My Personal Notes, A Technical Learning Report

//program.cs//

var builder = WebApplication.CreateBuilder(args);

- This single line initializes the whole .NET web framework.

WebApplication?

https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication?view=aspnetcore-10.0

is from using Microsoft.AspNetCore.Builder namespace an Microsoft.AspNetCore.App NuGet package that gets installed by defualt with .net!

WebApplication is a wrapper class that wraps around the Host structure.
https://www.c-sharpcorner.com/blogs/wrapper-class-in-c-sharp1

This class:
* Holds the Host (Ihost) // the main engine of the program that manages everything. Witch the programcan not work without

* Generates the WebApplicationBuilder
https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder?view=aspnetcore-10.0
prepares **everything** your web app needs before it actually runs: configuration, logging, dependency-injection container, environment info, and the web-server defaults (Kestrel + routing). It's a convenience wrapper that wires a Generic Host and Web Host together with sensible defaults.

* Handles all the web-specific stuff (Routing, Endpoints, Middleware)

.builder()?

builder is an instance of WebApplicationBuilder.

WebApplicationBuilder exposes useful properties:

- builder.Services → IServiceCollection (where you register DI services)
- builder.Configuration → IConfigurationBuilder/IConfiguration (app settings)
- builder.Environment → IHostEnvironment (Development/Production, content root, etc.)
- builder.Logging → logging config
- builder.WebHost → IWebHostBuilder (web host specific settings)
- builder.Host → IHostBuilder (generic host builder)

WebApplication.CreateBuilder(args) performs several coordinated steps (simplified):

1. **Create a HostBuilder** (Host.CreateDefaultBuilder(args)) — the generic host foundation.
2. **Set up default configuration providers** (appsettings.json, env-specific json, user secrets in dev, environment variables, command-line args. In this exact order).
3. **Set up default logging providers** (Console, Debug, EventSource by default).
4. **Create an IServiceCollection** and register many default services used by ASP.NET Core (routing, options, endpoint routing, controllers support if needed later, etc.).
5. **Create a IWebHostBuilder** and apply *web defaults* (register Kestrel, set content root, enable IIS integration hooks, etc.).
6. **Expose all those pieces as WebApplicationBuilder properties** so it can customize before building the final WebApplication.

One of the services that gets registers is: DI — (ServiceCollection / ServiceProvider) Lifetime management, dependency injection, and service structure.

https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

https://learn.microsoft.com/en-us/iis/get-started/introduction-to-iis/introduction-to-iis-architecture

Args?

args is the string[] passed from the command line (same as Main(string[] args)). They are parsed into configuration so you can override values at runtime:

Example: dotnet run --Urls "http://*:5001"

The --Urls argument would be read and applied to the config, affecting which addresses Kestrel listens on.

AddDbContext

https://learn.microsoft.com/en-us/ef/core/dbcontext-configuration/

what DbContext do?
* Creates the connection

* Maintains tables as entities

* Translates models to SQL

* Tracks and stores changes

* Manages transactions

builder.Services.AddDbContext<BlogContext>(options => options.UseMySql(builder.Configuration.GetConnectionString("DefaultConnection"), new MySqlServerVersion(new Version(8,0,32)), mySqlOptions => mySqlOptions.EnableRetryOnFailure()));

- This line registers a DbContext called BlogContext in the application services and tells EF Core to use MySQL, with that connection string, with the specified server version (8.0.32), and with the ability to automatically retry on transient errors.

After executing this line:

When in code we request BlogContext somewhere (like a controller), DI creates a BlogContext instance with the DbContextOptions<BlogContext> that we set — that is, with the connection string, MySQL provider, and retry settings.

The instance is disposed (connections are closed) when the scope ends (the HTTP request ends).

builder.Services

AddDbContext inside builder.Services records that for each HTTP request Create a new scoped BlogContext.

builder.Services is an IServiceCollection — a "list of services" that the application converts into a container (ServiceProvider) when Build() is executed. Any service registered here will be able to be injected into the rest of the code in a specific way (Scoped/Singleton/Transient).

AddDbContext<BlogContext>(...)

