Contents

[the complete and discrete steps in Visual Studio 2022, for adding the entire code to a .NET Core Web API Controller file that returns a list of data but I have not implemented any Entity Framework Core code yet, though I need to use Entity Framework Core with an actual SQL Server database (not in-memory) so please include any steps required for use of Entity Framework Core and SQL Server. 2](#_Toc204929949)

[1. Create a New .NET Core Web API Project 2](#_Toc204929950)

[2. Install EF Core and SQL Server Packages 2](#_Toc204929951)

[3. Create the Data Model 2](#_Toc204929952)

[4. Create the EF Core DbContext 2](#_Toc204929953)

[5. Add Connection String in appsettings.json 2](#_Toc204929954)

[6. Register DbContext in Program.cs 3](#_Toc204929955)

[7. Create the API Controller 3](#_Toc204929956)

[8. Create the Database via EF Core Migrations 3](#_Toc204929957)

[9. Run and Test the API 3](#_Toc204929958)

[Optional: Seed Data (for testing) 4](#_Toc204929959)

[APPENDIX 5](#_Toc204929960)

[Explanation of Each Step 5](#_Toc204929961)

[1. Create a New .NET Core Web API Project 5](#_Toc204929962)

[2. Install EF Core and SQL Server Packages 5](#_Toc204929963)

[3. Create the Data Model 5](#_Toc204929964)

[4. Create the EF Core DbContext 5](#_Toc204929965)

[5. Add Connection String in appsettings.json 5](#_Toc204929966)

[6. Register DbContext in Program.cs 5](#_Toc204929967)

[7. Create the API Controller 5](#_Toc204929968)

[8. Create the Database via EF Core Migrations 6](#_Toc204929969)

[9. Run and Test the API 6](#_Toc204929970)

[Optional: Seed Data (for testing) 6](#_Toc204929971)

# the complete and discrete steps in Visual Studio 2022, for adding the entire code to a .NET Core Web API Controller file that returns a list of data but I have not implemented any Entity Framework Core code yet, though I need to use Entity Framework Core with an actual SQL Server database (not in-memory) so please include any steps required for use of Entity Framework Core and SQL Server.

**ChatGPT said:**

Got it! Here’s a **step-by-step Visual Studio 2022 guide** to create a **.NET Core Web API Controller** that returns a list of data, using **Entity Framework Core** with a **real SQL Server database** — assuming no EF Core code exists yet.

**Step-by-step guide in Visual Studio 2022**

## 1. Create a New .NET Core Web API Project

* Open **Visual Studio 2022**.
* Click **Create a new project**.
* Select **ASP.NET Core Web API** and click **Next**.
* Name your project, e.g., ResourceBooksApi, and choose a folder. Click **Next**.
* Choose:
  + Framework: **.NET 6** or **.NET 7** (whichever is latest/stable)
  + Authentication: **None**
  + Enable OpenAPI support: checked (optional, useful for Swagger)
* Click **Create**.

## 2. Install EF Core and SQL Server Packages

* In Visual Studio, open the **Package Manager Console** from **Tools > NuGet Package Manager > Package Manager Console**.
* Run the following commands one by one:

Install-Package Microsoft.EntityFrameworkCore

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

These packages add EF Core and SQL Server provider support.

## 3. Create the Data Model

* In **Solution Explorer**, right-click your project > **Add > New Folder**, name it Models.
* Right-click the Models folder > **Add > Class**, name it ResourceBook.cs.
* Replace the content with:

namespace ResourceBooksApi.Models

{

public class ResourceBook

{

public int Id { get; set; } // Primary Key

public string Title { get; set; }

public string Author { get; set; }

public string Category { get; set; }

public DateTime PublishedDate { get; set; }

}

}

## 4. Create the EF Core DbContext

* Right-click the project > **Add > New Folder**, name it Data.
* Right-click Data folder > **Add > Class**, name it AppDbContext.cs.
* Paste this code:

using Microsoft.EntityFrameworkCore;

using ResourceBooksApi.Models;

namespace ResourceBooksApi.Data

{

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options)

: base(options) { }

public DbSet<ResourceBook> ResourceBooks { get; set; }

}

}

## 5. Add Connection String in appsettings.json

* Open appsettings.json.
* Add a connection string under the "ConnectionStrings" section:

{

"ConnectionStrings": {

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

},

// ... rest of the file

}

Adjust Server=localhost to your SQL Server instance name if different, and update credentials if not using Windows Authentication.

## 6. Register DbContext in Program.cs

* Open Program.cs.
* Add the necessary using directives at the top:

using Microsoft.EntityFrameworkCore;

using ResourceBooksApi.Data;

* Inside the var builder = WebApplication.CreateBuilder(args); section, register the AppDbContext:

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

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

* Ensure you have:

builder.Services.AddControllers();

* At the bottom, make sure to map controllers:

app.MapControllers();

Your Program.cs should look something like:

var builder = WebApplication.CreateBuilder(args);

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

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

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

## 7. Create the API Controller

* Right-click the Controllers folder > **Add > Controller**.
* Choose **API Controller - Empty**, click **Add**.
* Name it ResourceBooksController.
* Replace its content with:

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using ResourceBooksApi.Data;

using ResourceBooksApi.Models;

namespace ResourceBooksApi.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class ResourceBooksController : ControllerBase

{

private readonly AppDbContext \_context;

public ResourceBooksController(AppDbContext context)

{

\_context = context;

}

// GET: api/resourcebooks

[HttpGet]

public async Task<ActionResult<IEnumerable<ResourceBook>>> GetResourceBooks()

{

return await \_context.ResourceBooks.ToListAsync();

}

}

}

## 8. Create the Database via EF Core Migrations

* Open **Package Manager Console** again.
* Run:

Add-Migration InitialCreate

Update-Database

* This creates your database and tables in SQL Server.

