ASP.NET

Page Models

C#

<IActionResult> - Appropriate return type when multiple ActionResult return types are possible.

<ActionResult> - represent various HTTP status codes.

OnPost() – used to insert/update remote data.

OnGet() – used to retrieve remote data.(shouldn't be used for operations that have side-effects. Such as taking actions in web applications)

**don't allow users to set values in GET requests.

**Set attributes in Post requests because they have built-in security from ASP.NET

Model Binding- [BindProperty] above attribute (data is retrieved from an HTTP request, it is converted into .NET types, and it is stored in corresponding model properties) ViewModel binding.

OnGetAsync() – allows the computer to "move-on" to other tasks while waiting for asynchronous operation to complete. Time-consuming operations don't bring the entire program to a halt waiting on the completion of one operation.

EX – network requests, querying a database

Enables → async, await, Task types.

Summary

1.Add async to the signature

2.Change the return type to task.

3.Rename the method to OnGetAsync()

using → statement ensures the correct use of an IDisposable instance

1.When the control leaves the block of the using statement, the IDisposable instance is disposed.

await using → Used to correctly dispose of a IAsyncDisposable

1.To do this within a request handler like OnGet()

2.We have to label it as an asynchronous method using async,

3.Change return type to Task

4. rename method to OnGetAsync() **Optional but conventional.

**Can't have OnGet() and OnGetasync()….Use one or the other.

```
EX →  public async Task OnGetAsync()

    {

      using (StreamWriter writer = new StreamWriter("log.txt", append: true))
      {
        await writer.WriteLineAsync($"OnGetAsync() called at {DateTime.Now}.");
      }
    }
```

OnPostAsync() – OnPostAsync is it's own version of OnGetAsync() and the same rules apply

```
EX→ public async Task OnPostAsync()

    {
      using (StreamWriter writer = new StreamWriter("log.txt", append: true))
      {
        await writer.WriteLineAsync($"OnPostAsync() called at {DateTime.Now}.");
      }
    }
```

DATABASES

_ViewStart.cshtml – commonly used for setting the layout page so all pages have a consistent header, left navigation, and footer.

_ViewImports – commonly used to add using statements so all code has access to the correct namespaces and assemblies.

_Layout.cshtml – defines the HTML for the header, left navigation, and footer.

@RenderBody() – gets replaced with the markup from each page that uses _Layout.

@RenderSection – replaced by script blocks as needed for things like input validation.

Dotnet command-line tools for working with databases

```
dotnet add package Microsoft.EntityFrameworkCore.SQLite
       dotnet add package Microsoft.EntityFrameworkCore.Design
       dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Building the Entity Framework Context

The Context is a class that acts as a middle man. Contains methods that can be called to ask the database to something useful i.e. → Create, Update, Add. Often referred to as a service.

```
public class CountryContext : DbContext
  {
    public CountryContext(DbContextOptions<CountryContext> options) : base(options)
    {

    }

    public DbSet<Continent> Continents { get; set; }

    public DbSet<Country> Countries { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<Continent>().ToTable("Continent");
      modelBuilder.Entity<Country>().ToTable("Country");
    }
  }
```

Add created EF context so it can be injected into all the pages that need

```
public void ConfigureServices(IServiceCollection services)
    {
      services.AddRazorPages().AddRazorPagesOptions(options =>
      {
        options.Conventions.ConfigureFilter(new IgnoreAntiforgeryTokenAttribute());
      });
      services.AddDbContext<CountryContext>(options =>
      {
        options.UseSqlite(Configuration.GetConnectionString("CountryContext"));
       });

      }
```

Create Method

```
private static void CreateDbIfNotExists(IHost host)
      {
        using (var scope = host.Services.CreateScope())
        {
          var services = scope.ServiceProvider;
          var context = services.GetRequiredService<CountryContext>();
          context.Database.EnsureCreated();
        }
```

Update main method to check if the database should be created

```
public static void Main(string[] args)
      {
        var host = CreateHostBuilder(args).Build();
        CreateDbIfNotExists(host);
        host.Run();
      }
```

HTML

`btn btn-danger btn-sm` - make a small red button

<input asp-for="name"> → Renders in html as → <input type="text" name="Author" id="Author">

asp-for – create a form that submits data with a POST request.

1.saves hassle of writing html attributes.

2.Checks for errors before you run the code.

3.Changes to the property(in a model) are automatically carried into the view page.

4.Advanced settings control data validation.

