ASP.NET Core Web API .NET 8 2024 -1. Create Project + Install

Installing Development Tools

- **Visual Studio Code**
    - Search "Visual Studio Code" in Google.
    - Click the official **VS Code** website → *Download for Windows*.
    - Run the installer (single-click install).
- **Visual Studio Community**
    - Search "Visual Studio" (not VS Code) and go to the official site.
    - Choose the **Community** edition (free forever) and start the installer.
    - During setup, select only the components you need for this course:
        - *.NET desktop development* (includes the .NET runtime)
        - *ASP.NET and web development*
    - *Do not* install Node.js or Python through Visual Studio; install them separately if required.

Component	Why Needed?
.NET runtime	Runs the ASP .NET Core Web API.
ASP.NET & web development	Provides templates and tools for APIs.
Node.js / Python	Optional; not required for the API part.

Setting Up SQL Server

1. Search "SQL Server" and open the **Microsoft** download page.
2. Choose **SQL Server Express** (sufficient for 99 % of cases).
3. Run the installer – accept the default "Basic" setup.
4. After SQL Server is installed, download **SQL Server Management Studio (SSMS)** from the same page and install it (or use Azure Data Studio if preferred).

Definition: *SQL Server Express* is a free, lightweight edition of Microsoft SQL Server that provides core database capabilities for development and small-scale production.

Creating the Project Folder

- In Windows Explorer, create a new folder (e.g., FinShark).
- Open the folder with **VS Code**: File → Open Folder.
- The VS Code explorer will now show an empty workspace ready for the API.

Generating the Web API Project

1. Open the integrated terminal in VS Code (Ctrl+ `).
2. Ensure you are inside the FinShark folder, then run:
3. dotnet new webapi -o API
    - -o API creates a subfolder named **API** and places all generated files there.
4. Change directory into the new project:
5. cd API

    Tip: Always verify you are inside the API folder before running further commands.

Running the API with Swagger

- Start the development server with hot-reload:
- dotnet watch run
- The command launches the app and automatically opens **Swagger UI** in your default browser.

Definition: *Swagger* (OpenAPI) is a specification and set of tools that generate interactive documentation for RESTful APIs, allowing you to test endpoints directly from the browser.

Understanding Program.cs

Program.cs is the entry point that configures the application's services and request pipeline.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var builder = WebApplication.CreateBuilder(args);



//Register services (dependency injection)
builder.Services.AddControllers();          // MVC controllers
builder.Services.AddEndpointsApiExplorer(); // Enables Swagger generation
builder.Services.AddSwaggerGen();           // Swagger middleware

var app = builder.Build();
```

```
//Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
   app.UseSwagger();
   app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

- **Builder (WebApplication.CreateBuilder)** – sets up dependency injection, configuration, logging, etc.
- **builder.Services** – where you add services (e.g., controllers, Swagger).
- **app** – the HTTP request pipeline; middleware runs here in the order added.
- **Middleware examples**
    - app.UseHttpsRedirection() – redirects HTTP ⟶ HTTPS.
    - app.UseAuthorization() – enforces authorization policies.
- **app.MapControllers()** – maps attribute-routed controller actions to endpoints.

Key Concept: The **pipeline** processes each incoming HTTP request through a series of middleware components before reaching the controller action that handles the request.

Optional: Adding the React Front-End

- If you want the UI, clone the provided GitHub repository (link supplied in the course material).
- The front-end lives in a separate folder and can be started independently.
- **Not required** for completing the API portion; you may work solely within the API project.## 🚀 API Fundamentals
- **CRUD** represents the four basic operations a computer can perform on data: **Create**, **Read**, **Update**, **Delete**.
- In the context of web APIs, these operations correspond to specific HTTP verbs:

HTTP Verb	CRUD Operation	Common Verb Meaning
GET	**Read**	Retrieve data
POST	**Create**	Add new data
PUT / PATCH	**Update**	Modify existing data
DELETE	**Delete**	Remove data

Definition: *CRUD* is the acronym for the four fundamental data manipulation actions—Create, Read, Update, and Delete—that underlie most computer operations.

- Remember:
    - **POST → Create**
    - **GET → Read**
    - **PUT/PATCH → Update**
    - **DELETE → Delete**

Understanding this mapping helps you translate familiar database terminology into the language of RESTful APIs.

Setting Up VS Code Extensions

- Open **Visual Studio Code** → go to the **Extensions** view.
- Install the following extensions (most can be added in one batch by searching for a pack):

Extension	Purpose
C/C++ Dev Kit (or just **C** if you prefer)	Provides IntelliSense, debugging, and build tools for C/C++ projects.
.NET Extension Pack	Adds .NET Core support, project templates, and debugging features.
NuGet Gallery	Enables easy browsing and installing of NuGet packages directly from VS Code.
Prettier	Auto-formats code for consistent style and readability.
C Extension Pack (by *JosKreative*)	Quick scaffolding of C files and additional language utilities.

- **Installation tip:** Search for the pack name (e.g., "C Extension Pack") and click **Install**; the pack will pull in all its constituent extensions automatically.
- After installing, you'll have:
    1. Seamless creation of new C files.
    2. Integrated .NET tooling for building APIs.
    3. Automatic code formatting with Prettier.

These extensions streamline the development workflow, letting you focus on building the API rather than configuring the editor.

ASP.NET Core Web API .NET 8 2024 - 2. Models

Why APIs Matter

API: *code that sits on top of a database and provides a safe, self-contained way to interact with the data.*

- Corporations pay heavily for APIs because they protect **sensitive data** and expose only the operations you allow.
- An API acts as a **gatekeeper**, preventing arbitrary reads/writes and ensuring business rules are enforced.

Databases and Models

Database: *a structured collection of data, comparable to a filing cabinet where each drawer is a table.*

- **Model** → blueprint for a table (a "drawer").
- Each record in a model corresponds to a **row** (a piece of paper) stored in the database.
- Think of a database as an **enhanced Excel spreadsheet** linked by **primary keys** and **foreign keys**.

Model Properties

Property	Data Type	Purpose / Notes
Id	int	Unique identifier (primary key). Must be an **integer** to guarantee uniqueness.
Symbol	string	Stock ticker symbol (e.g., "AAPL"). Stored as text, never as an integer.
CompanyName	string	Full name of the company. Initialized with string.Empty to avoid null-reference errors.
Price	decimal(18,2)	Monetary amount; 18 total digits, 2 after the decimal point.
LastDiv	decimal(18,2)	Most recent dividend; same precision as Price.
Industry	string	Sector classification; also starts as string.Empty.
MarketCap	long	Total market value (can exceed 2 billion, hence long).

Example Model (C# / Entity Framework)

```
using System.ComponentModel.DataAnnotations;


public class Stock
{
    [Key]                   // Primary key
    public int Id { get; set; }


    public string Symbol { get; set; } = string.Empty;
    public string CompanyName { get; set; } = string.Empty;
    public decimal Price { get; set; }          // decimal(18,2) in DB
```

```
    public decimal LastDiv { get; set; }        // decimal(18,2) in DB
    public string Industry { get; set; } = string.Empty;
    public long MarketCap { get; set; }

    // Navigation property for one-to-many comments
    public List<Comment> Comments { get; set; } = new();
}
public class Comment
{
    [Key]
    public int Id { get; set; }

    public string Text { get; set; } = string.Empty;

    // Foreign key linking to Stock
    public int StockId { get; set; }
    public Stock Stock { get; set; } = null!;
}
```

Primary Key & Foreign Key

Primary Key (PK): *the unique identifier for a record; acts as the "parent."*

Foreign Key (FK): *a reference to a primary key in another table; acts as the "child."*

- A **PK** guarantees each row can be distinguished from every other row.
- An **FK** creates a link, enabling a **one-to-many** relationship (one parent → many children).

One-to-Many Relationship

In a **one-to-many** setup, a single **parent** record (e.g., a Stock) can be associated with multiple **child** records (e.g., Comments).

Steps to Implement (Entity Framework Core by convention)

1. **Add a collection navigation property** to the parent model (List<Comment> Comments).
2. **Add a scalar foreign-key property** (int StockId) to the child model.
3. **Add a reference navigation property** (Stock Stock) to the child model.
4. **Run migrations** so EF Core creates the appropriate PK and FK columns in the database.

Resulting Schema (simplified)

Table	Columns
Stock	Id (PK), Symbol, CompanyName, Price, LastDiv, Industry, MarketCap
Comment	Id (PK), Text, StockId (FK → Stock.Id)

- EF Core follows **convention**: if a property is named <Parent>Id and a navigation property <Parent> exists, it automatically treats it as a foreign key.

Entity Framework Conventions

Convention: *default rules that EF Core applies when you don't explicitly configure the model.*

- Naming pattern <EntityName>Id → recognized as a foreign key.
- Presence of a collection navigation property on the parent signals a **one-to-many** relationship.
- No extra configuration needed unless you want to customize column names, types, or cascade behavior.
- **Entity Framework Core** can scan your code and infer foreign-key relationships without needing explicit Fluent API configuration.
- The framework uses the presence of a **foreign-key property** (e.g., StockId) together with a matching **navigation property** to create the database relationship automatically.

Navigation Properties

Navigation property – a property in an entity class that holds a reference to a related entity (or a collection of related entities). It enables you to **dot-navigate** from one model to another, e.g., comment.Stock or stock.Comments.

- Provide **object-graph traversal** in LINQ queries (Include, ThenInclude).
- Allow lazy/eager loading of related data.
- Must be defined alongside the foreign-key field for EF Core to wire up the relationship.

Comment Model Definition

Below is a typical **Comment** entity that includes an ID, foreign key, navigation property, and other fields.

```
using System;
public class Comment
{
    // Primary key
    public int Id { get; set; }
    // Foreign key to Stock (or another related entity)
    public int StockId { get; set; }
```

```
    // Navigation property – lets us access the related Stock
    public Stock Stock { get; set; }

    // Comment details
    public string Title { get; set; } = string.Empty;
    public string Content { get; set; } = string.Empty;

    // Timestamp set when the row is inserted
    public DateTime CreatedOn { get; set; } = DateTime.Now;
}
```

Property Overview

Property	Type	Purpose
Id	int	Primary key for the Comment table
StockId	int	Foreign key linking to the Stock entity
Stock	Stock	Navigation property for relationship traversal
Title	string	Short heading for the comment (initialized to string.Empty)
Content	string	Main body of the comment (initialized to string.Empty)
CreatedOn	DateTime	Auto-populated timestamp when the comment is created

Setting the CreatedOn Timestamp

- The CreatedOn property is initialized with DateTime.Now.
- EF Core assigns the current date-time **at the moment the entity is added to the database**, ensuring an accurate creation record.

Steps to Ensure Automatic Timestamping

1. Declare the property with a default value: public DateTime CreatedOn { get; set; } = DateTime.Now;.
2. Add a new Comment instance via the DbContext (context.Comments.Add(comment);).
3. Call context.SaveChanges(); – EF Core writes the current timestamp to the column.

Using the Relationship in Queries

- To retrieve a comment with its related stock:

```
var commentWithStock = context.Comments
    .Include(c => c.Stock)
```

```
    .FirstOrDefault(c => c.Id == commentId);
```

- To get all comments for a specific stock:

```
var stockComments = context.Comments
    .Where(c => c.StockId == stockId)
    .ToList();
```

ASP.NET Core Web API .NET 8 2024 - 3. Entity Framework

Analogy: Excel Spreadsheets = Database Tables

- **Excel spreadsheets** are visually similar to **database tables**.
- A **database table** stores rows and columns, but on its own it isn't directly usable in code.
- An **Object-Relational Mapper (ORM)** bridges this gap by converting tables into **objects** that developers can manipulate.

Definition: *An ORM maps relational database structures to object-oriented programming constructs, allowing developers to work with data as native objects.*

Entity Framework (EF) Overview

- **Entity Framework** is the .NET ORM, turning each table row into a C# **object**.
- A single object represents one row; a **list of objects** represents multiple rows.

Installing EF Core & SQL Server Provider

Step	Action	NuGet Package	Purpose
1	Open **VS Code** → **Extensions** → search **NuGet Gallery**	NuGet Gallery	Provides UI for managing packages.
2	Open command palette (Ctrl+Shift+P) → run **NuGet Gallery**	–	Launches the NuGet manager.
3	Install **EF Core for SQL Server** (match version to your .NET SDK)	Microsoft.EntityFrameworkCore.SqlServer	Adds EF Core runtime and SQL Server provider.
4	Install **EF Core Tools** (optional but useful)	Microsoft.EntityFrameworkCore.Tools	Enables CLI commands like dotnet ef migrations.
5	Install **EF Core Design** (required for migrations)	Microsoft.EntityFrameworkCore.Design	Supplies design-time services for scaffolding.

Tip: Use the same major version (e.g., 8.x for .NET 8) across all EF packages to avoid compatibility errors.

Creating the Data Folder & DbContext

1. **Create folder** Data.
2. **Add class** ApplicationDbContext.cs inside Data.

```
using Microsoft.EntityFrameworkCore;
namespace YourNamespace.Data
{
    public class ApplicationDbContext : DbContext
    {
        // Constructor receiving DbContextOptions
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) { }

        // DbSet properties represent tables
        public DbSet<Stock> Stocks { get; set; }
        public DbSet<Comment> Comments { get; set; }
    }
}
```

- DbContext provides all the ORM "goodies".
- The **constructor** injects DbContextOptions and passes them to the base DbContext.
- Each DbSet<T> lets EF query and manipulate the corresponding table (T is the entity class).

Registering DbContext in Program.cs

```
var builder = WebApplication.CreateBuilder(args);

//Add EF Core DbContext with SQL Server provider
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
```

- Place the registration **before** app.Build() (i.e., before builder.Build() or any app calls).
- GetConnectionString("DefaultConnection") reads the connection string from *appsettings.json*.

Configuring the Connection String

1. Open **appsettings.json**.
2. Ensure the **ConnectionStrings** section exists (note the plural "s").

```
{
```

```
  "ConnectionStrings": {
    "DefaultConnection":
"Server=YOUR_DESKTOP_NAME\\SQLEXPRESS;Database=FinShark;Trusted_Connection=True;TrustServerCertificate=True;"
  }
}
```

- Replace YOUR_DESKTOP_NAME with the actual computer name shown in **SQL Server Management Studio (SSMS)** → right-click the server → **Properties** → **Computer Name**.
- Key flags: Trusted_Connection=True, TrustServerCertificate=True, Integrated Security=True.
- Adjust Database=FinShark if you chose a different database name.

Creating the Physical Database in SSMS

1. Launch **SQL Server Management Studio**.
2. Right-click **Databases** → **New Database...** → name it (e.g., FinShark).
3. Click **OK**.

The database will be created automatically when EF runs its first migration.

Running a Migration (Database Creation)

dotnet ef migrations add InitialCreate

dotnet ef database update

- **add InitialCreate**: scaffolds migration code that represents the current model (entities & DbSets).
- **database update**: applies the migration, generating the actual tables in the SQL Server database.

Note: Migrations are optional in a learning environment, but they illustrate how EF translates models into SQL schema.

Summary of Required Files

File	Key Content
Data/ApplicationDbContext.cs	Inherits DbContext, defines DbSet<Stock> & DbSet<Comment>.
Program.cs	Registers ApplicationDbContext with UseSqlServer.
appsettings.json	Contains "ConnectionStrings": { "DefaultConnection": "..."} .
*.csproj	Includes EF Core NuGet package references (matching .NET version).

Quick Checklist

- [ ] Install **NuGet Gallery** extension.
- [ ] Add EF Core packages (SqlServer, Tools, Design).
- [ ] Create **Data** folder & **ApplicationDbContext** class.
- [ ] Register DbContext in **Program.cs** before building the app.

- [ ] Add correct **connection string** to *appsettings.json*.
- [ ] Create the database in **SSMS** (matching name).
- [ ] (Optional) Run EF **migration** commands to generate tables. ## Generating a Migration
- Use the **Entity Framework Core (EF Core)** CLI to create migration code.
- Command format:

dotnet ef migrations add <MigrationName>

- Example (adds a migration named *Enit*):

dotnet ef migrations add Enit

- When the command succeeds **no errors** should appear.
- The generated file contains the C# code that describes how to **build the new entity** and its schema changes.

Applying the Migration to the Database

- The migration code is only **generated**, not executed.
- To actually create or update the database, run:

dotnet ef database update

- If the project does not require globalization, set the related option to **false** before running the update.

Verifying the Database in SSMS

1. Open **SQL Server Management Studio (SSMS)** and connect to the target database.
2. Right-click the database → **Refresh**.
3. Expand **Tables** → you should see:
    - __EFMigrationsHistory – tracks every migration that has been applied.
    - Tables created by the migration (e.g., Comments, Rows, Relationships).
4. To inspect data, right-click a table → **Select Top 100 Rows** (or any number you prefer).

