Step-by-Step Guide: Setting Up a Basic Cloth Management API with .NET

Introduction

This guide is designed for students who are new to .NET development as well as those with prior experience. It will walk through setting up a simple **Cloth Management API**, explaining key concepts along the way.

---

0. Downloading Dependencies

- **Dependencies**: Packages and services required for the application to run.
  - Managed through **NuGet** in `.csproj` files.
  - Dependencies:
    - `Microsoft.EntityFrameworkCore.SqlServer`: Handles database connections.
    - `Microsoft.EntityFrameworkCore`: Enables Entity Framework.
    - `Swashbuckle.AspNetCore`: Enables Swagger documentation.
    - Serilog.AspNetCore: Enables Logging
    - Serilog.Sinks.Console: Enables Serilog to write onto the console
  - Dependencies ensure the application has the necessary libraries to function properly.

1. Understanding the Components

How `get;` `set;` Relates to Queries in EF Core

- When you use `get;`, Entity Framework (EF Core) retrieves the value from the database when querying an entity.
- When you use `set;`, EF Core tracks changes to the property and updates the database when `SaveChanges()` is called.

Example:

```
 var cloth = _context.Cloths.FirstOrDefault(c => c.ClothID == 1);
cloth.Name = "Updated Name";  // set; is used here
_context.SaveChanges();  // EF Core updates the database
```

-
  - The `get;` retrieves the `Name` from the database.

    - The `set;` updates it in memory, and `SaveChanges()` commits it to the database.

Quick Explanation of { get; set; }

- `get;` allows you to **retrieve** the value of a property.
- `set;` allows you to **assign** a value to a property.

Example:

public string Name { get; set; }

-
    - `Name` can be read (`get`) and changed (`set`).
    - This is shorthand for defining a property with an internal backing field.

1.1. Understanding Project Folder Structure

📌 Common Folders in a .NET API Project

- **`Models` Folder**: Contains classes that define the database entities.
    - Example: `Cloth.cs` and `Order.cs` define the structure of the database tables.
- **`DTO` (Data Transfer Object) Folder**: Contains objects used to shape data before sending it through the API.
    - Example: `ClothDTO.cs` ensures only relevant `Cloth` data is exposed.
- **`Controllers` Folder**: Contains API controllers that handle incoming HTTP requests and responses.
    - Example: `ClothController.cs` defines endpoints for managing cloth data.
- **`Data` Folder**: Contains the database context (`DbContext.cs`), which handles interactions with the database.
    - Example: `AppDbContext.cs` configures EF Core to map models to database tables.
- **`appsettings.json`:** Stores configuration settings for the application.

    Example of contents:

    ```
    {

      "ConnectionStrings": {

        "DefaultConnection":
    "Server=.;Database=MyDb;Trusted_Connection=True;"

      },
    ```

```
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

- **`Program.cs`**: The entry point of the application.
  - It sets up the web host, registers services, and configures middleware.

    Example setup:

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

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

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

    builder.Services.AddControllers();

    var app = builder.Build();

    app.UseAuthorization();

    app.MapControllers();
    ```

      - app.Run();
      - This configures services such as **database connections, controllers, and middleware**.
      - Used to configure **database connections, logging levels, API keys, and other environment settings**.

These folders and configurations help keep code **organized, scalable, and maintainable** while ensuring proper setup for an efficient API.

1.2. Models (`Cloth.cs`)

A **model** represents the data structure used in our application. The `Cloth` model defines the properties of a cloth item in our database.

- Defined with **Entity Framework** attributes to specify table and column names.
- Uses **data annotations** like `[Key]` to mark the primary key.
- Includes a **relationship** with `Orders`.

Code Example (`Cloth.cs` without Foreign Key):

```
[Table("cloths")]
public partial class Cloth
{
    [Key]
    [Column("ClothID")]
    public int ClothID { get; set; }

    [Column("Name")]
    public required string Name { get; set; }

    [Column("Quantity")]
    public int Quantity { get; set; }

    public ICollection<Order> Orders { get; set; } = new List<Order>();
}
```

Why Use `ICollection<Order>`?

- This represents a **one-to-many relationship** where a single `Cloth` item can be associated with multiple `Order` entries.
- EF Core uses this collection to understand and **map the foreign key relationship in the `Order` table`**.

Code Example (`Order.cs` with Foreign Key Relationship to Cloth):

```
[Table("orders")]
public partial class Order
```

```
{
    [Key]
    [Column("OrderID")]
    public int OrderID { get; set; }

    [ForeignKey("ClothID")]
    public int ClothID { get; set; }
    public Cloth Cloth { get; set; }

    public DateTime OrderDate { get; set; }
}
```

Why is `ClothID` a Foreign Key in `Order`?

- `Order` needs to be associated with a specific `Cloth` item.
- The `[ForeignKey("ClothID")]` annotation explicitly tells EF Core that `ClothID` is a foreign key linking to the `Cloth` table.
- The `public Cloth Cloth { get; set; }` navigation property allows easy retrieval of the related `Cloth` entity when querying an `Order`.
- This ensures that every `Order` is always linked to an existing `Cloth`, preventing `Order` records from being left without a related `Cloth`.

1.3. Data Transfer Objects (`ClothDTO.cs`)

- A **DTO (Data Transfer Object)** is used to control what data is exposed from our API.
- It helps in preventing the overexposure of sensitive database fields.

Code Example (`ClothDTO.cs`):