```
<a asp-page="./Authors" asp-route-fullname="Roald Dahl">Roald</a> → renders → <a
href=”./Authors?fullname=Roald+dahl”>Roald</a>
```

asp-route-{value} – Adds route values to an ‘href’. The {value} placeholder is interpreted as a potential route parameter.

asp-page – Sets an anchor tag’s ‘href’ attribute value to a specific page.

Routing

With page-based framework of Razor Pages, instead of URLs you can rename and make variable route parameters.

1.Define custom URL segments

2.Define route templates

3.Add constraints on route templates

4.Understand how Tag Helpers adapt to custom routing

Routing URL Segments

@page – We can add and/or change URL segments by adding a string after the @page directive.

EX → @page “/Pirates” at the top of the privacy page would be available at https::/…/Pirates

EX → @page “Pirates” would be available at https::../Privacy/Pirates

Routing Templates

POST request → localhost:8000/Movies?title=Inception → localhost:8000/Movies/Inception

➔ @page “{title}” would generalize the format of the URL and make it more readable

Optional Route Templates

? – We can mark route values as optional with a question mark. → @page “{title?}”

If – else – combined with an if else statement, the handler method an change behavior on whether the value was provided.

EX:

```
public void OnGet(string? title)
{
  if (String.IsNullOrEmpty(title)) **For non-string types we use .HasValue**
  {
    IsGeneralDisplay = true;
  }
  else
  {
    Title = title.Value;
    IsGeneralDisplay = false;
  }
}
```

Constrained Route Templates

Strict type constraints help avoid errors.

Within the @page directive we can specify that constraint lik below, where int stands fo "integer"

Would cause an error → `localhost:8000/veggies/YES/fruits/NO/grains/IDUNNO/protein/SORTA/dairy/NEVER`

Adding strictly typed constraints will help avoid this issue.

EX → `@page "/veggies/{veggies:int}..." ** means value must be an integer`

```
General format: @page "{routevalue:constraint}" **can add ? at the end of constraint to
make it optional
```

Tons of constraints → 1. Int 2. Alpha (a-z case insensitive) 3. Bool ....

Asp-route-{value} →previously used to append to a query string

→if the url template accepts url segments instead of a query string it automatically formats urls to match the defined template.

EX: `<a asp-page="./Blogpost" asp-route-id="4">ID 4</a>`

```
    renders →
```

```
   <a href="./Blogpost?id=4">ID 4</a>
```

EX: @page {id} → `<a asp-page="./Blogpost" asp-route-id="4">ID 4</a>`

```
    renders →
```

```
   <a href="./Blogpost/4">ID 4</a>
```

REVIEW ROUTES

```
<!-- No URL segments -->
@page
<!-- Edit the default route -->
@page "/Days"
```