Tip: Using *Select Top N Rows* is a quick way to confirm that tables and relationships were created correctly and to view sample data.

Summary of Key Commands

Command	Purpose	When to Use
dotnet ef migrations add <Name>	Generates migration code for schema changes	After modifying your EF Core model
dotnet ef database update	Applies pending migrations to the database	After verifying migration code, before running the app
Refresh (SSMS)	Reloads database objects to show new tables	After running database update

Checklist Before Running database update

- Migration added without errors (dotnet ef migrations add ...).
- Globalization flag set to **false** if not needed.
- Confirm the migration file reflects the intended entity changes.

Common Practices

- **Never** assume the database is created after adding a migration; always run dotnet ef database update.
- Regularly **refresh** SSMS to see the latest schema.
- Use the **Select Top ... Rows** feature for quick data validation.

ASP.NET Core Web API .NET 8 2024 - 4. Controllers

API Controllers Overview

API controllers act like the doors of a house, providing the entry points (URLs) through which clients interact with an API.

- Controllers map HTTP requests to code that handles them.
- Each controller typically corresponds to a **model** (e.g., Stock).

List vs. Detail Endpoints

- **List endpoint** – returns a collection of resources (e.g., a feed).
- **Detail endpoint** – returns a single resource identified by an ID.

Endpoint Type	Typical URL Pattern	Return Type
List	GET /api/stocks	IEnumerable<Stock>
Detail	GET /api/stocks/{id}	Stock (or 404)

Setting Up the Controllers Folder

1. In the API project, create a new folder named **Controllers**.
2. Add a controller file for each model, e.g., StockController.cs.
3. Keep controllers separate (e.g., CommentController.cs) to follow **separation of concerns**.

Implementing StockController

```
using Microsoft.AspNetCore.Mvc;
using YourProject.Data;   // DB context namespace
using YourProject.Models; // Stock model namespace

namespace YourProject.Controllers
{
   [ApiController]
   [Route("api/[controller]")]
   public class StockController : ControllerBase   // ← base class first
   {
      private readonly ApplicationDbContext _context; // read-only DB context
      public StockController(ApplicationDbContext context)
      {
         _context = context;
      }
```

```
    // GET list endpoint (see next section)
    // GET detail endpoint (see later)
  }
}
```

- **Why ControllerBase first?** Adding it after attributes avoids namespace conflicts and keeps the code tidy.
- Making _context **read-only** prevents accidental modification of the DB context reference.

GET List Endpoint (Read Operation)

```
[HttpGet]
public async Task<ActionResult<IEnumerable<Stock>>> GetStocks()
{
  // Deferred execution ⟶ SQL generated only when enumerated
  var stocks = await _context.Stocks.ToListAsync();
  return Ok(stocks);
}
```

- **GET** = "read".
- ToListAsync() forces execution of the query, turning the deferred LINQ query into an actual SQL call.

Deferred Execution Explained

Deferred execution means a LINQ query isn't run until its results are needed (e.g., when ToListAsync() is called). This lets EF Core build the SQL statement on the fly, improving performance and flexibility.

GET Detail Endpoint (Single Record)

```
[HttpGet("{id}")]
public async Task<IActionResult> GetStockById(int id)
{
  var stock = await _context.Stocks.FindAsync(id); // searches by primary key

  if (stock == null)
  {
    return NotFound(); // 404 wrapped in IActionResult
  }
  return Ok(stock); // 200 with the found entity
}
```

- **FindAsync** is optimized for primary-key lookups.
- IActionResult is a **wrapper** that lets you return different HTTP status codes without writing low-level response code.

Key Points

- **Model binding** automatically converts the {id} segment from the URL into an int.
- Always perform a **null check** and return NotFound() when the entity isn't present.

Registering Controllers in Program.cs

```
var builder = WebApplication.CreateBuilder(args);

// Add MVC services
builder.Services.AddControllers();

var app = builder.Build();

// Map attribute-routed controllers
app.MapControllers();

app.Run();
```

- Omitting AddControllers or MapControllers disables Swagger and leads to an HTTPS-redirect error.

Seeding Dummy Data in SQL Server

1. Open **SQL Server Management Studio**.
2. Insert sample rows into the Stocks table, e.g.:

```
INSERT INTO Stocks (Name, PurchasePrice, Dividend, Industry, MarketCap)
VALUES
('Vanguard Total Index', 200, 0.00, 'Index Fund', 2000000000000),
('Palmer', 20, 0.00, 'Technology', 5000000000);
```

- Populate enough records to test list and detail endpoints.

Running & Testing with Swagger

- Execute dotnet watch run (or dotnet run) to start the API.
- Navigate to https://localhost:{port}/swagger to explore the auto-generated UI.
- Swagger displays the **list** (GET /api/stocks) and **detail** (GET /api/stocks/{id}) endpoints, allowing you to send test requests and view responses.

ASP.NET Core Web API .NET 8 2024 - 5. DTOs

What Is a DTO?

Data Transfer Object (DTO) – a simple object that carries data between processes. It is used to **shape** the data you expose through an API, often trimming or reshaping the domain model.

- DTOs **do not contain business logic**; they only hold serialized data.
- In .NET Core APIs, most DTOs fall into **response** or **request** categories.

Response DTO vs. Request DTO

Aspect	Response DTO	Request DTO
Purpose	Send only the data needed by the client (e.g., hide passwords, internal fields).	Accept data from the client in a controlled format.
Typical Content	Subset of model properties, often without navigation properties or sensitive fields.	Fields required to create or update an entity, possibly with validation attributes.
Example	UserResponseDto containing Username but **excluding** Password.	UserCreateDto containing Username, Password, Email with data-annotation validation.

Organizing DTOs in a .NET Project

- Create a top-level **Dtos** folder.
- Optionally split into subfolders per domain (e.g., Dtos/Stocks, Dtos/Users).
- Keeps the project tidy as the number of DTOs grows.

```
/ProjectRoot
|
├── Controllers
├── Models
├── Dtos
|   ├── StockDto.cs
|   └── UserDto.cs
└── Mappers
    └── StockMapper.cs
```

Manual Mapping with Extension Methods

Instead of using AutoMapper, you can write **extension methods** that convert a model to its DTO.

```
// File: Mappers/StockMapper.cs
using Project.Models;
using Project.Dtos;
public static class StockMapper
{
   // Convert Stock model ⟶ StockDto
   public static StockDto ToStockDto(this Stock model)
   {
      return new StockDto
      {
         Id         = model.Id,
         Symbol     = model.Symbol,
         CompanyName = model.CompanyName,
         LastDiv    = model.LastDiv,
         Industry   = model.Industry,
         MarketCap  = model.MarketCap
         // Omit Comments, Sensitive fields, etc.
      };
   }
}
```

Key points

1. **Static class** holds the mapping logic.
2. Method is **static** and takes the source object as this – makes it an **extension method**.
3. Return a **new DTO instance** populated only with the desired properties.

Using LINQ Select as a Mapper

When you need to transform a collection of models into DTOs, use Select, which is the LINQ equivalent of map.

```
// Inside StockController.cs
[HttpGet]
public async Task<IEnumerable<StockDto>> GetAll()
{
```

```
    var stocks = await _context.Stocks.ToListAsync();
    // Map each Stock ⟶ StockDto
    return stocks.Select(s => s.ToStockDto());
}
```

- Select projects each element of stocks using the ToStockDto extension.
- Returns an IEnumerable<StockDto> ready for serialization.

Common DTO Practices

- **Never expose raw passwords** or other security-sensitive fields.
- **Trim navigation properties** (e.g., comments) unless the client explicitly needs them.
- **Apply data-annotation attributes** ([Required], [StringLength], etc.) on **request DTOs** for validation.
- Keep DTOs **flat** when possible; nested DTOs can increase payload size.

Example: User DTOs

```
// Dtos/UserResponseDto.cs
public class UserResponseDto
{
    public int    Id       { get; set; }
    public string Username { get; set; }
    // No Password field!
}
// Dtos/UserCreateDto.cs
public class UserCreateDto
{
    [Required]
    public string Username { get; set; }
    [Required]
    [StringLength(100, MinimumLength = 8)]
    public string Password { get; set; }
    [EmailAddress]
    public string Email { get; set; }
}
```

- **Response DTO** excludes the password.
- **Request DTO** includes validation attributes for automatic model-state checking.

ASP.NET Core Web API .NET 8 2024 - 6. POST (Create)

Overview of Data Creation in APIs

POST is the HTTP verb used to create new data on the server.
In a web-API context, a POST request typically carries a JSON payload that the server processes and persists to a database.

Entity Framework: Add vs. SaveChanges

Entity Framework (EF) tracks entities you add to the context but does **not** write them to the database until SaveChanges() is called.

EF Operation	What Happens	When Data Is Stored
context.Add(entity)	EF starts **tracking** the entity.	Not yet saved.
context.SaveChanges()	All tracked changes are flushed to the database.	Immediately after the call.

DTOs (Data Transfer Objects) and Their Purpose

DTOs define exactly what data a client is allowed to send or receive, preventing over-exposure of model fields (e.g., IDs, internal comments).
Typical DTO fields for a CreateStockRequestDto

DTO Property	Corresponding Model Property
CompanyName	CompanyName
PurchasePrice	PurchasePrice
LastDiv	LastDiv
Industry	Industry
MarketCap	MarketCap

The Id and Comments fields are deliberately omitted.

Implementing the POST Endpoint (C# / ASP.NET Core)

Define the Request DTO

```
// CreateStockRequestDto.cs
public class CreateStockRequestDto
{
    public string CompanyName { get; set; }
    public decimal PurchasePrice { get; set; }
    public decimal LastDiv { get; set; }
    public string Industry { get; set; }
```

```
    public long MarketCap { get; set; }
}
```

Build a Mapper to Convert DTO → Model

```
// StockMapper.cs
public static class StockMapper
{
    public static Stock FromCreateDto(CreateStockRequestDto dto)
    {
        return new Stock
        {
            CompanyName = dto.CompanyName,
            PurchasePrice = dto.PurchasePrice,
            LastDiv = dto.LastDiv,
            Industry = dto.Industry,
            MarketCap = dto.MarketCap
        };
    }
}
```

Create the Controller Action

```
// StocksController.cs
[HttpPost]
public IActionResult Create([FromBody] CreateStockRequestDto requestDto)
{
    // Map DTO to domain model
    var stock = StockMapper.FromCreateDto(requestDto);

    // Add to EF context (starts tracking)
    _context.Stocks.Add(stock);

    // Persist changes
    _context.SaveChanges();
```

```
    // Return the created object as a response DTO
    var responseDto = StockMapper.ToResponseDto(stock);
    return CreatedAtAction(nameof(GetById), new { id = stock.Id }, responseDto);
}
```

Key points

- **[FromBody]** tells ASP.NET Core to bind the JSON payload from the request body.
- CreatedAtAction automatically generates a **201 Created** response with a location header pointing to GetById.

Supporting GET-by-ID Action (used by CreatedAtAction)

```
[HttpGet("{id}")]
public IActionResult GetById(int id)
{
    var stock = _context.Stocks.Find(id);
    if (stock == null) return NotFound();

    var dto = StockMapper.ToResponseDto(stock);
    return Ok(dto);
}
```

End-to-End Create Request Flow

1. **Client** sends a POST request with JSON matching CreateStockRequestDto.
2. ASP.NET Core binds the JSON to requestDto ([FromBody]).
3. **Mapper** converts requestDto to a Stock entity.
4. EF **adds** the entity (_context.Stocks.Add) – the entity is now tracked.
5. **SaveChanges** writes the tracked entity to the database.
6. The newly saved entity (now with an Id) is mapped back to a **response DTO**.
7. CreatedAtAction returns **201 Created** with the location of the new resource.

Verifying the POST Endpoint

- Run the API: dotnet watch run (or cd api && dotnet watch run).
- Use a tool like **Postman** or **cURL** to POST JSON:

```
curl -X POST http://localhost:5000/api/stocks \
    -H "Content-Type: application/json" \
    -d '{
```

```
  "CompanyName":"Acme Corp",
  "PurchasePrice":125.50,
  "LastDiv":2.3,
  "Industry":"Manufacturing",
  "MarketCap":5000000000
}'
```

- Expected result: response contains the newly created stock **without** Id or Comments fields in the request, but the response includes the generated Id.

Summary of Files Created

File	Purpose
CreateStockRequestDto.cs	Defines allowed input fields for creating a stock.
StockMapper.cs	Handles conversion between DTOs and the Stock model.
StocksController.cs	Contains the POST (Create) and GET (GetById) actions.
Stock.cs (model)	Entity representing a stock record in the database.

These components together enable a clean, secure, and testable **POST** workflow in an ASP.NET Core API using Entity Framework.

ASP.NET Core Web API .NET 8 2024 - 7. PUT (Update)

Update vs Create Overview

Update – an operation that replaces the data of an existing record with a new representation.

Create – an operation that inserts a brand-new record into the data store.

- **Update** requires knowledge of the *existing* entity (usually its **ID**).
- **Create** does **not** need an identifier because the record does not yet exist.

Identifying Records for Update

- The request must include an **identifier** (most often the primary key Id).
- Identifier can be supplied:
    - In the **URL** (e.g., PUT /api/stocks/5) ⟶ FromRoute.
    - In the **body** as part of the DTO ⟶ FromBody.
- Without a valid ID the system cannot locate the target record, so the update cannot proceed.

Full Object Replacement vs Patch

Operation	What is sent	Fields affected	Typical HTTP verb
Create	Complete object (no Id)	All fields become new record	POST
Update	Complete object (including Id)	**All** fields are overwritten; null values replace existing data	PUT
Patch	Partial object (only changed fields)	Only listed fields are modified; others stay unchanged	PATCH

- Sending null in a **PUT** will set the corresponding column to NULL in the database.
- **PATCH** ignores omitted fields, leaving their current values untouched.

Searching for Existing Record

1. Use a **searching algorithm** (e.g., FirstOrDefaultAsync) to retrieve the entity by its Id.
2. If the entity is not found, return **404 Not Found**.
3. The retrieved entity becomes *tracked* by Entity Framework, enabling change detection.

Modifying and Tracking with Entity Framework

- Once the entity is loaded, modify its properties **directly** on the object.
- EF Core automatically tracks these changes.
- **Important:** modifications alone do **not** persist to the database; you must call SaveChangesAsync().

```
// Example: change company name
stock.CompanyName = updateDto.CompanyName;
```

DTO Design for Update Endpoint

- Create a **dedicated DTO** for each endpoint to avoid accidental field mismatches.
- Example DTO for updating a stock:

```
public class UpdateStockRequestDto
{
    public string CompanyName { get; set; }
    public decimal PurchasePrice { get; set; }
    public string Industry { get; set; }
    public decimal MarketCap { get; set; }
}
```

- Keeping DTOs separate from the **Create** DTO ensures that required/optional fields are correctly enforced.

Sample PUT Endpoint Code (C#)

```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateStock(
    [FromRoute] int id,
    [FromBody] UpdateStockRequestDto updateDto)
{
    //Retrieve existing record
    var stock = await _context.Stocks.FirstOrDefaultAsync(s => s.Id == id);

    //Null check
    if (stock == null)
        return NotFound();

    //Apply updates (EF Core tracks changes)
    stock.CompanyName   = updateDto.CompanyName;
    stock.PurchasePrice = updateDto.PurchasePrice;
    stock.Industry      = updateDto.Industry;
    stock.MarketCap     = updateDto.MarketCap;

    //Persist changes
```

```
    await _context.SaveChangesAsync();

    //Return updated DTO
    var resultDto = _mapper.Map<ReadStockDto>(stock);
    return Ok(resultDto);
}
```

Null Check and Response Handling

- Perform a **null check** immediately after the search:

```
if (stock == null)
    return NotFound();   // HTTP 404
```

- Prevents unnecessary processing and returns a clear client signal.

SaveChanges and Persisting Updates

- await _context.SaveChangesAsync(); generates the necessary **SQL UPDATE** statement and sends it to the database.
- Without this call, changes remain only in memory and are lost when the request ends.

Comparison Table (Create vs Update vs Patch)

Feature	Create (POST)	Update (PUT)	Patch (PATCH)
Identifier required	No	Yes (in URL or body)	Yes (in URL)
Payload	Full object (no Id)	Full object (including Id)	Partial object (only changed fields)
Null handling	N/A (new row)	Sets column to NULL if sent	Ignored (field unchanged)
EF Core tracking	New entity added	Existing entity tracked after retrieval	Tracked entity; only modified fields are sent
Typical response	201 Created + location header	200 OK + updated DTO	200 OK + updated DTO

ASP.NET Core Web API .NET 8 2024 - 8. DELETE

HTTP DELETE Endpoint Overview

Definition: An HTTP DELETE request removes a resource identified by a URI. In RESTful APIs, it typically returns **204 No Content** on success.

