

Hosting



...

ASP.NET Core Razor Pages : CRUD Operations with Repository Pattern and Entity Framework Core

...



Repository Pattern is one of the most popular patterns to create apps. It removed duplicate database operation codes and De-couples the application from the Data Access Layer. This gives added advantages to developers when they are creating the apps.

In this tutorial we will be creating **CRUD Operations in ASP.NET Core Razor Pages using Repository Pattern.**



The app which we will build will be in ASP.NET Core Razor Pages and there will be Entity Framework Core “ORM” on the data access layer.

Download the complete source code from my [GitHub Repository](#).

Page Contents

Repository Pattern

Repository Pattern's Simple Example

Benefits of Repository Pattern

Database Context and Entity Framework Core Migrations

Building the Repository Pattern

CRUD operations using Repository Pattern

Create Movie with Repository Pattern

Read Movies with Repository Pattern

Creating Pagination for Records

Update Movie with Repository Pattern

Update Movie with Repository Pattern

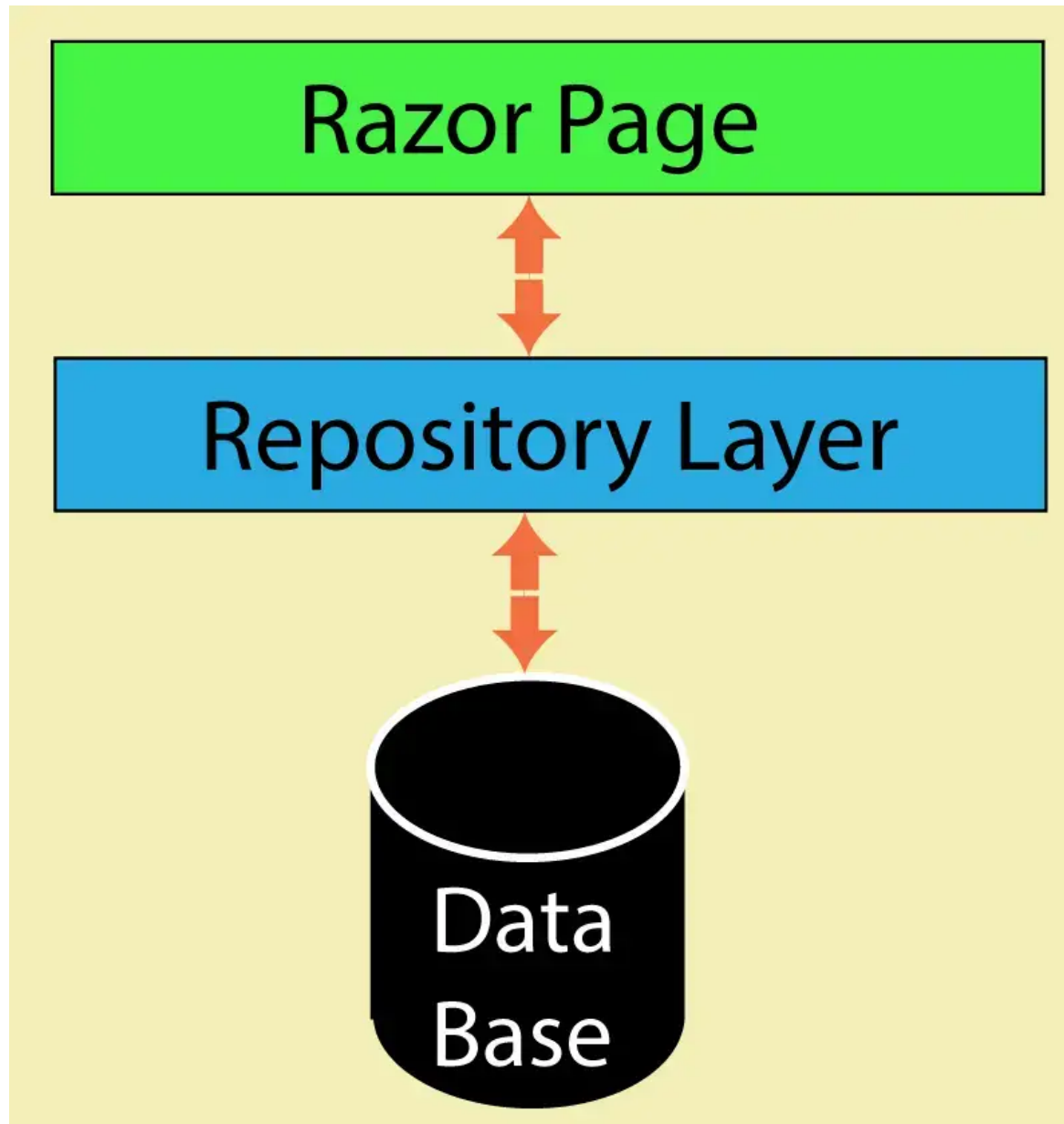
Delete Movie with Repository Pattern

Filtering Entity by LINQ Expression

Repository Pattern

Repository Pattern is a design pattern where we have a layer that separates the Domain Layer with the Data Access Layer. Domain will call the Repository on every database related work like Create, Read, Update and Delete operations. The Repository will then communicate with the Data Access Layer to perform the operation and return the result back to the domain layer.

See the below image which explains the architecture:



Note that Repository is made with an Interface and a Class which will

...



Repository Pattern's Simple Example

When there is no Repository Pattern then the database operation is performed straight from the Razor Page. See the below code where the Razor Page `OnPostAsync` method is using the database context class of Entity Framework Core and inserting a record on the database.

```
1 | public void OnPostAsync(Movie movie)
2 | {
3 |     context.Add(movie);
4 |     await context.SaveChangesAsync();
5 | }
```

and then the repository will perform the creation of the record on the database.

```
1 | public void OnPostAsync(Movie movie)
2 | {
3 |     repository.CreateAsync(movie);
4 | }
```

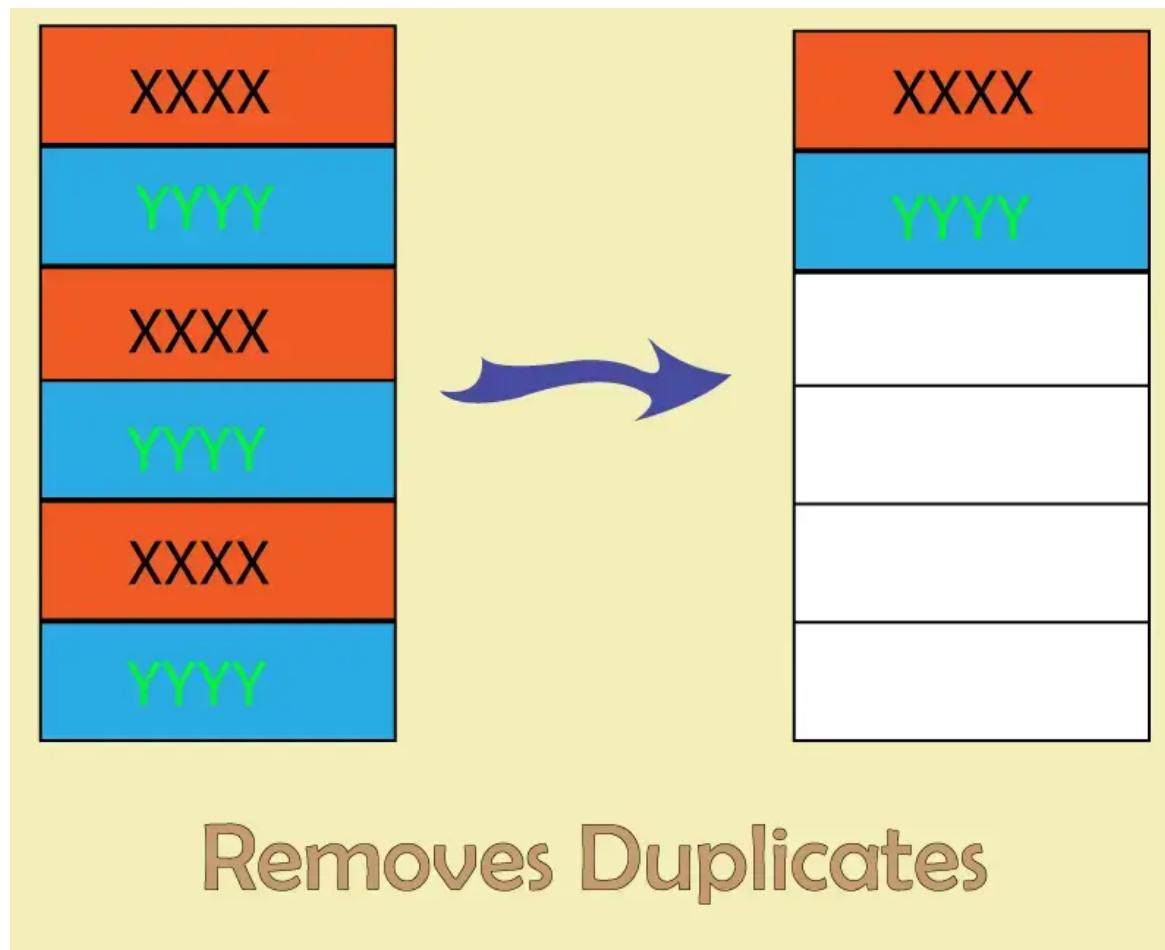
★ A quick note: I used Repository Pattern to create Microservices in ASP.NET Core, you can read it at [First ASP.NET Core Microservice with Web API CRUD Operations on a MongoDB database \[Clean Architecture\]](#).

