



Dotnet Core Docs

This is training documentation for dotnet core

Table of Contents

Dotnet Core Documentation	2
Minimal Apis	30

Dotnet Core Documentation

Introduction to Minimal API

We will create an event API to build on the concept of minimal APIs. This API will help you learn about:

1. Project structure
2. REST API
3. CRUD Functionality

We will use an in-memory database for now, meaning the data will not be permanently saved to the database. When the a### How it Works in .NET Core

- ASP.NET Core has a built-in DI Container. Just the same way we registered the DbContext, the DI tells the system "When someone asks for an `AppDbContext`, here's how to create it."
- Later, when your app needs that `AppDbContext` in a class (e.g., a controller or service), you don't create it manually — the system **automatically injects it for you**.
- An example is registering `AppDbContext` with the In-Memory Database. It will register **`AppDbContext`** as a service. It then tells the system to use an in-memory database when creating `AppDbContext`.
- Later, when we create a class, here's how the DI is injected:tion starts, it will clear any previous data. We will later connect to Microsoft SQL Server.

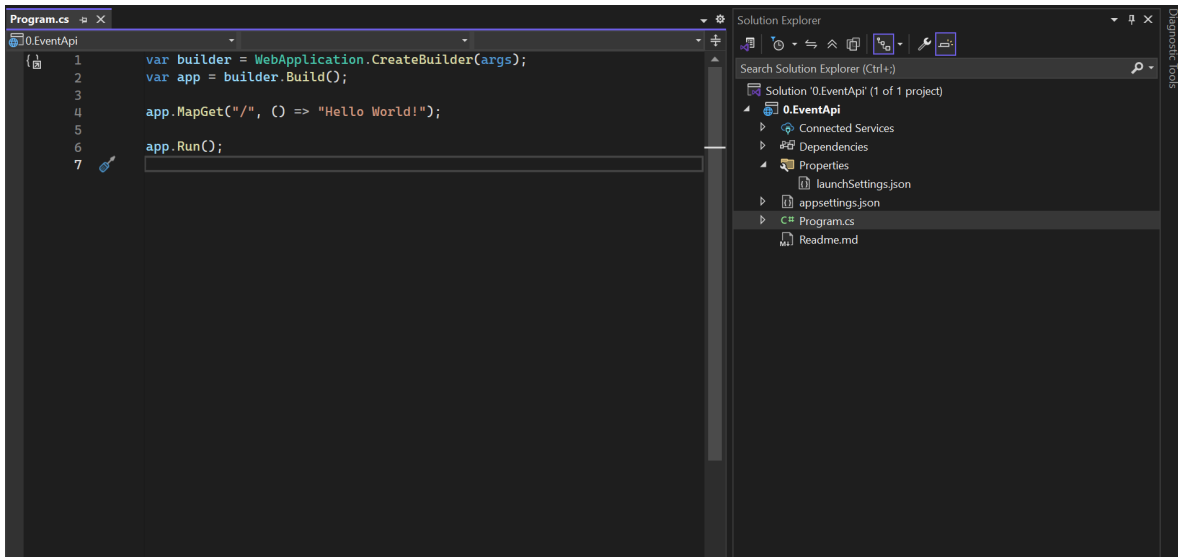
Requirements

1. Code Editor - Visual Studio is preferred
2. Fundamental concepts of C# language
3. Readiness to learn 😊

Step 1: Create an Empty .NET Core Project

Step 2: What is Included in the Project and Running the Scaffolded Project

You should have the following code in the *Program.cs* file. Program.cs is the entry point of the application, and we will be referencing it several times.



1753097002946.png

- Press the keys Ctrl + F5 or click the "Run without debugging" button to run the application.
- You should have a browser launched with *Hello World* being displayed. This shows that the project configuration looks good.
- Let's break down the scaffolded project:

Properties/launchSettings.json

This file tells Visual Studio or dotnet run how to launch the app during development, including:

- [] Ports
- [] Whether a browser should open

- [] Environment variables
- [] HTTPS or HTTP settings

Schema

```
"$schema": "https://json.schemastore.org/launchsettings.json",
```

The above line defines the JSON Schema - this helps editors like VS Code with IntelliSense and validation.

Profiles

Defines a way to run the application - we have two profiles: "http" and "https"

- [] ***commandName***: tells dotnet to launch the main project with dotnet run
- [] ***dotnetRunMessages***: shows the usual logs like ("now listening on http://")
- [] ***launchBrowser***: used to auto-launch the browser
- [] ***applicationUrl***: Port where the application will be running
- [] ***environmentVariables***: Sets the environment variables like "Development"

To avoid having the browser launch every time you run your project, head to the Properties folder and open the *launchSettings.json* file. You should see a property called *****launchBrowser: "true"*****. Change that to ***launchBrowser: "false"*** for both the profiles (http and https).

```
1 {
2   "$schema": "https://json.schemastore.org/launchsettings.json",
3   "profiles": {
4     "http": {
5       "commandName": "Project",
6       "dotnetRunMessages": true,
7       "launchBrowser": false,
8       "applicationUrl": "http://localhost:5141",
9       "environmentVariables": {
10        "ASPNETCORE_ENVIRONMENT": "Development"
11      }
12    },
13    "https": {
14      "commandName": "Project",
15      "dotnetRunMessages": true,
16      "launchBrowser": false,
17      "applicationUrl": "https://localhost:5000;http://localhost:5001",
18      "environmentVariables": {
19        "ASPNETCORE_ENVIRONMENT": "Development"
20      }
21    }
22  }
23 }
24
```

1753097399085.png

The Dependencies folder will hold all the packages we will be installing from NuGet Package Manager (<https://www.nuget.org/>)

What is NuGet?

