

.NET Programming

Chapter 3: Working with data

TABLE CONTENT

- Introduction to Entity Framework
- CRUD Operations
- Migrations
- Complex Models
- Inheritance

INTRODUCTION TO ENTITY FRAMEWORK

Entity Framework (EF) is an Object-Relational Mapper (ORM) for .NET, allowing developers to interact with databases using .NET objects rather than writing raw SQL queries. It simplifies data access by abstracting much of the database layer and translating C# code into SQL commands.



Key Features:

- Database Abstraction: EF allows you to work with data as strongly-typed .NET objects. Instead of interacting with raw database tables and columns, you use entities (classes) and their properties.
- Code-First Approach: In EF, you can define your database structure through C# classes (entities), and EF will generate the corresponding database schema.
- Database-First Approach: EF also supports generating C# classes from an existing database, making it easier to map database tables to entities.
- Querying Data: With LINQ (Language Integrated Query), you can write queries in C# that EF will translate into SQL and execute on the database.
- Change Tracking: EF keeps track of changes made to the objects and automatically generates the appropriate SQL commands to update the database.
- Migrations: EF provides a way to manage database schema changes over time. With migrations, you can evolve the database schema while preserving the existing data.

Entity Framework Versions

- EF 6.x: The classic, stable version of Entity Framework.
- EF Core: A lightweight, cross-platform version, offering better performance and additional features.

Example:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }
}
```

- Student is an entity class representing a table in the database.
- SchoolContext inherits from DbContext, which is the class responsible for interacting with the database.

You can then query the database as:

```
using (var context = new SchoolContext())
{
    var students = context.Students.ToList();
}
```


EF Workflows

- Code First: Define your models in code and generate the database schema from those models.
- Database First: Start with an existing database and generate models from it.
- Model First: Use a visual designer to design your models, which will generate both the database schema and the classes.

EF helps to minimize the impedance mismatch between the object-oriented world and relational databases, making it easier to manage data persistence and manipulation in your .NET application

CRUD OPERATIONS

Implement CRUD Functionality - ASP.NET MVC with EF Core

CRUD Operations: Basic operations to Create, Read, Update, and Delete data from a database.

Goals: Learn how to implement these operations in an ASP.NET MVC application using EF Core.

Prerequisites:

- Create an ASP.NET Core MVC Project
- Set Up the Database Context (The previous chapter was completed in the model)
- Create the Model

Create Operation

Create View: Displays a form to create a new record (HTML form).

Controller Actions:

HttpGet to show the form:

```
0 references  
public IActionResult Create()  
{  
    return View();  
}
```

Create Operation

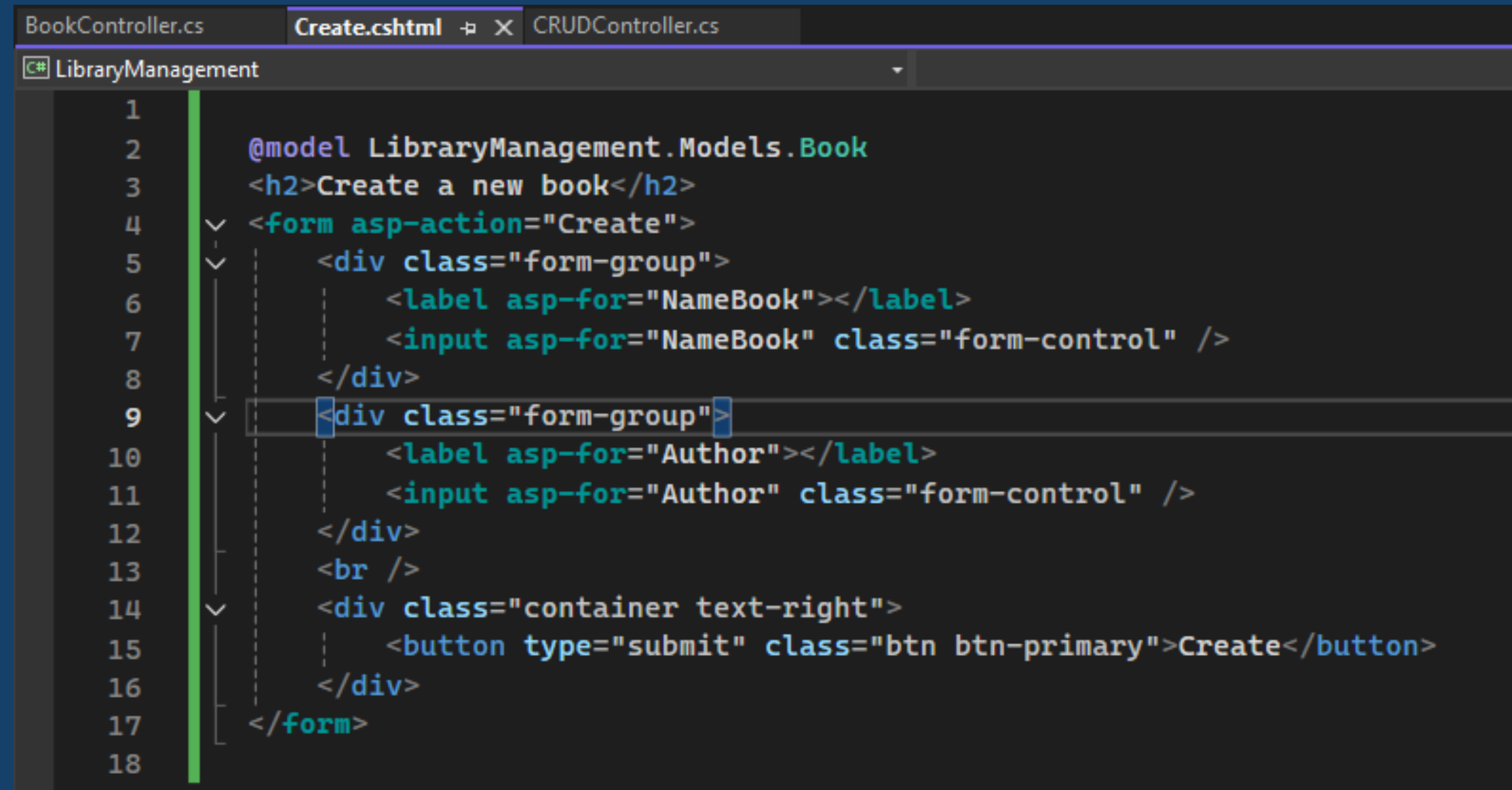
HttpPost to handle form submission:

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Create([Bind("NameBook,Author")] Book book)
{
    if (ModelState.IsValid)
    {
        var maxId = _Bookcontext.Book.Max(b => (int?)b.Id) ?? 0;
        book.Id = maxId + 1;

        _Bookcontext.Add(book);
        await _Bookcontext.SaveChangesAsync();
        return RedirectToAction("List", "Book"); // (Action, Controller)
    }
    return RedirectToAction("List", "Book");
}
```

Create Operation

Create View (Create.cshtml): Renders an HTML form for creating a new book.



```
1
2  @model LibraryManagement.Models.Book
3  <h2>Create a new book</h2>
4  <form asp-action="Create">
5      <div class="form-group">
6          <label asp-for="NameBook"></label>
7          <input asp-for="NameBook" class="form-control" />
8      </div>
9      <div class="form-group">
10         <label asp-for="Author"></label>
11         <input asp-for="Author" class="form-control" />
12     </div>
13     <br />
14     <div class="container text-right">
15         <button type="submit" class="btn btn-primary">Create</button>
16     </div>
17 </form>
18
```

Read Operation

Displays a list of books.

```
public IActionResult List()
{
    var books = _Bookcontext.Book.OrderBy(b => b.Id).ToList(); //Linq
    ViewBag.Books = books;
    return View();
}
```

```
<tbody>
  @foreach (var item in ViewBag.Books)
  {
    <tr>
      <td>@item.Id</td>
      <td>@item.NameBook</td>
      <td>@item.Author</td>
      <td>
        <a href="@Url.Action("Edit", "Books", new { id = item.Id })" class="btn btn-primary">Edit</a>
        <form asp-action="Delete" asp-controller="Books" method="post" class="d-inline">
          <input type="hidden" name="id" value="@item.Id" />
          <button type="submit" class="btn btn-danger">Delete</button>
        </form>
      </td>
    </tr>
  }
</tbody>
```

CRUD OPERATIONS

Update Operation

Controller Edit Action:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var book = await _Bookcontext.Book.FindAsync(id);
    if (book == null)
    {
        return NotFound();
    }
    return View(book);
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Edit(int id, [Bind("Id,NameBook,Author")] Book book)
{
    if (id != book.Id)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        _Bookcontext.Update(book);
        await _Bookcontext.SaveChangesAsync();
        return RedirectToAction("List", "CRUD");
    }
    return RedirectToAction("List", "CRUD");
}
```

CRUD OPERATIONS

Update Operation

Edit View (Edit.cshtml):

Renders an HTML form to update an existing book.

