**1. Introduction to Entity Framework Core (EF Core)**

**Entity Framework Core (EF Core)** is an **Object-Relational Mapper (ORM)** developed by Microsoft that allows developers to work with databases using **.NET objects** instead of writing SQL queries manually.  
EF Core simplifies data access by mapping **C# classes to database tables**.

**2. Why Use EF Core**

* **Abstraction**: Developers can focus on business logic instead of SQL code.
* **Maintainability**: Changes to models can automatically update the database.
* **Portability**: Works with multiple databases like SQL Server, MySQL, PostgreSQL, and SQLite.
* **Productivity**: Reduces boilerplate data-access code.

**3. EF Core Architecture**

The EF Core architecture is built around the following key components:

|  |  |
| --- | --- |
| **Component** | **Description** |
| **DbContext** | The main class that manages database connections and transactions. |
| **DbSet** | Represents a table in the database; used for CRUD operations. |
| **Model** | Defines the structure of tables, relationships, and constraints. |
| **Migration** | Mechanism to keep database schema in sync with C# models. |
| **LINQ (Language Integrated Query)** | Allows writing database queries using C# syntax. |

**4. Database Approaches**

EF Core supports **two main approaches** for connecting applications with databases:

1. **Code First Approach**
   * The developer defines **C# classes**, and EF Core generates the database schema automatically.
   * Best suited for new projects where the database doesn’t exist yet.
2. **Database First Approach**
   * The database is already created.
   * EF Core automatically generates C# model classes and DbContext based on the existing tables.
   * Ideal for projects integrating with existing databases.

**5. Setting up EF Core in ASP.NET Core**

To use EF Core, the following steps are required:

**Step 1: Install EF Core Packages**

Run these commands in the terminal:

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

**Step 2: Configure Database Connection**

In the appsettings.json file, define a connection string:

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=StudentDB;Trusted\_Connection=True;TrustServerCertificate=True;"

}

**Step 3: Register DbContext in Program.cs**

builder.Services.AddDbContext<StudentDBContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

**Step 4: Create DbContext and Model**

* **Model (Entity)** → represents a database table.
* **DbContext** → manages database access and queries.

**6. Understanding DbContext**

DbContext is the heart of EF Core. It:

* Opens and closes database connections.
* Tracks entity changes.
* Executes SQL commands.
* Maps database tables to C# objects.

Example:

public class StudentDBContext : DbContext

{

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

}

**7. How EF Core Translates C# to SQL**

EF Core automatically converts your C# code into SQL statements.  
For example:

var data = context.Students.ToList();

Internally executes:

SELECT \* FROM Students;

**8. LINQ in EF Core**

**LINQ (Language Integrated Query)** allows writing queries directly in C#.  
Examples:

var student = context.Students.FirstOrDefault(s => s.Id == 1);

var topStudents = context.Students.Where(s => s.Grade == "A").ToList();

EF Core translates these into optimized SQL queries.

**9. EF Core and SQL Server Integration Steps**

1. Create a **SQL Server database** (e.g., StudentDB).
2. Create tables (e.g., Students table).
3. Scaffold database into C# using EF Core Database First.
4. Connect Web API with the database context.
5. Test data retrieval through Swagger or Postman.

**Testing the Connection**

Run the application and open Swagger UI.  
Check endpoints like:

GET /api/students

GET /api/students/{id}

If the database connection is correct, data will appear as JSON output.

**Benefits of EF Core in Real Projects**

* Simplifies CRUD operations.
* Automatically handles parameterization (prevents SQL injection).
* Integrates with ASP.NET Core dependency injection.
* Supports transactions and concurrency control.
* Works well with REST APIs and microservices.

**Common Issues and Fixes**

|  |  |  |
| --- | --- | --- |
| **Issue** | **Cause** | **Solution** |
| Connection string error | Wrong SQL Server name or missing database | Verify connection string in appsettings.json |
| Migration error | Missing EF Core tools | Install Microsoft.EntityFrameworkCore.Tools |
| Table not found | Wrong schema or table name | Check OnModelCreating in DbContext |
| Null reference on DbContext | Forgot to register DbContext | Register in Program.cs |

**Day-End Mini Task**

* Create a StudentsController with endpoints:
  + GET /api/students → Retrieve all students
  + GET /api/students/{id} → Retrieve a single student

**Snapshots :**

A screenshot of a computer

AI-generated content may be incorrect.

Code : Program.cs

A screenshot of a computer

AI-generated content may be incorrect.

Code : Student.cs

A screenshot of a computer program

AI-generated content may be incorrect.

Code : StudentDBContext.cs

A screen shot of a computer program

AI-generated content may be incorrect.

Code : StudentController.cs

A screenshot of a computer

AI-generated content may be incorrect.

Code : appsettings.json

A screenshot of a computer

AI-generated content may be incorrect.

Output : GET method

A screenshot of a computer

AI-generated content may be incorrect.

Output : GET (All data from database)

A screenshot of a computer

AI-generated content may be incorrect.

Output : GET (Spacific data from database)

**A screenshot of a computer program

AI-generated content may be incorrect.**

Code : Database Code