- Sends only the **resource ID** to the API.
- Entity Framework (EF) handles most of the heavy lifting.
- The controller action must:
    1. Locate the entity by its ID.
    2. Verify the entity exists.
    3. Remove the entity from the context.
    4. Persist changes with SaveChanges().
    5. Return an appropriate status code.

Route and Method Signature

```
[HttpDelete("{id}")]
public IActionResult Delete([FromRoute] int id)
{
   // implementation...
}
```

- HttpDelete("{id}") defines the route template and binds the **id** from the URL.
- [FromRoute] explicitly tells ASP.NET Core to pull the value from the route (good practice for clarity).

Step-by-Step Deletion Logic

1. **Retrieve the entity**
2. var stock = _context.Stocks.FirstOrDefault(s => s.Id == id);
3. **Null check** – if stock is null, the resource doesn't exist ⟶ return **404 Not Found**.
4. if (stock == null)
5. return NotFound();
6. **Remove the entity**
7. _context.Stocks.Remove(stock);
8. **Commit the change**
9. _context.SaveChanges();
10. **Return success** – use NoContent() which maps to **204 No Content**.

11. return NoContent();

HTTP Status Codes for DELETE Operations

Status Code	Meaning	Typical Use Case
204	**No Content** – successful delete	Resource removed
404	**Not Found** – ID does not exist	Invalid ID
400	**Bad Request** – malformed request	Invalid payload
500	**Internal Server Error**	Unexpected failure

Testing with Swagger

- After implementing the endpoint, **restart** the application.
- Open **Swagger UI** (usually at /swagger).
- Use the **DELETE** button, provide the target ID (e.g., vti), and execute.
- Expect a **204** response with no body. Verify the item no longer appears in the GET list.

Key Takeaways

- Always include the **route attribute** ([HttpDelete("{id}")]) for clarity and routing safety.
- A **null check** prevents attempts to delete non-existent records.
- Remove() followed by SaveChanges() triggers EF to generate the appropriate DELETE SQL.
- Returning NoContent() aligns with REST conventions for successful deletions.

ASP.NET Core Web API .NET 8 2024 - 9. Async/Await

Why Database & Network Calls Are Slow

- **Database calls** and **network requests** require reaching out to external systems, which introduces latency (often measured in milliseconds to seconds).
- These operations are considered the **slowest actions** in typical application code because they must travel across hardware, routers, and remote servers.

Definition: *I/O-bound operation* – any task that spends most of its time waiting for external resources (e.g., disk, network, database) rather than using the CPU.

Async as the "Secret Weapon"

- **Asynchronous programming** lets the CPU work on other tasks while waiting for I/O, dramatically improving throughput.
- Originally introduced in **C#**, the pattern quickly spread to many languages because it solves the I/O bottleneck elegantly.

Core Concepts

Concept	Purpose	Typical Syntax (C#)
async	Marks a method as capable of asynchronous execution.	public async Task<T> MethodAsync()
Task<T>	Represents a future value; a **wrapper** for the result of an async operation.	Task<StockDto>
await	Pauses the method until the awaited Task completes, without blocking the thread.	var result = await dbContext.FindAsync(id);
Coroutine (generated by the compiler)	Under the hood mechanism that yields control back to the caller while awaiting.	–

Converting Synchronous Code to Async (Step-by-Step)

1. **Add the async keyword** to the method signature.
2. **Change the return type** to Task<T> (or Task for void).
3. **Identify I/O calls** (e.g., Find, Select, SaveChanges) and replace them with their async equivalents (FindAsync, ToListAsync, etc.).
4. **Prepend await** to each async call.
5. **Ensure the method body only contains async code** where necessary; code without I/O remains synchronous.

Example Transformation

```
// Synchronous version
public ActionResult<StockDto> GetStock(int id)
{
    var stock = _context.Stocks.Find(id);
    var dto = new StockDto(stock);
    return Ok(dto);
}

// Asynchronous version
public async Task<ActionResult<StockDto>> GetStockAsync(int id)
{
    var stock = await _context.Stocks.FindAsync(id);
    var dto = new StockDto(stock);
    return Ok(dto);
}
```

Common Async Patterns in Controllers

- **GET (single item)** → FindAsync + await
- **GET (list)** → ToListAsync + await
- **POST / CREATE** → AddAsync (or just Add) + await _context.SaveChangesAsync()
- **PUT / UPDATE** → Update + await _context.SaveChangesAsync()
- **DELETE** → **Synchronous** (Remove) – the framework does not provide RemoveAsync; keep it synchronous and wrap the method in Task if needed.

Note: The delete operation is *not* async in EF Core; the reason is undocumented. Feel free to comment if you discover why.

Practical Tips & Debugging

- The async keyword **does not change execution**; it only signals to the compiler and developers that the method contains awaitable code.
- **Debugging async code** can be harder; look for the await keyword to locate the actual asynchronous suspension points.
- When a method returns Task<T>, the compiler expects a value **immediately** (even if the underlying operation hasn't finished). The Task acts as a placeholder.

Running & Testing the Async API

Start the application with hot-reload

dotnet watch run

Sample HTTP Interactions

Verb	Endpoint	Expected Result	Status Code
GET	/api/stocks	List of StockDto objects	200
GET	/api/stocks/{id}	Single StockDto	200
POST	/api/stocks	Created StockDto	201
PUT	/api/stocks/{id}	Updated StockDto (e.g., change PurchasePrice to 90)	200
DELETE	/api/stocks/{id}	Resource removed	204

- After converting to async, requests **respond faster** because the server can handle more concurrent I/O operations.

Checklist for Async Refactor

- [ ] All controller actions marked async.
- [ ] Return types changed to Task<ActionResult<T>> (or Task<IActionResult>).
- [ ] All EF Core calls (Find, ToList, SaveChanges) replaced with their Async counterparts.
- [ ] await added before each async call.
- [ ] Delete actions left synchronous but wrapped in Task if method signature requires async.
- [ ] Application runs without compile-time errors and all endpoints return the expected status codes.

ASP.NET Core Web API .NET 8 2024 - 10. Repository Pattern + DI

Problem with Controllers

- Controllers have become **too large** and contain **direct database calls**.
- A controller's responsibility is to handle **HTTP requests and route URLs**, not to manage data access.
- Mixing concerns makes the code **hard to maintain** and **hard to test**.

Principle: *Separate concerns* – keep data-access logic out of controllers.

Repository Pattern

- **"Code to an interface"** = code to an **abstraction**.
- The abstraction groups repetitive data-access code (e.g., repeated FirstOrDefault queries) into a single method.
- The repository provides a **central place** for all data-access operations.

Benefits	How it works
Single point to change data logic	Define an interface (e.g., IStockRepository)
Easier unit testing	Implement the interface in a concrete class (StockRepository)
Swappable data source	Swap the implementation without touching callers

Dependency Injection (DI)

- **Constructor-based DI** supplies required objects (e.g., DbContext) when a class is instantiated.
- DI "pre-heats the oven" by ensuring the **database context** is available before any repository method runs.
- The DI container resolves the concrete implementation of an interface at runtime.

Definition: *Dependency Injection* is a technique where an object's dependencies are provided externally rather than created inside the object.

Project Structure

/Interfaces

  IStockRepository.cs

/Repositories

  StockRepository.cs

/Controllers

  StockController.cs

Program.cs

- **Interfaces folder** – holds all contracts (I*Repository).
- **Repositories folder** – concrete implementations that interact with the DB.

Implementation Steps

1. **Create the interface**
    - Add IStockRepository with a method signature Task<List<Stock>> GetAllAsync();.
2. **Implement the repository**
    - Create StockRepository class that **inherits** IStockRepository.
    - Inject ApplicationDbContext via the constructor.
    - Implement GetAllAsync using await _context.Stocks.ToListAsync();.
3. **Register the service** in Program.cs
4. builder.Services.AddScoped<IStockRepository, StockRepository>();
5. **Refactor the controller**
    - Remove direct DbContext calls.
    - Add a private readonly IStockRepository _stockRepo;.
    - Inject the repository through the controller's constructor.
    - Call _stockRepo.GetAllAsync() instead of querying the context.
6. **Restart the application** to apply the DI changes.

Code Samples

Interface Definition (IStockRepository.cs)

```
using System.Collections.Generic;
using System.Threading.Tasks;

public interface IStockRepository
{
    Task<List<Stock>> GetAllAsync();
}
```

Repository Implementation (StockRepository.cs)

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

public class StockRepository : IStockRepository
{
    private readonly ApplicationDbContext _context;
```

```
    // Constructor injection
    public StockRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<List<Stock>> GetAllAsync()
    {
        return await _context.Stocks.ToListAsync();
    }
}
```

Updated Controller (StockController.cs)

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

[ApiController]
[Route("[controller]")]
public class StockController : ControllerBase
{
    private readonly IStockRepository _stockRepo;

    public StockController(IStockRepository stockRepo)
    {
        _stockRepo = stockRepo;
    }

    [HttpGet]
    public async Task<IActionResult> GetAll()
    {
        var stocks = await _stockRepo.GetAllAsync();
        return Ok(stocks);
```

```
    }
}
```

DI Registration (Program.cs)

```
var builder = WebApplication.CreateBuilder(args);

// Other service registrations...

builder.Services.AddScoped<IStockRepository, StockRepository>();

var app = builder.Build();

// Middleware pipeline...

app.Run();
```

Comparison: Before vs. After Refactor

Aspect	Before (Controller-centric)	After (Repository + DI)
Location of DB calls	Inside controller methods	Inside StockRepository
Testability	Hard – requires DB in tests	Easy – mock IStockRepository
Code duplication	Repeated queries across actions	Single method GetAllAsync
Flexibility	Changing data source requires controller changes	Swap repository implementation only

Key Takeaways

- **Separate concerns**: Controllers handle routing, repositories handle data access.
- **Code to abstractions**: Use interfaces (IStockRepository) as contracts.
- **Leverage DI**: Constructor injection provides required services and keeps classes loosely coupled.
- **Maintainability**: Centralizing data logic makes future changes (filters, sorting, different DB) straightforward.

ASP.NET Core Web API .NET 8 2024 - 11. Refactor To Repository

Repository Pattern Overview

Repository Pattern – an abstraction layer that encapsulates data-access logic, providing a clean contract (interface) for CRUD operations while keeping the rest of the application agnostic of the underlying data source.

- Centralizes **database calls**.
- Improves **testability** and **maintainability**.
- Typically paired with an **interface** that defines the contract.

Defining the Repository Interface

```
public interface IStockRepository
{
    Task<StockModel?> GetByIdAsync(int id);          // returns null if not found
    Task<StockModel> CreateAsync(StockModel model); // never null – creates a new record
    Task<StockModel?> UpdateAsync(int id, UpdateStockRequestDto dto);
    Task<StockModel?> DeleteAsync(int id);
    Task<IEnumerable<StockModel>> GetAllAsync();
}
```

- ? after StockModel signals a **nullable** return (used for methods that may not find a record).
- All methods are **asynchronous** (Task<...>).

Implementing the Repository

Add async to every method

- Always ensure the async keyword is present; otherwise the contract is broken and the compiler will flag a mismatch.

Create (Insert)

```
public async Task<StockModel> CreateAsync(StockModel model)
{
    _context.Stocks.Add(model);
    await _context.SaveChangesAsync();
    return model;
}
```

Delete

```
public async Task<StockModel?> DeleteAsync(int id)
{
    var stock = await _context.Stocks.FirstOrDefaultAsync(s => s.Id == id);
    if (stock == null) return null;
    _context.Stocks.Remove(stock);   // Remove is synchronous
    await _context.SaveChangesAsync();
    return stock;
}
```

Get All

```
public async Task<IEnumerable<StockModel>> GetAllAsync()
{
    return await _context.Stocks.ToListAsync();
}
```

Get By ID

```
public async Task<StockModel?> GetByIdAsync(int id)
{
    return await _context.Stocks.FindAsync(id);
}
```

Update

```
public async Task<StockModel?> UpdateAsync(int id, UpdateStockRequestDto dto)
{
    var existing = await _context.Stocks.FirstOrDefaultAsync(s => s.Id == id);
    if (existing == null) return null;

    // map DTO fields onto the entity
    existing.CompanyName = dto.CompanyName;
    existing.PurchasePrice = dto.PurchasePrice;
    existing.LastDividend = dto.LastDividend;
    existing.Industry = dto.Industry;
    existing.MarketCap = dto.MarketCap;
    await _context.SaveChangesAsync();
```

```
    return existing;
}
```

Method Summary Table

Method	Return Type	Nullable?	Async?
GetByIdAsync	StockModel	Yes	Yes
CreateAsync	StockModel	No	Yes
UpdateAsync	StockModel	Yes	Yes
DeleteAsync	StockModel	Yes	Yes
GetAllAsync	IEnumerable<StockModel>	No	Yes

Updating the Controller

Replace direct DbContext usage with repository calls.

```
[ApiController]
[Route("api/[controller]")]
public class StocksController : ControllerBase
{
    private readonly IStockRepository _repo;

    public StocksController(IStockRepository repo) => _repo = repo;

    // GET /api/stocks/{id}
    [HttpGet("{id}")]
    public async Task<ActionResult<StockModel>> Get(int id) =>
        await _repo.GetByIdAsync(id) ?? NotFound();

    // GET /api/stocks
    [HttpGet]
    public async Task<ActionResult<IEnumerable<StockModel>>> GetAll() =>
        Ok(await _repo.GetAllAsync());

    // POST /api/stocks
    [HttpPost]
```

```
    public async Task<ActionResult<StockModel>> Create(StockModel model)
    {
        var created = await _repo.CreateAsync(model);
        return CreatedAtAction(nameof(Get), new { id = created.Id }, created);
    }

    // PUT /api/stocks/{id}
    [HttpPut("{id}")]
    public async Task<ActionResult<StockModel>> Update(int id, UpdateStockRequestDto dto)
    {
        var updated = await _repo.UpdateAsync(id, dto);
        return updated is null ? NotFound() : Ok(updated);
    }

    // DELETE /api/stocks/{id}
    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
        var deleted = await _repo.DeleteAsync(id);
        return deleted is null ? NotFound() : NoContent();
    }
}
```

- All controller actions now **delegate** to the repository.
- Boilerplate DB code has been removed, making the controller concise.

Testing the Refactored Endpoints

1. **Restart** the server to ensure the new services are loaded.
2. **Create** a new stock (e.g., Tesla) via POST – verify a 201 Created response with the new Id.
3. **Read** the newly created record with GET /api/stocks/{id} – confirm the returned JSON matches the input.
4. **Update** the record using PUT – change fields like PurchasePrice and verify the response reflects the changes.
5. **Delete** the record with DELETE – expect a 204 No Content.
6. **Confirm Deletion** by attempting another GET; the API should return 404 Not Found.

All CRUD operations now flow through the **repository**, producing cleaner, more testable code.

ASP.NET Core Web API .NET 8 2024 - 12. Comment System

Repository Interface

- Create an **interface** for comment data access: ICommentRepository.
- Add a single method for now: Task<List<Comment>> GetAllAsync();

```
public interface ICommentRepository
{
    Task<List<Comment>> GetAllAsync();
}
```

Definition: *Repository pattern* – an abstraction that separates the data-access logic from business logic, allowing easier testing and swapping of data sources.

Repository Implementation

1. Add a concrete class CommentRepository that **implements** ICommentRepository.
2. Inject the ApplicationDbContext (or simply Context) via the constructor.
3. Store the context in a **private readonly** field (_context).
4. Implement GetAllAsync by querying Comments and returning a List<Comment>.

```
public class CommentRepository : ICommentRepository
{
    private readonly ApplicationDbContext _context;

    public CommentRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<List<Comment>> GetAllAsync()
    {
        return await _context.Comments.ToListAsync();
    }
}
```

Dependency Injection Setup

- Register the repository in **Program.cs** (or Startup.cs) so it can be injected into controllers.

builder.Services.AddScoped<ICommentRepository, CommentRepository>();

Tip: Even if the app works without explicit registration, adding it keeps the DI container explicit and maintainable.