NuGet is the package manager for .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers.

Connected Services Folder

This folder provides tooling to connect your application to external services like:

- [] Web APIs - OpenAPI/Swagger for API calls
- [] gRPC services - Add and consume gRPC endpoints
- [] Azure Services - Connect to older SOAP/WCF services
- [] Databases - connect to a database and generate models with EF

Program.cs File

This is the entry point of the application. Let's break it down:

1. **var builder = WebApplication.CreateBuilder(args);**

- Used to create a *WebApplicationBuilder* instance
- It configures services, logging, environment, and app settings

2. **var app = builder.Build();**

- This builds the application
- Creates a *WebApplication* instance, making the app ready to define endpoints and middlewares

3. **app.MapGet("/", () => "Hello World!");**

- This is where you **define your first API endpoint**. Whenever you visit the endpoint, it returns "Hello World".
- `() => "Hello World!"` - this is a lambda function that returns a string. Lambda is a concept in C#.

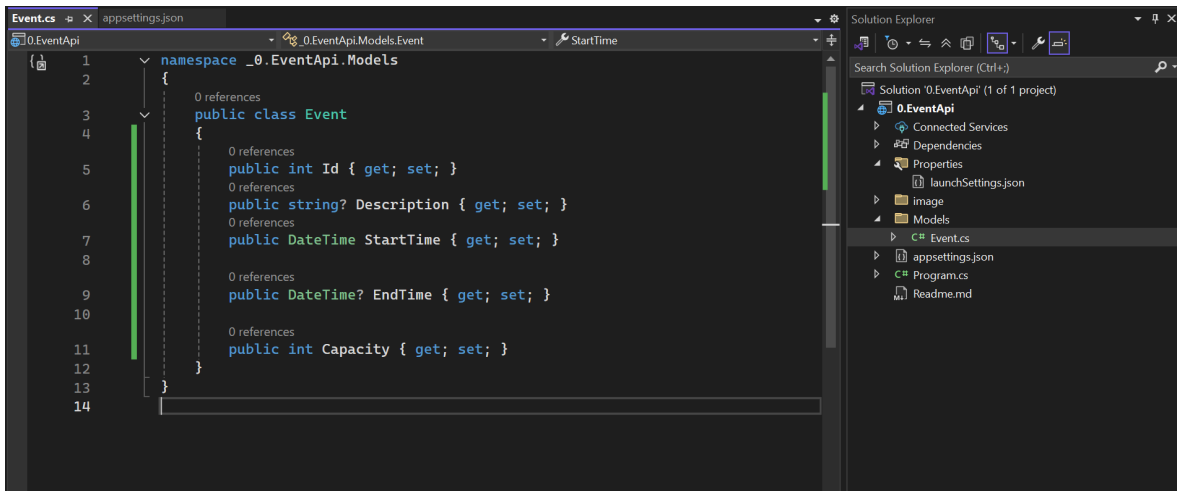
4. **app.Run();**

- This **starts the web server**.
- It starts listening for HTTP requests on the specified port(s)
- It keeps the application running

Step 3: Create Models Folder

Add a folder at the root of the project and name it Models. The Models folder will hold all the entities we will be using. An entity is similar to a table in relational databases.

Inside the newly created Models folder, add a class named Event. This will hold the properties of an event.



1753171415175.png

- All the fields are made public, meaning they can be accessed from anywhere in the application
- These properties are of different data types which simulate the real-world object of an event
- The string property has "?" because by default, all strings are nullable

Concept of Nullable Reference Types

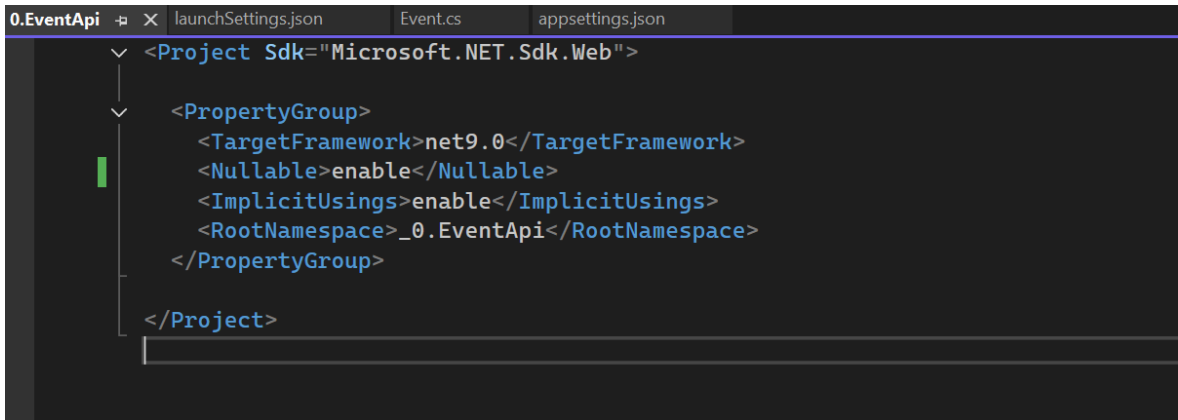
This is a concept that was introduced in C# 8.0. In older versions of C#, you could not be warned if there was a potential null value. With nullable reference types, C# became more strict and smart by helping you catch null issues at compile time.

- ***public string Description { get; set; }*** - This means that you guarantee that the Description will never be null. If you do not initialize any value, the compiler will warn you.
- ***public string? Description { get; set; }*** - This means that Description can be null. This tells the compiler that it's okay if the property is null, and it won't warn you.