Benefits of Repository Pattern

Repository Pattern offers many benefits which are given below:

Removes Duplicate Queries

An app will read data from database at multiple places, this will lead to redundant codes. With repository in place, you can simply call the repository which will provide you with the data. Got the point?



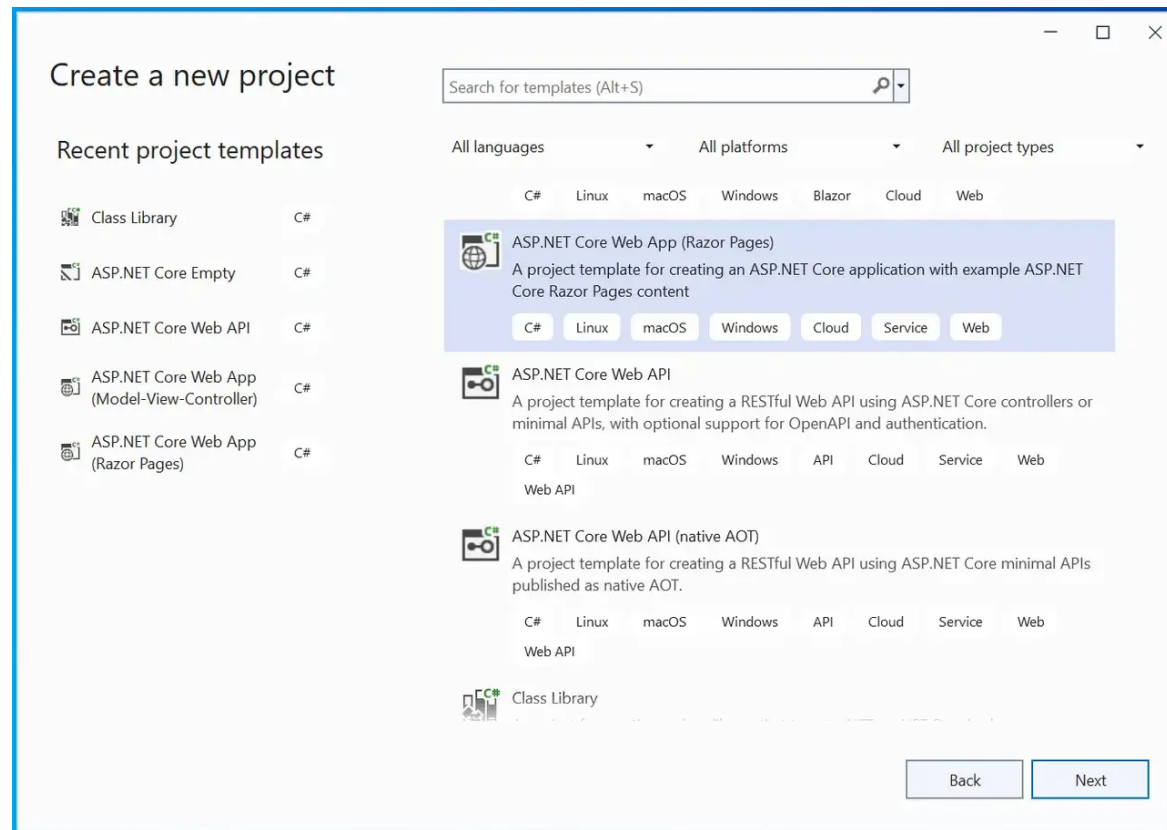
Loosely Coupled Architecture

Suppose after sometime you need to change the ORM from Entity Framework Core to Dapper. Repository Pattern will help you to achieve this quickly with minimum changes. You will only need to make changes to the Repository according to Dapper without doing any change to the business, domain, UI, Controller, Razor Page, etc.



Implementing Repository Pattern in ASP.NET Core Razor Pages

Let us now implement Repository Pattern, so create a new project in Visual Studio and select the ASP.NET Core Web App (Razor Pages) template as shown by the below image.



Give your app the name MovieCrud.

Configure your new project

ASP.NET Core Web App (Razor Pages)

C#

Linux

macOS

Windows

Cloud

Service

Project name

MovieCrud

Location

E:\Yogesh\SEO\Codes\Hosting

Solution name ⓘ

MovieCrud

☐

Place solution and project in the same directory

Next, select .NET 8.0 for framework and click Create button.

Additional information

ASP.NET Core Web App (Razor Pages) C# Linux macOS Windows Cloud Service Web

Framework ⓘ
.NET 8.0 (Long Term Support)

Authentication type ⓘ
None

☒ Configure for HTTPS ⓘ

☐ Enable Docker ⓘ

Docker OS ⓘ
Linux

☐ Do not use top-level statements ⓘ

Back Create

Withing a few second the Razor Pages app will be created and then you can add new features to it. I will start with installing Entity Framework Core to the app because it will be used as a database access layer to create CRUD features.

■ ■ ■

■ ■ ■



Entity Framework Core and Entities

Entity Framework Core is the ORM which we will use to communicate with the database and perform CRUD operations. Our Repository will be taking to Entity Framework Core so that these database operations can be performed. So, we will have to install following 3 packages from NuGet –



Microsoft.EntityFrameworkCore.SqlServer


Microsoft.EntityFrameworkCore.Design


Microsoft.EntityFrameworkCore.Tools


These packages can be installed from [Manage NuGet Packages for Solution](#) window which can be opened from Tools ► NuGet Package Manager ► Manage NuGet Packages for Solution.

Browse Installed Updates Consolidate

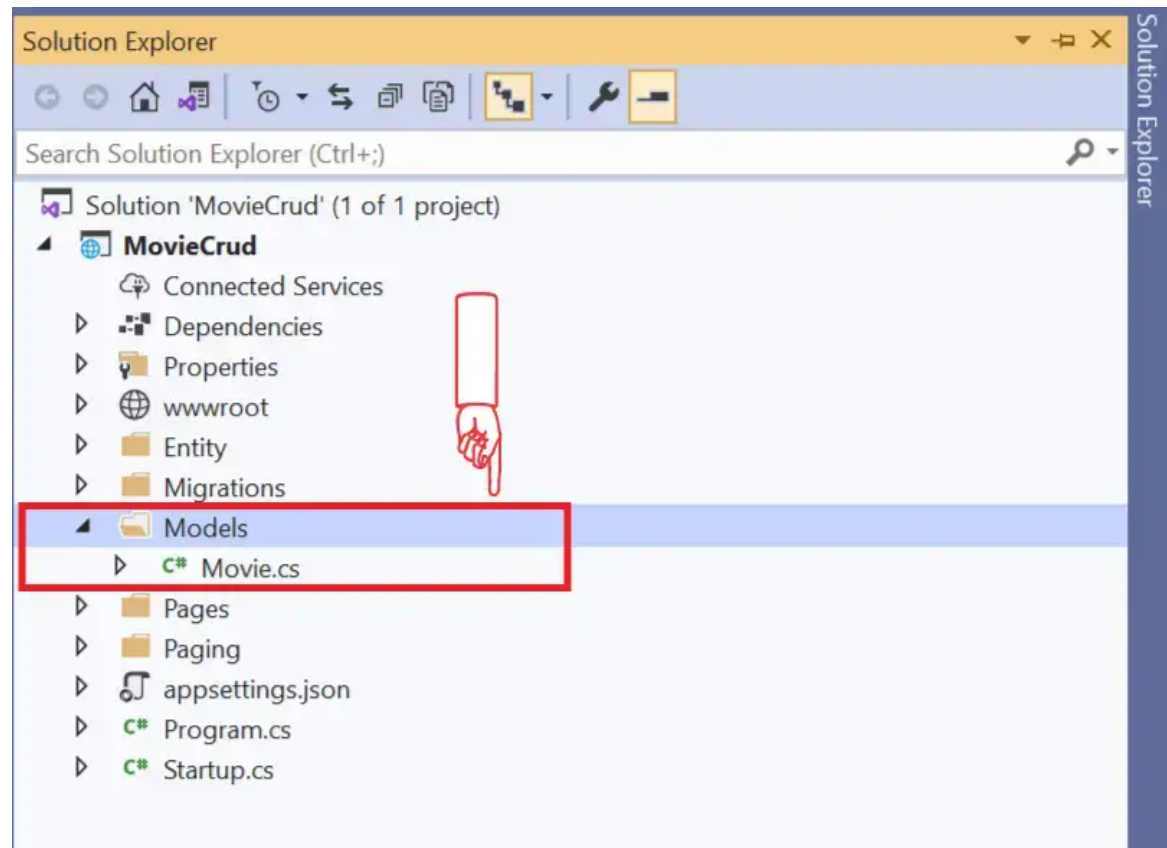
Search (Ctrl+L)   ☐ Include prerelease

 **Microsoft.EntityFrameworkCore.Design** by Microsoft
Shared design-time components for Entity Framework Core tools.

 **Microsoft.EntityFrameworkCore.SqlServer** by Microsoft
Microsoft SQL Server database provider for Entity Framework Core.