Controller Setup

1. Create CommentController that inherits from ControllerBase.
2. Annotate with [ApiController] and [Route("api/[controller]")].
3. Inject ICommentRepository through the constructor.
4. Add an async **GET** action returning IActionResult.

```
[ApiController]
[Route("api/[controller]")]
public class CommentController : ControllerBase
{
    private readonly ICommentRepository _commentRepo;

    public CommentController(ICommentRepository commentRepo)
    {
        _commentRepo = commentRepo;
    }

    [HttpGet]
    public async Task<IActionResult> GetAll()
    {
        var comments = await _commentRepo.GetAllAsync();
        var dtos = comments.Select(CommentMapper.ToDto);
        return Ok(dtos);
    }
}
```

DTO Creation

- Add CommentDto in the **DTO** folder.
- Copy properties from the Comment entity **except** navigation properties.
- Include: Id, Title, Content, CreatedOn, StockId.

```
public class CommentDto
{
```

```
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public string Content { get; set; } = string.Empty;
    public DateTime CreatedOn { get; set; }
    public int StockId { get; set; }
}
```

Entity Property	DTO Inclusion	Reason
Id	Yes	Primary key needed by client
Title	Yes	Display field
Content	Yes	Main comment body
CreatedOn	Yes	Timestamp for ordering
StockId	Yes	Links comment to a stock
NavigationProp	No	Would embed full Stock object, unnecessary for API

Mapper

- Create a static **mapper** class to convert Comment ⟶ CommentDto.
- Use a LINQ Select in the controller to map collections.

```
public static class CommentMapper
{
    public static CommentDto ToDto(Comment model)
    {
        return new CommentDto
        {
            Id = model.Id,
            Title = model.Title,
            Content = model.Content,
            CreatedOn = model.CreatedOn,
            StockId = model.StockId
        };
    }
}
```

Note: Select works like JavaScript's map, producing a new collection without altering the original list.

Testing & Seeding

1. Add temporary test comments directly in code (or via a seeder) to verify the endpoint.
2. Example seed data:

```
_context.Comments.AddRange(
    new Comment
    {
        Title = "Test Comment",
        Content = "Test content",
        StockId = 21, // Tesla example
        CreatedOn = DateTime.UtcNow
    },
    new Comment
    {
        Title = "Another Test",
        Content = "More content",
        StockId = 21,
        CreatedOn = DateTime.UtcNow
    });

await _context.SaveChangesAsync();
```

3. Run the app with dotnet watch run and hit GET /api/comment to confirm the JSON payload matches CommentDto.

Successful response confirms the **infrastructure** (repository, DI, controller, DTO, mapper) is wired correctly.

ASP.NET Core Web API .NET 8 2024 - 13. Comment GET + Include()

Adding GetById to ICommentRepository

- Open **ICommentRepository.cs**.
- Add the asynchronous signature that returns a single comment (or null):

```
Task<Comment?> GetByIdAsync(int id);
```

- The method takes an **int** identifier because each comment is uniquely identified by its primary key.

Definition – *Include*: In Entity Framework Core, Include tells the query to eagerly load related navigation properties (e.g., load a stock's comments together with the stock entity).

Implementing GetByIdAsync in CommentRepository

1. Call the repository's GetByIdAsync from the controller.
2. Retrieve the comment with await _commentRepo.GetByIdAsync(id);.
3. Check for null:

```
if (comment == null)
    return NotFound();
```

4. Return the comment mapped to a DTO:

```
return Ok(_mapper.Map<CommentDto>(comment));
```

- The controller now responds with **200 OK** and a JSON payload or **404 Not Found** when the ID does not exist.

Updating DTOs to Carry Comments

- **StockDto**: add a collection property for comments.

```
public class StockDto
{
    // existing properties ...
    public List<CommentDto> Comments { get; set; } = new();
}
```

- **CommentDto** (example):

```
public class CommentDto
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public string Content { get; set; } = string.Empty;
    public DateTime CreatedOn { get; set; }
```

```
}
```

Mapping Comments with AutoMapper

- Extend the stock-to-DTO mapping to include the comment list:

```
CreateMap<Stock, StockDto>()
   .ForMember(dest => dest.Comments,
         opt => opt.MapFrom(src => src.Comments));
```

- Ensure the comment mapping exists:

```
CreateMap<Comment, CommentDto>();
```

- Convert the result to a List<CommentDto> when needed:

```
var commentDtos = stock.Comments
                .Select(c => _mapper.Map<CommentDto>(c))
                .ToList();
```

Eager-Loading Comments in Stock Queries

- Modify the **stock repository** to use Include so comments are loaded with each stock:

```
using Microsoft.EntityFrameworkCore;

public async Task<Stock?> GetByIdAsync(int id)
{
   return await _context.Stocks
                .Include(s => s.Comments)
                .FirstOrDefaultAsync(s => s.Id == id);
}
```

- Use FirstOrDefaultAsync instead of Find because Find bypasses Include.

Installing & Configuring Newtonsoft.Json

Step	Action
1	Open the command palette (Ctrl+Shift+P) → **NuGet: Install Package**.
2	Install **Newtonsoft.Json** (latest version).
3	Install **Microsoft.AspNetCore.Mvc.NewtonsoftJson** (version 8.x for .NET 8).
4	In **Program.cs**, add the JSON serializer with reference-loop handling:

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services
    .AddControllers()
    .AddNewtonsoftJson(options =>
      options.SerializerSettings.ReferenceLoopHandling =
        Newtonsoft.Json.ReferenceLoopHandling.Ignore);
```

- This setting prevents **object cycles** that EF Core can create when serializing navigation properties (e.g., Stock Comments).

Running & Testing the Endpoint

1. Rebuild the solution (dotnet watch run).
2. Open **Swagger UI**.
3. Call **GET** /api/comments/{id} to verify a single comment is returned.
4. Call **GET** /api/stocks/{id} to confirm the stock payload now includes a comments array.

- If a stock has no comments, the array will be empty ([]).
- Successful responses show the newly added comment data (title, content, created date) alongside the stock fields.

ASP.NET Core Web API .NET 8 2024 - 14. 1-To-Many CREATE (Post)

Database Relationship (Stock <->Comment)

Foreign Key – a column in a child table that references the primary key of a parent table, establishing a parent-child relationship.

- **Stock** is the **parent** entity (primary key: Id).
- **Comment** is the **child** entity.
- Comment.StockId is the **foreign key** that points to Stock.Id.
- One stock can have **many** comments (one-to-many relationship).

Creating a Comment (API Design)

Route & Action Signature

```
// POST: /api/stocks/{stockId}/comments
[HttpPost("{stockId}")]
public async Task<IActionResult> CreateCommentAsync(
    [FromRoute] int stockId,
    [FromBody] CreateCommentDto createDto)
{
    // implementation ...
}
```

- The **stockId** in the route ties the new comment to its parent stock.
- The body contains only the fields needed to create a comment.

CreateCommentDto

```
public class CreateCommentDto
{
    public string Title { get; set; }      // comment title
    public string Content { get; set; }    // comment body
}
```

- No Id or CreatedOn – those are generated by the system.

DTO → Model Mapping

Mapper Method

```
public static Comment ToComment(this CreateCommentDto dto, int stockId)
{
    return new Comment
    {
        Title   = dto.Title,
        Content = dto.Content,
        StockId = stockId         // set foreign key
        // CreatedOn is set automatically by EF Core
    };
}
```

- Extends CreateCommentDto with a **conversion** to the Comment entity.
- Assigns the passed-in stockId to Comment.StockId.

Repository Methods

Repository	Method	Return Type	Description
IStockRepository	Task<bool> StockExistsAsync(int id)	bool	Checks whether a stock with the given id exists.
ICommentRepository	Task<Comment> AddAsync(Comment comment)	Comment	Inserts a new comment and returns the persisted entity.
CommentRepository (implementation)	await _context.Comments.AddAsync(comment); await _context.SaveChangesAsync();	_	Persists the comment to the database.

Controller Flow – Steps to Create a Comment

1. **Extract stockId** from the route and the DTO from the request body.
2. **Validate stock existence** by calling await _stockRepository.StockExistsAsync(stockId).
    - If false, return **404 Not Found** ("Stock does not exist").
3. **Map DTO → Model** using createDto.ToComment(stockId).
4. **Persist comment** with await _commentRepository.AddAsync(comment).
5. **Return** CreatedAtAction("GetById", new { id = comment.Id }, comment.ToDto()).

Validation Logic (Stock Existence)

```
if (!await _stockRepository.StockExistsAsync(stockId))
{
    return NotFound("Stock does not exist");
}
```

- Centralizes the existence check in the **stock repository** for reuse across the API.

Example: Adding Comments to Stocks

- **Tesla (Id = 21)**

  POST /api/stocks/21/comments ⟶ creates a comment linked to Tesla.
- **Microsoft (Id = 22)**

  POST /api/stocks/22/comments ⟶ creates a comment linked to Microsoft.

Both requests follow the same flow described above, demonstrating that the **foreign-key relationship** automatically associates each comment with the correct stock.

ASP.NET Core Web API .NET 8 2024 - 15. Comment UPDATE (Put)

Update Endpoint Overview

- Updating a comment only requires the **comment ID** and the new data; the many-to-one relationship is already established.
- The client sends a **PUT** request with the comment ID in the route and a JSON body containing the updated fields.

Definition: A **PUT** request replaces the existing resource (or part of it) identified by the URL with the supplied representation.

DTO Definitions

DTO Name	Purpose	Fields
UpdateCommentRequestDto	Carries the data needed to modify a comment	string Title string Content
CreateCommentRequestDto (reference)	Used for creating comments; similar structure to the update DTO	string Title string Content

```
// UpdateCommentRequestDto.cs
public class UpdateCommentRequestDto
{
    public string Title { get; set; }
    public string Content { get; set; }
}
```

Repository Update Method

1. Define the async update signature in the repository interface.
2. Implement the method:
    - Retrieve the existing comment by **ID**.
    - Return null if not found.
    - Update only the **Title** and **Content** (leave other fields untouched).
    - Call SaveChangesAsync() to persist.

```
// ICommentRepository.cs
public interface ICommentRepository
{
    Task<Comment?> UpdateAsync(int id, Comment comment);
}

// CommentRepository.cs
public class CommentRepository : ICommentRepository
{
    private readonly AppDbContext _context;

    public CommentRepository(AppDbContext context) => _context = context;

    public async Task<Comment?> UpdateAsync(int id, Comment updatedComment)
    {
        var existing = await _context.Comments.FindAsync(id);
        if (existing == null) return null;
        existing.Title   = updatedComment.Title;
        existing.Content = updatedComment.Content;

        await _context.SaveChangesAsync();
        return existing;
    }
}
```

Controller Action

- Route: PUT /api/comments/{id}
- Parameters: int id from route, UpdateCommentRequestDto dto from body.
- Steps:
    1. Map the DTO to a **Comment** entity.
    2. Call UpdateAsync(id, comment).
    3. Return **404 NotFound** if the repository returns null.
    4. Return **200 OK** with the updated comment.

```
// CommentController.cs
[ApiController]
[Route("api/[controller]")]
public class CommentController : ControllerBase
{
    private readonly ICommentRepository _repo;
    private readonly IMapper _mapper;

    public CommentController(ICommentRepository repo, IMapper mapper)
    {
        _repo   = repo;
        _mapper = mapper;
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> Update(int id, [FromBody] UpdateCommentRequestDto dto)
    {
        var commentEntity = _mapper.Map<Comment>(dto);
        var updated = await _repo.UpdateAsync(id, commentEntity);

        if (updated == null)
            return NotFound("Comment not found");

        var resultDto = _mapper.Map<UpdateCommentRequestDto>(updated);
        return Ok(resultDto);
    }
}
```

Validation & Error Handling

- **Existence Check:** if (existing == null) return null; ensures we don't attempt to update a non-existent record.
- **HTTP Responses:**
    - 404 NotFound ⟶ comment with supplied ID does not exist.
    - 200 OK ⟶ successful update, returns the updated DTO.

Saving Changes with EF Core

- EF Core automatically tracks the retrieved entity (existing).
- Modifying its properties marks those fields as changed.
- SaveChangesAsync() commits only the altered columns to the database.

Testing with Swagger

1. Open Swagger UI (/swagger).
2. Locate **PUT /api/comments/{id}**.
3. Provide an existing comment ID (e.g., 2) and a JSON body:

```
{
  "title": "Apple",
  "content": "Apple is the best stock"
}
```

4. Execute the request and verify the response reflects the updated values.
5. Retrieve the comment via **GET** to confirm persistence.

ASP.NET Core Web API .NET 8 2024 - 16. Comment DELETE

Setting Up the HTTP DELETE Endpoint

- Add the **DELETE** action method **below** the existing PUT method in the controller.
- Use the [HttpDelete("{id}")] attribute to bind the route parameter.

```
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(string id)
{
    // implementation will go here
}
```

- The method receives the **ID** of the record to delete.

Repository Interface & Implementation

Interface definition

```
public interface ICommentRepository
{
    Task<Comment?> DeleteAsync(string id);
    // other members...
}
```

- DeleteAsync returns the deleted **Comment** object or null if not found.

Implementation steps

1. Retrieve the comment by ID.
2. Verify it exists; return null if it doesn't.
3. Call Remove on the DbContext.
4. Persist changes with SaveChangesAsync.
5. Return the removed comment.

```
public async Task<Comment?> DeleteAsync(string id)
{
    var comment = await _context.Comments
                        .FirstOrDefaultAsync(c => c.Id == id);
```

```
    if (comment == null) return null;   // not found

    _context.Comments.Remove(comment);   // **Remove** is NOT async
    await _context.SaveChangesAsync();  // persist deletion

    return comment;
}
```

Note: Remove does **not** have an async overload; attempting to await it will cause a compile-time error.

Delete Logic Flow in the Controller

1. Call the repository's DeleteAsync with the supplied ID.
2. If the result is null, respond with **404 NotFound**.
3. Otherwise, respond with **200 OK** (or NoContent if you prefer).

```
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(string id)
{
    var deletedComment = await _commentRepository.DeleteAsync(id);

    if (deletedComment == null)
        return NotFound($"Comment with ID {id} does not exist."); // 404

    return Ok(deletedComment); // 200, optionally return the deleted object
}
```

Persisting Changes – SaveChangesAsync

- After calling Remove, you must **save** the changes for the SQL DELETE statement to be sent to the database.
- await _context.SaveChangesAsync(); ensures the operation is committed.

Response Summary Table

Situation	HTTP Status	Response Body
Record not found	404 NotFound	"Comment with ID {id} does not exist."
Successful deletion	200 OK	Deleted comment object (optional)
Alternative (no content)	204 NoContent	*empty*
Unexpected server error	500 Internal Server Error	Error details (handled by middleware)

Testing the DELETE Endpoint

1. **Run** the application with dotnet watch run (cold restart recommended).
2. Use a tool like **Postman**, **cURL**, or **Swagger UI** to send a DELETE request:
3. curl -X DELETE https://localhost:5001/api/comments/2
4. Verify:
    - The response status matches expectations (404 if ID = 2 no longer exists, 200 if it was removed).
    - A subsequent GET request no longer returns the deleted comment.

Key Points & Common Pitfalls

- **Always** check for the entity's existence before attempting to delete. Deleting a non-existent record will cause a null reference issue.
- Remove **cannot** be awaited; only SaveChangesAsync is asynchronous.
- Returning the deleted entity is optional; some APIs prefer NoContent (204).
- Ensure the route template ("{id}") matches the parameter name in the action method.

ASP.NET Core Web API .NET 8 2024 - 17. Data Validation

Data Validation Overview

Data validation ensures that data submitted to an API conforms to the expected format and type before it is processed.

- Common entry points: **URL parameters** and **JSON payloads**.
- Validation can be performed at the **routing level** (URL constraints) or **model level** (DTO annotations).
- Proper validation prevents malformed requests and returns clear error responses (e.g., 404 for bad routes, 400 for invalid models).

URL (Route) Constraints

Route constraints enforce type rules directly on URL segments.

- The most frequent constraint is int, which forces a route segment to be an integer.
- If a non-integer value is supplied, the framework returns a **404 Not Found** before the controller action runs.

Example route with int constraint

```
// GET /api/comments/{id}
[HttpGet("{id:int}")]
public IActionResult GetComment(int id) { ... }
```

Testing the constraint

1. Run the application (dotnet watch run).
2. Navigate to a valid URL, e.g., https://localhost:5001/api/comments/5.
3. Replace the numeric ID with a string, e.g., .../api/comments/abc.
4. Observe the 404 response, confirming the constraint works.

DTO Data Annotations

Data annotations are attributes applied to DTO properties to describe validation rules.

Annotation	Purpose	Example Usage
[Required]	Property must be supplied	[Required] public string Title { get; set; }
[MinLength(int)]	Minimum number of characters	[MinLength(5, ErrorMessage = "Title must be at least 5 characters")]
[MaxLength(int)]	Maximum number of characters	[MaxLength(280, ErrorMessage = "Title cannot exceed 280 characters")]
[Range(double min, double max)]	Numeric range (inclusive)	[Range(0.001, 100, ErrorMessage = "Dividend must be between 0.001 and 100")]

Key practice:

- **Do not** place annotations on the domain **model**; use a separate **DTO** to keep validation scoped to API contracts.

Example DTOs with Annotations

CreateCommentRequest DTO

```
using System.ComponentModel.DataAnnotations;

public class CreateCommentRequest
{
    [Required]
    [MinLength(5, ErrorMessage = "Title must be at least 5 characters")]
    [MaxLength(280, ErrorMessage = "Title cannot exceed 280 characters")]
    public string Title { get; set; }

