Entity Framework Core

ARCTECH INFO PRIVATE LIMITED

Entity Framework Core

Entity framework is an ORM (Object Relational Mapping) tool

- ORM is a technique of accessing a relational database

Earlier developers had to write ADO.NET code to save or retrieve data from underlying database

- Prior to .NET 3.5
- This was a cumbersome and error prone process
- Microsoft introduced a framework called “Entity Framework” to automate all database related activities

Developers can work with objects and properties rather than database tables and columns

Data-oriented applications can be created with less code compared with traditional applications.

EF Core classes (.NET 5)

`DbContext` is an integral part of Entity Framework

It represents a session with the database

It includes `DbSet` objects which represent the tables

It can be used to query from a database

It is used to group together changes like insert, update, delete

These changes will be written to the underlying database as a single unit

Use EF Core (.NET 5)

Install the necessary Nuget packages

- Microsoft.EntityFrameworkCore & Microsoft.EntityFrameworkCore.SqlServer
- Or Oracle.EntityFrameworkCore for Oracle Database (ver 5)
- *Note: For .NET 5 install the 5.x.y versions of the packages*

Create a folder in the project, called [Data] and a class in this folder called `ApplicationDbContext` inheriting from .

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(
        DbContextOptions<ApplicationDbContext> options)  : base(options)
    {
    }
}
```

Use EF Core (.NET 5)

In Startup.cs add EF Core Service to the DI Container as below (Program.cs for .NET 6)

-
  ```
  services.AddDbContext<ApplicationDbContext>(
      options => options.UseSqlServer(
          Configuration.GetConnectionString("DefaultConnectionString")));
  ```
- In case of oracle use the following
  ```
  services.AddDbContext<ApplicationDbContext>(
      options => options.UseOracle(
          Configuration.GetConnectionString("DefaultConnectionString")));
  ```

EF Core is ready to use in your project.

Use EF Core (.NET 5)

Now for any table in the database which your application requires

- Create a Model class with the same name as the Table (usually in the Models folder), and properties matching the fields in the Student table. E.g.,

```
public class Student
{
      [Key]
      public int RollNo { get; set; }
      public string Name { get; set; }
}
```

- Create a property in the `ApplicationDbContext` class of type `Microsoft.EntityFrameworkCore.DbSet<T>`

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

Use EF Core (.NET 5)

In any controller or service class, use the `Students` property from the `ApplicationDbContext` class to perform CRUD operations as below

- Inject the `ApplicationDbContext` in any controller or service class. See example below.

```
public StudentsController(ApplicationDbContext applicationDbContext)
{
    _applicationDbContext = applicationDbContext;
}
```

- Select all students

```
        var students = await _applicationDbContext.Students.ToListAsync();
```

- Insert a student

```
        await _applicationDbContext.Students.AddAsync(student);
```

- Update a student

```
        _applicationDbContext.Update(student);
```

- Delete a student

```
        _applicationDbContext.Students.Remove(student);
```

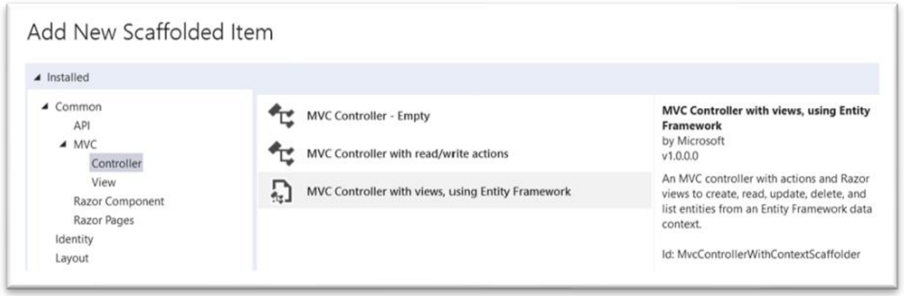
Points to consider

- Insert, update and delete statements are performed as a unit by EF Core
- So, to commit the changes to the database you have to call the following before exiting the controller or service method