```
@model LibraryManagement.Models.Book

<h2>Edit Book</h2>

<form asp-action="Edit">
  <input type="hidden" asp-for="Id" />
  <div class="form-group">
    <label asp-for="NameBook"></label>
    <input asp-for="NameBook" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Author"></label>
    <input asp-for="Author" class="form-control" />
  </div>
  <br />
  <!-- When submit call that auto cal post method-->
  <button type="submit" class="btn btn-primary">Save</button>
</form>
```


CRUD OPERATIONS

Delete Operation

Controller Delete Action:

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var book = await _Bookcontext.Book.FirstOrDefaultAsync(m => m.Id == id);
    if (book == null)
    {
        return NotFound();
    }
    return View(book);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var book = await _Bookcontext.Book.FindAsync(id);
    _Bookcontext.Book.Remove(book);
    await _Bookcontext.SaveChangesAsync();
    return RedirectToAction("List", "CRUD");
}
```

CRUD OPERATIONS

Delete Operation

Delete View (Delete.cshtml):

Displays a confirmation prompt before deleting a book.

```

List.cshtml Delete.cshtml Edit.cshtml CRUDController.cs
LibraryManagement
1      @model LibraryManagement.Models.Book
2
3      <h2>Delete Book</h2>
4
5      <p>Are you sure you want to delete this Book?</p>
6      <dl class="row">
7          <dt class="col-sm-2">Book Name</dt>
8          <dd class="col-sm-10">@Model.NameBook</dd>
9
10         <dt class="col-sm-2">Author</dt>
11         <dd class="col-sm-10">@Model.Author</dd>
12     </dl>
13
14     <form asp-action="Delete">
15         <input type="hidden" asp-for="Id" />
16         <button type="submit" class="btn btn-danger">Delete</button>
17     </form>
18

```

CRUD OPERATIONS

Recap:

- Implemented CRUD operations in an ASP.NET Core MVC app using Entity Framework Core.
- Razor Views used to render forms and data for each operation (Create, Read, Update, Delete).
- Scaffolded the controller and views to quickly set up a functional web interface.

MIGRATIONS

What are Migrations?

- **Migrations:** A feature in Entity Framework Core (EF Core) to manage changes to the data model.
- **Purpose:**
 - Keeps the database schema in sync with the application's data model.
 - Facilitates incremental updates without losing existing data.
- **Use Cases:**
 - Adding, modifying, or removing columns or tables.
 - Changing relationships between entities.

Why Use Migrations?

- **Maintain Data Integrity:** Allows you to update the schema while preserving existing data.
- **Version Control:** Each migration acts as a version of the database schema, enabling easy tracking of changes.
- **Automation:** Simplifies database updates during development and deployment.

Key Components of Migrations

1. Migration Class:

- Contains `Up` and `Down` methods.
- `Up`: Defines the operations to apply changes.
- `Down`: Defines the operations to revert changes.

2. Model Snapshot:

- A snapshot file representing the current state of the data model.
- Helps track what has changed since the last migration.

Creating a Migration

- **Command to Create a Migration:**

```
dotnet ef migrations add MigrationName
```

- **Example:**

- If you add a new column to an existing table, create a migration to apply that change.

- **Generated Files:**

- A new migration file in the Migrations folder with the specified name.

Applying Migrations

- **Command to Apply Migrations:**

```
dotnet ef database update
```

- **Outcome:**

- Updates the database schema to match the current model.
- Creates a special table (__EFMigrationsHistory) to track which migrations have been applied.

Rolling Back Migrations

- **Command to Remove the Last Migration:**

```
dotnet ef migrations remove
```

- **Use Case:**

- If a migration introduces errors, you can roll it back to restore the previous state.

Conclusion

•Summary:

- Migrations are essential for managing database schema changes in ASP.NET Core MVC applications.
- They enable developers to evolve their data models while preserving existing data and maintaining application stability.

•Next Steps:

- Explore advanced migration scenarios such as seeding data and handling complex changes.

COMPLEX MODELS

Overview

- Expand on a simple data model with additional entities and relationships.
- Customize the data model with formatting, validation, and database mapping rules.

The ASP.NET Core logo is centered within a dark blue rectangular frame. The text "ASP.NET" is in white, and "Core" is in a lighter blue color. The "C" in "Core" is stylized with a blue dot in the center. The frame has white lines on the left and right sides, and a white line on the top right corner.