    [Required]
    [MinLength(5, ErrorMessage = "Content must be at least 5 characters")]
    [MaxLength(280, ErrorMessage = "Content cannot exceed 280 characters")]
    public string Content { get; set; }
}
```

UpdateCommentRequest DTO

```
public class UpdateCommentRequest : CreateCommentRequest { }
```

CreateStockRequest DTO

```
public class CreateStockRequest
{
    [Required]
    [MaxLength(10, ErrorMessage = "Symbol cannot exceed 10 characters")]
    public string Symbol { get; set; }

    [Required]
    [MaxLength(10, ErrorMessage = "Company name cannot exceed 10 characters")]
    public string CompanyName { get; set; }
```

```
    [Required]
    [Range(1, 1_000_000_000_000, ErrorMessage = "Purchase price must be between 1 and 1 trillion")]
    public decimal PurchasePrice { get; set; }
    [Range(0.001, 100, ErrorMessage = "Dividend must be between 0.001 and 100")]
    public decimal Dividend { get; set; }
    [MaxLength(10, ErrorMessage = "Industry cannot exceed 10 characters")]
    public string Industry { get; set; }
    [Range(1, 5_000_000_000_000, ErrorMessage = "Market cap must be between 1 and 5 trillion")]
    public decimal MarketCap { get; set; }
}
```

UpdateStockRequest DTO

```
public class UpdateStockRequest : CreateStockRequest { }
```

Implementing Validation in Controllers

ModelState is provided by ControllerBase and aggregates validation results.

1. Add the DTO as a parameter to the action (automatically bound from the request body).
2. Check ModelState.IsValid.
3. If invalid, return BadRequest(ModelState); otherwise, proceed with business logic.

Controller snippet

```
[ApiController]
[Route("api/[controller]")]
public class CommentsController : ControllerBase
{
    [HttpPost]
    public IActionResult Create([FromBody] CreateCommentRequest request)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        // Process the valid request...
        return Ok();
    }
}
```

- The same pattern works for **PUT**, **PATCH**, and other actions that accept DTOs.
- **GET** actions typically do not need ModelState because route constraints already enforce type safety.

Summary of Validation Steps

1. **Define route constraints** ({id:int}) for simple URL parameters.
2. **Create DTOs** that mirror the expected JSON shape.
3. **Apply data annotations** ([Required], [MinLength], [MaxLength], [Range]) to DTO properties.
4. **Enable model validation** in controller actions by checking ModelState.IsValid.
5. **Return appropriate error responses** (404 for route violations, 400 for model validation failures). ## 🛠 Adding CRUD Endpoints

- **Copy-paste** the same boiler-plate code for each action (GET by ID, POST, PUT, DELETE) into the controller.
- Keep the code **well-structured**; separate each method visually so it isn't a jumbled block.
- Example skeleton (C# ASP.NET Core):

```
// GET: api/comments/{id}
[HttpGet("{id}")]
public IActionResult GetById(int id)
{
    var comment = _repo.Get(id);
    if (comment == null) return NotFound();
    return Ok(comment);
}

// POST: api/comments
[HttpPost]
public IActionResult Create([FromBody] CommentDto dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);
    var created = _repo.Add(dto);
    return CreatedAtAction(nameof(GetById), new { id = created.Id }, created);
}
```

```
// PUT: api/comments/{id}
[HttpPut("{id}")]
public IActionResult Update(int id, [FromBody] CommentDto dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);
    var updated = _repo.Update(id, dto);
    if (updated == null) return NotFound();
    return NoContent();
}

// DELETE: api/comments/{id}
[HttpDelete("{id}")]
public IActionResult Delete(int id)
{
    var deleted = _repo.Delete(id);
    if (!deleted) return NotFound();
    return NoContent();
}
```

Endpoint Overview

HTTP Method	Route	Purpose
GET	/api/comments/{id}	Retrieve a comment by ID
POST	/api/comments	Create a new comment
PUT	/api/comments/{id}	Update an existing comment
DELETE	/api/comments/{id}	Remove a comment

Testing Endpoints

1. **Run the API** and open your API-testing tool (e.g., Postman, Insomnia).
2. **Test GET**:
    - Request GET /api/comments/1 ⟶ expect the comment "Apple".
3. **Test POST**:
    - Send JSON with title and content.
    - Example payload:
    - {
    - "title": "Apple 2 was the best computer",
    - "content": "My thoughts on the Apple II."
    - }
    - Verify a 201 Created response and the returned object includes a new id.
4. **Test PUT**:
    - Update an existing comment (e.g., ID 5).
    - Payload can change title or content.
    - Expect 204 No Content on success.
5. **Test DELETE**:
    - Delete a comment (e.g., ID 18).
    - Expect 204 No Content.

Tip: After each operation, re-run a GET on the same ID to confirm the change took effect.

Validation Rules

Field	Requirement	Error Message (example)
title	**Required**; non-empty string	"Title field is required."
symbol	**Required** for stock objects	"Symbol field is required."
content	Optional, but if present must be a string	—

- Validation is enforced automatically by the framework's **model state** (ModelState.IsValid).
- When validation fails, the API returns 400 Bad Request with details.

Sample Data Operations

- **Add Apple comment**
    - POST /api/comments with title "Apple".
    - Verify response includes the new ID.
- **Add Apple 2 comment**
    - Title: "Apple 2 was the best computer".
    - Content: personal note.
- **Delete a comment**
    - DELETE /api/comments/4 (or whichever ID was used).
    - Confirm "Comment does not exist" if the ID is wrong.
- **Add Tesla stock entry**
    - POST /api/stocks (assuming a separate stock controller).
    - Payload: { "symbol": "TSLA", "title": "Tesla stock", "content": "..." }.
- **Update Microsoft stock**
    - PUT /api/stocks/22 with bogus data to test validation (e.g., empty symbol).

Future Extensions

- **Complex JSON validation** can be added later without altering existing endpoints—just plug in a custom validator or data-annotation attributes.
- Remember to **update the Swagger/OpenAPI spec** whenever new validation rules are introduced, so consumers see the latest contract.

Definition: *Model validation* ensures incoming request bodies meet defined schema constraints before business logic executes, reducing runtime errors and improving API reliability.

ASP.NET Core Web API .NET 8 2024 - 18. Filtering

Importance of ToList and IQueryable

ToList – Executes the generated SQL query and materializes the results as a list in memory.

IQueryable – Defers execution of the SQL query, allowing additional query operators (e.g., Where, Take) to be appended before the database call.

Feature	ToList	IQueryable
Execution timing	Immediate (eager)	Deferred (lazy)
Allows further filtering after call?	No	Yes
Typical use case	Return final result set	Build up query step-by-step

Creating a Query Object for Filtering

1. Add a new folder **Helpers**.
2. Create a class **QueryObject** (or an interface) with two optional string properties:

```
// Helpers/QueryObject.cs
public class QueryObject
{
    public string? Symbol { get; set; }         // e.g., "MSFT"
    public string? CompanyName { get; set; }    // e.g., "Microsoft"
}
```

- The properties are nullable (string?) so they can be omitted when a client does not filter on that field.

Updating the Repository Interface

Modify the stock repository contract to accept the new query object:

```
// Interfaces/IStockRepository.cs
Task<List<Stock>> GetAllAsync(QueryObject query);
```

Implementing Filtering in the Repository

1. Convert the DbSet<Stock> to an IQueryable<Stock>.
2. Apply conditional Where clauses based on the presence of query parameters.
3. Materialize the result with ToListAsync() only after all filters are applied.

```
// Repositories/StockRepository.cs
public async Task<List<Stock>> GetAllAsync(QueryObject query)
{
    IQueryable<Stock> stocks = _context.Stocks.AsQueryable();
```

```
    if (!string.IsNullOrWhiteSpace(query.CompanyName))
    {
        stocks = stocks.Where(s => s.CompanyName.Contains(query.CompanyName));
    }

    if (!string.IsNullOrWhiteSpace(query.Symbol))
    {
        stocks = stocks.Where(s => s.Symbol.Contains(query.Symbol));
    }

    return await stocks.ToListAsync();
}
```

- AsQueryable() enables deferred execution.
- Each Where clause refines the SQL that will eventually be sent to the database.

Adjusting the Controller to Accept Query Parameters

```
// Controllers/StockController.cs
[HttpGet]
public async Task<ActionResult<List<Stock>>> GetAllAsync([FromQuery] QueryObject query)
{
    var stocks = await _stockRepository.GetAllAsync(query);
    return Ok(stocks);
}
```

- [FromQuery] tells ASP.NET Core to bind URL query parameters (e.g., ?symbol=MSFT&companyName=Microsoft) to the QueryObject.

Verifying the API with Swagger

1. Run the application (dotnet watch run).
2. Open Swagger UI.
3. Test without parameters ⟶ returns the full list.
4. Test with companyName=Microsoft ⟶ returns only Microsoft records.
5. Test with symbol=TSLA ⟶ returns only Tesla records.

The API now supports **search-by-name** and **search-by-symbol** through a single, extensible query object.

Comparison Summary

Aspect	Before (no query object)	After (with IQueryable & QueryObject)
Filtering capability	None (full list only)	Name & Symbol filters via URL params
SQL generation timing	Immediate (ToList)	Deferred until ToListAsync() call
Extensibility	Hard – need new method signatures	Easy – add more properties to QueryObject
Performance impact	Potentially large data transfer	Smaller result sets, less network I/O

ASP.NET Core Web API .NET 8 2024 - 19. Sorting

Setting Up the Query Object

- Create a **query object** to hold sorting information.
- Include a **string** property named SortBy – identifies the column to sort.
- Include a **bool** property IsDescending – indicates direction (default false).

```
public class Query
{
    public string? SortBy { get; set; } = null;   // column name, e.g., "Symbol"
    public bool IsDescending { get; set; } = false; // false ⟶ ASC, true ⟶ DESC
}
```

Note: Keeping sorting parameters together makes the API signature cleaner and easier to extend.

Repository Sorting Logic

1. **Validate input** – ignore null, empty, or whitespace SortBy values.
2. **Build the ORDER BY clause** only after the main query is assembled (so the generated SQL is sent correctly).
3. Use **nested if statements** to handle the presence of a sort column and its direction.

```
public IQueryable<Stock> GetStocks(Query query)
{
    var result = _context.Stocks.AsQueryable();

    //Guard clause for missing sort column
    if (!string.IsNullOrWhiteSpace(query.SortBy))
    {
        //Determine which column to sort by
        if (query.SortBy.Equals("Symbol", StringComparison.OrdinalIgnoreCase))
        {
            //Apply direction using ternary operator
            result = query.IsDescending
                ? result.OrderByDescending(s => s.Symbol)
                : result.OrderBy(s => s.Symbol);
        }
        // Add more columns here with additional if blocks
```

```
    }

    return result;
}
```

Handling Sort Direction

Property	Meaning	Default
IsDescending	true ⟶ ORDER BY ... DESC false ⟶ ORDER BY ... ASC	false

- The ternary operator ? : decides between OrderBy and OrderByDescending.
- Example of the ternary expression:

```
result = query.IsDescending
    ? result.OrderByDescending(s => s.Symbol)
    : result.OrderBy(s => s.Symbol);
```

String Comparison Options

Comparison Type	Description	When to Use
StringComparison.OrdinalIgnoreCase	Fast, case-insensitive, culture-independent. Works for standard English/Latin characters.	Most typical scenarios.
StringComparison.Ordinal	Case-sensitive, culture-independent.	When exact case matters.
StringComparison.CurrentCultureIgnoreCase	Case-insensitive, respects the current culture.	When cultural sorting rules are required.
StringComparison.InvariantCultureIgnoreCase	Case-insensitive, invariant culture (stable across locales).	When you need consistent behavior across deployments.

Tip: Use OrdinalIgnoreCase unless you have special Unicode or locale requirements. Exotic Unicode symbols may need a different comparison mode.

Testing the Sorting Feature with Swagger

1. **Restart** the application (cold restart) to ensure the latest code is loaded.
2. Open **Swagger UI** and locate the endpoint for retrieving stocks.
3. Fill the query parameters:
    - sortBy ⟶ "Symbol" (or another column).
    - isDescending ⟶ true for descending, false for ascending.
4. Execute the request.

5. Verify the returned list order:
    - true ⟶ e.g., Tesla, Microsoft, Amazon (DESC).
    - false ⟶ e.g., Amazon, Microsoft, Tesla (ASC).

GET /api/stocks?sortBy=Symbol&isDescending=true

The response should reflect the selected sort direction, confirming that the repository logic and query object are correctly integrated.

ASP.NET Core Web API .NET 8 2024 - 20. Pagination

Overview of Pagination

Pagination is a technique that limits the number of records returned by a query, splitting the full result set into manageable **pages**. This prevents performance issues (e.g., crashes) when dealing with large datasets such as 10,000 records.

- Returns only a subset of records per request.
- Allows clients to request specific pages by providing a **page number** and **page size**.

Skip and Take Mechanics

Skip and **Take** are query operators that together enable pagination.

- **Skip(n)** – bypasses the first *n* elements of a sequence.
- **Take(m)** – selects the next *m* elements after the skip.

When combined, they retrieve exactly the records for the desired page:

skipCount = (pageNumber - 1) * pageSize

result = source.Skip(skipCount).Take(pageSize)

Implementing Pagination in Code

Step-by-step implementation (C# / Entity Framework):

1. **Add pagination parameters** to the request DTO.
2. **Set default values** (page number = 1, page size = 20).
3. **Calculate the skip amount** inside the repository.
4. **Apply Skip and Take** to the query before execution.

```
//DTO with pagination fields
public class StockQueryDto
{
   public int PageNumber { get; set; } = 1;   // default = first page
   public int PageSize   { get; set; } = 20;  // default = 20 records per page
   // other filter properties (e.g., Symbol, SortBy) omitted for brevity
}
```

```
//Repository method applying pagination
public async Task<List<Stock>> GetAllAsync(StockQueryDto query)
{
    int skip = (query.PageNumber - 1) * query.PageSize;   // calculate skip
    return await _context.Stocks
        .Skip(skip)              // skip previous pages
        .Take(query.PageSize)    // take current page size
        .ToListAsync();
}
```

Parameter Defaults Table

Parameter	Default Value	Description
PageNumber	1	The first page (no skip applied).
PageSize	20	Number of records returned per page.

Testing Pagination with Swagger

- Open Swagger UI and locate the **GET /stocks** endpoint.
- Provide values for **pageNumber** and **pageSize** to verify results.
- Example tests:
    1. **No parameters** → returns all records (useful for sanity check).
    2. **pageSize = 3**, **pageNumber = 1** → returns records 1-3.
    3. **pageSize = 3**, **pageNumber = 2** → returns records 4-6, etc.

These tests confirm that the Skip and Take logic correctly slices the dataset.

Key Points

- Always subtract **1** from PageNumber when calculating skip to avoid an off-by-one error.
- Keep PageSize reasonable (e.g., 20) to balance data transfer size and UI usability.
- Pagination logic lives in the **repository** layer, keeping controllers clean.
- Swagger is a quick way to manually validate pagination behavior during development.

ASP.NET Core Web API Identity JWT 2024 - 21. Install Identity

JWT Basics

JWT (JSON Web Token) is a compact, URL-safe string used to transmit claims between a client and a server.

- Consists of three Base64URL-encoded parts separated by dots (.): **Header**, **Payload**, **Signature**.
- The **Header** specifies the signing algorithm (e.g., HS256).
- The **Payload** carries claims such as sub (subject), exp (expiration), iss (issuer), etc.
- The **Signature** is created by hashing the header and payload with a secret key, ensuring integrity.

Required NuGet Packages

Package	Purpose
Microsoft.Extensions.Identity.Core	Provides ASP.NET Core Identity services and base IdentityUser class.
Microsoft.EntityFrameworkCore (matching .NET version)	ORM for database access.
Microsoft.AspNetCore.Authentication.JwtBearer	Adds JWT bearer token handling middleware.

Installation (inside VS Code terminal):

```
dotnet add package Microsoft.Extensions.Identity.Core
dotnet add package Microsoft.EntityFrameworkCore --version 8.*
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Defining the Application User Model

Create a custom user class that inherits from IdentityUser. This gives you all default Identity fields (email, password hash, etc.) and lets you add custom properties.

```
// File: Models/AppUser.cs
using Microsoft.AspNetCore.Identity;

public class AppUser : IdentityUser
{
    // Example of a custom field
    public int RiskLevel { get; set; }
}
```

- **Key point:** No need to manually add password or email fields; they are already in IdentityUser.

Configuring the DbContext for Identity

Replace the plain DbContext with IdentityDbContext<AppUser> so Identity can manage its own tables.

```
// File: Data/ApplicationDbContext.cs
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : IdentityDbContext<AppUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options) { }

    // Your other DbSet<T> properties (e.g., Stocks, Comments) go here
}
```

- Identity will automatically create tables like AspNetUsers, AspNetRoles, etc.