 **Microsoft.EntityFrameworkCore.Tools** by Microsoft
Entity Framework Core Tools for the NuGet Package Manager Console in V

Next, we will need to create the Entity which will be a Movie.cs class. You have to create this class inside the Models folder. So first create “Models” folder on the root of the app and then inside it create the Movie.cs class.



The code of the Movie.cs class is given below:

```
using System.ComponentModel.DataAnnotations;

namespace MovieCrud.Models
{
    public class Movie
    {
        public int Id { get; set; }
    }
}
```

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

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

Points to note:

1. We will be performing the CRUD operations for Movie records, so the Movie entity is created for this purpose.
2. It has 3 fields “Id, Name & Actors”. Name and Actors are made required by putting `[Required]` attribute over them. So movie records cannot have empty name and actors values.
3. EF Core will make the Name and Actors columns on the database when migration is performed. Both of them will be made `NOT NULL` types.
3. The Id field’s value is of type int. EF core migrations will create Id column on the database as primary key and of type `IDENTITY (1, 1)` `NOT NULL` . The id value will be autogenerated from 1 and incremented by the value of 1. Since the database itself will be adding int value for this column therefore there is no need to add `[Required]` attribute over the “Id” field.

Database Context and Entity Framework Core Migrations

Database context is a primary class which will interact with the database. We will need to create it inside the Models folder. So, create MovieContext .cs class to the Models folder with the following code:

■ ■ ■

```
1 using Microsoft.EntityFrameworkCore;
2
3 namespace MovieCrud.Models
4 {
5     public class MovieContext : DbContext
6     {
```

```
10 |  
11 |         public DbSet<Movie> Movie { get; set; }  
12 |     }  
13 | }
```

The MovieContext inherits the DbContext class of Microsoft.EntityFrameworkCore namespace. It has the object DbContextOptions<MovieContext> defined on it's constructor as a dependency. ASP.NET Core dependency injection will provide this object to it.

... We have also defined all the entities where EF Core will use DbSet on this class. Since we only have the Movie entity so it is defined as DbSet<Movie> .

...

Next, we need to register the MovieContext on the Program.cs. So add the following highlighted line which will register our database context.

```
1  using Microsoft.EntityFrameworkCore;
2  using MovieCrud.Models;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  // Add services to the container.
7  builder.Services.AddDbContext<MovieContext>(options =>
8      options.UseSqlServer(builder.Configuration.GetCo
9  builder.Services.AddRazorPages());
10
11  var app = builder.Build();
12
13  // Configure the HTTP request pipeline.
14  if (!app.Environment.IsDevelopment())
15  {
16      app.UseExceptionHandler("/Error");
17      // The default HSTS value is 30 days. You may want t
18      app.UseHsts();
19  }
20
21  app.UseHttpsRedirection();
22  app.UseStaticFiles();
23
24  app.UseRouting();
25
26  app.UseAuthorization();
27
28  app.MapRazorPages();
29
30  app.Run();
```

appsettings.json file as shown below:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=MovieDb;Trusted_Connection=True;Mu
ltipleActiveResultSets=true"
  }
}
```

I am using mssqllocaldb and the database name as "MovieDb".

So let us perform the migrations. Go to Tools ► NuGet Package Manager ► Package Manager Console, and enter the following commands one by one.

```
add-migration Migration1
Update-Database
```

Note: Enter the first command and press enter to execute it. Then enter the second command and press enter to execute it.

After the Migration commands finish executing you will find a Migrations folder created on the app. This folder contains newly created files that were used to create the database.

Next, go to View ► SQL Server Object Explorer where you will find the MovieDb database. This database is created when we ran the migrations. This database will contain only one table called Movie which corresponds to the Movie entity we created in our ASP.NET Core Razor Pages app.

If you can't see the database then click the refresh icon on the SQL Server Object Explorer window.

Right click on the Movie table and select View Code. This will show you the tables definition.

```
5 | CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([Id] ASC  
6 | );
```

As discussed earlier all columns are NOT NULL type and the Id column is defined as IDENTITY (1, 1).

Building the Repository Pattern

Repository Pattern needs 2 things:

1. Interface
2. Class which will implement/inherit the interface.

The interface will contain all the methods that will be dealing with database operations like CRUD. Commonly you will have methods for

Sometimes we will have a few more methods for dealing with filtering of records. We will see them in the latter sections.

■ ■ ■



Related tutorial – I have also covered [Onion Architecture in a very detailed and illustrative post, see Implementing Onion Architecture in ASP.NET Core with CQRS](#).

Anyway for now create a new folder called Entity on the root of the app. Then inside it, add a new class called IRepository.cs, this class will define an Interface which we just talked about. So add the following code to this class


```
namespace MovieCrud.Entity
{
    public interface IRepository<T> where T : class
    {
        Task CreateAsync(T entity);
    }
}
```

Things to note:

The IRepository<T> is a generic type where T is constrained as a class. This mean T must be a reference type like a class, interface, delegate, or array type..

I have defined a single asynchronous method called CreateAsync in the interface – Task CreateAsync(T entity) .

Next, we create a new class called Repository.cs inside the same “Entity” folder. This class will implement the interface we defined earlier. So add the below given code to the class.

```
using MovieCrud.Models;

namespace MovieCrud.Entity
{
    public class Repository<T> : IRepository<T> where T : class
    {

```

```
public Repository(MovieContext context)
{
    this.context = context;
}

public async Task CreateAsync(T entity)
{
    if (entity == null)
        throw new
ArgumentNullException(nameof(entity));

    context.Add(entity);
    await context.SaveChangesAsync();
}
}
```

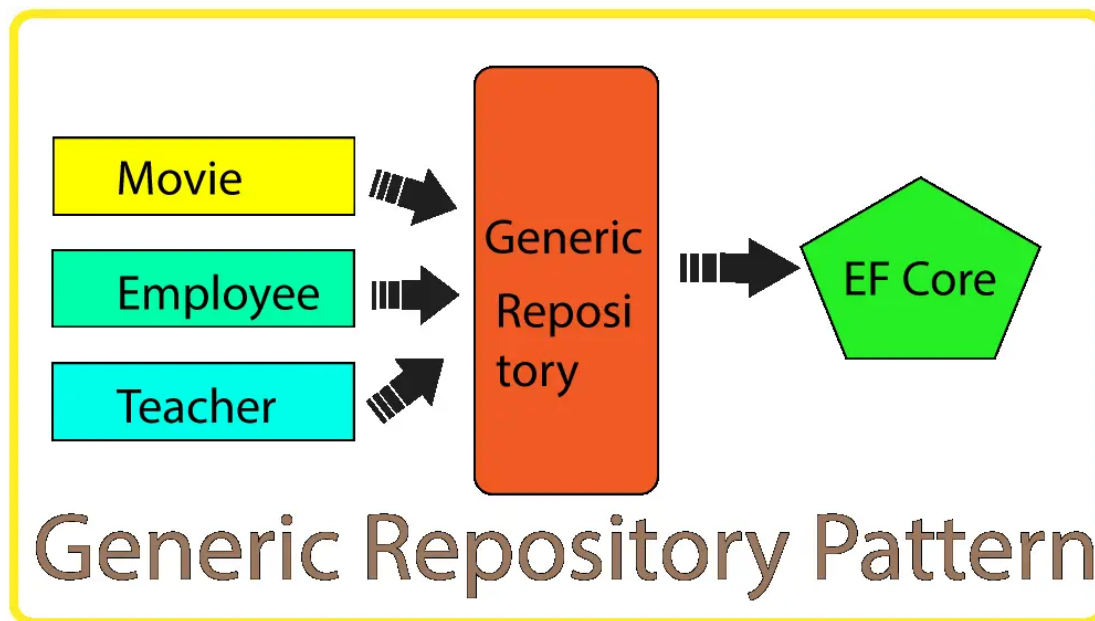
Points to note:

Point 1 – The constructor of the class receives the “MovieContext” object, which is the database context of EF core, in its parameter. The ASP.NET Core dependency injection engine will provide this object to the class.