Turning Nullable ON and OFF

This behavior is controlled by a setting in the .csproj file:


```
<Nullable>enable</Nullable>
```



1753172173059.png

Read more about Nullable reference types (https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-reference-types?WT.mc_id=studentamb_244628)

.csproj File

This is an important file - it's like the heart of the project configuration. It is an XML file that defines:

- What files are part of the project
- What dependencies (NuGet packages) are needed
- What version of .NET to use
- Build settings, project properties, and more

Step 4: EF Core and DB Context

Documentation/Resources

Official docs (https://learn.microsoft.com/en-us/ef/core/?WT.mc_id=studentamb_244628)

Readme Presentation (<https://github.com/kenya-data-platform-user-group/EF-Core-Presentation>)

YouTube Tutorial by Nick Chapsas (<https://www.youtube.com/watch?v=2t88FOeQ898>)

What is EF Core?

EF Core is a modern, open-source, and cross-platform **Object-Relational Mapper (ORM)** for .NET that eliminates the need to write complex SQL queries manually. It allows developers to work with databases using **C# classes and LINQ** instead of SQL.

EF Core acts as a bridge between **.NET applications** and **databases**, allowing developers to perform operations using object-oriented techniques.

Why EF Core?

1. Quick Story/Example

Imagine you're building a .NET application and need to interact with a database. How do you do it efficiently? You could write raw SQL queries, but that can be error-prone and hard to maintain. This is where an ORM (Object-Relational Mapper) like EF Core comes in handy.

2. How Do We Interact with Databases in .NET?

Before Entity Framework Core (EF Core), developers interacted with databases using **ADO.NET**, **Dapper**, or raw SQL queries. While these approaches provided control and performance, they often required writing a lot of boilerplate code for CRUD operations:

- Open a database connection
- Write SQL queries manually
- Handle result mappings to objects
- Manage transactions and exceptions explicitly

This process is repetitive and error-prone. This is where **EF Core** comes in.

3. Why Use EF Core?

a. Simplifies Data Access

EF Core abstracts database interactions, allowing developers to use **C# objects** instead of SQL queries.

b. Boosts Productivity

- Eliminates the need to write repetitive SQL queries
- Supports **automatic migrations** to handle database schema changes
- Works seamlessly with **LINQ queries** for data retrieval

c. Supports Multiple Databases

EF Core is database-agnostic and supports multiple database providers, including:

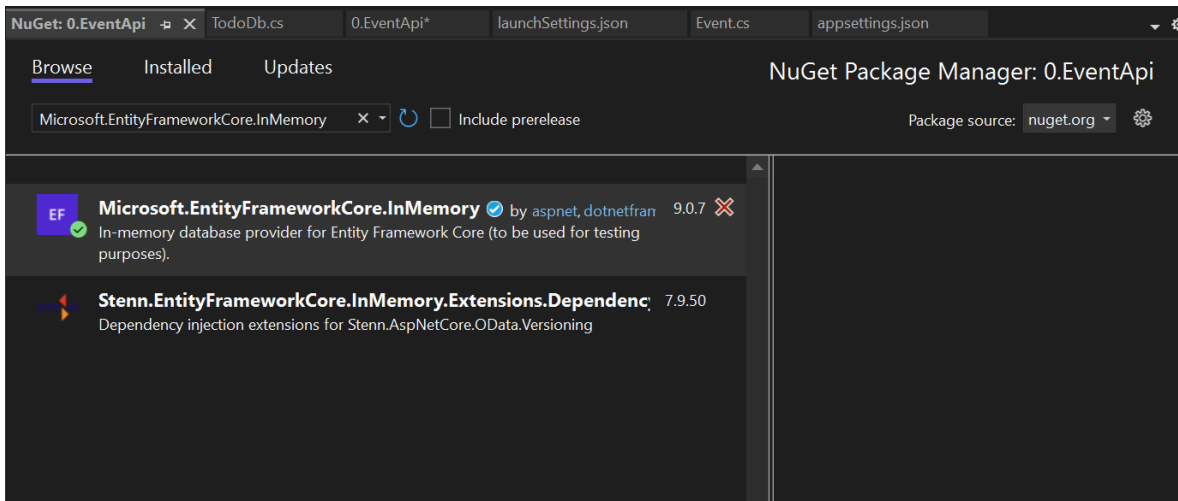
- SQL Server
- PostgreSQL
- MySQL
- SQLite
- Azure Cosmos DB

Setting Up EF Core

Step 1: Installing Database Provider

In Visual Studio, click on Tools → NuGet Package Manager and search for ***Microsoft.EntityFrameworkCore.InMemory*** and install it in your project.

This is an **Entity Framework Core (EF Core) provider** that allows you to use an **in-memory database**, mainly for **testing or prototyping**.