Registering Services in Program.cs

1. Add Identity and EF Core Store

```
builder.Services.AddIdentity<AppUser, IdentityRole>(options =>
{
    // Password policy (see next section)
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```

2. Configure Password Requirements

```
options.Password.RequireDigit = true;
options.Password.RequireLowercase = true;
options.Password.RequireUppercase = true;
options.Password.RequireNonAlphanumeric = true;
options.Password.RequiredLength = 12;
```

3. Set Up JWT Authentication Scheme

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme    = JwtBearerDefaults.AuthenticationScheme;
```

```
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidIssuer    = builder.Configuration["Jwt:Issuer"],

        ValidateAudience = true,
        ValidAudience    = builder.Configuration["Jwt:Audience"],

        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(builder.Configuration["Jwt:SigningKey"]))
    };
});
```

- **Default scheme** is set to JwtBearerDefaults.AuthenticationScheme so every request is automatically checked for a bearer token.

4. Add Authorization (optional)

```
builder.Services.AddAuthorization();
```

AppSettings for JWT

Add a Jwt section under the root of appsettings.json.

```
{
  "AllowedHosts": "*",
  "Jwt": {
    "Issuer": "YourAppIssuer",
    "Audience": "YourAppAudience",
    "SigningKey": "SuperSecretKeyThatShouldBeLongEnough"
  }
}
```

- **Issuer** and **Audience** are identifiers used during validation; they become crucial in production environments.

- **SigningKey** must be a strong secret (minimum 256-bit for HS256).

End-to-End Flow (summary)

1. **User registers / logs in** with email & password.
2. Identity validates credentials and **issues a JWT** containing user claims.
3. Client stores the token (e.g., in local storage) and sends it in the Authorization: Bearer <token> header on each request.
4. The JWT middleware validates the token using the parameters defined above.
5. If valid, the request proceeds; otherwise a 401 Unauthorized response is returned.

Quick Checklist

- [ ] Install Identity.Core, EntityFrameworkCore, JwtBearer.
- [ ] Create AppUser inheriting from IdentityUser.
- [ ] Switch DbContext to IdentityDbContext<AppUser>.
- [ ] Register Identity with EF store in Program.cs.
- [ ] Set strong password policies.
- [ ] Configure JWT authentication scheme and validation parameters.
- [ ] Add Jwt section (Issuer, Audience, SigningKey) to appsettings.json.

These steps give you a secure, email-based login flow backed by JWTs in a .NET 8 application.## JWT Configuration

- **Issuer** – the server that creates the token.
- **Audience** – the client(s) that will consume the token.

Important: The **signing key** must remain secret. If exposed, attackers can forge valid tokens.

Key points

- Use a strong, unpredictable signing key (e.g., a generated secret, not a placeholder like "swordfish").
- Store the signing key in *appsettings.json* and reference it via configuration.

```
// Startup.cs – Configure JWT authentication
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme    = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
```

```
    {
        ValidateIssuer          = true,
        ValidIssuer             = builder.Configuration["Jwt:Issuer"],

        ValidateAudience        = true,
        ValidAudience           = builder.Configuration["Jwt:Audience"],

        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(builder.Configuration["Jwt:SigningKey"]))
    };
});
```

Configuration Summary

Setting	Description	Example Value
Jwt:Issuer	Identifier of the token-issuing server	https://localhost:5246
Jwt:Audience	Intended recipient(s) of the token	MyAppClient
Jwt:SigningKey	Secret used to sign the JWT (must be strong)	A_Very_Secure_Secret!23!

Adding Authentication & Authorization Middleware

1. In the HTTP request pipeline, enable authentication.
2. Follow with authorization to enforce policies.

```
app.UseAuthentication();   // Validate JWT on incoming requests
app.UseAuthorization();    // Apply role/claim checks
```

EF Core Migrations for Identity

Steps to create and apply migrations

1. **Add a migration** for the Identity schema.
2. dotnet ef migrations add Identity
3. **Update the database** with the new schema.
4. dotnet ef database update

Resulting tables in SQL Server

Table	Purpose
AspNetUsers	Stores user accounts

AspNetRoles	Defines roles (e.g., Admin, User)
AspNetUserRoles	Links users to their roles
AspNetUserLogins	External login providers (Google, etc.)
AspNetUserClaims	Custom claims attached to users
AspNetUserTokens	Refresh/Access tokens for users

Ready to Test

- After migrations, verify tables exist via **SQL Server Management Studio**.
- Proceed to implement **login** and **register** endpoints that issue JWTs using the configured settings.

Tip: Always keep the signing key out of source control and rotate it periodically for enhanced security.

ASP.NET Core Web API Identity JWT 2024 - 22. Register

User Registration Workflow

- **UserManager** handles most of the heavy lifting for creating users.
- The method CreateAsync receives a user object and a plain-text password, then:
    1. **Hashes & salts** the password.
    2. Stores the hashed password in the **database**.
    3. Returns an IdentityResult indicating success or failure.

Definition: *Hashing* is a one-way function that converts an input (e.g., a password) into a fixed-size string of characters. The same input always produces the same hash, but any change (even a single character) yields a completely different hash.

Why Passwords Are Hashed

- Prevents exposure of **clear-text passwords** if the database is compromised.
- Even with powerful hardware, reversing a secure hash is computationally infeasible.
- Hashes are **deterministic**: the same password → same hash; a different password → different hash.

Property	Reason
One-way function	Attackers cannot derive the original password from the hash.
Salted	Adds random data to each password, preventing pre-computed attacks.
Deterministic per user	Allows the system to verify a password by re-hashing the input.

Setting Up the Account Controller

1. **Create a new controller file** (e.g., AccountController.cs).
2. Import the base controller and **inject** UserManager<AppUser> as a **private readonly** field.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;

[ApiController]
[Route("api/account")]
public class AccountController : ControllerBase
{
    private readonly UserManager<AppUser> _userManager;

    public AccountController(UserManager<AppUser> userManager)
```

```
    {
        _userManager = userManager;
    }
}
```

3. Add a **POST** endpoint for registration.

```
[HttpPost("register")]
public async Task<IActionResult> Register([FromBody] RegisterDto dto)
{
    // Implementation goes here
}
```

Register DTO (Data Transfer Object)

The DTO enforces **strict validation** for incoming registration data.

```
public class RegisterDto
{
    [Required]                      // Username must be supplied
    public string Username { get; set; }

    [Required, EmailAddress]        // Built-in email validation
    public string Email { get; set; }

    [Required]                      // Password must be supplied
    public string Password { get; set; }
}
```

- Validation attributes ([Required], [EmailAddress]) automatically produce **model-state errors** if the payload is invalid.

Registration Logic Inside the Controller

1. **Validate model state** – catches DTO-level errors.
2. Build an AppUser instance from the DTO.
3. Call UserManager.CreateAsync(user, password) inside a **try-catch** block to handle Identity-specific exceptions.

```
[HttpPost("register")]
public async Task<IActionResult> Register([FromBody] RegisterDto dto)
{
```

```
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    try
    {
        var appUser = new AppUser
        {
            UserName = dto.Username,
            Email = dto.Email
        };

        var result = await _userManager.CreateAsync(appUser, dto.Password);

        if (result.Succeeded)
            return Ok(new { Message = "User created successfully." });

        // Return all identity errors to the client
        return BadRequest(result.Errors);
    }
    catch (Exception ex)
    {
        // Log the exception as needed
        return StatusCode(500, new { Error = ex.Message });
    }
}
```

- The **IdentityResult** contains a collection of IdentityError objects that explain why creation failed (e.g., password complexity, duplicate email).

Seeding Default Roles

Before any user can be assigned a role, **at least one role must exist** in the database.

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
```

```
    var roles = new List<IdentityRole>
    {
        new IdentityRole
        {
            Name = "Admin",
            NormalizedName = "ADMIN"
        },
        new IdentityRole
        {
            Name = "User",
            NormalizedName = "USER"
        }
    };

    builder.Entity<IdentityRole>().HasData(roles);
}
```

- NormalizedName is the uppercase version required by ASP.NET Core Identity.
- After seeding, you can assign roles to users via UserManager.AddToRoleAsync.

Common Errors & How to Handle Them

Situation	Typical Cause	Mitigation
IdentityResult not succeeded	Password doesn't meet complexity rules, duplicate email/username	Ensure DTO validation and configure password policy if needed
Unhandled exception during CreateAsync	Missing required services, null UserManager	Wrap call in try-catch, verify DI registration
Role not found when assigning	Roles not seeded before user creation	Seed roles in OnModelCreating (see above)

Summary of Key Concepts

- **UserManager** → central API for creating, updating, and managing identity users.
- **CreateAsync** → automatically hashes passwords and stores them securely.
- **DTO validation** → prevents malformed data from reaching the service layer.

- **Role seeding** → required prerequisite for role-based authorization.
- **Error handling** → always validate ModelState and catch IdentityResult errors. ## 🚀 User Creation Flow
- **CreateUserAsync** returns a **result object** that indicates whether the operation succeeded.
- The result includes properties such as Succeeded, Errors, and other metadata used for further logic.

```
var createResult = await _userManager.CreateAsync(user, password);
if (!createResult.Succeeded)
{
   // Handle creation failure
}
```

Role Assignment

- After a user registers, assign the **default role** (e.g., User) via AddToRoleAsync.
- **Admin role** should **never** be granted through a public register endpoint.
    - Assign admin roles manually or through a secure, restricted endpoint.

```
var roleResult = await _userManager.AddToRoleAsync(user, "User");
if (!roleResult.Succeeded)
{
   // Handle role-assignment failure
}
```

Definition: *Assigning a role* attaches a claim to the user's identity, enabling role-based authorization throughout the application.

Error Handling

Scenario	HTTP Status Code	Typical Reason
Creation fails	500 (Internal Server Error)	Unexpected server issue
Bad request (e.g., validation)	400 (Bad Request)	Client supplied invalid data
Role assignment fails	500 or 400 (depending on cause)	Database or logic error

- Return the appropriate status code together with the error details from the Identity result.

```
if (!createResult.Succeeded)
   return StatusCode(500, new { Errors = createResult.Errors });

if (!roleResult.Succeeded)
   return StatusCode(500, new { Errors = roleResult.Errors });
```

Identity UserManager Return Object

Key Point: Every call to UserManager (CreateAsync, AddToRoleAsync, etc.) returns an **IdentityResult** object containing:

- Succeeded (bool) – overall success flag
- Errors (ICollection<IdentityError>) – collection of error messages
- Additional helper properties (e.g., IsSuccess, FailedCount) that can be used for detailed logging.

Database Migration & Seeding

1. **Create a migration** to seed roles.
2. dotnet ef migrations add SeedRoles # name can be anything descriptive
3. **Apply the migration** to update the database.
4. dotnet ef database update
5. In the generated migration file, insert code to ensure the **User** and **Admin** roles exist:
6. protected override void Up(MigrationBuilder migrationBuilder)
7. {
8. migrationBuilder.InsertData(
9. table: "AspNetRoles",
10. columns: new[] { "Id", "Name", "NormalizedName" },
11. values: new object[,]
12. {
13. { Guid.NewGuid().ToString(), "User", "USER" },
14. { Guid.NewGuid().ToString(), "Admin", "ADMIN" }
15. });
16. }

Verifying Roles & Testing

- **Check role seeding** by querying the AspNetRoles table; you should see rows for **User** and **Admin**.
- Use **Swagger** (or any API client) to hit the register endpoint:
    - Provide a unique username (e.g., investor444) and a strong password.
    - Expect a response indicating **user created** if everything is seeded correctly.

dotnet watch run # launches the API with hot-reload

Database Verification

- Open your database console (e.g., SSMS, Azure Data Studio) and run:
- SELECT TOP 1000 *
- FROM AspNetUsers;

- Verify that the newly registered user (investor444@gmail.com) appears with the correct hashed password and assigned role.

Quick Recap of Critical Steps

- **Create** user → check IdentityResult.
- **Assign** default role → never expose admin role via public endpoint.
- **Handle** errors with proper HTTP codes.
- **Seed** roles via EF migration before testing.
- **Validate** through Swagger and direct DB queries.

ASP.NET Core Web API Identity JWT 2024 - 23. Token Service

Claims vs. Roles

Claim – a piece of information about a user (e.g., email, username, time-zone) stored directly in the JWT.

Role – a traditional permission grouping that typically requires a database lookup each time it is needed.

Aspect	Claims	Roles
Storage	Embedded in the JWT (no DB hit)	Stored in DB, fetched on each request
Flexibility	Arbitrary key-value pairs	Fixed set of predefined groups
Performance	Fast – token already carries data	Slower – extra DB round-trip
Use case	Fine-grained user attributes, dynamic permissions	Simple, static permission groups

Defining the JWT Service Interface

```
// ITokenService.cs
public interface ITokenService
{
    string CreateToken(AppUser user);
}
```

- **AppUser** – your domain model that contains the data needed for claims (e.g., Email, UserName, IsPayingUser, TimeZone).

Implementing the JWT Service

Constructor Dependencies

```
public class TokenService : ITokenService
{
    private readonly IConfiguration _config;
    private readonly SymmetricSecurityKey _key;

    public TokenService(IConfiguration config)
    {
        _config = config;

        // Convert the secret key string from appsettings.json to bytes
        _key = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
```

}

- **IConfiguration** – provides access to *appsettings.json* where the secret key lives.
- **SymmetricSecurityKey** – the cryptographic key used to sign the token.

Building Claims

```
var claims = new List<Claim>
{
    new Claim(JwtRegisteredClaimNames.Email, user.Email),
    new Claim(JwtRegisteredClaimNames.Sub,   user.UserName), // "sub" = subject
    new Claim("isPayingUser", user.IsPayingUser.ToString()),
    new Claim("timeZone", user.TimeZone)
};
```

- Claims are just **key-value pairs**; the JWT standard defines many names (e.g., email, sub).
- Custom claim types (e.g., isPayingUser) can be added as needed.

Creating Signing Credentials

```
var creds = new SigningCredentials(
    _key, SecurityAlgorithms.HmacSha512Signature);
```

- **SigningCredentials** tell .NET how to encrypt the token (HMAC-SHA-512 in this case).**Defining the Token Descriptor**

```
var tokenDescriptor = new SecurityTokenDescriptor
{
    Subject = new ClaimsIdentity(claims),
    Expires = DateTime.UtcNow.AddHours(2),   // token lifetime
    SigningCredentials = creds,
    Issuer = _config["Jwt:Issuer"],
    Audience = _config["Jwt:Audience"]
};
```

- **Expires** – limits token lifetime to mitigate theft risk.
- **Issuer / Audience** – optional validation parameters.

Generating the JWT

```
var tokenHandler = new JwtSecurityTokenHandler();
var securityToken = tokenHandler.CreateToken(tokenDescriptor);
return tokenHandler.WriteToken(securityToken);
```

- CreateToken builds the token object; WriteToken serializes it to the compact JWT string. **appsettings.json – Storing Secrets Safely**

```
{
  "Jwt": {
    "Key": "YOUR_SUPER_SECRET_256_BIT_KEY",
    "Issuer": "YourApp",
    "Audience": "YourAppClients"
  }
}
```

- **Never** commit the actual secret key to source control. Use environment variables or secret managers in production.

Accessing Claims at Runtime

```
// Inside any controller or middleware
var email = HttpContext.User.FindFirst(ClaimTypes.Email)?.Value;
var userName = HttpContext.User.FindFirst(JwtRegisteredClaimNames.Sub)?.Value;
var isPaying = HttpContext.User.FindFirst("isPayingUser")?.Value;
var timeZone = HttpContext.User.FindFirst("timeZone")?.Value;
```

- The **ClaimsPrincipal** (often accessed via HttpContext.User) acts like a wallet that holds all claim values extracted from the JWT after it is validated.

Summary Checklist

1. Add **IConfiguration** to the service constructor.
2. Create a **SymmetricSecurityKey** from the secret in *appsettings.json*.
3. Assemble a **List<Claim>** with both standard and custom claim types.
4. Build **SigningCredentials** using HMAC-SHA-512.
5. Populate a **SecurityTokenDescriptor** (subject, expiry, issuer, audience, signing credentials).
6. Use **JwtSecurityTokenHandler** to create and serialize the token.
7. Retrieve claims from HttpContext.User wherever needed in the app. ## JWT Overview

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties.

It consists of three Base64-URL encoded parts: **Header**, **Payload**, and **Signature**.

- **Header** – specifies the signing algorithm (e.g., HS256).
- **Payload** – contains registered claims such as iss (issuer), aud (audience), exp (expiration), plus custom claims.

- **Signature** – created by hashing the encoded header and payload with a **symmetric security key** (or an asymmetric key).