```
public Repository(MovieContext context)
{
```

Point 2 – The CreateAsync method is doing the insertion of the record to the database. It gets the entity of type “T” in it’s parameter and inserts this entity to the database using Entity Framework Core.

Point 3 – The “T” type gives us a great benefit because we it helps to extend the generic repository as we can insert any entity to the database not just the “Movie”. We can simply add more entities like “Employee”, “School”, “Teacher” and the same repository Repository.cs will perform the insertion of that entity without needing any code addition to the Repository. This is the power of Generics.



So, in short, we define a new entity called employee in an Employee.cs class. This employee entity can then be added to the database by the

```
1 public class Employee
2 {
3     public int Id { get; set; }
4
5     [Required]
6     public string Name { get; set; }
7
8     [Required]
9     public int Salary { get; set; }
10
11    [Required]
12    public string Address { get; set; }
13 }
```



Our repository pattern should be more likely be called as **Generic Repository Pattern**. Generic Repository Pattern is much more powerful than simple Repository Pattern.

...

Point 4 – The code `context.Add(entity)` keeps the track of this new entity. Then the code `context.SaveChangesAsync()` creates the record of this entity in the database. The entity has a default value of 0 for the Id. Entity Framework Core knows that since Id value is 0 therefore it has to create a new record on the database. Had the id value not 0 but in some positive integer value, then EF core would perform updation of the record for this Id value.

Finally, we need to register the Interface to its implementation in the Program.cs. The code that has to be added is shown below.

```
builder.Services.AddTransient(typeof(IRepository<>),  
    typeof(Repository<>));
```

The above code tells ASP.NET Core to provide a new instance of Repository whenever a dependency of IRepository is present. The typeof specifies that the type can be anything like Movie, Teacher, Employee, etc.

Great, we just completed building our Generic Repository Pattern. All we are left is creating the CRUD Operations in Razor Pages.

CRUD operations using Repository Pattern

Create Movie with Repository Pattern

The first **CRUD operation is the Create Record** operation. We already have added the `CreateAsync` method to the `IRepository<T>` interface:

```
Task CreateAsync(T entity);
```

We have also added it's implementation to the `Repository.cs` class:

```
1 public async Task CreateAsync(T entity)
2 {
3     if (entity == null)
4         throw new ArgumentNullException(nameof(entity));
5
6     context.Add(entity);
7     await context.SaveChangesAsync();
8 }
```

There is Pages folder on the app's root folder. You need to create a new Razor Page inside this Pages folder and call it `Create.cshtml`.

So, right click the Pages folder and select Add ► New Item.

Next, on the Add New Item window, select Razor Page – Empty template and name it as Create.cshtml.



The razor page will be created and opened in Visual Studio. Delete it's initial code and add the following code to it.

■ ■ ■

```
1  @page
2  @model CreateModel
3  @using Microsoft.AspNetCore.Mvc.RazorPages;
4  @using MovieCrud.Entity;
5  @using Models;
6
7  @{
8      ViewData["Title"] = "Create a Movie";
9  }
10
11 <h1 class="bg-info text-white">Create a Movie</h1>
```

```

16         <label asp-for="@Model.movie.Name"></label>
17         <input type="text" asp-for="@Model.movie.Name"
18         <span asp-validation-for="@Model.movie.Name"
19     </div>
20     <div class="form-group">
21         <label asp-for="@Model.movie.actors"></label>
22         <input type="text" asp-for="@Model.movie.actors"
23         <span asp-validation-for="@Model.movie.actors"
24     </div>
25     <button type="submit" class="btn btn-primary">Create
26 </form>
27
28 @functions {
29     public class CreateModel : PageModel
30     {
31         private readonly IRepository<Movie> repository;
32         public CreateModel(IRepository<Movie> repository)
33         {
34             this.repository = repository;
35         }
36
37         [BindProperty]
38         public Movie movie { get; set; }
39
40         public IActionResult OnGet()
41         {
42             return Page();
43         }
44
45         public async Task<IActionResult> OnPostAsync()
46         {
47             if (ModelState.IsValid)
48                 await repository.CreateAsync(movie);
49             return Page();
50         }
51     }

```

Understanding the code: – The page has both razor and C# codes. The top part contains razor code while C# codes are placed inside the functions body. The `@page` applied on the top specify that this is a Razor Page and can handle http requests.

Now run Visual Studio and open this page on the browser by it's name, i.e. the url of this page will be – <https://localhost:44329/Create>. The localhost port will be different for you.

I would also like you to check the `_ViewStart.cshtml` located inside the “Pages” folder. It contains code that is executed at the start of each Razor Page’s execution. Double click this file and see that it specifies the layout to be `_Layout` for the razor pages.

■ ■ ■

```
@{  
    Layout = "_Layout";  
}
```



You will find the layout of the razor pages called `_Layout.cshtml` inside the Pages ► Shared folder. Open this file to find html and razor codes

An important thing to note is the `@RenderBody()` code. It renders all the content of the Razor Pages. So this means the Create.cshtml Razor Page will be rendered by `@RenderBody()`, it will have header on the top and footer on the bottom. The header and footer are defined on the layout.

■ ■ ■

Next, there is a model defined for the Create.cshtml Razor Page and it is named “CreateModel”.

```
@model CreateModel
```

On the functions block, the class by the same model is defined. The class inherits the PageModel abstract class.

```
@functions {  
    public class CreateModel : PageModel  
    {  
        ...  
    }  
}
```

The constructor of the CreateModel class has a dependency for `IRepository<Movie>` .

```
private readonly IRepository<Movie> repository;  
public CreateModel(IRepository<Movie> repository)  
{  
    this.repository = repository;  
}
```

The dependency injection feature solves this by providing the `Repository<Movie>` object to the constructor. Recall we earlier registered this dependency in the program class:

■ ■ ■

```
services.AddTransient(typeof(IRepository<>),
typeof(Repository<>));
```

Next see Movie type property defined and it has [BindProperty] attribute. This means this property will get the value from the html elements defined on the form, when the form is submitted. This type of binding is done automatically by ASP.NET Core through Model Binding feature.

```
[BindProperty]
public Movie movie { get; set; }
```

We defined 2 input type text elements on the form which are binding with the "Name" and "Actors" properties of the Movie type object.

```
1  <form method="post">
2    <div class="form-group">
3      <label asp-for="@Model.movie.Name"></label>
4      <input type="text" asp-for="@Model.movie.Name"
5      <span asp-validation-for="@Model.movie.Name"
6    </div>
7    <div class="form-group">
8      <label asp-for="@Model.movie.Actors"></label>
9      <input type="text" asp-for="@Model.movie.Actors"
10     <span asp-validation-for="@Model.movie.Actors"
11    </div>
12     <button type="submit" class="btn btn-primary">Cre
13  </form>
```



```
<input type="text" asp-for="@Model.movie.Name" class="form-control" />  
<input type="text" asp-for="@Model.movie.actors" class="form-control" />
```

Tag helpers enable the server-side code to create and render HTML elements. The `asp-for` tag helper adds 4 HTML attributes to the input elements. You can see these attributes by “Inspecting” them on the browser. Right click on the browser and select Inspect.

A new window which will show you the HTML source of the page. Now



Then move your mouse over the name input element, this will highlight it in dark background color. Click it so that it gets selected. Check below image.



Now the below window will show the html code of the name input element in highlighted color. See the below image where I have marked it.

■ ■ ■



The html codes of both the Name and Actors input elements are given below.

...



```
<input type="text" class="form-control input-validation-
error" data-val="true" data-val-required="The Name field is
required." id="movie_Name" name="movie.Name" value="">
```

```
<input type="text" class="form-control" data-val="true" data-
val-required="The Actors field is required."
id="movie_Actors" name="movie.Actors" value="">
```

The 4 added attributes which you can clearly see are:

```
data-val="true"
data-val-required="The Name/Actors field is required."
id="movie_Name" or id="movie_Actors"
name="movie.Name" or name="movie.Actors"
```

The name attribute is provided with the value of movie.Name. It helps to bind the value with the object's property with Model Binding. The id field is also binded similarly except that there is '_' in place of '.', the id attribute is used in Client Side Validation which is done with jQuery Validation and jQuery Unobtrusive Validation. I will not be covering client side validation in this tutorial.

★ The `asp-for` tag helper on the label generates "for" attribute

is **Name**, notice the 'for' attribute value is equal to the 'id' of the input field.

■ ■ ■

■ ■ ■

■ ■ ■

The `_ViewImports.cshtml` inside the “Pages” folder is used to make directives available to Razor pages globally so that you don’t have to add them to Razor pages individually. If you open it then you will find that the Tag Helpers are also imported there with the following code.

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Model Validation

The `data-val` and `data-val-required` creates the model validation features. Let us understand them in details.

The `data-val` with true value flags the field as being subject to

is the same what we gave to the Name and Actors fields of the Movie class.

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

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

```
<span asp-validation-for="@Model.movie.Name" class="text-  
danger"></span>  
<span asp-validation-for="@Model.movie.actors" class="text-  
danger"></span>
```

Notice also the asp-validation-for tag helper applied on the 2 span element to display the validation errors for the Name and Actors fields. So, when validation fails, the 2 span will display the values of the data-val-required attributes of the input elements. This is the way Validation works in ASP.NET Core Razor Pages.



I have also added a div before the form on the razor page. This div will show all the validation errors together. This is done by adding the tag helper `asp-validation-summary="All"` to the div.