This method (from the Microsoft.EntityFrameworkCore package) performs a default registration that:

Registers BlogContext as a Scoped service (one instance per Scope — for each HTTP request create a new separate DbContext scope).

Why Scoped? Because DbContext has state (change tracking) and should not be shared as a Singleton between requests. Each web request usually requires an independent instance to avoid interference and to allow for automatic disposal.

In addition to the BlogContext itself, it also places DbContextOptions<BlogContext> in the container so that EF knows how to instantiate it.

DbContextOptions / Options => …

Inside the parentheses we provide a delegate that sets up the DbContextOptionsBuilder. Here we are telling EF which provider and what options to use (like in here MySQL, connection string, and server version).

Chosing the provider

Enables the UseMySql provider

Dictionary "C# to MySQL Conversion Rules"

Building SQL queries

MySQL behaviors (LIMIT, DATE, CHARSET, AUTO_INCREMENT)

Version to be followed (8.0.32)

builder.Configuration.GetConnectionString("DefaultConnection")

builder.Configuration is the IConfiguration that CreateBuilder created — that is, a combination of appsettings.json, environment variables, command-line args, etc.

GetConnectionString("DefaultConnection") reads the connection string value from the ConnectionStrings:DefaultConnection section of the configuration.(from appsetings.json)

pick a version and EnableRetryOnFailure

EnableRetryOnFailure() tells EF Core to use an Execution Strategy that retries operations multiple times in the event of temporary network errors or timeouts (such as a short disconnect).

Benefit: In cloud or network environments where transient errors are common, simple requests like connecting or executing a query are automatically retried, and you'll encounter fewer errors.

Build()

- Host is actually get built

builder.Build() does the following:

* Final DI container is built

* All services are compiled (CreateServiceProvider)

* WebApplication pipeline is prepared

* Kestrel server is configured

* Middleware is stored in builder list

* A WebApplication is built which is basically a wrapper on Ihost

Migration and Seed

App is an object of WebApplication and beacuase of DI everything that I wrote for seeding and migrating in DataExtention.cs file is gonna happen and called on this object.

MigrateDbAsync() and SeedDbAsync() are not by defualt WebApplication methods but they are extention methods. So they can be access when we using the right namespace for it. And because this methods are coded that have this WebApplication app for there first prameter and also written as statics methods on static class.

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members?redirectedfrom=MSDN

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static

Using await allows the runtime to do other work while performing I/O (although there may not be other work at startup, it is future-proof and standard behavior).

we want to make sure that the migration and seed are complete before the server starts listening. If the server responds to traffic before the migration is complete, a race condition may occur between the requests and the schema (for example, a request trying to read a table that has not yet been created).

If the migration encounters an error, await + throw will cause the startup to stop and the host to not start until you fix the error (instead of coming up with a half-configured database).

in general these 2 line make sure the data base is fully updated befor listening to the port. learn more in DataExtionsion.cs review part.

Run()

The moment the application enters "running state". Detailed description:

* Host.StartAsync is executed
* Kestrel server is started
* Starts listening on URLs
* Middleware pipeline is created and frozen
* Every HTTP request is sent to this pipeline

//DataExtension.cs//

why static?

These methods are Extension method they must be static. Extention methodes won't compiles unless they are static.

These methods are not dependent on the internal state of the class

The extension content has no private state, such as: private fields, properties nor constructors

we also don't want any object created from this class, this class is just a utility class So being static makes perfect sense.

We also have these rule that if the method is static the class of it should be static as well.

Why async?

https://learn.microsoft.com/en-us/shows/c-advanced/introduction-to-async-await-and-tasks--c-advanced-5-of-8

https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/

https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/

- Task represents an asynchronous operation.

Database operations are **I/O-bound** and if they use syncronize the could be **blocking thread.**

Migration could take a while by making it async the thread is free and cpu is also free to do anything else. So it is recommended that for operation related to database we use the async version of methods. It prevent deadlock.

Microsoft Docs said, "Do not block on async calls. Use async all the way

So we chose these versions MigrateAsync(), EnsureCreatedAsync(), SaveChangesAsync(), ExecuteSqlAsync() instead and make it asyncronize.

