# Entity Framework in C#: A Comprehensive Guide

## Contents

# 1 Beginner Level: Understanding the Basics

## 1.1 What is an Entity in C#?

An entity in C# is a class that represents a table in a database, with each property mapping to a column in the table.

**Interview Answer:** "In C#, an entity is a class that defines the structure of data corresponding to a database table. For instance, a `Student` class with properties like `Id`, `Name`, and `Age` maps to a `Students` table in the database."

## 1.2 What is a Model Class in ASP.NET/Entity Framework?

A model class encapsulates data and business logic for a specific entity, forming part of the MVC pattern and mapping to a database table in Entity Framework.

**Interview Answer:** "In ASP.NET, a model class outlines the data structure and its rules. In Entity Framework, it also serves as an entity for database interactions."

## 1.3 What is Entity Framework (EF)?

Entity Framework is an Object Relational Mapper (ORM) that enables developers to interact with databases using C# classes instead of raw SQL queries.

**Interview Answer:** "Entity Framework is a Microsoft ORM tool that streamlines data access in .NET applications by allowing developers to manipulate data as .NET objects, reducing the need for manual SQL queries."

## 1.4 Advantages of Entity Framework

- Reduces boilerplate SQL code
- Automatically maps classes to database tables
- Supports LINQ queries for data retrieval
- Simplifies schema management with migrations
- Provides built-in change tracking and validation

**Interview Answer:** "EF enhances productivity by abstracting database operations, minimizes coding errors, and integrates seamlessly with ASP.NET projects."

## 1.5 What is an ORM (Object Relational Mapper)?

An ORM is a tool that facilitates automatic mapping between relational databases and object-oriented code.

**Interview Answer:** "ORMs like Entity Framework manage the mapping between object-oriented models and relational databases, allowing me to focus on C# code rather than SQL."

# 2 Entity Framework Approaches

## 2.1 Code First

In the Code First approach, entities are defined as C# classes, and Entity Framework generates the database schema based on these classes.

**Interview Quote:** "I define my entities in C#, and EF uses migrations to create or update the database schema."

### 2.2 Database First

In the Database First approach, an existing database is used, and Entity Framework reverse-engineers the tables into C# entity classes.

**Interview Quote:** "I use scaffolding to generate model classes from an existing database."

### 2.3 Model First

In the Model First approach, a visual designer is used to create the entity model, and Entity Framework generates both the code and the database schema.

**Interview Quote:** "I use a visual designer to define entities and relationships, and EF generates both the code and database based on that model."

## 3 Code First Approach

### 3.1 How to Define an Entity

```
public class Student {
    public int Id { get; set; }
    public string Name { get; set; }
}
```

### 3.2 What is DbContext?

`DbContext` acts as a bridge between the application and the database, managing entities, connections, and operations.

**Interview Answer:** "It handles entities, database connections, and operations like `SaveChanges` and migrations."

### 3.3 Create and Use DbContext

```
public class AppDbContext : DbContext {
    public DbSet<Student> Students { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(
        options) {}
}
```

### 3.4 Configure Connection String

In `appsettings.json`:

```
"ConnectionStrings": {
  "DefaultConnection": "Server=.;Database=MyDB;Trusted_Connection=True;
      "
}
```

Register in `Program.cs`:

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("
        DefaultConnection")));
```

### 3.5 What is DbSet?

`DbSet<T>` represents a table in the database within the `DbContext`.

**Interview Answer:** "Each `DbSet<T>` in `DbContext` corresponds to a database table."

### 3.6 What is OnModelCreating?

`OnModelCreating` is used to configure entity rules via the Fluent API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Student>().HasKey(s => s.Id);
}
```

### 3.7 Apply Migrations in Code First

Commands (EF Core CLI):

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

**Interview Answer:** "Migrations ensure the database schema stays in sync with the model."

## 4 Database First Approach

### 4.1 Generate Models

Use the `Scaffold-DbContext` command in the Package Manager Console:

```
Scaffold-DbContext "YourConnectionString" Microsoft.EntityFrameworkCore
    .SqlServer -OutputDir Models
```

### 4.2 Tools for Scaffolding

- EF Core CLI (`dotnet ef`)
- Visual Studio: Add > New Scaffolded Item

## 5 Relationships and Constraints

### 5.1 One-to-Many

```
public class Student {
    public int ClassId { get; set; }
    public Class Class { get; set; }
}
```

### 5.2 Many-to-Many

```
public class StudentCourse {
    public int StudentId;
    public Student Student;
    public int CourseId;
    public Course Course;
}
```

### 5.3 [ForeignKey], [Required]

```
[ForeignKey("ClassId")]
public int ClassId { get; set; }

[Required]
public string Name { get; set; }
```

### 5.4 Fluent API vs Data Annotations

- **Data Annotations:** Inline attributes on properties
- **Fluent API:** Code-based configuration in `OnModelCreating`

## 6 CRUD Operations with EF

### 6.1 Add

```
var student = new Student { Name = "Ravi" };
_context.Students.Add(student);
_context.SaveChanges();
```

### 6.2 Read

```
var students = _context.Students.ToList();
```

### 6.3 Update

```
student.Name = "Raj";
_context.Students.Update(student);
_context.SaveChanges();
```

### 6.4 Delete

```
_context.Students.Remove(student);
_context.SaveChanges();
```

### 6.5 SaveChanges()

Commits all changes to the database.

## 7 Intermediate/Advanced EF Core

### 7.1 Loading Strategies

| Type | Description |
|---|---|

| | |
|---|---|
| Lazy | Loads related data on demand |
| Eager | Uses `.Include()` to load data early |
| Explicit | Manually loads related data via code |

### 7.2 Migrations

Track and apply schema changes:

```
dotnet ef migrations add AddEmail
dotnet ef database update
```

### 7.3 Rollback

Delete the last migration:

```
dotnet ef migrations remove
```

### 7.4 Change Tracking

Entity Framework tracks entity changes and applies only updated values.

### 7.5 Shadow Properties

Properties not defined in the class but tracked in the model.

### 7.6 Seeding Data

```
modelBuilder.Entity<Student>().HasData(
    new Student { Id = 1, Name = "Admin" });
```

### 7.7 Concurrency Handling

Use the `[Timestamp]` attribute or a `RowVersion` column.

### 7.8 AsNoTracking()

Disables tracking for read-only queries to improve performance.

## 8 Testing & Real-Time Scenarios

### 8.1 Repository Pattern

Creates an abstraction layer between business logic and Entity Framework.

```
public interface IStudentRepo {
    Task<IEnumerable<Student>> GetAll();
}
```

### 8.2 Unit Testing with In-Memory DB

```
options.UseInMemoryDatabase("TestDB");
```

### 8.3 Performance Optimization

- Use `AsNoTracking()`
- Limit columns with `Select()`
- Use indexes
- Avoid lazy loading in loops

### 8.4 Include()

Used to eager load related data:

```
var students = _context.Students.Include(s => s.Class).ToList();
```

### 8.5 Raw SQL in EF Core

```
var students = _context.Students.FromSqlRaw("SELECT * FROM Students").
    ToList();
```