```
<div asp-validation-summary="All" class="text-danger"></div>
```

The razor pages have Handler methods that are automatically executed as a result of a http request. There are 2 handler methods – OnGet and OnPostAsync. The “OnGet” will be called when http type get request is made to the Create Razor Page while the “OnPostAsync”, which is async version of OnPost, is called when http type post request is made to the Create Razor Page.

```
1 public IActionResult OnGet()  
2 {  
3     return Page();  
4 }  
5  
6 public async Task<IActionResult> OnPostAsync()  
7 {  
8     if (ModelState.IsValid)  
9         await repository.CreateAsync(movie);  
10    return Page();
```

The `onGet()` handler simply renders back the razor page. The `OnPostAsync()` handler checks if model state is valid.

...

```
if (ModelState.IsValid)
```

It then calls the repository's `CreateAsync` method. This `CreateAsync` method is provided with the movie object so that it is inserted to the database.

```
await repository.CreateAsync(movie);
```

Note that the `ModelState.IsValid` is true only when there are no

Well, that's all said, let us see how Model Binding and Validation works. So run visual studio and go to the url of the Create razor page – <https://localhost:44329/Create>. Without filling any of the text boxes, click the Create button to submit the form. You will see validation error messages in red color.

Now fill the values of Name and Actors and then submit the form once



Now on the SQL Server Object Explorer, right click on the Movie table and select View Data.

...



You will see the record is inserted on the database.



Congratulations, we successfully inserted the record with **Generic Repository Pattern**. Next we will see the Reading part.

...



Read Movies with Repository Pattern

Now we will build the Read Movie functionality. So, we need to add a new method to our repository which will perform the reading task. So go to the IRepository interface and add ReadAllAsync method as shown below:

```
1 public interface IRepository<T> where T : class
2 {
3     Task CreateAsync(T entity);
```

The ReadAllAsync method returns a list of T types asynchronously. Next, we will have to implement this method on the “Repository.cs” class. The code is shown below in highlighted manner.

```
1  using Microsoft.EntityFrameworkCore;
2  using MovieCrud.Models;
3
4  namespace MovieCrud.Entity
5  {
6      public class Repository<T> : IRepository<T> where T : class
7      {
8          private MovieContext context;
9
10         public Repository(MovieContext context)
11         {
12             this.context = context;
13         }
14
15         public async Task CreateAsync(T entity)
16         {
17             if (entity == null)
18                 throw new ArgumentNullException(nameof(entity));
19
20             context.Add(entity);
21             await context.SaveChangesAsync();
22         }
23
24         public async Task<List<T>> ReadAllAsync()
25         {
26             return await context.Set<T>().ToListAsync();
27         }
28     }
29 }
```

the database. In general term it will read the “T” entity from the database and then we convert it to a list by from the `ToListAsync()` method.

Moving to the Razor page, create a new Razor Page inside the “Pages” folder and name is Read.cshtml and add the following code to it.

```
1  @page
2  @model ReadModel
3  @using Microsoft.AspNetCore.Mvc.RazorPages;
4  @using MovieCrud.Entity;
5  @using Models;
6
7  @{
8      ViewData["Title"] = "Movies";
9  }
10
11 <h1 class="bg-info text-white">Movies</h1>
12 <a asp-page="Create" class="btn btn-secondary">Create</a>
13
14 <table class="table table-sm table-bordered">
15     <tr>
16         <th>Id</th>
17         <th>Name</th>
18         <th>Actors</th>
19     </tr>
20     @foreach (Movie m in Model.movieList)
21     {
22         <tr>
23             <td>@m.Id</td>
24             <td>@m.Name</td>
25             <td>@m.Actors</td>
26         </tr>
```

```
31     public class ReadModel : PageModel
32     {
33         private readonly IRepository<Movie> repository;
34         public ReadModel(IRepository<Movie> repository)
35         {
36             this.repository = repository;
37         }
38
39         public List<Movie> movieList { get; set; }
40
41         public async Task OnGet()
42         {
43             movieList = await repository.ReadAllAsync();
44         }
45     }
46 }
```

The code of this Razor Page is quite simple, firstly I have created a link to the “Create” Razor Page with the help of `asp-page` tag helper.

■ ■ ■



```
<a asp-page="Create" class="btn btn-secondary">Create a  
Movie</a>
```

Next we have an HTML table which contains 3 columns for the 3 fields of the Movie entity. The table will be showing all the Movies which are currently stored in the database. A `List<Movie>` property is defined on the functions block, it is provided with the list of movie by the `OnGet()` method.

```
1 public List<Movie> movieList { get; set; }  
2  
3 public async Task OnGet()  
4 {  
5     movieList = await repository.ReadAllAsync();  
6 }
```

The html table has a foreach loop which structures each record per table row.

```
1 @foreach (Movie m in Model.movieList)  
2 {  
3     <tr>  
4         <td>@m.Id</td>  
5         <td>@m.Name</td>  
6         <td>@m.Actors</td>  
7     </tr>  
8 }
```

Since the Read Page is created, so we should redirect user to the Read Page when a new Record is created. So go to Create.html page and

the Read Page. The redirection is done by the `RedirectToPage("Read")` method. See the change which is shown in highlighted manner.

■ ■ ■

```
1 public async Task<IActionResult> OnPostAsync()  
2 {  
3     if (ModelState.IsValid)  
4         await repository.CreateAsync(movie);  
5     return RedirectToPage("Read");  
6 }
```

Let us test this page, run visual studio and navigate to <https://localhost:44329/Read> where you will be shown the Movie Record created earlier (see below image). This shown the Read

Creating Pagination for Records

We have created the Reading of the records but there is no paging. I created a few more movies which will show together on the page.

This will also cause problem like slowing the page when the number of records increases. To solve this problem, we will create Pagination feature. So, create a new folder called Paging on the root of the app. To this folder add 3 classes which are:

■ ■ ■



PagingInfo.cs – keep track total records, current page, items per page and total pages.

MovieList.cs – contains the list of records of the current page and PagingInfo.

PageLinkTagHelper.cs – it is a tag helper which will build the paging links inside a div.

PagingInfo.cs code is:

```
1 | namespace MovieCrud.Paging
2 | {
```

```
7      public int CurrentPage { get; set; }
8      public int TotalPages
9      {
10         get
11         {
12             return (int)Math.Ceiling((decimal>Total
13                 ItemsPerPage);
14         }
15     }
16 }
17 }
```

MovieList.cs code is:



```

5 |     public class MovieList
6 |     {
7 |         public IEnumerable<Movie> movie { get; set; }
8 |         public PagingInfo pagingInfo { get; set; }
9 |     }
10| }

```

PageLinkTagHelper.cs code is:

```

1 | using Microsoft.AspNetCore.Mvc.Rendering;
2 | using Microsoft.AspNetCore.Mvc.Routing;
3 | using Microsoft.AspNetCore.Mvc.ViewFeatures;
4 | using Microsoft.AspNetCore.Mvc;
5 | using Microsoft.AspNetCore.Razor.TagHelpers;
6 | using System.Dynamic;
7 |
8 | namespace MovieCrud.Paging
9 | {
10|     [HtmlTargetElement("div", Attributes = "page-model")]
11|     public class PageLinkTagHelper : TagHelper
12|     {
13|         private IUrlHelperFactory urlHelperFactory;
14|
15|         public PageLinkTagHelper(IUrlHelperFactory urlHelperFactory)
16|         {
17|             urlHelperFactory = urlHelperFactory;
18|         }
19|
20|         [ViewContext]
21|         [HtmlAttributeNotBound]
22|         public ViewContext ViewContext { get; set; }
23|
24|         public PagingInfo PageModel { get; set; }
25|
26|         public string PageName { get; set; }

```

```

30         public Dictionary<string, object> PageOtherValues { get; set; }
31
32         public bool PageClassesEnabled { get; set; }
33
34         public string PageClass { get; set; }
35
36         public string PageClassSelected { get; set; }
37
38         public override void Process(TagHelperContext context, TagHelperOutput output)
39         {
40             IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(context);
41             TagBuilder result = new TagBuilder("div");
42             string anchorInnerHtml = "";
43
44             for (int i = 1; i <= PageModel.TotalPages; i++)
45             {
46                 TagBuilder tag = new TagBuilder("a");
47                 anchorInnerHtml = AnchorInnerHtml(i);
48
49                 if (anchorInnerHtml == "..")
50                     tag.Attributes["href"] = "#";
51                 else if (PageOtherValues.Keys.Count > 0)
52                     tag.Attributes["href"] = urlHelper.Action("Index", "Home", new { page = i });
53                 else
54                     tag.Attributes["href"] = urlHelper.Action("Index", "Home", new { page = i });
55
56                 if (PageClassesEnabled)
57                 {
58                     tag.AddCssClass(PageClass);
59                     tag.AddCssClass(i == PageModel.CurrentPage ? "active" : "");
60                 }
61                 tag.InnerHtml.Append(anchorInnerHtml);
62                 if (anchorInnerHtml != "")
63                     result.InnerHtml.AppendHtml(tag);
64             }
65             output.Content.AppendHtml(result.InnerHtml);

```