```
<!-- Add a route template -->
@page "/Days/{day}"
<!-- Constrain route value -->
@page "/Days/{day:int}"
<!-- Make route value optional -->
@page "/Days/{day:int?}"
```

<u>Redirection</u>

→Redirecting users to a different page

- EX: A blog website has a “Posts” page showing blog posts and a separate “Create” page. After submitting on “Create”, the user should be redirected to the list of blog posts on “Posts”.

  1.RedirectToPage() → causes the server to render the Razor view page described in the method argument. Generates a 302 HTTP status code(temporary redirection message that means the requested resource or page has been moved to a different location).

  2.NotFound() → “Status 404 Not Found” response

  3.Page() → causes server to render the Razor view page associated with the current page model.

→To perform redirection, methods need to return an object.

<u>IActionResult</u> → type of return used for redirection methods.

RedirectToPage() →

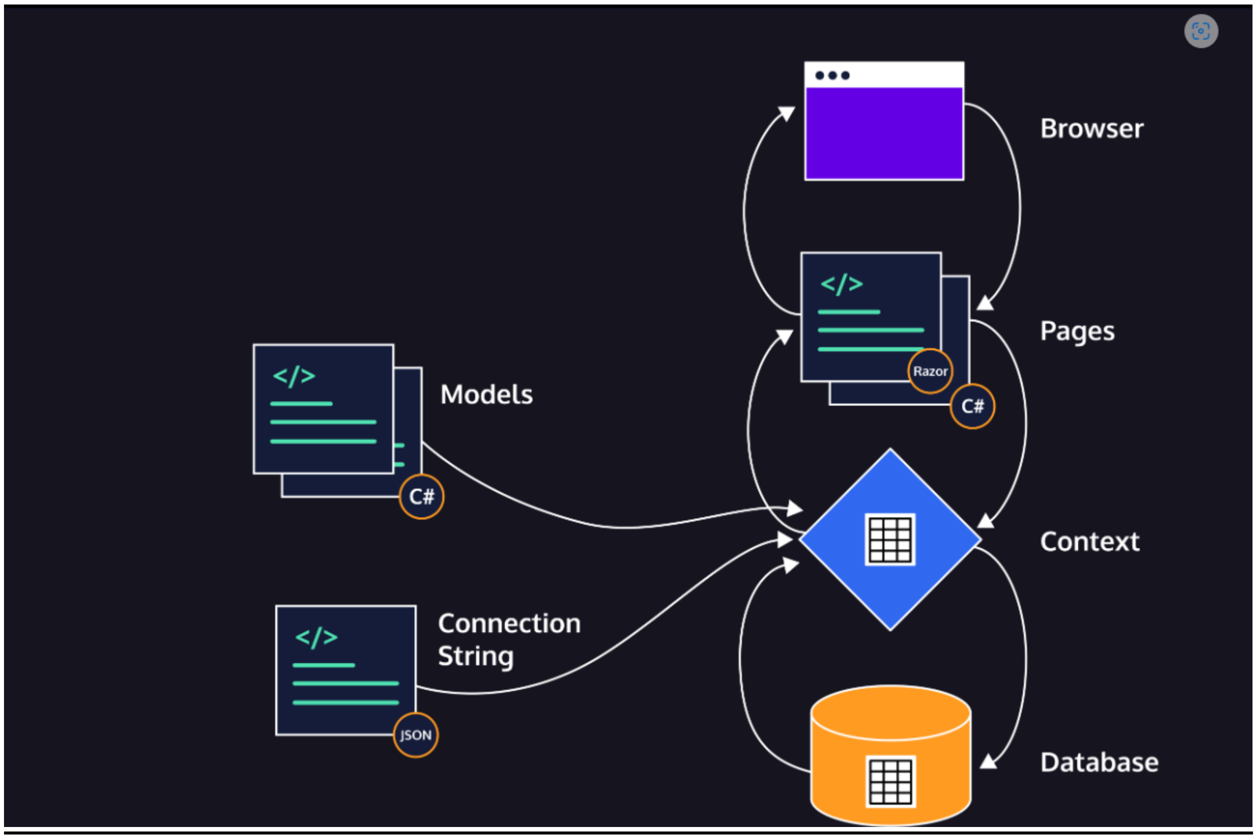
EX :
```
public IActionResult OnPost()
{
  return RedirectToPage("/Checkout");
}
```

Page() → Returning the Page() method will lead to the same behavior as an empty void OnGet()

EX:
```
public IActionResult OnGet()
{
  return Page();
}
```

IActionResult → When a redirect method is used in a handler method, it must return the type(or some type that implements the IActionResult interface)

EntityFrameworkCore Diagram



- The database connection string is stored in the **AppSettings.json** file
- Models are coded as C# classes that define field properties
- The Context is also a C# class that references the connection string and model classes forming a bridge from your code to physical table storage
- The EF Context can create the database and tables or reference an existing database
- Razor pages use dependency injection to consume the EF context making all referenced tables available to your page model code
- Code in the page model interacts with the EF context to pass data from the database to the page model member variables
- This data is merged with HTML markup to create web pages that are displayed in the browser

Code Delete in the Index Page Model

Remove() – Deleting a row in a database is handled by the EF Remove() method.

1.Takes a row object as the parameter

EX:

```
Country country = await _context.Countries.FindAsync(001);
_context.Remove(country);
```

The ID must be passed in the <form> postback. Triggered by a submit button in an <a> tag. A hidden <input> tag can be used to store the ID in the browser until the user clicks Delete and does the <form> postback.

1.All EF changes are only in memory until the SaveChangesAsync() method is called.

EX: shows the same code as above but with the SaveChangesAsync() added to ensure that the changes are persisted.