```
        await _applicationDbContext.SaveChangesAsync();
```

Scaffolding MVC (.NET 5+)

Once you have the Model class and `ApplicationDbContext` setup, Visual Studio can automatically generate the Controller and View files with readymade code



Add New Scaffolded Item

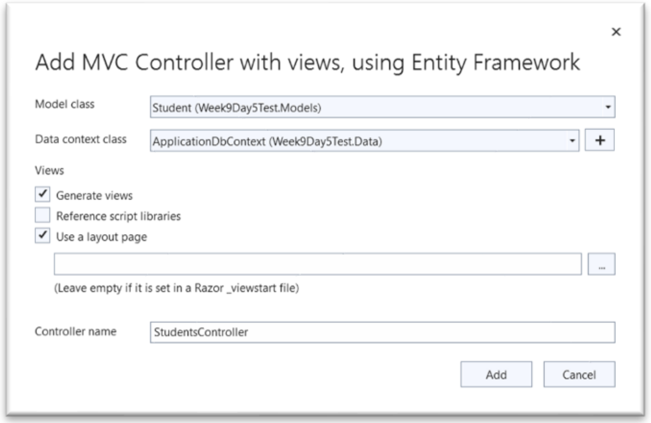
- Installed
  - Common
    - API
    - MVC
      - Controller
      - View
    - Razor Component
    - Razor Pages
  - Identity
  - Layout

MVC Controller - Empty
MVC Controller with read/write actions
MVC Controller with views, using Entity Framework

MVC Controller with views, using Entity Framework
by Microsoft
v1.0.0.0

An MVC controller with actions and Razor views to create, read, update, delete, and list entities from an Entity Framework data context.

Id: MvcControllerWithContextScaffolder



Add MVC Controller with views, using Entity Framework

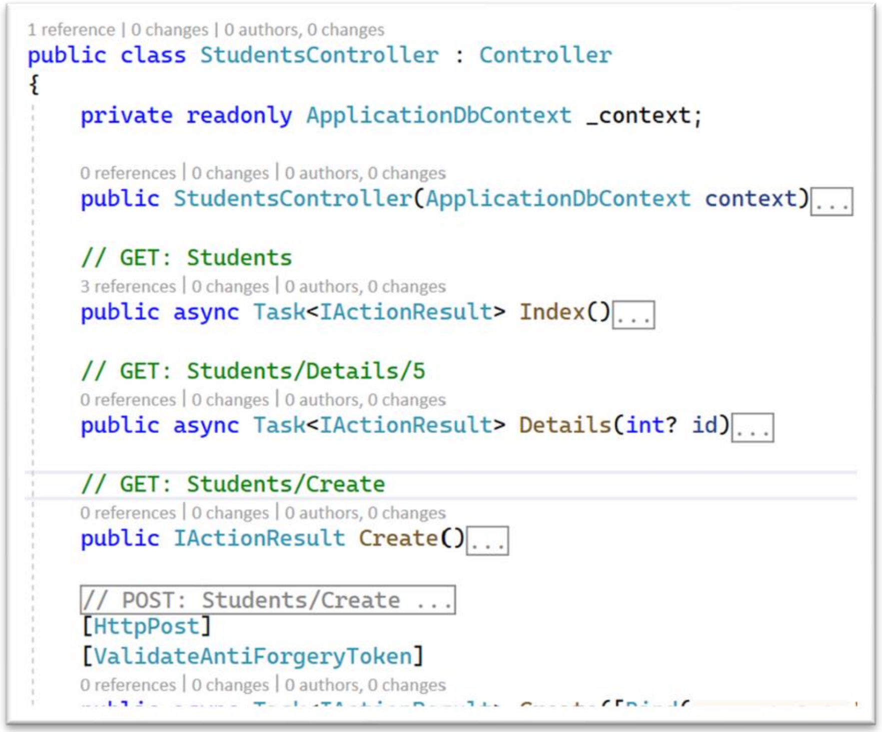
Model class	Student (Week9Day5Test.Models)
Data context class	ApplicationDbContext (Week9Day5Test.Data)

Views
- [✓] Generate views
- [ ] Reference script libraries
- [✓] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Controller name	StudentsController

Add   Cancel



```
1 reference | 0 changes | 0 authors, 0 changes
public class StudentsController : Controller
{
    private readonly ApplicationDbContext _context;

    0 references | 0 changes | 0 authors, 0 changes
    public StudentsController(ApplicationDbContext context)...

    // GET: Students
    3 references | 0 changes | 0 authors, 0 changes
    public async Task<IActionResult> Index()...

    // GET: Students/Details/5
    0 references | 0 changes | 0 authors, 0 changes
    public async Task<IActionResult> Details(int? id)...

    // GET: Students/Create
    0 references | 0 changes | 0 authors, 0 changes
    public IActionResult Create()...