```

69         {
70             object routeValues = null;
71             var dict = (routeValues != null) ? new Dictionary<string, object>() : null;
72             dict.Add("id", i);
73             foreach (string key in PageOtherValues.Keys)
74             {
75                 dict.Add(key, PageOtherValues[key]);
76             }
77
78             var expandoObject = new ExpandoObject();
79             var expandoDictionary = (IDictionary<string, object>)expandoObject;
80             foreach (var keyValuePair in dict)
81             {
82                 expandoDictionary.Add(keyValuePair);
83             }
84
85             return expandoDictionary;
86         }
87
88         public static string AnchorInnerHtml(int i,
89         {
90             string anchorInnerHtml = "";
91             if (pagingInfo.TotalPages <= 10)
92                 anchorInnerHtml = i.ToString();
93             else
94             {
95                 if (pagingInfo.CurrentPage <= 5)
96                 {
97                     if ((i <= 8) || (i == pagingInfo.TotalPages))
98                         anchorInnerHtml = i.ToString();
99                     else if (i == pagingInfo.TotalPages)
100                         anchorInnerHtml = "..";
101                 }
102                 else if ((pagingInfo.CurrentPage > 5) || (i == 1))
103                 {
104                     if ((i == 1) || (i == pagingInfo.TotalPages))

```

```

108         }
109         else if (pagingInfo.TotalPages - pa
110         {
111             if ((i == 1) || (pagingInfo.Tota
112                 anchorInnerHtml = i.ToString
113             else if (pagingInfo.TotalPages .
114                 anchorInnerHtml = "...";
115         }
116     }
117     return anchorInnerHtml;
118 }
119 }
120 }

```

The PageLinkTagHelper does the main work of creating paging links. Let me explain how it works.

The tag helpers must inherit TagHelper class and should override the Process function. The process function is the place where we write our tag helper code. Here in our case we will be creating anchor tags for the paging links and show them inside a div.

The tag helper class is applied with HtmlTargetElement attribute which specifies that it will apply to any div which has page-model attribute.

```
[HtmlTargetElement("div", Attributes = "page-model")]
```

The tag helper class has defined a number of properties which will receive the value from the Razor Page. These properties are PageModel, PageName, PageClassesEnabled, PageClass and

There is another property of type `ViewContext` which is binded with the View Context Data which includes routing data, `ViewData`, `ViewBag`, `TempData`, `ModelState`, current HTTP request, etc.

```
1 | [ViewContext]
2 | [HtmlAttributeNotBound]
3 | public ViewContext ViewContext { get; set; }
```

The use of `[HtmlAttributeNotBound]` attribute basically says that this attribute isn't one that you intend to set via a tag helper attribute in the razor page.

The tag helper gets the object of `IUrlHelperFactory` from the dependency injection feature and uses it to create the paging anchor tags.

```
IUrlHelper urlHelper =
urlHelperFactory.GetUrlHelper(ViewContext);
```

There is also a function called [AnchorInnerHtml](#) whose work is to create the text for the paging links. The next thing we have to do is to make this tag helper available to the razor pages, which we can do by adding the below given code line inside the `_ViewImports.cshtml` file.

```
@addTagHelper MovieCrud.Paging.*, MovieCrud
```

The `@addTagHelper` directive makes Tag Helpers available to the Razor

so this means to load all tag helper that have MovieCrud.Paging namespace or any namespace that starts with MovieCrud.Paging like:

```
MovieCrud.Paging.CustomCode
MovieCrud.Paging.Abc
MovieCrud.Paging.Secret
MovieCrud.Paging.Something
...
```

And the second parameter “MovieCrud” specifies the assembly containing the Tag Helpers. This is the name of the app.

Next, we need to integrate this tag helper on the Read Razor Page. So first we need to add new method to our repository. Add method called ReadAllFilterAsync to the IRepository. See the highlighted code below:

```
1 public interface IRepository<T> where T : class
2 {
3     Task CreateAsync(T entity);
4     Task<List<T>> ReadAllAsync();
5     Task<(List<T>, int)> ReadAllFilterAsync(int skip,
6     }
```

This method returns a Tuple of type `List<T>` and `int`. This obviously means it will return a list of records of the current page and total number of records in the database. Other than that, it takes 2 parameters – skip and take, they help us to build the logic to fetch only

Next add the implementation of this method on the Repository.cs class as shown below.

```

1  using Microsoft.EntityFrameworkCore;
2  using MovieCrud.Models;
3
4  namespace MovieCrud.Entity
5  {
6      public class Repository<T> : IRepository<T> where T : class
7      {
8          private MovieContext context;
9
10         public Repository(MovieContext context)
11         {
12             this.context = context;
13         }
14
15         public async Task CreateAsync(T entity)
16         {
17             if (entity == null)
18                 throw new ArgumentNullException(nameof(entity));
19
20             context.Add(entity);
21             await context.SaveChangesAsync();
22         }
23
24         public async Task<List<T>> ReadAllAsync()
25         {
26             return await context.Set<T>().ToListAsync();
27         }
28
29         public async Task<(List<T>, int)> ReadAllFilteredAsync()
30         {
31             var all = context.Set<T>();
32             var relevant = await all.Skip(skin).Take(skin).ToListAsync();

```

```

36 |
37 |         return result;
38 |     }
39 | }
40 |

```

See that now we are fetching only the records of the current page by the use of Linq Skip and Take methods.

```
var relevant = await all.Skip(skip).Take(take).ToListAsync();
```

Then returning the records along with the count of all the records in a Tuple.

```
(List<T>, int) result = (relevant, total);
```

Finally, go to Read.cshtml and do the changes which are shown below in highlighted way.

```

1 | @page "{id:int?}"
2 | @model ReadModel
3 | @using Microsoft.AspNetCore.Mvc.RazorPages;
4 | @using MovieCrud.Entity;
5 | @using Models;
6 | @using Paging;
7 |
8 | @{
9 |     ViewData["Title"] = "Movies";
10 | }
11 |
12 | <style>
13 |

```

```

17         .pagingDiv > a {
18             display: inline-block;
19             padding: 0px 9px;
20             margin-right: 4px;
21             border-radius: 3px;
22             border: solid 1px #c0c0c0;
23             background: #e9e9e9;
24             box-shadow: inset 0px 1px 0px rgba(255,255,255,0.5);
25             font-size: .875em;
26             font-weight: bold;
27             text-decoration: none;
28             color: #717171;
29             text-shadow: 0px 1px 0px rgba(255,255,255,0.5);
30         }
31
32         .pagingDiv > a:hover {
33             background: #fefefe;
34             background: -webkit-gradient(linear,
35             background: -moz-linear-gradient(0% 0% 0% 0%);
36         }
37
38         .pagingDiv > a.active {
39             border: none;
40             background: #616161;
41             box-shadow: inset 0px 0px 8px rgba(0,0,0,0.5);
42             color: #f0f0f0;
43             text-shadow: 0px 0px 3px rgba(0,0,0,0.5);
44         }
45     </style>
46
47     <h1 class="bg-info text-white">Movies</h1>
48     <a asp-page="Create" class="btn btn-secondary">Create</a>
49
50     <table class="table table-sm table-bordered">
51         <tr>
52             <th>Id</th>

```

```

56     @foreach (Movie m in Model.movieList.movie)
57     {
58         <tr>
59             <td>@m.Id</td>
60             <td>@m.Name</td>
61             <td>@m.Actors</td>
62         </tr>
63     }
64 </table>
65
66 <div class="pagingDiv" page-model="Model.movieList.pa
67
68 @functions{
69     public class ReadModel : PageModel
70     {
71         private readonly IRepository<Movie> repository
72         public ReadModel(IRepository<Movie> repository
73         {
74             this.repository = repository;
75         }
76
77         public MovieList movieList { get; set; }
78
79         public async Task OnGet(int id)
80         {
81             movieList = new MovieList();
82
83             int pageSize = 3;
84             PagingInfo pagingInfo = new PagingInfo();
85             pagingInfo.CurrentPage = id == 0 ? 1 : id
86             pagingInfo.ItemsPerPage = pageSize;
87
88             var skip = pageSize * (Convert.ToInt32(id
89             var resultTuple = await repository.ReadAl
90
91             pagingInfo.TotalItems = resultTuple.Item1

```

```
95 |     }  
96 | }
```

Let us understand these changes one by one. From the top id route is added to the page directive.

```
@page "{id:int?}"
```

This is done because the page number will come in the url as a last segment like:

```
https://localhost:44329/Read/1  
https://localhost:44329/Read/2  
https://localhost:44329/Read/3  
https://localhost:44329/Read/10
```

The above type of routing is created ASP.NET Core by default.

The next change is the MovieList property added to the functions block.

```
public MovieList movieList { get; set; }
```

This property is then used in the foreach loop which is creating the table rows from the records. The Model.movieList.movie will contain the list of movies.

