorId { get; set; }
    [ForeignKey("MajorId")]
    public virtual Major Major { get; set; }
}
[Table("Majors")]
public class Major
{
    [Key]
    public int MajorId { get; set; }
    public string Name { get; set; }
}
```

Data Annotations Attributes

- Table
- Column
- Key
- NotMapped
- ForeignKey
- InverseProperty
- Required
- MaxLength
- StringLength

Table

Table Attribute: `[Table(string name, Properties:[Schema = string])`

- name: Name of the Db table.
- Schema: Name of the Db Schema in which a specified table should be created. (Optional)

```
using System.ComponentModel.DataAnnotations.Schema;

[Table("StudentMaster")]
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```

```
using System.ComponentModel.DataAnnotations.Schema;

[Table("StudentMaster", Schema="Admin")]
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```

Column

```
[Column (string name, Properties:[Order = int],[TypeName = string])
```

- name: Name of a column in a db table.
- Order: Order of a column, starting with zero index. (Optional)
- TypeName: Data type of a column. (Optional)

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int StudentID { get; set; }

    [Column("Name")]
    public string StudentName { get; set; }
    [Column("DoB", TypeName="DateTime2")]
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
}
```

Key

- The default convention creates a primary key column for a property whose name is Id or <Entity Class Name>Id.
- The Key attribute overrides this default convention.

```
using System.ComponentModel.DataAnnotations;

public class Student
{
    [Key]
    public int StudentKey { get; set; }
    public string StudentName { get; set; }
}
```

Composite Keys

```
public class SampleContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Order>()
            .HasKey(o => new { o.CustomerAbbreviation, o.OrderNumber });
    }
}

public class Order
{
    public string CustomerAbbreviation { get; set; }
    public int OrderNumber { get; set; }
    public DateTime DateCreated { get; set; }
    public Customer Customer { get; set; }
    ...
)
```

NotMapped

- You can apply the [NotMapped] attribute on one or more properties for which you do NOT want to create a corresponding column in a database table.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    [NotMapped]
    public int Age { get; set; }
}
```

ForeignKey

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }

    public int StandardRefId { get; set; }

    [ForeignKey("StandardRefId")]
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

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

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }

    public int StandardRefId { get; set; }
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

    [ForeignKey("StandardRefId")]
    public ICollection<Student> Students { get; set; }
}
```

InverseProperty

- The InverseProperty attribute is used when two entities have more than one relationship

```
public class Course
{
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public string Description { get; set; }

    public Teacher OnlineTeacher { get; set; }
    public Teacher ClassRoomTeacher { get; set; }
}

public class Teacher
{
    public int TeacherId { get; set; }
    public string Name { get; set; }

    [InverseProperty("OnlineTeacher")]
    public ICollection<Course> OnlineCourses { get; set; }
    [InverseProperty("ClassRoomTeacher")]
    public ICollection<Course> ClassRoomCourses { get; set; }
}
```

Required

The Required attribute can be applied to one or more properties in an entity class. EF will create a NOT NULL column in a database table for a property on which the Required attribute is applied

```
using System.ComponentModel.DataAnnotations;

public class Student
{
    public int StudentID { get; set; }
    [Required]
    public string StudentName { get; set; }
}
```

MaxLength

- The MaxLength attribute specifies the maximum length of data value allowed for a property which in turn sets the size of a corresponding column in the database. It can be applied to the string or byte[] properties of an entity.

```
using System.ComponentModel.DataAnnotations;

public class Student
{
    public int StudentID { get; set; }
    [MaxLength(50)]
    public string StudentName { get; set; }

}
```

StringLength

- The StringLength attribute can be applied to the string properties of an entity class. It specifies the maximum characters allowed for a string property which in turn sets the size of a corresponding column (nvarchar in SQL Server) in the database

```
using System.ComponentModel.DataAnnotations;

public class Student
{
    public int StudentID { get; set; }
    [StringLength(50)]
    public string StudentName { get; set; }
}
```

THE END