Token Creation Process

1. **Define the token descriptor** – includes issuer, audience, signing credentials, and claims.
2. **Instantiate JwtSecurityTokenHandler.**
3. **Create the token** with CreateToken(tokenDescriptor).
4. **Serialize to string** using WriteToken(token).

```
using System.IdentityModel.Tokens.Jwt;
using Microsoft.IdentityModel.Tokens;

//Token descriptor
var tokenDescriptor = new SecurityTokenDescriptor
{
   Issuer = "YourIssuer",
   Audience = "YourAudience",
   Subject = new ClaimsIdentity(new[]
   {
      new Claim(ClaimTypes.Name, userName),
      new Claim(ClaimTypes.Email, userEmail)
   }),
   Expires = DateTime.UtcNow.AddHours(1),
   SigningCredentials = new SigningCredentials(
      new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey)),
      SecurityAlgorithms.HmacSha256Signature)
};

//Handler
var tokenHandler = new JwtSecurityTokenHandler();

//Create token object
SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);
```

```
//Convert to string
string jwt = tokenHandler.WriteToken(token);
```

Symmetric Security Key Requirements

The **symmetric security key** is a secret known only to the server. It encrypts the JWT signature and guarantees integrity.

- Must be sufficiently long (recommended ≥ 512 bits ≈ 64 bytes).
- Short keys trigger an error like *"key size is not long enough"*.
- Example of a strong key (64-character random string):

```
string secretKey = "a7f9d3c2e5b1f6a8d9e0c3b4a1d2e7f8g9h0i1j2k3l4m5n6o7p8q9r0s1t2u3v4";
```

Dependency Injection Setup (Program.cs)

Step	Code	Purpose
1	builder.Services.AddScoped<ITokenService, TokenService>();	Register the token service for DI.
2	builder.Services.AddAuthentication(...);	(Typically) configure JWT bearer authentication.

```
var builder = WebApplication.CreateBuilder(args);

// Register custom token service
builder.Services.AddScoped<ITokenService, TokenService>();

var app = builder.Build();
app.Run();
```

DTOs for Register Response

Creating a **Data Transfer Object** (DTO) lets the API return user details together with the JWT.

```
public class RegisterResponseDto
{
    public string UserName { get; set; }   // e.g., "investor2222"
    public string Email    { get; set; }   // e.g., "investor2222@gmail.com"
    public string Token    { get; set; }   // JWT string
}
```

Why use a DTO?

- Bundles related data (username, email, token) in a single response.
- Keeps the API contract clean and extensible.
- Improves client-side readability.

Controller Integration

1. **Inject** ITokenService via constructor.
2. **Generate** the DTO after successful registration.
3. **Return** the DTO to the client.

```
[ApiController]
[Route("api/[controller]")]
public class AccountController : ControllerBase
{
    private readonly ITokenService _tokenService;

    public AccountController(ITokenService tokenService)
    {
        _tokenService = tokenService;
    }

    [HttpPost("register")]
    public async Task<ActionResult<RegisterResponseDto>> Register(RegisterDto model)
    {
        // (User creation logic omitted)

        var token = _tokenService.GenerateToken(appUser); // returns JWT string

        var response = new RegisterResponseDto
        {
            UserName = appUser.UserName,
            Email    = appUser.Email,
            Token    = token
        };

        return Ok(response);
    }
}
```

Testing the Generated Token

1. Run the API (dotnet watch run).
2. Register a user (e.g., investor2222, investor2222@gmail.com, password 2222).
3. Copy the returned token.
4. Paste it into an online **JWT decoder** (e.g., jwt.io) to verify payload contents:

Claim	Expected Value
iss	Your configured issuer
aud	Your configured audience
email	investor2222@gmail.com
name	investor2222

If the decoded payload matches, the token creation pipeline is correct.

Common Pitfalls

- **Key too short** – ensure the symmetric key meets the minimum byte length.
- **Missing DI registration** – forgetting AddScoped<ITokenService, TokenService>() results in null injection.
- **Returning the token object** – always use WriteToken to serialize to a string before sending to the client.
- **Incorrect audience/issuer** – mismatch causes validation failures during login.

ASP.NET Core Web API Identity JWT 2024 - 24. Login

Login Endpoint Overview

- **HTTP POST** is used because a new login request creates data (a session token).
- The action method returns Task<IActionResult> and is marked **async**.

```
[HttpPost("login")]
public async Task<IActionResult> Login(LoginDto loginDto)
{
    // implementation ...
}
```

DTO Definition

A **Data Transfer Object (DTO)** is created to receive the username and password.

Data annotations enforce required fields.

```
public class LoginDto
{
    [Required]
    public string Username { get; set; }

    [Required]
    public string Password { get; set; }
}
```

Model Validation

- Before any processing, check ModelState.IsValid.
- If validation fails, return **400 Bad Request** with the model errors.

```
if (!ModelState.IsValid)
    return BadRequest(ModelState);
```

Finding User with UserManager

- UserManager<AppUser> abstracts the data store; no need to query the DbContext directly.
- The user is located by **username** (or email) and normalized to lower-case.

```
var user = await _userManager.FindByNameAsync(loginDto.Username.ToLower());
if (user == null)
    return Unauthorized("Invalid username or password");
```

UserManager – A service that provides CRUD operations and query capabilities for identity users.

Password Verification with SignInManager

- SignInManager<AppUser> handles password checks and sign-in logic.
- CheckPasswordAsync is called with the user, password, and a **lockoutOnFailure** flag (set to false to avoid automatic lockout).

```
var passwordCheck = await _signInManager.CheckPasswordAsync(user, loginDto.Password, lockoutOnFailure: false);
if (!passwordCheck.Succeeded)
    return Unauthorized("Invalid username or password");
```

SignInManager – Manages sign-in operations, including password validation and lockout handling.

Handling Login Results

- On successful authentication, a **JWT token** is generated and returned inside a response DTO.
- The response DTO includes Username, Email, and Token.

```
var response = new UserDto
{
    Username = user.UserName,
    Email    = user.Email,
    Token    = _tokenService.CreateToken(user)
};


return Ok(response);
```

Generating JWT Token

- A custom ITokenService (or similar) creates the JWT based on the authenticated user.
- The token contains claims such as user identifier, role, and expiration.

Component	Purpose
JwtSecurityTokenHandler	Serializes the token to a string.
SigningCredentials	Secures the token with a secret key.
Claims	Stores user data (e.g., sub, email, role).
Expires	Sets token lifetime (e.g., 1 hour).

Running and Testing the API

1. **Start the API**
2. cd YourApiFolder

3. dotnet watch run
4. **Register a test user** (via registration endpoint).
5. **Login** with the new credentials:
6. {
7. "username": "investor111",
8. "password": "Password111!"
9. }
10. **Copy the returned JWT** for use in subsequent requests.

Swagger Configuration for JWT

Add JWT support to Swagger in Program.cs right after builder.Services.AddSwaggerGen().

```
builder.Services.AddSwaggerGen(c =>
{
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Description = "JWT Authorization header using the Bearer scheme. Example: \"Bearer {token}\"",
        Name = "Authorization",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.Http,
        Scheme = "bearer"
    });

    c.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference { Type = ReferenceType.SecurityScheme, Id = "Bearer" }
            },
            Array.Empty<string>()
        }
    });
});
```

- After a **cold restart**, the Swagger UI shows an **Authorize** button.
- Paste the JWT token, then test secured endpoints (e.g., GET /stock) to confirm proper authorization.

Summary of Key Steps

1. Define LoginDto with required fields.
2. Validate the model state.
3. Retrieve the user via UserManager.
4. Verify password with SignInManager.CheckPasswordAsync.
5. Generate and return a JWT token.
6. Configure Swagger to accept the JWT for testing secured routes.

ASP.NET Core Web API .NET 8 2024 - 25. Many-To-Many

Many-to-Many Relationships Overview

Definition: A *many-to-many* relationship allows any number of records in one table to be associated with any number of records in another table.

Example use-cases: favorites, likes, invites, **stock portfolios** where a user can hold many stocks and a stock can belong to many users.

- One-to-many (or many-to-one) limits the parent side to a single record, which prevents the "infinite combinations" needed for portfolios.

Bad Approach: Direct Foreign Keys

Warning: Adding a UserId column to the **Stocks** table **and** a StockId column to the **Users** table creates a *bidirectional one-to-many* mapping.

This design:

- Couples the two entities tightly.
- Causes data anomalies when a stock belongs to multiple users (or vice-versa).
- Is only acceptable for very special cases.

Proper Approach: Join Table (Portfolio)

A **join table** stores pairs of foreign keys, representing the association between two entities without embedding either key in the other entity's table.

Model Definition (C# / EF Core)

```
// Models/Portfolio.cs
public class Portfolio
{
    public int AppUserId { get; set; }      // FK to AppUser
    public int StockId   { get; set; }      // FK to Stock

    // Navigation properties (developer convenience)
    public AppUser AppUser { get; set; }
    public Stock   Stock   { get; set; }
}
// Models/AppUser.cs (partial)
public class AppUser
```

```
{
    public int Id { get; set; }
    public string UserName { get; set; }

    // Collection navigation for many-to-many
    public List<Portfolio> Portfolios { get; set; } = new();
}
// Models/Stock.cs (partial)
public class Stock
{
    public int Id { get; set; }
    public string Symbol { get; set; }

    // Collection navigation for many-to-many
    public List<Portfolio> Portfolios { get; set; } = new();
}
```

Table-Naming Conventions (optional aesthetic step)

```
// Inside Stock.cs
[Table("Stocks")]
public class Stock { ... }

// Inside Portfolio.cs
[Table("Portfolios")]
public class Portfolio { ... }
```

Add DbSet to ApplicationDbContext

```
public class ApplicationDbContext : DbContext
{
    public DbSet<AppUser>   AppUsers   { get; set; }
    public DbSet<Stock>     Stocks     { get; set; }
    public DbSet<Portfolio> Portfolios { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
    {
        // Fluent API configuration (see next section)
    }
}
```

Configuring EF Core with Fluent API (OnModelCreating)

Purpose: Explicitly declare foreign keys and navigation relationships so EF Core generates the correct join table schema.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Configure primary key for the join table (composite key)
    modelBuilder.Entity<Portfolio>()
        .HasKey(p => new { p.AppUserId, p.StockId });

    // Relationship: Portfolio ⟶ AppUser (many-to-one)
    modelBuilder.Entity<Portfolio>()
        .HasOne(p => p.AppUser)
        .WithMany(u => u.Portfolios)
        .HasForeignKey(p => p.AppUserId);

    // Relationship: Portfolio ⟶ Stock (many-to-one)
    modelBuilder.Entity<Portfolio>()
        .HasOne(p => p.Stock)
        .WithMany(s => s.Portfolios)
        .HasForeignKey(p => p.StockId);
}
```

Key points

Entity	Navigation Property	Foreign Key	Collection on Other Side
Portfolio	AppUser	AppUserId	AppUser.Portfolios
Portfolio	Stock	StockId	Stock.Portfolios

- The composite primary key (AppUserId, StockId) guarantees each user-stock pair is unique.
- EF Core will automatically create the **Portfolios** table with the two FK columns.

Refreshing Migrations & Databas

When table names or relationships change, recreate migrations to avoid "invalid object" errors.

1. **Delete old migrations**
2. rm -r Migrations
3. **Drop the existing database** (only safe for solo projects)
    - Open SQL Server Management Studio → delete the database or run:
4. DROP DATABASE YourDatabaseName;
5. **Create a fresh migration**
6. dotnet ef migrations add PortfolioManyToMany
7. **Apply the migration**
8. dotnet ef database update
9. **Verify**
    - Re-open SSMS, connect to the database, and confirm tables: AppUsers, Stocks, Portfolios (with columns AppUserId, StockId).

ASP.NET Core Web API .NET 8 2024 - 26. Portfolio GET

Setting Up the Portfolio Controller

- Add a new **API controller** file (e.g., PortfolioController.cs).
- Decorate the class with [ApiController] and [Route("api/[controller]")].
- Inject the required services through the constructor:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
using YourApp.Data;
using YourApp.Models;
using YourApp.Repositories;

[ApiController]
[Route("api/[controller]")]
public class PortfolioController : ControllerBase
{
    private readonly UserManager<AppUser> _userManager;
    private readonly IStockRepository _stockRepo;
    private readonly IPortfolioRepository _portfolioRepo;

    public PortfolioController(UserManager<AppUser> userManager,
                               IStockRepository stockRepo,
                               IPortfolioRepository portfolioRepo)
    {
        _userManager = userManager;
        _stockRepo   = stockRepo;
        _portfolioRepo = portfolioRepo;
    }

    // ... actions go here ...
}
```

Service	Purpose
UserManager<AppUser>	Manage ASP.NET Identity users (lookup, create, etc.)
IStockRepository	Access stock data (read-only in this scenario)
IPortfolioRepository	Custom queries for a user's portfolio

Handling User Claims

- The current user is available via HttpContext.User.
- To obtain the **username** stored in the JWT claim, call a custom extension method.

```
[HttpGet]
[Authorize]
public async Task<IActionResult> GetUserPortfolio()
{
   var username = User.GetUserName();   // extension defined below
   var appUser  = await _userManager.FindByNameAsync(username);
   var portfolio = await _portfolioRepo.GetUserPortfolioAsync(appUser);
   return Ok(portfolio);
}
```

Definition: *Claims* are key-value pairs embedded in a security token (e.g., JWT) that describe the authenticated user.

Building the Claims Extension

Create a folder Extensions and add ClaimsExtensions.cs:

```
using System.Security.Claims;
namespace YourApp.Extensions
{
   public static class ClaimsExtensions
   {
      private const string UserNameClaim = "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name";
      public static string GetUserName(this ClaimsPrincipal user)
      {
         return user.FindFirst(UserNameClaim)?.Value;
      }
   }
}
```

- The constant holds the exact claim type used when the token was generated.
- GetUserName returns null if the claim is missing, so callers should handle that case.

Repository Pattern for Portfolio

Define the Interface

```
using System.Collections.Generic;
using System.Threading.Tasks;
using YourApp.Models;

public interface IPortfolioRepository
{
    Task<IReadOnlyList<StockDto>> GetUserPortfolioAsync(AppUser user);
}
```

Implement the Repository

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using YourApp.Data;
using YourApp.Models;

public class PortfolioRepository : IPortfolioRepository
{
    private readonly ApplicationDbContext _context;

    public PortfolioRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<IReadOnlyList<StockDto>> GetUserPortfolioAsync(AppUser user)
    {
        return await _context.Portfolios
```

```
        .Where(p => p.AppUserId == user.Id)        // filter by user
        .Select(p => new StockDto                  // map join row → Stock DTO
        {
            Symbol      = p.Stock.Symbol,
            CompanyName  = p.Stock.CompanyName,
            PurchaseDate = p.Stock.PurchaseDate,
            LastDiv     = p.Stock.LastDiv,
            Industry    = p.Stock.Industry,
            MarketCap   = p.Stock.MarketCap
        })
        .ToListAsync();
    }
}
```

- Portfolios is the **join table** linking AppUser <->Stock.
- The Select clause acts like a **map** operation, transforming each join entity into a clean StockDto.

Mapping the Join Table to a Stock DTO

```
public class StockDto
{
    public string Symbol { get; set; }
    public string CompanyName { get; set; }
    public DateTime PurchaseDate { get; set; }
    public decimal LastDiv { get; set; }
    public string Industry { get; set; }
    public long MarketCap { get; set; }
}
```

- Only the **stock-specific fields** are exposed; user information is omitted.
- This DTO is ideal for returning to the front-end via the API.

Asynchronous Data Retrieval

- All database calls are performed with async/await to avoid blocking threads.
- Example call chain:

1. Retrieve username from claims.
2. Find the AppUser with UserManager.FindByNameAsync.

3. Call PortfolioRepository.GetUserPortfolioAsync.
4. The repository executes an EF Core query ending in ToListAsync().
5. The controller returns Ok(portfolio).