```
@foreach (Movie m in Model.movieList.movie)
```

inside the style block.

```
<div class="pagingDiv" page-model="Model.movieList.pagingInfo"
page-name="Read" page-classes-enabled="true" page-
class="paging" page-class-selected="active"></div>
```

The div also has other attributes whose values will be binded to the respective property defined on the tag helper class.

```
page-model ---- PageModel
page-name ----  PageName
page-classes-enabled --- PageClassesEnabled
page-class --- PageClass
page-class-selected --- PageClassSelected
```

This type of binding is done by the tag helper automatically. This is how it works:

First remove dash “-” from the attribute name and then capitalize the first characters from the words before and after the dash sign. Now search this new name among the C# property (given on the tag helper class) and bind the value to this property.

Example : page-mode after removing dash and capitalization of first characters becomes “PageModel”. You have the PageModel property defined on the tag helper class so it binds the value to this property.

In the tag helper class I have `PageOtherValues` property defined as a dictionary type:

```
[HtmlAttributeName(DictionaryAttributePrefix = "page-other-")]  
public Dictionary<string, object> PageOtherValues { get; set; }  
= new Dictionary<string, object>();
```

This property gets the values in **Dictionary** type from the attributes that starts with “page-other-”. Examples of such attributes can be:

```
page-other-other  
page-other-data  
page-other-name
```

The values of these attributes will be added to the query string of url. The function `AddDictionaryToQueryString` defined on the class does this work.

Although I have not used it but it can be useful if you want to add more features to your tag helper class.

Now moving to the `OnGet` Handler which now gets the page number value in it's parameter. Recall we have added it to the page directive some time back.

I have set the page size as 3 which you can change according to your


```
int pageSize = 3;
```

The current page and the items per page are added to the PagingInfo class object. Also the value of the starting record for the page is calculated and added to the skip variable.

```
var skip = pageSize * (Convert.ToInt32(id) - 1);
```

Next, we call the `ReadAllFilterAsync` method with the value of skip and the number of records to fetch (i.e. pagesize).

```
var resultTuple = await repository.ReadAllFilterAsync(skip,
    pageSize);
```

The method returns Tuple whose value is extracted and provided to the TotalItems of pagingInfo and movie of movieList.

```
pagingInfo.TotalItems = resultTuple.Item2;
movieList.movie = resultTuple.Item1;
```

Finally we provide pagingInfo property of the movieList object the value of pagingInfo.

```
movieList.pagingInfo = pagingInfo;
```

As you can see the movieList's pagingInfo value is provided to the

One more change is needed now on the Create.cshtml Razor Page. This change is on the OnPostAsync handler, see highlighted code below:

```
1 public async Task<IActionResult> OnPostAsync()  
2 {  
3     if (ModelState.IsValid)  
4     {  
5         await repository.CreateAsync(movie);  
6         return RedirectToPage("Read", new { id = 1 });  
7     }  
8     else  
9         return Page();  
10 }
```

We are now redirecting to the first page of the Read razor page in case the new record is successfully created. The redirected url will be <https://localhost:44329/Read/1>.

```
return RedirectToPage("Read", new { id = 1 });
```

For the case when there happens to be validation errors, we simply return to the same page.

```
return Page();
```

Now run visual studio, create a few movie records, and navigate to the url – <https://localhost:44329/Read/1> where you will see the pagination links working excellently.

For large number of pages the pagination will add two dots “..” before and after the last and first page’s links (check the below image). This is just like what we see in professional sites.

Update Movie with Repository Pattern

Now moving to the Update Movie feature, like what we did previously, add a new method called UpdateAsync to the IRepository interface:

```
1 public interface IRepository<T> where T : class
2 {
3     Task CreateAsync(T entity);
4     Task<List<T>> ReadAllAsync();
5     Task<(List<T>, int)> ReadAllFilterAsync(int skip,
6     Task UpdateAsync(T entity);
7 }
```

Also implement this method on the Repository.cs class.

```
1 public async Task UpdateAsync(T entity)
2 {
3     if (entity == null)
4         throw new ArgumentNullException(nameof(entity));
5
6     context.Update(entity);
7     await context.SaveChangesAsync();
8 }
```

The UpdateAsync method is updating the entity in the database from the last 2 lines:

```
context.Update(entity);
await context.SaveChangesAsync();
```

```

1  @page
2  @model UpdateModel
3  @using Microsoft.AspNetCore.Mvc.RazorPages;
4  @using MovieCrud.Entity;
5  @using Models;
6
7  @{
8      ViewData["Title"] = "Update a Movie";
9  }
10
11 <h1 class="bg-info text-white">Update a Movie</h1>
12 <a asp-page="Read" class="btn btn-secondary">View all</a>
13
14 <div asp-validation-summary="All" class="text-danger">
15
16 <form method="post">
17     <div class="form-group">
18         <label asp-for="movie.Id"></label>
19         <input type="text" asp-for="movie.Id" readonl
20     </div>
21     <div class="form-group">
22         <label asp-for="movie.Name"></label>
23         <input type="text" asp-for="movie.Name" clas
24         <span asp-validation-for="movie.Name" class='
25     </div>
26     <div class="form-group">
27         <label asp-for="movie.Actors"></label>
28         <input type="text" asp-for="movie.Actors" cla
29         <span asp-validation-for="movie.Actors" clas
30     </div>
31     <button type="submit" class="btn btn-primary">Upd
32 </form>
33
34 @functions{
35     public class UpdateModel : PageModel
36     {

```

```
40         this.repository = repository;
41     }
42
43     [BindProperty]
44     public Movie movie { get; set; }
45
46     public async Task<IActionResult> OnGet(int id)
47     {
48         movie = await repository.ReadAsync(id);
49         return Page();
50     }
51
52     public async Task<IActionResult> OnPostAsync()
53     {
54         if (ModelState.IsValid)
55         {
56             await repository.UpdateAsync(movie);
57
58             return RedirectToPage("Read", new { id = movie.Id });
59         }
60         else
61             return Page();
62     }
63 }
64 }
```

The code of the Update Razor Page is very similar to the Create Razor Page, just a few changes which are:

■ ■ ■



1. In the `onGet()` handler the repository is called to fetch the record whose id the handler receives in it's parameter.

```
movie = await repository.ReadAsync(id);
```

The record's id will be sent to the Update Razor Pages as a query string parameter. I have shown such links below:

```
https://localhost:44329/Update?id=1  
https://localhost:44329/Update?id=2  
https://localhost:44329/Update?id=10  
...
```

The OnGet handler has (int id) in it's parameter and the model binding feature will automatically bind this id parameter with the value of the id given on the query string.

```
public async Task<IActionResult> OnGet(int id) {...}
```

I am calling the `ReadAsync()` method of the repository and passing the value of the record id to be fetched to it. This means we will have to add this method to our Generic Repository. So add this method to the IRepository interface.

```
1 public interface IRepository<T> where T : class
2 {
3     Task CreateAsync(T entity);
4     Task<List<T>> ReadAllAsync();
5     Task<(List<T>, int)> ReadAllFilterAsync(int skip,
6     Task UpdateAsync(T entity);
7     Task<T> ReadAsync(int id);
8 }
```

Also implement it on the IRepository.cs class. As shown below.

```
public async Task<T> ReadAsync(int id)
{
    return await context.Set<T>().FindAsync(id);
}
```

The above method used the `FindAsync()` method of Entity Framework

2. On the `OnPostAsync()` handler we check if model state is valid and then call the `UpdateAsync` method of the repository with the entity to be updated. Next redirecting the user to the first page of the `Read.cshtml`.

```
if (ModelState.IsValid)
{
    await repository.UpdateAsync(movie);
    return RedirectToPage("Read", new { id = 1 });
}
```

Another thing we need to do is to link the Update page from the Read page. The table on the `Read.cshtml` which shows the movie records is an ideal area for this. We will add another column for the table, an anchor tag will be added to this column, this anchor tag will be linking to the Update page. See the changes I have highlighted on the table.

```
1 <table class="table table-sm table-bordered">
2   <tr>
3     <th>Id</th>
4     <th>Name</th>
5     <th>Actors</th>
6     <th></th>
7   </tr>
8   @foreach (Movie m in Model.movieList.movie)
9   {
10     <tr>
11       <td>@m.Id</td>
12       <td>@m.Name</td>
```

```

16 |                                     Update
17 |                                     </a>
18 |                               </td>
19 |         </tr>
20 |     }
21 | </table>

```

The anchor tag's href value will be created by asp-page and asp-route tag helpers. The asp-page is provided with the name of the page which is Update, while asp-route is provided with the name of the route which is id. The id value is added to the tag helper from the foreach loop mechanism.