```
Country country = await _context.Countries.FindAsync(001);
_context.Remove(country);
await _context.SaveChangesAsync();
```

2.Once all of this is done, the Index page needs to refresh and call OnGetAsync() which will load all the rows in the table and display them in the loop.

**This provides visual confirmation that the deletion was a success.

Retrieve a single row from DbSet

```
Continent Continent = await _context.Continents.FindAsync(id);
```

Flag the row as deleted in-memory DbSet

```
if (Continent != null)
        {
          _context.Continents.Remove(Continent);
          await _context.SaveChangesAsync();
        }
```

Context → Context represents a session with the underlying database, allowing CRUD(create,read,update,delete)

1.An instance of a context class represents Unit Of Work and Repository patterns wherein it can combine multiple changes under a single database transaction.

2.Used to query or save data to the database.

1.Also used to configure domain classes, database related mappings, changing tracking settings, caching, transactions etc.

Markup the Delete Button

The code below adds the button to the page for deleting the item selected.

```
<input class="btn btn-danger btn-sm" type="submit" value="Delete" >
<input type="hidden" name="ID" value=@item.ID >
```

Code Create and Update in the Edit Page Model

Edit Page Model → has 2 roles: Create and edit

1.Edit

→ displays most or all the fields from the current record and allows the user to edit the fields in <input> tags.

→The column names are placed in <label> tags.

→The ID of the desired record arrives in the URL as a query parameter or URL segment.

→ID automatically becomes part of the page model

→OnGetAsync() accepts the ID as a parameter, makes sure it is not null, and retrieves the matching record with the EF context FindAsync()

→The record is passed to a public member which is referenced in the HTML markup. Since the record can be changed by the user the [BindProperty] annotation is applied to the member so it can return to the server on <form> post back.

2.Create

→ Similar to edit **BUT uses an empty record as a starting point.

→ OnGetAsync() must check for a value in ID. If it is null, then a new empty member record is created. If it has a value, then FindAsync() is used to retrieve the current row.

- ➔ When the user clicks the Save button, the <form> posts back with a copy of all <input> tag values.
- ➔ OnPostAsync() again checks for an ID, if it is null, we call the EF method Add()
- ➔ If the ID has a value, we call Attach() with a chained state value set to Modified.

SaveChangesAsync() must be called at the end of either case in order to persist the record to the database.

OnPostAsync(string id) → returns Task<IActionResult> which is an ASP.NET truck for redirection.

Once the record is saved, we display the index page with our new or modified record.

EDIT EXAMPLE

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorCountry.Models;
using RazorCountry.Data;

namespace RazorCountry.Pages.Continents
{
  public class EditModel : PageModel
  {
    private readonly CountryContext _context;

    public EditModel(CountryContext context)
    {
      _context = context;
    }

    [BindProperty]
    public Continent Continent { get; set; }

    public async Task<IActionResult> OnGetAsync(string id)
    {
      if (id == null)
      {
        Continent = new Continent();
      }
      else
      {
        Continent = await _context.Continents.FindAsync(id);
        if (Continent == null)
        {
          return NotFound();
        }
      }
      return Page();
    }

    public async Task<IActionResult> OnPostAsync(string id)
    {
      if (!ModelState.IsValid)
      {
        return Page();
      }
      else
      {
        if (id == null)
        {
          _context.Continents.Add(Continent);
        }
        else
        {
          _context.Attach(Continent).State = EntityState.Modified;
        }
      }
      await _context.SaveChangesAsync();
      return RedirectToPage("./Index");
    }
  }
}
```

Markup the Edit Page

Tag helpers are used as attributes of the `<label>` and `<input>` tags. The `asp-for` attribute needs the column name from the record schema so it can bind the column name for the `<label>` or the record data for the `<input>`. The helper uses the data type and other information from the EF model to determine which control type to display. This is clearly evident in the `Population` and `UnitedNationsDate` fields.