Also there is an c# rule that says if inside an method we have await, the whole methos should be async. Otherwise compiler will get and error.

public static async Task MigrateDbAsync(this WebApplication app)

public static → Since this is an extension method, it must be static and placed in a static class. public so that it is accessible in the Program.cs file.

async Task → The method has no return value but is async; the caller awaits it.

(this WebApplication app) → This is the syntax that says this method can be called as the app.MigrateDbAsync() method. The WebApplication type indicates that the method is dependent on the overall application context (so that it can access app.Services).

using var scope = app.Services.CreateScope();

Why need scope? Because BlogContext is registered as Scoped in AddDbContext.

Scoped means: an instance is created for each scope. On the web, each request has a separate scope; but now in startup we are in a non-request context, so we need to manually create a scope to use scoped services (like DbContext).

app.Services -> This is the IServiceProvider or root service provider created in builder.Build().

CreateScope() -> Creates a new IServiceScope that has a subcontainer (scope) and holds the scoped lifetimes in it.

using var scope = ... -> scope is disposable; when it exits the block, scope.Dispose() is called and any scoped service that is disposable (like BlogContext) is disposed — important for closing connections.

var dbContext = scope.ServiceProvider.GetRequiredService<BlogContext>();

scope.ServiceProvider → is the service provider specific to that scope (suitable for resolving scoped services).

GetRequiredService<BlogContext>() → asks DI to create a BlogContext. If it is not registered.

How is a BlogContext created?

https://learn.microsoft.com/en-us/ef/core/

When we execute AddDbContext<BlogContext>(options => ...) in Program.cs, a ServiceDescriptor is registered in the IServiceCollection, which contains:

a factory to create a DbContextOptions<BlogContext> based on options => ...

a registration for the BlogContext itself (Scoped), whose constructor accepts BlogContext(DbContextOptions<BlogContext> options).

At resolve time, the ServiceProvider executes the factory, constructs the DbContextOptions<>

BlogContext typically inherits from the DbContext base class and implements things like getting the IModel, IDbContextOptions, and ChangeTracker in the constructor

Result: The dbContext is a new instance that is ready to perform DB operations.

var logger = scope.ServiceProvider.GetRequiredService<ILogger<Program>>();

This is a generic logger from the Microsoft.Extensions.Logging system. With ILogger<Program> the category is usually named Program.

Why do we get from scope.ServiceProvider? Logger is usually Singleton but the simple logic is to get all services from the same provider.

Logger is used to report success or error.

Try and catch block

dbContext.Database.MigrateAsync()

This command forces EF Core to:

Check what migrations are in the Migrations folder (files generated by dotnet ef migrations add ...)

Read the state of the __EFMigrationsHistory table in the database (this table shows which migrations have already been run).

Take any migrations that have not yet been run, generate the necessary SQL (MigrationBuilder → SQL generator associated with the provider), and execute that SQL within a transaction or set of operations.

Every migration that runs add it's record in __EFMigrationsHistory table.

If this whole thing is successful we log its success otherwise Any errors are logged and then throw; re-raises the error (not throw ex;).

Why throw;? Because the original stack trace is preserved; throw ex; overwrites the stack trace.

Goal: If migration fails, the program should not continue running silently; usually we want startup to fail so that the error is detected and the program does not start.

SeedDbAsync mostly works like the this. Have an scoped make dbcontext and the logger try to do seeding and use DbInitializer that makes an admin if there isn't already one in our user tabel and also make some category for the start.

Notice the intraction with database has await so our methos is async

//BlogContex.sc//

“BlogContext is a DbContext — meaning EF Core will use it for tracking entities, querying, saving changes, and interacting with the database.”

DbContext is the EF Core base class. It represents a **session** with the database.

DbContext is a concrete class not a abstract class, we do not have to override it’s method however it has some virtual methods like OnModelCreating that we can override so we can expand the method and what it dose.

DbContext uses **Template Method Pattern** meaning the base dbcontext controls the main algorithm make some part virtual that we can override only this parts.

