🧩 Unit 2 – Entity Framework Core (EF Core) and LINQ

🔷 Section 1 – Entity Framework Core (EF Core)

🟢 1.1 Introduction to EF Core

💡 What is EF Core?

EF Core (Entity Framework Core) is an **Object-Relational Mapper (ORM)**.
It lets you **interact with a database using C# classes** instead of writing SQL queries manually.

Example:

- Without ORM → You write SQL:
  SELECT * FROM Students WHERE StudentId = 1;
- With ORM (EF Core) →
  var student = _context.Students.Find(1);

EF Core automatically:

1. Generates the SQL in background,
2. Executes it on the database, and
3. Converts the table row into a C# object.

⚙️ Packages Needed

To use EF Core with SQL Server:

- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

🔶 What is ORM?

ORM (Object-Relational Mapper) is a software tool that connects **objects** in your code to **tables** in a database.

Database	C# Object
Table	Class
Row	Object (Record)
Column	Property

✅ Benefits of ORM:

- Write less SQL code
- Type-checking at compile time (reduces errors)
- Use LINQ for data access
- Auto table creation with migrations
- Easier to maintain and read

🧩 **Popular ORMs:**

- EF Core → C#/.NET
- Hibernate → Java
- Django ORM → Python
- Active Record → Ruby on Rails

🟣 **Steps to Set Up EF Core**

1. **Install Packages**
2. Microsoft.EntityFrameworkCore.SqlServer
3. Microsoft.EntityFrameworkCore.Tools
4. **Create a Model Class**
5. public class Student
6. {
7. public int StudentId { get; set; }
8. public string FullName { get; set; }
9. public string Email { get; set; }
10. }
11. **Create DbContext Class**
12. public class AppDbContext : DbContext
13. {
14. public DbSet<Student> Students { get; set; }
15. public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
16. }
17. **Add Connection String in appsettings.json**
18. "ConnectionStrings": {

19. "DefaultConnection": "server=Naimish;database=NRVDemo;trusted_connection=true;TrustServerCertificate=True;"
20. }
21. **Configure in Program.cs**
22. builder.Services.AddDbContext<AppDbContext>(
23. options => options.UseSqlServer(
24. builder.Configuration.GetConnectionString("DefaultConnection")));
25. **Create Migration & Database**
    - Add-Migration InitialCreate
    - Update-Database

🔷 **EF Core CRUD Operations**

Create

```
[HttpPost]
public IActionResult InsertStudent(StuStudent student)
{
   _context.StuStudents.Add(student);
   _context.SaveChanges();
   return NoContent();
}
```

Read

```
[HttpGet("All")]
public IActionResult GetStudents()
{
   var students = _context.StuStudents.ToList();
   return Ok(students);
}
```

Update

```
[HttpPut("{id}")]
public IActionResult UpdateStudent(int id, StuStudent student)
{
```

```
    var existing = _context.StuStudents.Find(id);
    if (existing == null) return NotFound();

    existing.StudentName = student.StudentName;
    existing.Age = student.Age;
    existing.Email = student.Email;

    _context.StuStudents.Update(existing);
    _context.SaveChanges();
    return NoContent();
}
```

Delete

```
[HttpDelete("{id}")]
public IActionResult DeleteStudentById(int id)
{
    var student = _context.StuStudents.Find(id);
    if (student == null) return NotFound();

    _context.StuStudents.Remove(student);
    _context.SaveChanges();
    return NoContent();
}
```

🟡 1.2 Code-First vs Database-First

Approach	Description
Code-First	You create C# classes first; EF Core builds the database from them.
Database-First	You already have a database; EF Core generates model classes from tables.

🟢 1.3 DbContext and DbSet

- **DbContext** → Main bridge between your code and database.
  It manages connection, tracks changes, and handles CRUD.

- public class HMSContext : DbContext
- {
- public DbSet<DocDoctor> DocDoctors { get; set; }
- public DbSet<PatPatient> PatPatients { get; set; }
- public DbSet<StuStudent> StuStudents { get; set; }
- }
- **DbSet<TEntity>** → Represents a table.
  Allows querying and saving data.

Example:

var allDoctors = _context.DocDoctors.ToList();

var doctor = _context.DocDoctors.Find(1);

_context.DocDoctors.Add(new DocDoctor { DoctorName="Naimish", Age=20 });

_context.SaveChanges();

🟣 1.4 Migrations

Migration keeps the database schema in sync with your model.

🧩 Commands

Task	Command
Add migration	Add-Migration InitialCreate
Apply to DB	Update-Database
Remove last	Remove-Migration

Files Created:

1. <timestamp>_<MigrationName>.cs → Contains Up() (create) and Down() (remove) methods.
2. <ContextName>ModelSnapshot.cs → Snapshot of current model.

Update-Database:

- Runs Up() for new migrations.
- Runs Down() to revert old ones.

🟠 1.5 Tracking vs No-Tracking Queries

EF Core can track changes to objects.