    // POST: Students/Create ...
    [HttpPost]
    [ValidateAntiForgeryToken]
    0 references | 0 changes | 0 authors, 0 changes
```

EF Core – Two methodologies

Database First

- Using SQL, create the database tables and stored procedures in your preferred database
- Using C#, create the entities, method calls to stored procedures and the `ApplicationDbContext` class
  - Example: `_dbContext.Database.ExecuteSqlRawAsync("EXEC DeleteUser", sqlParameters)`
- Preferably use reverse engineering tools to automatically generate the Models and `ApplicationDbContext` class.
  - `Scaffold-DbContext` scaffolding command or Visual Studio extension: EF Core Power Tool

Code First

- Using C#, Create the entities classes as required by the application and the `ApplicationDbContext` class
- Let EF Core manage creation of the database objects like tables, indexes, relationship etc.
  - `Add-Migration` scaffolding command followed by `Update-Database` command
    - *Note: You need to install nuget package `Microsoft.EntityFrameworkCore.Tools` & `Microsoft.EntityFrameworkCore.Design`*

POCO Entities

A POCO entity is a class that doesn't depend on any framework-specific base class.

It is like any other normal .NET CLR class, which is why it is called "Plain Old CLR Objects".

The following is an example of Employee POCO entity.

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal BaseSalary { get; set;
    public float TaxRate { get; set; }

    public Address Address { get; set; }

    public Grade Grade { get; set; }
}
```

LINQ

Language-Integrated Query (LINQ) - integration of query capabilities directly into the C# language

Problem with traditional queries

- Expressed as simple strings
- No Type checking at compile time
- No IntelliSense support
- You have to learn a different query language for each type of data
  - For e.g. Structured Query Language (SQL) for Relational Databases

With LINQ,

- A query is a first-class language construct like classes, methods, etc.
- Queries are written by using language keywords and familiar operators on strongly typed collections
- You have a consistent query experience across multiple data source types

LINQ

Query expressions are written in a declarative query syntax.

By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code.

You use the same basic query expression patterns to query and transform data.

You can write LINQ queries in C# on any data that supports the `IEnumerable` or `IEnumerable<T>` interface

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
                from score in scores
                where score > 80
                select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

LINQ

LINQ binds the gap between relational and object-oriented approaches

LINQ speeds up development time as

- C# developers do not have to learn SQL
- It catches errors at compile time
- includes IntelliSense & Debugging support.

LINQ expressions are Strongly Typed.

See an example of IntelliSense when selecting all orders in last 2 days

```
var ordersQuery =
  from order in _dbContext.Orders
  where order.OrderDate >= DateTime.Now.AddDays(-2)
  select order;

var orders = ordersQuery.AsEnumerable();

foreach (var order in orders)
{
    // order
}
```

LINQ expressions overview 1/2

Query expressions can be used to query and to transform data from any LINQ-enabled data source.

Query expressions are easy to grasp because they use many familiar C# language constructs

The variables in a query expression are all strongly typed

A query is not executed until you iterate over the query variable

Any query that can be expressed by using query syntax can also be expressed by using method syntax.

```
IEnumerable<int> scoreQuery =
                from score in scores
                where score > 80
                select score;
```

```
IEnumerable<int> scoreQuery =
        scores.Where(score => score > 80);
```

LINQ expressions overview 2/2

As a rule, when you write LINQ queries, we recommend that

- you use query syntax whenever possible and
- method syntax whenever necessary.

There is no semantic or performance difference between the two different forms.

Query expressions are often more readable than equivalent expressions written in method syntax.

Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call.

Method syntax can be combined with query syntax in various ways

Lamda Expression 1/2

```
int[] numbers =
    { 5, 10, 8, 3, 6, 12};

//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

//Method syntax:
IEnumerable<int> numQuery2 = numbers.
    Where(num => num % 2 == 0).
    OrderBy(num => num);
```

Notice that the conditional expression (n % 2 == 0) is passed as an in-line argument to the Where method

```
Where(num => num % 2 == 0)
```

This inline expression is called a lambda expression

It is a convenient way to write code that would otherwise be very cumbersome

```
int[] numbers = { 5, 10, 8, 3, 6, 12};
numbers.Where(num => num % 2 == 0)
```

Lamda Expression 2/2

In C# => is the lambda operator and is read as "goes to”

The code inside the Where method is executed for every element in the array

The num on the left of the operator is the input variable which corresponds to num in the query expression and represents an element of the array

To get started using LINQ, you do not have to use lambdas extensively.

However, certain queries can only be expressed in method syntax and some of those require lambda expressions.

After you become more familiar with lambdas, you will find that they are a powerful and flexible tool