## 9. Run and Test the API

* Press **F5** or click **Run** in Visual Studio.
* The app will start with Swagger UI (if enabled).
* Go to the endpoint GET /api/resourcebooks to test the API.
* Initially, it will return an empty list because no data is inserted yet.

## Optional: Seed Data (for testing)

To have some data returned, you can add seeding inside ResourceBooksController constructor or better in AppDbContext.

Example seeding inside the controller (for quick test):

public ResourceBooksController(AppDbContext context)

{

\_context = context;

if (!\_context.ResourceBooks.Any())

{

\_context.ResourceBooks.AddRange(

new ResourceBook { Title = "C# Basics", Author = "Jane Doe", Category = "Programming", PublishedDate = DateTime.UtcNow.AddYears(-2) },

new ResourceBook { Title = "SQL Server Guide", Author = "John Smith", Category = "Database", PublishedDate = DateTime.UtcNow.AddYears(-1) }

);

\_context.SaveChanges();

}

}

**Summary**

| **Step** | **What You Did** |
| --- | --- |
| 1 | Created Web API project in Visual Studio |
| 2 | Installed EF Core and SQL Server NuGet packages |
| 3 | Created ResourceBook data model |
| 4 | Created AppDbContext with DbSet<ResourceBook> |
| 5 | Added SQL Server connection string in config |
| 6 | Registered DbContext in Program.cs |
| 7 | Created API controller returning list of data |
| 8 | Created database using EF Core migrations |
| 9 | Ran and tested the API |

If you want, I can also help you add endpoints for **create, update, delete**, or show how to handle errors and validation! Would you like that?

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.

# APPENDIX

## Explanation of Each Step

**Why each step is important** and the **background reasoning** behind including it in the solution:

### 1. Create a New .NET Core Web API Project

**Why:**  
This initializes the foundation of your application—a **Web API project template** provides all the necessary files and configurations to build RESTful HTTP services. It sets up middleware, routing, and a structure tailored for APIs.

**Background:**  
.NET Core supports different app types (console, MVC, API, etc.). The Web API template includes components specific to building APIs, like controllers, JSON serialization, and swagger support. Starting with this template avoids manual setup of boilerplate code.

### 2. Install EF Core and SQL Server Packages

**Why:**  
Entity Framework Core (EF Core) is the ORM (Object-Relational Mapper) that lets your application interact with the database through C# objects instead of raw SQL. The SQL Server package is needed to connect EF Core to a SQL Server database specifically.

**Background:**  
.NET Core projects don't include EF Core by default. You must install these packages to gain the ability to define data models, manage database connections, and run queries using LINQ.

### 3. Create the Data Model

**Why:**  
The data model defines the shape of your data in the application and maps to database tables. Creating a ResourceBook class models the structure of each book entity you want to store or retrieve.

**Background:**  
EF Core uses these model classes to create the database schema and to convert database rows into C# objects and vice versa. This abstraction simplifies working with data.

### 4. Create the EF Core DbContext

**Why:**  
The DbContext class represents a session with the database, allowing querying and saving data. Defining a DbSet<ResourceBook> in the context informs EF Core which tables to manage.

**Background:**  
DbContext encapsulates database connection and change tracking, providing the main API surface for EF Core operations. It’s essential to create a context tailored to your domain models.

### 5. Add Connection String in appsettings.json

**Why:**  
Your application needs to know **where** the SQL Server database is, how to authenticate, and which database to use. The connection string is this critical information.

**Background:**  
Storing connection strings in appsettings.json follows .NET’s configuration conventions, allowing easy modification without recompiling and supporting environment-specific overrides.

### 6. Register DbContext in Program.cs

**Why:**  
This step tells the ASP.NET Core Dependency Injection (DI) system how to create and manage the AppDbContext instances, including which database provider and connection string to use.

**Background:**  
Registering services in DI is fundamental in ASP.NET Core for loose coupling and testability. EF Core requires this to inject the context into controllers or services seamlessly.

### 7. Create the API Controller

**Why:**  
Controllers handle incoming HTTP requests, interact with data, and return responses. Creating a controller that returns a list of resource books exposes an endpoint clients can call to fetch data.

**Background:**  
API controllers are the interface between your server-side data and clients (web apps, mobile apps, etc.). This step translates your backend data into an API accessible over HTTP.

### 8. Create the Database via EF Core Migrations

**Why:**  
EF Core Migrations let you translate your model classes into an actual database schema (tables, columns). Applying migrations creates or updates the database without writing SQL manually.

**Background:**  
Migrations provide version control for your database schema and keep it in sync with your data model, simplifying deployment and updates across environments.

### 9. Run and Test the API

**Why:**  
Running the app and testing the API endpoint confirms that your Web API correctly talks to the database and returns the expected data.

**Background:**  
Testing early validates your setup and ensures your application works end-to-end — from HTTP request to data retrieval.

### Optional: Seed Data (for testing)

**Why:**  
Seeding populates the database with sample data so your API returns meaningful results during development and testing.

**Background:**  
Starting with an empty database can make early testing harder. Seeding automates adding initial data, speeding up development feedback loops.

**Summary**

Each step builds on the previous one to:

* **Set up the app structure (step 1)**
* **Enable data access (steps 2-6)**
* **Expose data via API (step 7)**
* **Create and synchronize the database (step 8)**
* **Validate functionality (step 9)**

Together, they form a standard, best-practice approach to building a maintainable and scalable .NET Core Web API connected to SQL Server using EF Core.

If you'd like, I can explain any step in more detail!

Get smarter responses, upload files and images, and more.

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.