- **Tracking Query (default)** → Good for insert/update/delete.
- var student = context.Students.FirstOrDefault(s => s.Id == 1);
- student.Name = "Updated";
- context.SaveChanges(); // tracked & saved
- **No-Tracking Query** → Faster for read-only data.
- var students = context.Students
- .AsNoTracking()
- .Where(s => s.IsActive)
- .ToList();

✅ **Use Tracking** → when you need to modify data.
✅ **Use No-Tracking** → for reports or read-only queries.

🔵 1.6 Async / Await

Async/await helps your program run without freezing while waiting for tasks like database queries or API calls.

Example

- **Sync (non-async):** You stand idle until pizza arrives.
- **Async:** You play games while waiting; when pizza arrives, you pause and take it.

API Example:

```
[HttpGet]
public async Task<IActionResult> GetAllStudents()
{
   var students = await _context.StuStudents.ToListAsync();
   return Ok(students);
}
```

💡 Async = non-blocking & keeps the app responsive.

🔷 Section 2 – LINQ (Language Integrated Query)

🟢 2.1 Introduction to LINQ

LINQ lets you write queries directly inside C# code to fetch data from:

- Objects (in-memory collections)
- Databases (EF Core / SQL)
- XML files

✅ **Advantages**

- Same syntax for objects, DB, XML
- Cleaner & readable code
- Type safe (compile-time error check)
- IntelliSense support
- No SQL knowledge needed

🔵 **Query Syntax vs Method Syntax**

Syntax	Description	Example
Query	SQL-like syntax (begin with from)	from s in students where s.Marks > 75 select s
Method	Uses methods & lambda expressions	students.Where(s => s.Marks > 75).ToList()

In practice:
Method syntax is more powerful and used in real projects.

🟢 **Query Syntax Example**

```
var result = from s in students
             where s.Marks >= 75
             orderby s.Name
             select s;
```

🟣 **Method Syntax Example**

```
var result = students
   .Where(s => s.Marks >= 75)
   .OrderBy(s => s.Name)
   .ToList();
```

🔶 **Important LINQ Operators**

Type	Operators	Purpose
Filtering	Where	Selects records based on condition
Projection	Select, SelectMany	Picks specific fields or flattens lists
Sorting	OrderBy, OrderByDescending, ThenBy	Arranges data
Aggregation	Sum, Min, Max, Count, Average	Performs math operations
Grouping	GroupBy	Categorizes data
Quantifiers	Any, All	True/false checks
Join	Join	Combines two collections

🔵 Where Operator

Select records based on conditions:

```
var result = students.Where(s => s.Sem == 4 && s.Branch == "CE");
```

🟢 Select / SelectMany

- **Select** → Chooses specific columns.
- **SelectMany** → Flattens nested collections into one list.

🔵 Aggregate Operators

Method	Purpose
Sum()	Total of numbers
Min()	Smallest value
Max()	Largest value
Count()	Number of records
Average()	Average value

Example:

```
var avg = students.Average(x => x.CPI);
```

🟣 Sorting Operators

Operator	Description
OrderBy	Ascending order
OrderByDescending	Descending order
ThenBy	Secondary sort (ascending)
ThenByDescending	Secondary sort (descending)

Example:

```
var sorted = employees.OrderBy(e => e.FirstName)
                      .ThenBy(e => e.LastName);
```

🟢 **GroupBy**

Groups data by a specific key:

```
var groups = students.GroupBy(s => s.Branch);
```

You can group by multiple columns:

```
.GroupBy(t => new { t.Subject, t.Age });
```

🟣 **Any / All**

- Any() → Checks if *at least one* record matches.
  Example: students.Any(s => s.CPI > 8)
- All() → Checks if *all* records match.
  Example: students.All(s => s.Sem > 2)

🔵 **First / FirstOrDefault**

Method	Behavior
First()	Returns first matching record — throws error if none
FirstOrDefault()	Returns first matching record or default (null)

🟠 **Join**

Combines two collections (tables) based on common key — like SQL INNER JOIN.

Example:

```
var result = _context.Students
```

```
    .Join(_context.Departments,
        s => s.DepartmentId,
        d => d.Id,
        (s, d) => new { s.Name, Department = d.Name })
    .ToList();
```

SQL Equivalent:

```
SELECT s.Name, d.Name FROM Students s
INNER JOIN Departments d ON s.DepartmentId = d.Id;
```

✅ **Summary for Exam**

Topic	Easy Definition
EF Core	Framework to work with DB using C# objects
ORM	Converts tables ↔ classes
DbContext	Bridge between DB and C#
DbSet	Represents a table
Migration	Sync model with database
Async/Await	Handles non-blocking operations
LINQ	Query language inside C#
Where	Filter data
Select	Choose fields
OrderBy	Sort data
GroupBy	Categorize records
Aggregate Functions	Sum, Count, Avg, Min, Max
Any/All	Condition checks
Join	Combine two tables