> **DbContext is concrete. because it provides behavior,**
> **but inheritance is required because the behavior must be specialized for each domain**

So blogcontexts is concrete class too it just customizes the base behavior

DbContext is concrete but not generic It has no knowledge of this specefic project models . Subclass is the only way to introduce models, DbSets are only meaningful in subclasses , Fluent API can only be used via overrides , Tooling and migration require subclasses

https://github.com/dotnet/efcore/blob/main/src/EFCore/DbContext.cs

public BlogContext(DbContextOptions<BlogContext> options) : base(options) {}

This is the constructor.
It is required because DI will pass DbContextOptions<BlogContext> into it.

This object contains:

- Database provider (MySQL)
- Connection string
- Lazy loading settings
- Query tracking settings
- Logging settings
- Migrations settings

https://medium.com/@wgyxxbf/introduction-to-lazy-objects-in-c-b5276ec981b9

https://learn.microsoft.com/en-us/ef/core/querying/tracking

This options object comes from Program.cs:

```
builder.Services.AddDbContext<BlogContext>(options =>
    options.UseMySql(...)
);
```

EF Core builds the options here, then injects them into the constructor.

: base(options)

This passes the options to the **base DbContext class**.

This is how EF Core learns:

- which database to use
- how to connect
- configuration settings

Empty body {}

There is nothing inside the constructor.

But just having the signature is enough so DI can pass the options. With out this constructor the DbContext won't work with DI
EF Core requires a constructor that matches: DbContextOptions<TContext> options

If we remove the constructor, EF Core tries to use:

- A parameterless constructor

But this does **not** work in ASP.NET Core unless you manually configure the DbContext differently.

DbSets — Your Database Tables

A DbSet is:

- A *C# representation* of a database table
- A way to query and manipulate that table
- What EF Core tracks internally

EF Core uses TWO rules to name the actual table:

1. Class name = Post

2. Property name = Posts

By default table name becomes **Posts**.

{ get; set; }

Required for EF Core to create and track the DbSet.

7.5 = null!;

EF Core initializes DbSets at runtime.

But the compiler warns:

> "Non-nullable property Posts must contain a non-null value."

To silence the warning, we use the *null-forgiving operator*:

null!

This means:

- “Compiler, trust me — EF Core will initialize this.”

This is the standard EF Core pattern.

OnModelCreating — Fluent API Configuration

This method customizes how EF Core builds the database schema.

https://learn.microsoft.com/en-us/ef/core/managing-schemas/scaffolding/?tabs=dotnet-core-cli

https://learn.microsoft.com/en-us/ef/core/querying/tracking

lives inside DbContext, for Explicitly configuring how EF should map C# classes to database tables.

OnModelCreating is where I configure database rules.

protected override

This is EF Core’s template method.
You override it to configure entities manually.

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override

ModelBuilder modelBuilder

EF Core gives you this builder so you can configure:

- Primary keys
- Relationships

- Table names
- Indexes
- Constraints
- Entity configurations

https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.modelbuilder?view=efcore-10.0

base.OnModelCreating(modelBuilder);

Always call this first.

The base class may contain built-in configurations (especially if using Identity).

Configuring PostCategory (Many-to-Many Join Table)

relationship:

- Post ↔ many ↔ PostCategory ↔ many ↔ Category

Composite Key

```
modelBuilder.Entity<PostCategory>()
   .HasKey(pc => new { pc.PostId, pc.CategoryId });
```

> “Primary key is a combination of both PostId and CategoryId.”

This is the classical relational DB design for many-to-many join tables.

Relationship: PostCategory → Post

```
modelBuilder.Entity<PostCategory>()
   .HasOne(pc => pc.Post)
   .WithMany(p => p.PostCategories)
   .HasForeignKey(pc => pc.PostId);
```

- each PostCategory has one Post
- each Post can have many PostCategory rows
- PostId is the FK

Relationship: PostCategory → Category

```
modelBuilder.Entity<PostCategory>()
   .HasOne(pc => pc.Category)
   .WithMany(c => c.PostCategories)
   .HasForeignKey(pc => pc.CategoryId);
```

Same logic as above.

//mini_blog.csproj//

When I look at .csproj, I should think:

> “This file defines:
>
> - what kind of app I’m building
> - which .NET version I use
> - how strict the compiler is
> - which libraries exist
> - which tools are available”

It is **not config**.
It is **architecture**.