1753175412478.png

Limitations

- **No relational features** (e.g., foreign keys, joins, transactions)
- Doesn't always behave the same as real databases like SQL Server
- Not suitable for production

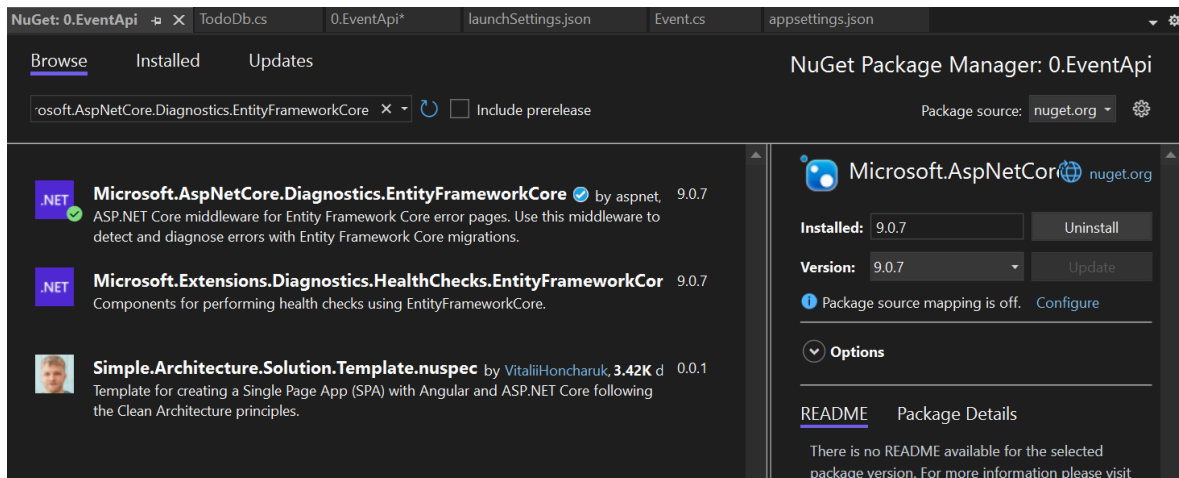
We will later advance to use other database providers.

Step 2: Installing EF Core Diagnostics

Search for *Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore* and install it in your project.

It provides **developer-friendly error pages** for **Entity Framework Core-related exceptions**, especially during development, such as database errors, migration errors, and more.

This should **only be used in development** because it exposes detailed error information that could be a security risk in production.



1753175682543.png

When to Use

- When you want **better debugging support** for EF Core issues during development
- In **educational or training environments**, to help students understand why certain DB errors are happening
- When you're building **apps that heavily rely on EF Core** and want improved developer feedback

We need to register our database provider in the Program.cs file, but first, let's learn about DbContext. 🙋

What is DbContext?

DbContext is the main class in Entity Framework Core that manages database connections and is used to query and save data.

Think of this as a bridge between your C# code and your database.

What does it do?

- Maps your **C# classes (entities)** to **database tables**
- Allows you to **query, insert, update, and delete** data
- Tracks changes to your data so it knows what to update in the database

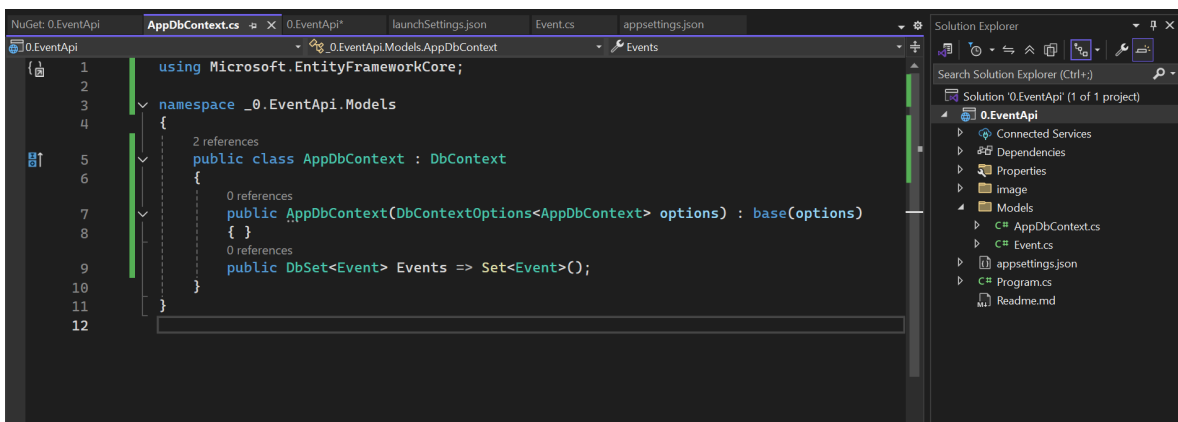
Step 3: Create a DbContext Class

In the Models folder, add a new class and name it AppDbContext.cs

In the file, add the following:

```
using Microsoft.EntityFrameworkCore;

namespace _0.EventApi.Models
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
        { }
        public DbSet<Event> Events => Set<Event>();
    }
}
```



1753176789869.png

Breakdown

The `AppDbContext` class is a central part of Entity Framework Core (EF Core). It represents a **session with the database**, allowing you to perform CRUD (Create, Read, Update, Delete) operations using C# classes.

Purpose

- Connects the application to the database

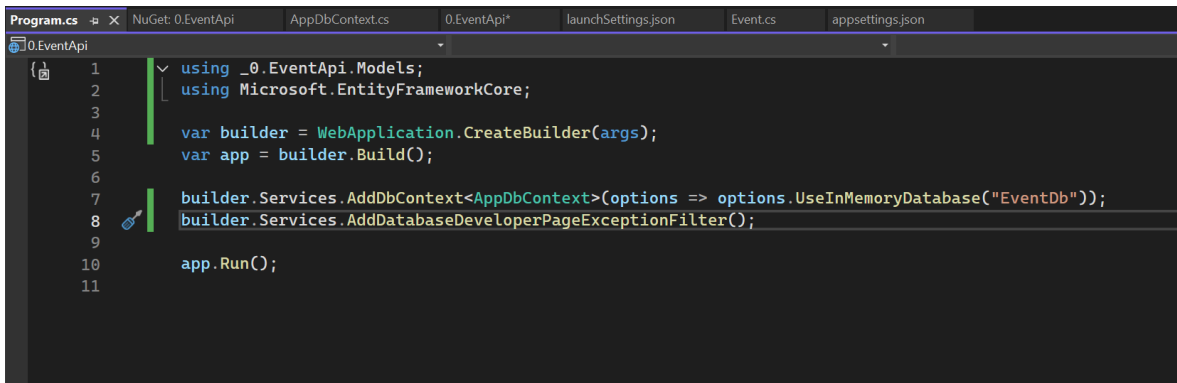
- Maps your models (e.g., `Event`) to database tables
- Enables querying and saving of data