```
public class ClothDTO

{

    public int ClothID { get; set; }

    public string Name { get; set; }

    public int Quantity { get; set; }
}
```

1.4. Database Context (`DbContext.cs`)

- Defines how entities interact with the **database**.
- Specifies **relationships** and constraints between tables.

Key Features:

- `DbSet<Cloth>`: Defines a table for `Cloths`.
- Configures **foreign keys** for `Order` relationships.

Code Example (`DbContext.cs` without Foreign Keys):

```
public class AppDbContext : DbContext
{
    public DbSet<Cloth> Cloths { get; set; }
}
```

Code Example (`DbContext.cs` with Foreign Keys):

```
public class AppDbContext : DbContext
{
    public DbSet<Cloth> Cloths { get; set; }
    public DbSet<Category> Categories { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Cloth>()
            .HasOne(c => c.Category)
            .WithMany(cat => cat.Cloths)

            .HasForeignKey(c => c.CategoryID);

    }

}
```

---

2. Implementing the API Controller (`ClothController.cs`)

The **controller** handles HTTP requests and responses, acting as a bridge between the client (frontend) and the database. It processes requests, executes business logic, and returns responses.

2.1. Setting Up the Controller

- Inherits from `ControllerBase`, following **RESTful principles**.
- Uses **dependency injection** to interact with `AppDbContext`.

Constructor:

```
public ClothController(AppDbContext context, ILogger<ClothController> logger)
{
    _context = context;
    _logger = logger;
}
```

Explanation:

- `_context` is the database context used to access the database.
- `_logger` is used for logging messages such as errors and successful operations.

2.2. CRUD Operations

GET all clothes

```
[HttpGet]
public async Task<ActionResult<IEnumerable<ClothDTO>>> GetCloth()
{
    var cloths = await _context.Cloths
        .Select(c => new ClothDTO
        {
            ClothID = c.ClothID,
            Name = c.Name,
            Quantity = c.Quantity
        })
        .ToListAsync();

    return Ok(cloths);
}
```

Explanation:

- This endpoint retrieves all `Cloth` records from the database.
- `Select` maps each `Cloth` to a `ClothDTO` to prevent exposing unnecessary details.
- `ToListAsync()` ensures the query is executed asynchronously.
- `return Ok(cloths);` sends a 200 OK response with the retrieved data.

GET a single cloth by ID

```
[HttpGet("{id}")]
public async Task<ActionResult<Cloth>> GetCloth(int id)
{
    var cloth = await _context.Cloths.FindAsync(id);

    if (cloth == null)
    {
        _logger.LogWarning("Cloth item with ID {ClothID} not found.", id);
        return NotFound();
    }

    _logger.LogInformation("Successfully fetched cloth item {ClothID}.", id);
    return cloth;
}
```

Explanation:

- Looks up a `Cloth` by `id`.
- If no record is found, returns `404 Not Found`.

- Otherwise, returns the cloth object.

POST (Create a cloth)

```
[HttpPost]
public async Task<ActionResult<Cloth>> CreateCloth(Cloth cloth)
{
    _context.Cloths.Add(cloth);
    await _context.SaveChangesAsync();

    _logger.LogInformation("Cloth item {ClothID} created successfully.",
cloth.ClothID);

    return CreatedAtAction(nameof(GetCloth), new { id = cloth.ClothID }, cloth);
}
```

Explanation:

- Adds a new `Cloth` entry to the database.
- Saves changes asynchronously.
- Returns a `201 Created` response with the new item's details.

PUT (Update a cloth)

```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateCloth(int id, Cloth cloth)
{
    if (id != cloth.ClothID)
    {
        _logger.LogWarning("Cloth ID mismatch: {ClothID} does not match request
ID {RequestID}.", cloth.ClothID, id);
        return BadRequest();
    }

    _context.Entry(cloth).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
        _logger.LogInformation("Cloth item {ClothID} updated successfully.", id);
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!_context.Cloths.Any(c => c.ClothID == id))
        {
            _logger.LogWarning("Cloth item {ClothID} not found during update.", id);
            return NotFound();
        }
        throw;
    }

    return NoContent();
}
```

Explanation:

- Ensures the provided `id` matches the `ClothID`.
- Marks the entity as modified and saves changes.
- Returns `204 No Content` if successful.

DELETE (Remove a cloth)

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteCloth(int id)
{
    var cloth = await _context.Cloths.FindAsync(id);
    if (cloth == null)
    {
        _logger.LogWarning("Cloth item {ClothID} not found for deletion.", id);
        return NotFound();
    }

    _context.Cloths.Remove(cloth);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Explanation:

- Searches for the cloth by `id`.
- If found, removes it from the database.

- Saves changes and returns `204 No Content` if successful.

Explanation of the Controller Code

Key Components:

- `var cloth = await _context.Cloths.FindAsync(id);`: Fetches a record asynchronously by its ID.
- `_context.Entry(cloth).State = EntityState.Modified;`: Marks an entity as modified so EF Core knows to update it in the database.
- `try { await _context.SaveChangesAsync(); }`: Attempts to save changes and logs success or failure.
- `_logger.LogInformation(...)`** and ****`_logger.LogWarning(...)`**: Logs important events such as successful retrieval, updates, and warnings when an item isn't found.

These components ensure proper database interactions, logging, and error handling in a structured way.

---

3. Running the API

Step 1: Run the API

- Click the **green arrow** ▶️ in Visual Studio to start the API.
- This will launch the application and provide the HTTPS URL for your API.
- Open the URL in your browser or test with Postman.

Hot Reload Feature

- **What it does:** Allows you to make changes in your code and see updates in real-time without restarting the entire application.
- **How to use it:**
  - Modify your code (e.g., update a controller or model).
  - Save the file, and the application updates automatically.
  - No need to restart the API manually.