```
var username = User.GetUserName();
var appUser  = await _userManager.FindByNameAsync(username);
var portfolio = await _portfolioRepo.GetUserPortfolioAsync(appUser);
return Ok(portfolio);
```

Principle: *Never block on asynchronous I/O in ASP.NET Core; always use the async variants (FindByNameAsync, ToListAsync, etc.) to keep the request pipeline scalable.*## Adding the Portfolio Repository to Dependency Injection

- Open **Program.cs**.
- Import the repository interface and implementation:

```
using YourNamespace.Repositories;   // IPortfolioRepository
using YourNamespace.Services;      // IPortfolioService
```

- Register the services in the DI container (usually inside builder.Services):

```
builder.Services.AddScoped<IPortfolioRepository, PortfolioRepository>();
builder.Services.AddScoped<IPortfolioService, PortfolioService>();
```

- Save and rebuild the solution.

Dependency Injection – a design pattern that supplies a class with its required dependencies from an external source rather than creating them internally.

Manually Creating a Portfolio Record in the Database

1. **Add a Stock**
    - Navigate to the **Stock** table in the stock database.
    - Insert a new stock row and **copy the generated Stock ID**.
2. **Ensure a User Exists**
    - Open the **Users** table.
    - If no user is present, create one and note its **User ID**.
3. **Insert a Portfolio Entry**
    - Go to the **Portfolios** table.
    - Choose **"Edit Top 200 Rows"** (or similar).
    - Delete any rows with NULL values.
    - Fill in the required fields:

Column	Value (example)
UserId	*User ID from step 2*
StockId	*Stock ID from step 1*
CreatedDate	GETDATE() (or appropriate date)
Quantity	10 (or desired amount)

- Press **Enter** to commit the row.

4. **Verify Data Types**
    - Ensure UserId and StockId are stored as **integers** (or the correct type) and not as strings.

Testing the Portfolio Endpoint with Swagger

- Launch the application and open **Swagger UI** (usually https://localhost:{port}/swagger).
- Authenticate:
    1. Click **"Authorize"**.
    2. Paste the JWT token into the input field.
    3. Confirm – the lock icon should appear, indicating you're authorized.
- Call the **Portfolio** endpoint (e.g., GET /api/portfolio/{userId}).
- The response should list the **stock(s)** linked to the user via the portfolio record you created.

Quick Reference Table

Action	File / Location	Code / Command
Register repository in DI	Program.cs	builder.Services.AddScoped<IPortfolioRepository, PortfolioRepository>();
Insert stock	stock database → **Stocks**	Manual row insertion, copy **StockId**
Insert user	stock database → **Users**	Manual row insertion, copy **UserId**
Insert portfolio record	stock database → **Portfolios**	Fill columns as per table above
Test via Swagger	https://localhost:{port}/swagger	Authorize → call /api/portfolio/{userId}

ASP.NET Core Web API .NET 8 2024 - 27. Portfolio CREATE

Many-to-Many Relationship Overview

Many-to-many relationships are modeled with a **join table** that stores pairs of foreign keys.

- In this example the join table is **Portfolio**.
- It links **AppUser** (user) <->**Stock** (stock symbol).

Definition: A *join table* contains the primary keys of the two related tables, allowing each side to have multiple links to the other.

Step-by-Step Process for Adding a Portfolio Entry

Step	Action	Purpose
1	Extract **username** from JWT claims.	Identify the current user.
2	Retrieve the **AppUser** entity via UserManager.FindByNameAsync.	Get the user's primary key (AppUserId).
3	Get the **Stock** entity by its **symbol** using a repository method.	Obtain StockId.
4	Validate that the stock exists.	Prevent adding nonexistent stocks.
5	Check the user's existing portfolio for the same symbol (case-insensitive).	Avoid duplicate entries.
6	Create a new **Portfolio** object with AppUserId and StockId.	Build the join-table record.
7	Persist the object through PortfolioRepository.CreateAsync.	Save to the database.
8	Return appropriate HTTP response (Created or error status).	Inform the client of success/failure.

Endpoint Implementation (C# / ASP.NET Core)

```
[HttpPost]
[Authorize]
public async Task<IActionResult> AddPortfolio([FromBody] string symbol)
{
    //Get username from claims
    var username = User.GetUsernameFromClaims();

    //Retrieve user
    var appUser = await _userManager.FindByNameAsync(username);
    if (appUser == null) return Unauthorized();

    //Retrieve stock by symbol
```

```
    var stock = await _stockRepository.GetBySymbolAsync(symbol);
    if (stock == null)
        return NotFound(new { Message = "Stock not found." });

    //Duplicate check
    var exists = await _portfolioRepository
        .AnyAsync(p => p.AppUserId == appUser.Id &&
                       p.Stock.Symbol.Equals(symbol, StringComparison.OrdinalIgnoreCase));
    if (exists)
        return BadRequest(new { Message = "Stock already in portfolio." });

    //Create Portfolio object
    var portfolio = new Portfolio
    {
        AppUserId = appUser.Id,
        StockId   = stock.Id
    };

    //Persist
    var created = await _portfolioRepository.CreateAsync(portfolio);
    if (created == null)
        return StatusCode(500, new { Message = "Could not create portfolio entry." });

    //Return success
    return CreatedAtAction(nameof(GetPortfolio), new { id = created.Id }, created);
}
```

Key points

- User.GetUsernameFromClaims() is a **claims extension** method.
- The endpoint is protected with [Authorize] so the JWT token supplies the claims.

Stock Repository – "Get By Symbol" Method

```
public interface IStockRepository
{
    Task<Stock?> GetBySymbolAsync(string symbol);
    // other members...
}

public class StockRepository : IStockRepository
{
    private readonly AppDbContext _context;
    public StockRepository(AppDbContext context) => _context = context;

    public async Task<Stock?> GetBySymbolAsync(string symbol) =>
        await _context.Stocks
            .FirstOrDefaultAsync(s => s.Symbol == symbol);
}
```

- The method returns null when the symbol isn't found, enabling the **stock-not-found** check in the controller.

Portfolio Repository – Create Method

```
public interface IPortfolioRepository
{
    Task<Portfolio?> CreateAsync(Portfolio portfolio);
    Task<bool> AnyAsync(Expression<Func<Portfolio, bool>> predicate);
    // other members...
}

public class PortfolioRepository : IPortfolioRepository
{
    private readonly AppDbContext _context;
    public PortfolioRepository(AppDbContext context) => _context = context;
```

```
    public async Task<Portfolio?> CreateAsync(Portfolio portfolio)
    {
        await _context.Portfolios.AddAsync(portfolio);
        await _context.SaveChangesAsync();
        return portfolio;
    }

    public async Task<bool> AnyAsync(Expression<Func<Portfolio, bool>> predicate) =>
        await _context.Portfolios.AnyAsync(predicate);
}
```

- CreateAsync **adds** the new Portfolio entity and calls SaveChangesAsync.
- AnyAsync is used for the duplicate-stock validation.

Validation Checklist

- **Stock existence** – stock == null ⇒ 404 Not Found.
- **Duplicate prevention** – use case-insensitive comparison of symbols.
- **User authorization** – handled by [Authorize] and claim extraction.
- **Null result from repository** – defensive 500 Internal Server Error.

Testing with Swagger

1. Start the API with dotnet watch run.
2. Open **Swagger UI**, ensure the lock icon shows the JWT is attached.
3. Use the **AddPortfolio** endpoint:
    - Input a valid stock symbol (e.g., PLTR).
    - Observe a **201 Created** response.
4. Verify the entry appears in the **Portfolio** table (or via a GET endpoint).

Key Terminology

- **Join table** – intermediary table storing foreign keys for many-to-many links.
- **Claims extension** – helper method to read custom data (e.g., username) from JWT claims.
- **Repository pattern** – abstraction layer for data access, promoting testability.
- **Idempotent check** – ensuring the same request doesn't create duplicate records.

ASP.NET Core Web API .NET 8 2024 - 28. Portfolio DELETE

Delete Portfolio Endpoint

Purpose: Remove a specific stock symbol from the authenticated user's portfolio.

- **HTTP verb:** DELETE
- **Route parameters:**
    - string username (extracted from claims)
    - string symbol (stock ticker to delete)

```
[HttpDelete("{symbol}")]
[Authorize]
public async Task<ActionResult> DeletePortfolio(string symbol)
{
   // implementation detailed below
}
```

Authorization & User Retrieval

Key step: Identify the user making the request via identity claims.

1. Extract the **username** from the JWT claims (extension created earlier).
2. Use UserManager<AppUser> to fetch the corresponding AppUser entity.

```
var username = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
var appUser = await _userManager.FindByNameAsync(username);
// Optional: null-check if you want extra safety
if (appUser == null) return BadRequest("User not found.");
```

Portfolio Repository Delete Method

Goal: Encapsulate the delete logic in a reusable repository method.

Method	Parameters	Return
DeletePortfolio(AppUser user, string symbol)	AppUser (owner), string (ticker)	Portfolio? (deleted entity or null)

```
public async Task<Portfolio?> DeletePortfolio(AppUser user, string symbol)
{
   var portfolio = await _context.Portfolios
      .FirstOrDefaultAsync(p => p.UserId == user.Id &&
                           p.Symbol.ToLower() == symbol.ToLower());
   if (portfolio == null) return null;
```

```
    _context.Portfolios.Remove(portfolio);
    await _context.SaveChangesAsync();

    return portfolio;
}
```

- **FirstOrDefaultAsync** fetches the exact record matching both the user ID and the lower-cased symbol.
- A **null check** ensures graceful handling when the stock isn't in the portfolio.
- After removal, SaveChangesAsync persists the change.

Filtering Logic in the Controller

Before calling the repository, confirm that the stock exists in the user's current portfolio.

```
var userPortfolio = await _portfolioRepo.GetUserPortfolioAsync(appUser);

var filteredStock = userPortfolio
    .Where(s => s.Symbol.Equals(symbol, StringComparison.OrdinalIgnoreCase))
    .ToList();

if (!filteredStock.Any())
    return BadRequest("Stock is not in your portfolio.");
```

- GetUserPortfolioAsync returns all portfolio items for the user.
- Where + Equals(..., OrdinalIgnoreCase) performs a case-insensitive match.
- If the filtered list is empty, a **400 Bad Request** is returned.

Testing with Swagger

1. **Log in** via Swagger UI to obtain a bearer token (lock icon → login).
2. **Verify current holdings** with a GET /portfolio call (e.g., Tesla, PLTR).
3. **Delete** each stock:
    - DELETE /portfolio/TESLA → should return 200 OK.
    - DELETE /portfolio/PLTR → should return 200 OK.
4. **Confirm removal** by re-issuing the GET request; the deleted symbols should no longer appear.
5. **Re-add** the stocks with POST /portfolio to ensure the delete-add cycle works end-to-end.

Summary of Delete Flow (Step-by-Step)

1. **Authorize** request and pull the username from claims.
2. **Retrieve** the AppUser via UserManager.
3. **Fetch** the full portfolio for that user.
4. **Filter** the list for the target symbol (case-insensitive).
5. **Return** BadRequest if the symbol isn't found.
6. **Call** PortfolioRepository.DeletePortfolio(appUser, symbol).
7. **Handle** a null result from the repo as a failure case.
8. **Respond** with Ok() on successful deletion.

ASP.NET Core Web API .NET 8 2024 - 29. One-To-One

One-to-One vs One-to-Many Relationships

Definition

One-to-One: Each entity instance is related to at most one instance of the other entity.

One-to-Many: One entity instance can be related to many instances of the other entity, represented by a collection (array/list).

Feature	One-to-One	One-to-Many
Navigation property	Single object (User)	Collection (ICollection<Comment>)
Database column	Foreign key column only	Foreign key column + separate join table or array
JSON representation	{ "UserId": "...", "User": { ... } }	{ "Comments": [{...}, {...}] }
EF Core configuration	No list on the dependent side	ICollection<T> on the principal side

Implementing a One-to-One Relationship in EF Core

1. **Add a foreign key property** to the dependent model (Comment).
2. **Add a navigation property** of the principal type (AppUser).
3. **Do not add a collection** on the dependent side.

Code Example (C#)

```
// Comment.cs
public class Comment
{
    public int Id { get; set; }

    //Foreign key matching the AppUser primary key
    public string AppUserId { get; set; }

    //Navigation property – single AppUser instance
    public AppUser AppUser { get; set; }

    // other comment fields...
}
// AppUser.cs (principal)
public class AppUser
```

```
{
    public string Id { get; set; }
    public string UserName { get; set; }
    // No collection of Comments here for a true one-to-one
}
```

Creating and Applying the Migration

1. Open a terminal in the API project folder.
2. Run the migration command:
3. dotnet ef migrations add CommentOneToOne
4. Apply the migration to the database:
5. dotnet ef database update

The migration will generate a new column AppUserId in the Comments table and set up the foreign-key constraint.

Verifying the Schema in SQL Server Management Studio

1. Connect to the database with **SQL Server Management Studio**.
2. Expand **Tables** → **Comments** and select **Select Top 1000 Rows**.
3. Confirm the presence of the new column **AppUserId** in the result set.

Column Name	Data Type	Description
Id	int	Primary key of the comment
AppUserId	nvarchar(max)	Foreign key referencing AspNetUsers.Id
...	...	Other comment fields

Step-by-Step Summary (One-to-One Setup)

1. Edit the dependent model (Comment) to include AppUserId and AppUser.
2. Ensure the principal model (AppUser) does **not** contain a collection of comments.
3. Run dotnet ef migrations add <Name> to scaffold the schema change.
4. Execute dotnet ef database update to apply the change.
5. Open SSMS and verify the new foreign-key column appears in the Comments table.

Key Concepts at a Glance

- **Sub-model**: A model embedded within another model (e.g., User inside Comment).
- **Navigation property**: Enables EF Core to load related data; can be a single object or a collection.
- **Foreign key**: Column that stores the identifier of the related entity.
- **Migration**: Incremental code-first change that updates the database schema.

ASP.NET Core Web API .NET 8 2024 - 30. User Generated Content

One-to-One Relationship Between Comment & User

Definition: A *one-to-one* relationship links a single record in one table (Comment) to a single record in another table (User), allowing direct access to the comment author's data.

- Goal: automatically attach the **AppUser** to a newly created comment so the author's name appears with the comment.
- Benefits: easy identification of trolls, richer UI data, minimal extra code.

Injecting UserManager<AppUser> into CommentController

1. Add a private read-only field:
2. private readonly UserManager<AppUser> _userManager;
3. Initialize it via constructor injection:
4. public CommentController(UserManager<AppUser> userManager, /* other deps */)
5. {
6. _userManager = userManager;
7. }

- The UserManager gives access to the currently authenticated user.

Creating a Comment with the Current User

Steps in the Create action

1. **Extract username** from JWT claims (using a claims extension).
2. **Fetch the user** from the database:
3. var user = await _userManager.FindByNameAsync(username);
4. **Attach the user** to the comment DTO/model before saving:
5. commentDto.AppUserId = user.Id; // or comment.AppUser = user;
6. **Save** the comment as usual.

- This "sneaky" step runs **before** the comment is persisted, ensuring the foreign key is set.

Updating Repository Queries with Include

Principle: EF Core uses *deferred execution*; navigation properties aren't loaded unless explicitly included.

Comment Repository

Method	Required Include
GetAllAsync	c => c.AppUser
GetByIdAsync	c => c.AppUser (via FirstOrDefaultAsync with Include)

```
public async Task<IEnumerable<Comment>> GetAllAsync()
{
    return await _context.Comments
                .Include(c => c.AppUser)
                .ToListAsync();
}
public async Task<Comment?> GetByIdAsync(int id)
{
    return await _context.Comments
                .Include(c => c.AppUser)
                .FirstOrDefaultAsync(c => c.Id == id);
}
```

Stock Repository – Nested Includes

- Stocks → Comments → AppUser

```
public async Task<IEnumerable<Stock>> GetAllAsync()
{
    return await _context.Stocks
                .Include(s => s.Comments)
                   .ThenInclude(c => c.AppUser)
                .ToListAsync();
}
```

- The ThenInclude ensures the **user** inside each comment is loaded.

Extending CommentDto

- Add a CreatedBy property to expose the author's username in API responses.

```
public class CommentDto
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
    // ... other fields ...
    public string CreatedBy { get; set; } = string.Empty;   // new
}
```

- Prevent null references by initializing with string.Empty.

Mapping Between Comment and CommentDto

```
public static CommentDto ToDto(Comment comment)
{
    return new CommentDto
    {
        Id = comment.Id,
        Title = comment.Title,
        // ... other mappings ...
        CreatedBy = comment.AppUser?.UserName ?? string.Empty
    };
```

- The mapper pulls UserName from the navigation property AppUser.

Adjusting StockController Return Types

- Ensure the stock-related endpoints return StockDto (which now contains comments with CreatedBy).
- Add the extra ToList() call if needed to materialize the query before returning.

```
var stocks = await _stockRepository.GetAllAsync();
return Ok(stocks.Select(s => _mapper.ToDto(s)).ToList());
```

Testing via Swagger

1. **Authenticate** in Swagger to obtain a JWT token.
2. **Create a comment** on a chosen stock (e.g., ticker "PALEN").
3. Verify the response contains:
    - createdBy with the correct username.
    - The comment appears under the stock's comments array.
4. **Retrieve the stock** again; the nested comment should show the author's name.

Quick Recap Checklist

- [ ] UserManager<AppUser> injected into CommentController.
- [ ] Username extracted from claims and user fetched with FindByNameAsync.
- [ ] Comment model receives AppUserId before saving.
- [ ] Repository queries use Include / ThenInclude to load the user.
- [ ] CommentDto includes a non-null CreatedBy field.
- [ ] Mapper sets CreatedBy = comment.AppUser.UserName.
- [ ] Stock endpoints return DTOs that contain comments with author info.

These steps establish a robust one-to-one link, enabling every comment to display its author without extra client-side work.