<Project Sdk="Microsoft.NET.Sdk.Web">

this line literally defines what kind of application this is.

It tells the .NET build system:

> “This project is a **Web project** and should use the **Web SDK**.”

What does the Web SDK give me?

By choosing Microsoft.NET.Sdk.Web, .NET automatically adds:

- ASP.NET Core runtime support
- Kestrel web server
- Middleware infrastructure
- Routing
- Minimal APIs
- Controllers
- Razor support (even if unused)
- Web-specific build targets

dotnet build, dotnet run, dotnet publish, Visual Studio / Rider reads this

<PropertyGroup>

This is a **configuration block**.

Everything inside it controls **how the compiler behaves**.

Think of it as:

> “Global rules for compiling this project”

<TargetFramework>net8.0</TargetFramework>

This line answers:

> “Which .NET runtime am I targeting?”

<TargetFramework>net8.0</TargetFramework>

Compile against **.NET ,** Use .NET 8 API, Require .NET 8 runtime to run

Why this matters A LOT

- Language features depend on it
- Available libraries depend on it
- EF Core version compatibility depends on it
- Hosting model depends on it

<Nullable>enable</Nullable>

This enables **Nullable Reference Types (NRT)**.

Meaning:

- string ≠ string?
- Null is treated as a *compile-time concept*
- Compiler warns you about possible null bugs

Why EF Core forces null!

Because EF Core initializes things **at runtime**, not in constructor.

Without nullable enabled:

- You’d miss many bugs
- But code would be lazier

With nullable enabled:

- Compiler forces you to be explicit
- Safer, more correct code

What if I disable it?

<Nullable>disable</Nullable>

- No null warnings
- Easier writing
- More runtime NullReferenceExceptions

Professional projects → **always enable**

<ImplicitUsings>enable</ImplicitUsings>

This is why you **don’t write these manually** anymore:

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
```

.NET injects them automatically.

What namespaces are added automatically?

Depends on SDK, but for Web:

- System
- System.Linq
- System.Collections.Generic
- Microsoft.AspNetCore.*
- Microsoft.Extensions.*

What if I disable it?

<ImplicitUsings>disable</ImplicitUsings>

Then:

- You must manually write all using statements
- Program.cs becomes long and ugly
- Nothing breaks, just annoying

This is purely a **developer experience feature**.

<ItemGroup>

This block lists **dependencies**.

Everything here becomes:

- NuGet package
- Available at compile time
- Sometimes runtime
- Sometimes design-time only

https://learn.microsoft.com/en-us/nuget/consume-packages/package-references-in-project-files

<PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.10" />

This is EF Core itself.

It gives us: DbContext, DbSet, ChangeTracker, LINQ → SQL translation, Migrations core logic, Query pipeline

This is the engine.

<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="9.0.10">

This package is NOT for runtime.

It exists for: dotnet ef migrations add, dotnet ef database update, Design-time DbContext discovery, Scaffolding, Tooling

IncludeAssets

- Specifies which **types of assets you want to include** from the package in your project.
- Only the specified assets will be used.

What Are "Assets" in a NuGet Package?

A NuGet package can contain different kinds of files and metadata that are called **assets**, such as:

- **Compile:** DLLs and assemblies used when compiling your project (the main code).

- **Runtime:** DLLs used at runtime (when the app runs).
- **ContentFiles:** Static files (like images, scripts) added to your project.
- **Build:** MSBuild targets and props files that can affect the build process.
- **Native:** Native (unmanaged) libraries.
- **Analyzers:** Code analyzers or Roslyn analyzers included in the package.
- **None:** Other content that does not belong in the above categories.

PrivateAssets>all</PrivateAssets> :

Package is NOT included in published output

NOT deployed to production

Used only at development time

This is tooling only.

"This package should only be used **privately** by *this* project. Don't propagate this package as a dependency downstream to any projects that consume my project."

<PackageReference Include="Pomelo.EntityFrameworkCore.MySql" Version="9.0.0" />

This is EF Core MySQL provider.

Translates LINQ → MySQL SQL

Handles MySQL limitations

Supports MySQL/MariaDB