Term	Description
<code>DbContext</code>	Base class provided by EF Core. It handles database operations and change tracking.
<code>AppDbContext</code>	Our custom context class inheriting from <code>DbContext</code> . We define models here as <code>DbSet<></code> .
<code>AppDbContext(DbContextOptions<AppDbContext> options)</code>	Constructor used to configure the context with settings like the connection string.
<code>DbSet<Event> Events</code>	Represents the <code>Events</code> table in the database. You use this to access and query event records.
<code>Set<Event>()</code>	Built-in EF Core method that initializes the <code>DbSet</code> for a given model type.

Step 4: Register the Context in Program.cs

Add the following:

```
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseInMemoryDatabase("EventDb"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```



```
1 using _0.EventApi.Models;
2 using Microsoft.EntityFrameworkCore;
3
4 var builder = WebApplication.CreateBuilder(args);
5 var app = builder.Build();
6
7 builder.Services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase("EventDb"));
8 builder.Services.AddDatabaseDeveloperPageExceptionFilter();
9
10 app.Run();
11
```

1753177804536.png

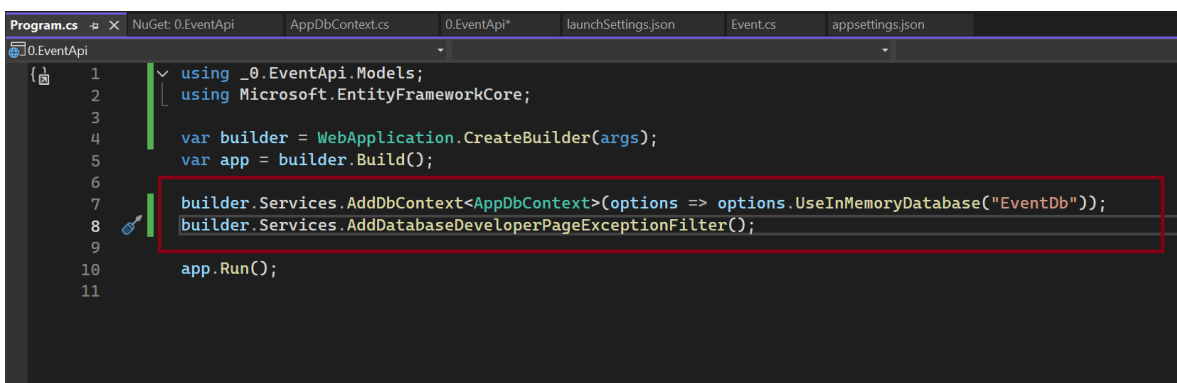
The first line registers the `AppDbContext` with the application's **dependency injection (DI) container** and configures it to use an **in-memory database** called "EventDb".

The second line adds a special **developer-friendly error page** that shows detailed database-related exceptions when running the app in **development mode**.

- Helps you see detailed errors if something goes wrong with EF Core (e.g., migrations or invalid queries)
- Makes debugging database issues easier during development
- Only shows detailed errors when `ASPNETCORE_ENVIRONMENT=Development`

This usage is possible because it is made available by the `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` package.

DbContext Flow



```
1 using _0.EventApi.Models;
2 using Microsoft.EntityFrameworkCore;
3
4 var builder = WebApplication.CreateBuilder(args);
5 var app = builder.Build();
6
7 builder.Services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase("EventDb"));
8 builder.Services.AddDatabaseDeveloperPageExceptionFilter();
9
10 app.Run();
11
```

1753177840347.png

Concept of Dependency Injection (DI)

Dependency Injection (DI) is a software design pattern that helps us **provide objects (dependencies)** that a class needs, **without creating them manually inside the class**.

Think of it like this 🤔

If a class needs a tool (like a hammer), instead of building the hammer itself, someone gives it the hammer ready to use.

Why Use DI?

- Keeps code clean and testable
- Reduces duplication
- Follows the principle of loose coupling (classes depend on *abstractions*, not on *concrete implementations*)

How it works in .NET Core

- ASP .NET Core has built-in DI Container. Just the same way we registered the DbContext, the DI tells the system "When someone asks for an `AppDbContext`, here's how to create it."
- Later, when your app needs that `AppDbContext` in a class (e.g., a controller or service), you don't create it manually — the system **automatically injects it for you**.
- An example is registering `AppDbContext` with the InMemory Database. It will register **`AppDbContext`** as a service. It then tells the system to use in-memory database when creating `AppDbContext`.
- Later when we shall be creating a class, here how the DI is injected.