ASP.NET Core

Customizing the Book Model

Goal:

- Customize the **Book** model by specifying **formatting**, **validation**, and **database mapping** rules using attributes.

DataType and DisplayFormat for Date Fields

Purpose:

- Customize how date fields, like **PublicationDate**, are displayed (date only, no time).

```
using System;
using System.ComponentModel.DataAnnotations;

public class Book
{
    public int ID { get; set; }
    public string Title { get; set; }

    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    public DateTime PublicationDate { get; set; }

    public string Author { get; set; }
}
```



StringLength and Validation

Purpose:

Validate string input, like Title and Author, to ensure they aren't too long.

```
using System.ComponentModel.DataAnnotations;

public class Book
{
    public int ID { get; set; }

    [StringLength(100, ErrorMessage = "Title can't be longer than 100 characters.")]
    public string Title { get; set; }

    [StringLength(50, ErrorMessage = "Author name can't be longer than 50 characters.")]
    public string Author { get; set; }
}
```

Regular Expression for Validation

Purpose:

Enforce patterns for certain fields, such as ensuring Author names start with an uppercase letter.

```
using System.ComponentModel.DataAnnotations;


namespace LibraryManagement.Models
{
    12 references
    public class Book
    {
        9 references
        public int Id { get; set; }
        5 references
        public string NameBook { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$", ErrorMessage = "Author name must start with a capital letter.")]
        5 references
        public string Author { get; set; }
    }
}
```

Column Attribute for Database Mapping

Purpose:

- Map properties in the **Book** class to custom column names in the database. For instance, mapping **Title** to BookTitle.



```
[Column("BookTitle")]  
[StringLength(100)]  
0 references  
public string Title { get; set; }
```

The Required Attribute

Purpose:

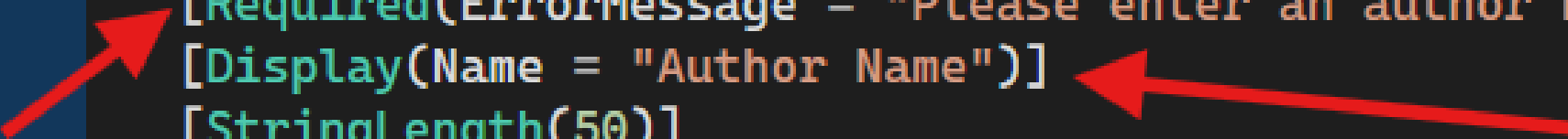
- Ensure that important fields, like Title and Author, are not left blank.

The Display Attribute

Purpose:

- Customize the label text that is shown on forms and views for properties like Title, Author.

```
[RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$", ErrorMessage = "Author name must start with a capital letter.")]
[Required(ErrorMessage = "Please enter an author name.")]
[Display(Name = "Author Name")]
[StringLength(50)]
5 references
public string Author { get; set; }
```



The Key Attribute

Purpose:

- The Key attribute is used to explicitly define a primary key in an entity when it's not following the convention (e.g., when the property name is not Id or ClassNameId).

The Key Attribute in the Book Model

Scenario:

- Suppose the Book model uses a non-standard field, like BookCode, as the primary key instead of the conventional ID or BookId.
- Key Attribute: Marks BookCode as the primary key.
- This is useful when your primary key field has a non-standard name.
- By default, ASP.NET assumes properties named Id or BookId to be the primary key without the need for the Key attribute.
- Use the Key attribute when your primary key does not follow this convention (e.g., using BookCode instead of ID).

```
[Key]
```

```
0 references
```

```
public string BookCode { get; set; }
```

The DatabaseGenerated attribute

The DatabaseGenerated attribute with the None parameter on the Id property specifies that primary key values are provided by the user rather than generated by the database.

```
[Key]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
9 references
public int Id { get; set; }
```

Foreign Key & Navigation Properties in Book Entity

Goal:

- Understand how to define foreign key and navigation properties to establish relationships between entities, using Book as the primary example.

Foreign Key Property - Author

One-to-Many Relationship:

- A Book is written by one author, so we define a foreign key and a navigation property for the Author.