```

<a class="btn btn-sm btn-primary" asp-page="Update" asp-route-
id="@m.Id">Update</a>

```

Run the app on visual studio and go to the Read Page's URL

<https://localhost:44329/Read/1>. Here you will see blue Update link

(looking like a button) against each record on the table. Click the Update link for the 2nd record.

You will be taken to the Update Razor page whose url will be <https://localhost:44329/Update?id=2>. Here you can update the 2nd record. I have shown this whole thing in the below image.

Notice the id of the record which is 2 is send in the URL as a query string:

```
https://localhost:44329/Update?id=2
```

Similarly, if you click on the 10th record then the URL will obviously become:

```
https://localhost:44329/Update?id=10
```

This completes the Update Record CRUD operations. We are now left with only the Delete operation so kindly proceed with it quickly.

Delete Movie with Repository Pattern

Start by adding method called DeleteAsync to the interface. This method accepts id of the entity as a parameter.

```
Task DeleteAsync(int id);
```

Also implement this method on the IRepository.cs class.

```
1 | public async Task DeleteAsync(int id)
2 | {
```

```

6 |
7 |     context.Set<T>().Remove(entity);
8 |     await context.SaveChangesAsync();
9 | }

```

In this method we used FindAsync method of the repository to find the entity by it's id.

```
var entity = await context.Set<T>().FindAsync(id);
```

And then deleted the entity by Entity Framework Core Remove method.

```

context.Set<T>().Remove(entity);
await context.SaveChangesAsync();

```

Next, moving to the razor page. We do not need to create a new Razor Page for the Delete operation in-fact we will use the Read.cshtml page for this. To create the Delete CRUD operation, we will add another column to the table on the Read.cshtml Razor Page. This column will contain a delete button which on clicking will delete the record.

The change to make to the table is shown below:

```

1 | <table class="table table-sm table-bordered">
2 |     <tr>
3 |         <th>Id</th>
4 |         <th>Name</th>
5 |         <th>Actors</th>
6 |         <th></th>
7 |         <th></th>

```

```

11         <tr>
12             <td>@m.Id</td>
13             <td>@m.Name</td>
14             <td>@m.Actors</td>
15             <td>
16                 <a class="btn btn-sm btn-primary" asp-
17                     Update
18                 </a>
19             </td>
20             <td>
21                 <form asp-page-handler="Delete" asp-r
22                     <button type="submit" class="btn
23                         Delete
24                     </button>
25                 </form>
26             </td>
27         </tr>
28     }
29 </table>

```

Notice we created a form and a button inside it. This button will post the form when clicked.

```

1 <form asp-page-handler="Delete" asp-route-id="@m.Id" r
2     <button type="submit" class="btn btn-sm btn-danger
3         Delete
4     </button>
5 </form>

```

The form has 2 tag helpers which are:

asp-page-handler – specifies which handler to call. Here I have specified “delete” handler method to be called when the form is

asp-route-id – specifies the value of id to be passes on the route.

This will contain the id of the record.

Note that handler can be both synchronous and asynchronous.

Asynchronous handlers have the term “Async” at the end of their name.

The name of the method is appended to “OnPost” or “OnGet”, depending on whether the handler should be called as a result of a POST or GET request. So, I added `asp-page-handler="delete"` and not `asp-page-handler="OnPostDelete"`.

Finally we will have to add OnPostDeleteAsync handler to the Read.cshtml file. This handler be called on the click of the delete button.

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    await repository.DeleteAsync(id);
    return RedirectToPage("Read", new { id = 1 });
}
```

The OnPostDeleteAsync handler is an asynchronous handler as it has “Async” on it’s end. The term OnPost at the beginning of it’s name specify that it is Post type handler.

This method calls the DeleteAsync() method of the repository and sends the id of the entity to it’s parameter.

After the record is deleted the user is redirected to the first page of the “Read.cshml”.

```
return RedirectToPage("Read", new { id = 1 });
```

Run the app on Visual Studio, on the Read Page you will see Delete button on each row of the table. Click on any button to delete the entity from the database. Check the below image.

I would like to explain you how the handlers work. If you inspect the html of the form, you will see the form’s action contains the handler query string with value as the name of the handler – `/Read/14?handler=delete` . See the below image.

Since the form method is defined as post i.e. `method="post"` and we have our handler defined as Post one i.e. `OnPost` at the beginning on it's name. Therefore when the form is submitted we are sure the correct handler will be called and the deletion of the entity will be done successfully.

We can also use this concept to call a Get type handler on a Razor Page by targeting it with an anchor tag.

Suppose we have a handler called `OnGetSomething` on a Razor Page called `Job.cshtml`

```
1 | public async Task<IActionResult> OnGetSomething()  
2 | {  
3 |     ...  
4 | }
```

```
<a href="/Job?handler=something">Job</a>
```

Congratulations you for building CRUD Operations using Razor Pages and Generic Repository Pattern & Entity Framework Core. You now have enough confidence to create and build any type of Database operations in Razor Pages using a cleaner code by **Generic Repository Patter**. Here is a bonus topic to you.

Filtering Entity by LINQ Expression

Let us create a feature to search an entity by it's name using LINQ Expression. So, to the interface and it's implementation class, import the below given namespace.

```
using System.Linq.Expressions;
```

Next, add a new method called ReadAllAsync to the IRepository interface. This method will have `Expression<Func<T, bool>>` type parameter which can be used to send filter expression.

```
Task<List<T>> ReadAllAsync(Expression<Func<T, bool>> filter);
```

Next, to the Repository.cs add this method:

```
public async Task<List<T>> ReadAllAsync(Expression<Func<T, bool>> filter)
```

```
}
```

The `Func<T, bool>` means the filter express will be on “T” type entity and returns bool type. So we can apply this filter express on the “Where” clause as shown below.

```
context.Set<T>().Where(filter)
```

Now we create a new Razor Page inside the “Pages” folder. Name this page as Search.cshtml and add the following code to it:

```
1  @page
2  @model SearchModel
3  @using Microsoft.AspNetCore.Mvc.RazorPages;
4  @using MovieCrud.Entity;
5  @using System.Linq.Expressions;
6  @using Models;
7
8  @{
9      ViewData["Title"] = "Search Movies";
10 }
11
12 <h1 class="bg-info text-white">Movies</h1>
13 <a asp-page="/Read" asp-route-id="1" class="btn btn-;
14
15 <form method="post">
16     <div class="form-group">
17         <label asp-for="@Model.movie.Name"></label>
18         <input type="text" asp-for="@Model.movie.Name"
19     </div>
20     <button type="submit" class="btn btn-primary">Se;
21 </form>
```

```

25 {
26     <h2 class="bg-danger text-white m-2">Result</h2>
27     <table class="table table-sm table-bordered">
28         <tr>
29             <th>Id</th>
30             <th>Name</th>
31             <th>Actors</th>
32         </tr>
33         @foreach (Movie m in Model.movieList)
34         {
35             <tr>
36                 <td>@m.Id</td>
37                 <td>@m.Name</td>
38                 <td>@m.Actors</td>
39             </tr>
40         }
41     </table>
42 }
43
44 @functions{
45     public class SearchModel : PageModel
46     {
47         private readonly IRepository<Movie> repository;
48         public SearchModel(IRepository<Movie> repository)
49         {
50             this.repository = repository;
51         }
52
53         [BindProperty]
54         public Movie movie { get; set; }
55
56         public List<Movie> movieList { get; set; }
57
58         public void OnGet()
59         {
60

```

```

64 |         Expression

```

Points to note:

The page has an input tag to enter the name for the entity to be searched.

```

<input type="text" asp-for="@Model.movie.Name" class="form-
control" />

```

The value of the input tag is bind to the Movie property.

```

[BindProperty]
public Movie movie { get; set; }

```

Inside the OnPostAsync we create the filter expression to match the name as that entered on the input tag.

```

Expression

```

Next we pass the expression to the ReadAllAsync method of the repository.

So if we entered “Rear Window” in the text box then the repository method will create the where express as shown below.

```
context.Set<T>().Where(= m => m.Name == "Rear  
Window").ToListAsync()
```

Finally run the app and go to the search page and perform the search, url – <https://localhost:44329/Search>. The search layout and result is shown in the below image.


In this long tutorial on Repository Pattern we successfully implemented in on ASP.NET Core Razor Pages app. Enjoy and happy coding.

■ ■ ■



SHARE THIS ARTICLE

ABOUT THE AUTHOR

I am Yogi S. I write DOT NET artciles on my sites hosting.work and yogihosting.com. You can connect with me on [Twitter](#). I hope my articles are helping you in some way or the other, if you like my articles consider buying me a coffee -  [Buy Me A Coffee](#)

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Related Posts based on your interest

[Necessary .NET
Security features for
securing your web
applications](#)

[Automated UI Testing
with Selenium in
ASP.NET Core](#)

[How to perform
Integration Testing in
ASP.NET Core](#)

Hello Everyone

Welcome to
Hosting.Work - A
Programming

Tutorial Website. It covers
ASP.NET Core topics. I hope
you enjoy reading it.

Enter your email address to subscribe to
this blog and receive notifications of new
posts by email

Email Address *

Subscribe



Subscribe to our Newsletter and receive "1 email per week" for the article
published on the site. No spamming.

Copyright ©2024