```
public class EventsController : ControllerBase
{
    private readonly AppDbContext _context;

    public EventsController(AppDbContext context)
```

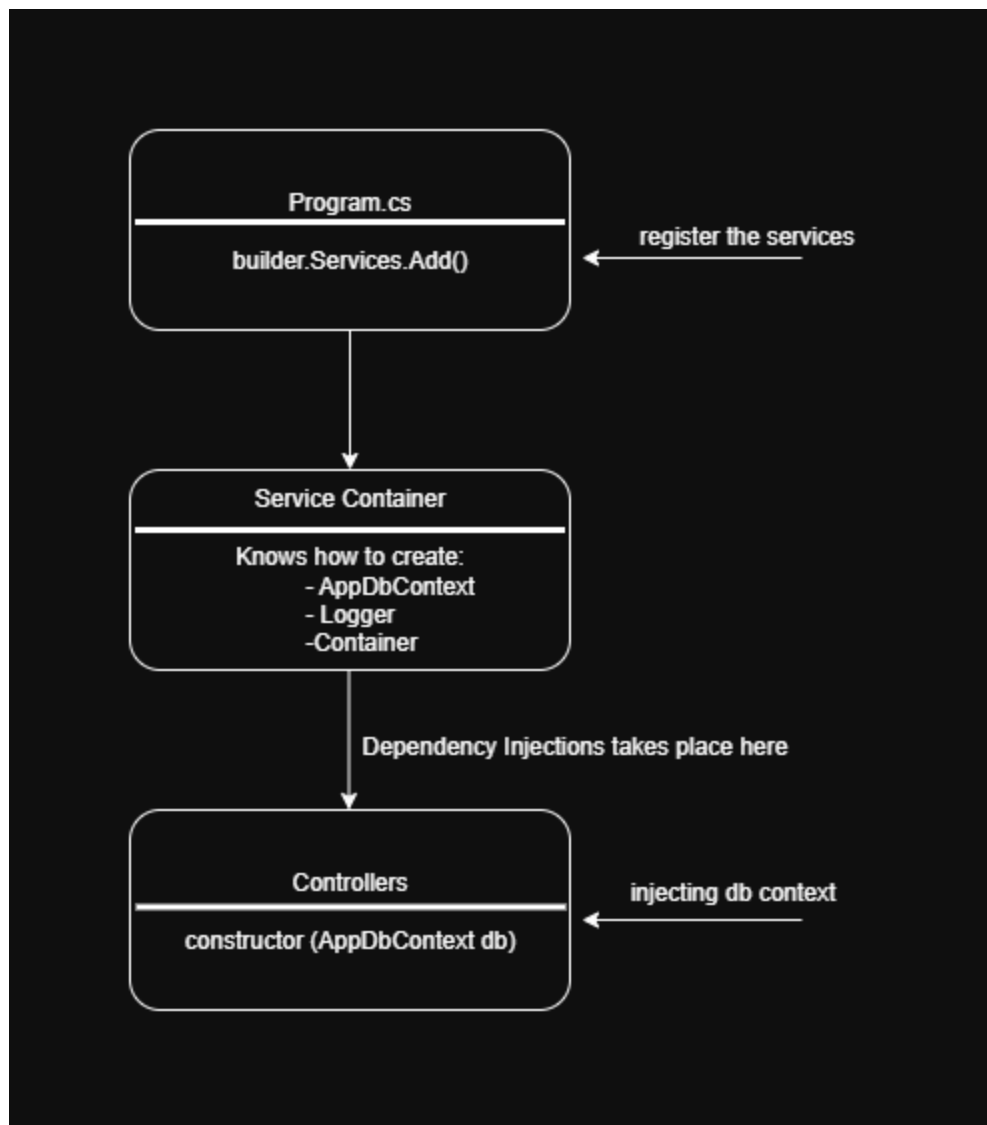
```

{
    _context = context; // <-- Injected automatically from the
container
}

// Now you can use _context.Events to access your database
}

```

- You do not have to use "new AppDbContext(...)" because the framework injected it automatically.



1753185534393.png

Terms to Remember

Term	Meaning
Dependency	An object a class depends on (e.g., <code>AppDbContext</code>)
Injection	Supplying the dependency from outside
Service Container	Built-in system in .NET Core that holds all registered services
Register	Telling the container how to create a dependency
Resolve	The container gives you the dependency when needed

Summary

How does `AppDbContext` work with `Program.cs`?

- `AppDbContext` tells EF that you want to work with a table of Event objects (a database table)
- `Program.cs` registers `AppDbContext` with the built-in Dependency Injection (DI). This tells the app that "Whenever someone needs ***AppDbContext***, give them an instance that uses an in-memory database called ***EventDb*** "
- In-memory database is used for testing and learning as it acts as a fake database stored in memory (RAM), so no actual SQL server is needed
- When `Program.cs` registers `AppDbContext`, you can inject it in any part of your application. We will explore this when we write CRUD operations of our Minimal API

Step 5: CRUD Operations for REST API

We will now proceed to write CRUD operations for an Event API. The aim is to Create, Read, Update, and Delete. Once we have these foundations, it will be much easier to grasp more concepts.

Just after `builder.Build();` you should have the following source code:

```

//create an event
app.MapPost("/event", async (Event newEvent, AppDbContext db) =>
{
    db.Events.Add(newEvent);
    await db.SaveChangesAsync();
    return Results.Created($"/event/{newEvent.Id}", newEvent);
});

//get all events
app.MapGet("/events", async(AppDbContext db) =>
    await db.Events.ToListAsync());

//get event by id
app.MapGet("/event/{id}", async (int id, AppDbContext db) =>
    await db.Events.FindAsync(id)
    is Event @event
    ? Results.Ok(@event)
    : Results.NotFound());

//update an event
app.MapPut("/event/{id}", async (int id, Event inputEvent, AppDbContext
db) =>
{
    var @event = await db.Events.FindAsync(id);

    if (@event is null) return Results.NotFound();