- **AuthorID** is the foreign key.
- **Author** is the navigation property that enables navigation to the related **Author** entity.

```
12 references
public class Book
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    9 references
    public int Id { get; set; }

    // Foreign Key
    0 references
    public int AuthorID { get; set; }

    // Navigation Property
    5 references
    public Author Author { get; set; }
}
```


Foreign Key Property - Category

Many-to-One Relationship:

- A **Book** belongs to **one** category, but a **Category** can have **many** books.
- The CategoryID property is the foreign key.
- The Category property defines a one-to-many relationship between Category and Book.

```
12 references
public class Book
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    9 references
    public int Id { get; set; }

    // Foreign Key for Category
    0 references
    public int CategoryID { get; set; }

    // Navigation Property for Category
    0 references
    public Category Category { get; set; }
}
```

Foreign Key Property – Category

One-to-Many Relationship:

- A **Book** can be borrowed multiple times, so the navigation property for **Borrowings** is a collection.
- Borrowings is a collection that represents all instances when this book was borrowed.
- This defines a one-to-many relationship between Book and Borrowing.

```
12 references
public class Book
{
    [Key]
    9 references
    public int Id { get; set; }

    // Navigation Property for Borrowings
    0 references
    public ICollection<Borrowing> Borrowings { get; set; }
}
```

Foreign Key Property – Category

Many-to-Many Relationship

- A Book can belong to multiple Genres, and a Genre can have many Books.
- BookGenres defines a many-to-many relationship between Book and Genre, where BookGenre is a join entity.

```
12 references
public class Book
{
    [Key]
    9 references
    public int Id { get; set; }

    // Many-to-Many Relationship with Genres
    0 references
    public ICollection<BookGenre> BookGenres { get; set; }
}
```

Column Attribute in ASP.NET Core MVC

Learn how to use the **Column attribute** to customize the SQL data type mapping for properties in an entity.

Column Attribute for Custom SQL Data Types

Changing Data Type Mapping:

- The **Column attribute** can change how a property in the entity maps to a SQL Server data type.
- Example: Mapping a decimal property to the **money** type in SQL Server for a **Book** price.
- Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property

Why Use It?

- Normally, the Entity Framework automatically maps .NET data types to the appropriate SQL Server types.
- However, in some cases, like currency, you may prefer to explicitly map a property to a specific SQL data type (e.g., **money** instead of **decimal**).

Example - Mapping Book Price to SQL money Type

- Let's assume you have a **Book** entity that includes a **Price** property. You want to store **Price** using SQL Server's **money** data type.

```
12 references
public class Book
{
    [Key]
    9 references
    public int Id { get; set; }

    // Use Column attribute to map Price to SQL Server 'money' type
    [Column(TypeName = "money")]
    0 references
    public decimal Price { get; set; }
}
```

Applying Migrations

Purpose:

- Update the database to match the changes in the **Book** model.
- **Run these commands** to apply the changes such as StringLength, RegularExpression, and Column.

```
bash
```

```
dotnet ef migrations add CustomizeBookModel  
dotnet ef database update
```

*(Optional) Run this command to switch to the database if you are using code-first approach; otherwise, skip this command.

INHERITANCE

Implementing Inheritance in ASP.NET Core MVC with EF Core

- Learn how to model inheritance in the database using Entity Framework Core with ASP.NET Core MVC.

Why Use Inheritance?

- Inheritance helps organize related entities more efficiently by sharing common properties across different classes.
- Example: In a bookstore, both **EBook** and **PrintBook** can share properties like Title and Author, but have distinct attributes (e.g., **file size** for **EBook** and **weight** for **PrintBook**).

Key Inheritance Patterns:

- **Table-per-Hierarchy (TPH)**: One table is used for all entities in the hierarchy, with a discriminator column to differentiate types.
- **Table-per-Type (TPT)**: Each entity type has its own table, with relationships between the tables.

Table-per-Hierarchy (TPH) Inheritance Pattern

What is TPH?

- **TPH** stores all types in a single database table. It adds a **discriminator column** to specify the type of entity.

Example:

You have a base class **Book** with two derived types **EBook** and **PrintBook**.

Configuration in EF Core:

```
modelBuilder.Entity<EBook>().HasBaseType<Book>();  
modelBuilder.Entity<PrintBook>().HasBaseType<Book>();
```

This creates a single table **Books** in the database, with an additional discriminator column to distinguish between **EBook** and **PrintBook**.

```
0 references  
public class EBook : Book  
{  
    0 references  
    public double FileSizeMB { get; set; }  
}  
  
0 references  
public class PrintBook : Book  
{  
    0 references  
    public double WeightKg { get; set; }  
}
```

TPH Database Table Example

In TPH, the table will look like this:

BookID	Title	Author	FileSizeMB	WeightKg	Discriminator
1	Digital Book	John Smith	5.2	NULL	EBook
2	Paper Book	Jane Doe	NULL	0.75	PrintBook

Start your future at EIU

Thank You