    @event.Title = inputEvent.Title;
    @event.Description = inputEvent.Description;
    @event.Capacity = inputEvent.Capacity;
    @event.StartTime = inputEvent.StartTime;
    @event.EndTime = inputEvent.EndTime;

    await db.SaveChangesAsync();
    return Results.NoContent();
});

```

```
//delete an event
app.MapDelete("/event/{id}", async (int id, AppDbContext db) =>
{
    if (await db.Events.FindAsync(id) is null)
    {
        return Results.NotFound();
    }
    else if (await db.Events.FindAsync(id) is Event @event)
    {
        db.Remove(@event);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }
    return Results.NoContent();
});
```

The last line of code should be **app.Run();**

Code Breakdown

Let's break down the source code for better understanding:

1. Create Event

```
//create an event
app.MapPost("/event", async (Event newEvent, AppDbContext db) =>
{
    db.Events.Add(newEvent);
    await db.SaveChangesAsync();
    return Results.Created($"/event/{newEvent.Id}", newEvent);
});
```

This route handles HTTP **POST** requests sent to **/event**. Its purpose is to **create and save a new event** in the in-memory database.

Code	Explanation
<code>app.MapPost("/event")</code>	<code>/event</code> is the path where the clients send the event data
<code>(Event newEvent, AppDbContext db)</code>	Two parameters are injected: the event data from the request body i.e., Postman or REST Client (<code>newEvent</code>) and AppDbContext from the service container (<code>db</code>)
<code>db.Events.Add(newEvent)</code>	Adds the new event to the in-memory database's Events Collection
<code>await db.SaveChangesAsync();</code>	Saves changes to the database asynchronously (non-blocking)
<code>return Results.Created(\$"/event/{newEvent.Id}", newEvent)</code>	Returns an HTTP 201 Created response with the newly created event and its URL

Testing with REST Client now

The scaffolded project comes with REST Client which is used to test the endpoints of the application.

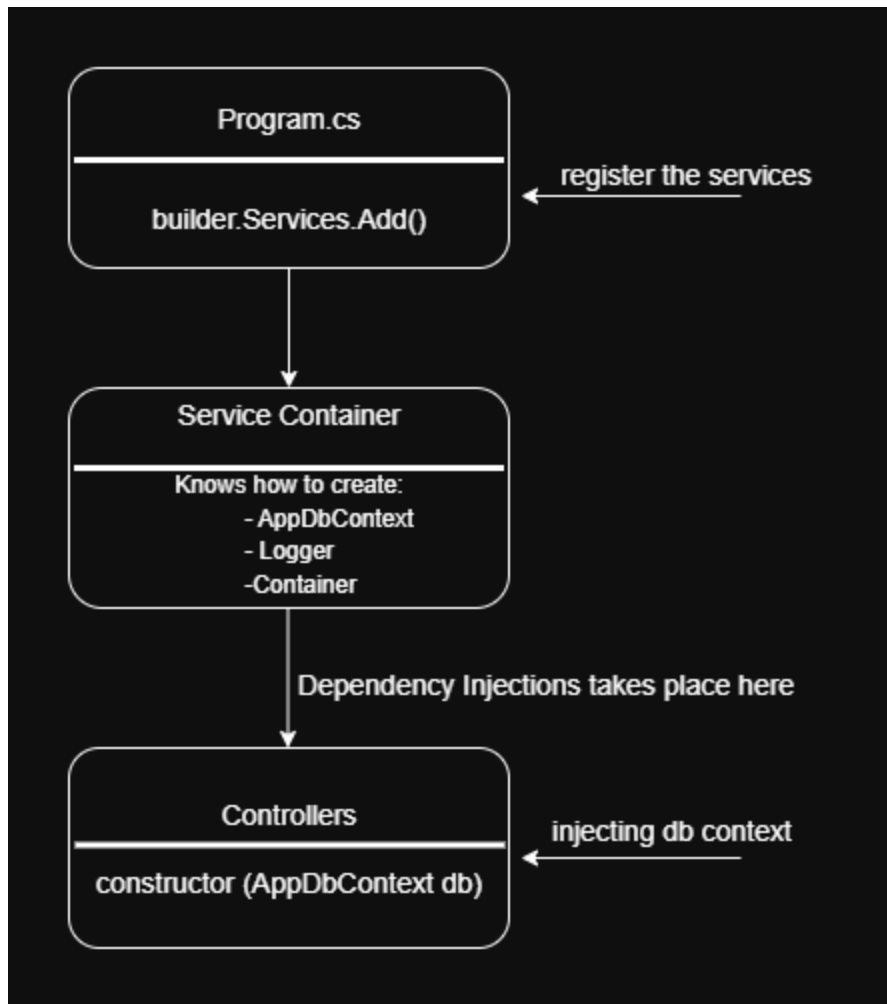
To generate the endpoints without needing to do them manually:

1. Hold `Ctrl + Q` on your keyboard and search for "Endpoint Explorer"
2. It will give you all the available endpoints
3. Right-click on the `POST` request and generate the request

On the generated endpoint, add the following data for testing:

```
{
  "title": "JS hackathon",
  "description": "Networking and talks",
}
```

```
"startTime": "2025-09-10T09:00:00",  
"endTime": "2025-09-10T17:00:00",  
"capacity": 150  
}
```



1753185753966.png

2. Get All Events

```
//get all events  
app.MapGet("/events", async(AppDbContext db) =>  
    await db.Events.ToListAsync());
```

What This Does:

This route handles HTTP GET requests sent to `/events`. It retrieves a list of all events

stored in the in-memory database.

Code	Explanation
<code>app.MapGet("/events")</code>	Route to the path <code>/events</code>
<code>AppDbContext db</code>	Injects <code>AppDbContext</code> via dependency injection to interact with the database
<code>await db.Events.ToListAsync()</code>	Asynchronously fetches all event records from the Events table and returns them as a list

Testing with REST Client

1. Hold `Ctrl + Q` on your keyboard and search for "Endpoint Explorer"
2. It will give you all the available endpoints
3. Right-click on the `GET` request and generate the request

On the generated endpoint, add a valid endpoint:

```
GET {{0.EventApi_HostAddress}}/events
```

3. Get an Event by ID

```
//get event by id
app.MapGet("/event/{id}", async (int id, AppDbContext db) =>
    await db.Events.FindAsync(id)
    is Event @event
    ? Results.Ok(@event)
    : Results.NotFound());
```

What This Does:

This route handles HTTP GET requests sent to `/event/`. It looks up a single event using the provided `id` from the database. If the event exists, it returns the event data; if not, it

returns a "Not Found" response.

code	Explanation
<code>app.MapGet("/event/")</code>	route a GET with dynamic id in the url path i.e /event/3
<code>(int id, AppDbContext db)</code>	The route handler takes the id from the URL and uses Dependency Injection to get the AppDbContext.
<code>await db.Events.FindAsync(id)</code>	Looks for the event with the matching id in the database. Returns null if not found.
<code>is Event @event ? Results.Ok (@event) : Results.NotFound()</code>	If the event exists, returns a 200 OK response with the event. If it doesn't, returns 404 Not Found.

Testing with REST Client (Get by ID)

1. Hold **Ctrl + Q** on your keyboard and search for "Endpoint Explorer"
2. It will give you all the available endpoints
3. Right-click on the **GET{id}** request and generate the request

On the generated endpoint, add a valid id

```
GET {{0.EventApi_HostAddress}}/event/4
```

4. Update an Event

```
//update an event
app.MapPut("/event/{id}", async (int id, Event inputEvent, AppDbContext db) =>
{
    var @event = await db.Events.FindAsync(id);

    if (@event is null) return Results.NotFound();
```

```
@event.Title = inputEvent.Title;
@event.Description = inputEvent.Description;
@event.Capacity = inputEvent.Capacity;
@event.StartTime = inputEvent.StartTime;
@event.EndTime = inputEvent.EndTime;

await db.SaveChangesAsync();
return Results.NoContent();
});
```

What This Does:

This route handles HTTP PUT requests to update an existing event in the database. The client provides an updated version of the event, and if the event exists, its values are modified and saved.

Code	Explanation
<code>MapPut("event/{id}")</code>	Sets up a PUT route for updating an event by ID
<code>(int id, Event inputEvent, AppDbContext db)</code>	The route handler receives the event <code>id</code> from the URL, the updated event data from the request body, and a database context via Dependency Injection
<code>await db.Events.FindAsync(id)</code>	Tries to find the event in the database using the provided <code>id</code>
<code>if (@event is null) return Results.NotFound()</code>	If no event is found, it returns a 404 Not Found response
<code>@event.Title = inputEvent.Title ...and the rest</code>	Updates each field of the found event with the new values provided
<code>await db.SaveChangesAsync()</code>	Commits the changes to the database
<code>return Results.NoContent()</code>	Returns a 204 No Content response to indicate the update was successful, but there's nothing to return in the body

Testing with REST Client (Update)

1. Hold `Ctrl + Q` on your keyboard and search for "Endpoint Explorer"
2. It will give you all the available endpoints
3. Right-click on the `UPDATE` request and generate the request

On the generated endpoint, add the following data for testing:

```
{
  "id": 1,
  "title": "Annual Meetup - Updated",
  "description": "Networking, and workshops",
  "startTime": "2025-09-10T09:00:00",
  "endTime": "2025-09-10T18:00:00",
  "capacity": 245
}
```

5. Delete an Event

```
//delete an event
app.MapDelete("/event/{id}", async (int id, AppDbContext db) =>
{
    if (await db.Events.FindAsync(id) is null)
    {
        return Results.NotFound();
    }
    else if (await db.Events.FindAsync(id) is Event @event)
    {
        db.Remove(@event);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }
    return Results.NoContent();
});
```

What This Does:

This endpoint handles HTTP **DELETE** requests to remove an event from the database by its ID. It first checks if the event exists and deletes it if found.

Code	Explanation
<code>app.MapDelete("/event/{id}")</code>	Sets up a DELETE route for removing an event by ID
<code>(int id, AppDbContext db)</code>	The handler receives the event id from the URL and uses the injected database context db
<code>if (await db.Events.FindAsync(id) is null)</code>	Checks if the event exists. If not, returns 404 Not Found
<code>else if (await db.Events.FindAsync(id) is Event @event)</code>	Retrieves the event again and stores it in @event
<code>db.Remove(@event)</code>	Marks the event for deletion
<code>await db.SaveChangesAsync()</code>	Commits the deletion to the database
<code>return Results.NoContent()</code>	Returns a 204 No Content response indicating successful deletion

Testing with REST Client (Delete)

1. Hold **Ctrl + Q** on your keyboard and search for "Endpoint Explorer"
2. It will give you all the available endpoints
3. Right-click on the **DELETE** request and generate the request

On the generated endpoint, add a valid id:

```
DELETE {{0.EventApi_HostAddress}}/event/5
```

Step 6: MapGroup

👉 Next, we will learn about TypedResults API to help us handle different responses of our system.

Minimal Apis

Start typing here...