Bootstrap CSS note:

- `form-control` - built-in class that rounds corners and shows a colored border while editing a field

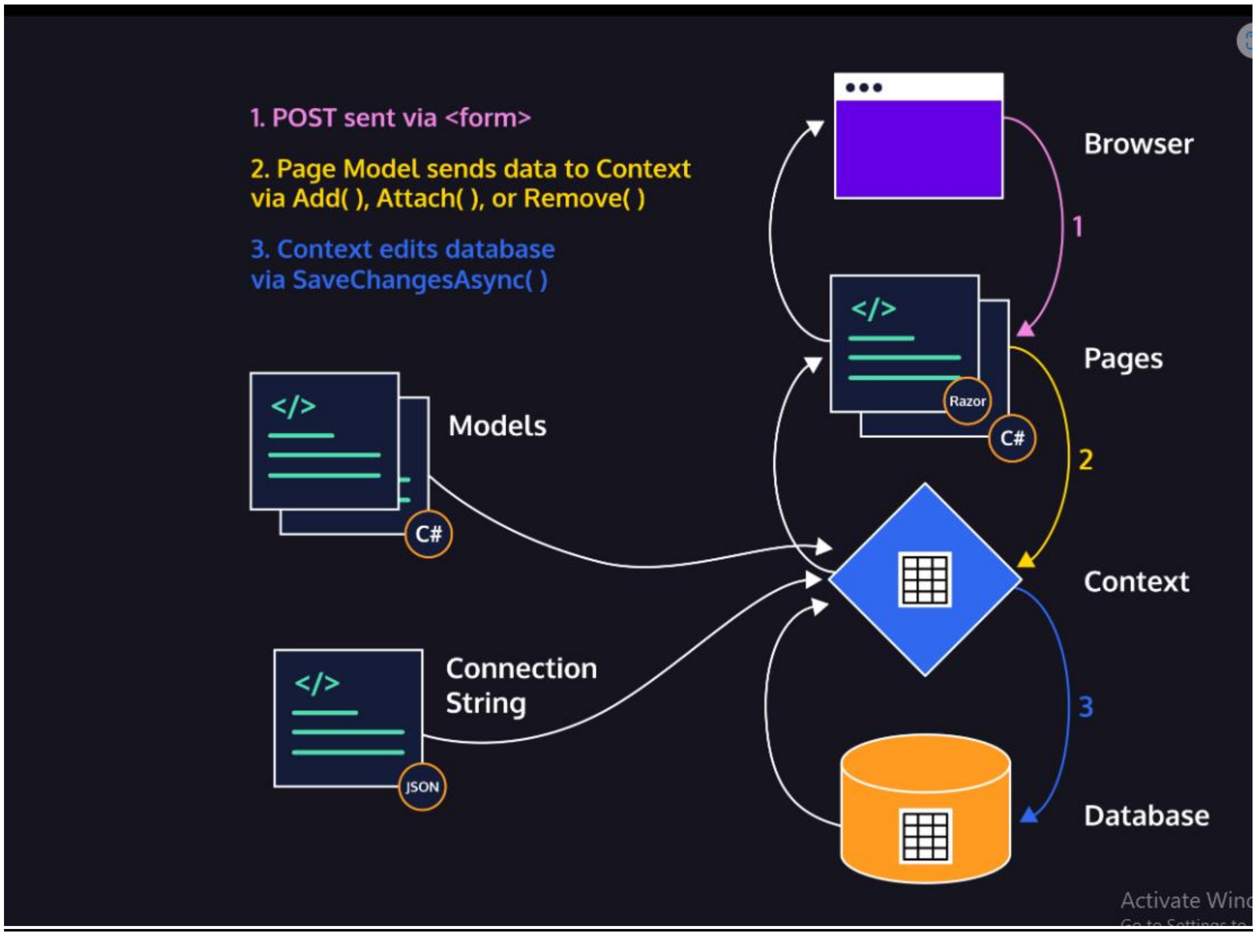
Markup for Edit Page

```
<form method="post">
  <div class="form-group d-flex">
    <label asp-for="Continent.ID" class="p-2 text-right" style="flex:0 0 15%"></label>
      <input asp-for="Continent.ID" class="form-control" style="flex:1 0 auto;width:auto"
/>
  </div>
</form>
```

EF Diagram

The EF diagram now shows how data is saved from the browser to the database:

- The user makes changes to existing data or adds new data
- The data is sent to the Razor page with a POST method that usually is triggered by a button click
- The Razor page parses the data fields from the `<form>` POST and assigns it to a member variable
- The variable is merged into the EF context `DBSet` using `Add()` for a new record, `Attach()` for an update, and `Remove()` for a deletion
- Each of these three context methods generate the SQL statements necessary to complete the task
- The SQL is sent to the database so the data can persist


1. POST sent via <form>
2. Page Model sends data to Context via Add( ), Attach( ), or Remove( )
3. Context edits database via SaveChangesAsync( )
Browser
1
Pages
Razor
C#
2
Models
C#
Context
Connection String
JSON
3
Database

Review Data Edits

Deleting a row in the database is handled by the EF `Remove()` method. This takes a row object as the parameter so we need to know which row to remove and somehow need to load that row. We find the row using the built-in EF method `FindAsync()` which takes the row ID as a parameter and returns a single row.

The ID is passed in the `<form>` postback. This is triggered by a submit button in an `<a>` tag. A hidden `<input>` tag can be used to pass the ID when the user clicks Delete forcing a `<form>` postback.

All EF changes are only in memory until the `SaveChangesAsync()` method is called to persist changes to the database file.

Once all this is done, the Index page needs to refresh and call `OnGetAsync()` which will load all the rows in the table and display them in the loop. This provides visual confirmation that the deletion was a success.

The edit page in our example has two roles: edit and create. For edit, it displays most or all fields from the current record and allows the user to edit fields in `<input>` tags. The column names are placed in `<label>` tags. The ID of the desired record arrives in the URL as a query parameter or URL segment. This ID automatically becomes part of the page model. The `OnGetAsync()` method accepts the ID as a parameter, makes sure it is not null, and retrieves the matching record with the EF context `FindAsync()` method. The record is passed to a public member which is referenced in the HTML markup. Since the record can be changed by the user, the `[Bind Property]` annotation is applied to the member so it can return to the server on `<form>` post back.

The create mode is similar but uses an empty record as a starting point. Our `OnGetAsync()` method must check for a value in ID. If it is null, then a new empty member record is created. If it has a value, then `FindAsync()` is used to retrieve the current row. When the user clicks the Save button, the `<form>` posts back with a copy of all `<input>` tag values. The `OnPostAsync()` again checks for an ID. If it is null, we call the EF method `Add()`. If the ID has a value, we call `Attach()` with a chained State value set to Modified. In either case, `SaveChangesAsync()` must be called to persist the record to the database.

Note that `OnPostAsync()` returns a `Task` of `IActionResult`. This is an ASP.NET trick for redirection. In simpler terms, once the record is saved, we display the Index page with our new or modified record.

Tag helpers are used as attributes of the `<label>` and `<input>` tags. The `asp-for` attribute needs the column name from the record schema so it can bind the column name for the `<label>` or the record data for the `<input>`. The helper uses the data type and other information from the EF model to determine which control type to display. This is clearly evident in the `Population` and `UnitedNationsDate` fields.

Code Continent Model Annotations

Now that we have afull working set of CRUD pages, we need to add VALIDATION. (could be bad dates, negative population values, etc.)

Refine database schema and ensure that the correct data types, string lengths, and character types are stored.

System.ComponentModel.DataAnnotations;

Annotations → Entered in the EF model class above each property field.

**Some annotations accept additional parameters like [ErrorMessage]

[Required] → field must contain a value

[StringLength] → set the minimum and maximum string lengths

[RegularExpression] → Match input patterns for things like email addresses and zipcodes

[Range] → Numeric or date limits

[DataType] → Define a data type which determines the database column type

[Display] → Change the field label

[DisplayFormat] → Change how the value is displayed

[BindProperty(SupportsGet=true)] → by default properties are not bound for HTTP GET requests.

EX:

```
using System.ComponentModel.DataAnnotations;

namespace RazorCountry.Models
{
  public class Continent
  {
    [Required, StringLength(2, MinimumLength=2), Display(Name="Code")]
    [RegularExpression(@"[A-Z]+", ErrorMessage="Only upper case characters are
allowed.")]
    public string ID { get; set; }
    [Required]
    public string Name { get; set; }
  }
}
```

Code Country Model Annotations

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

namespace RazorCountry.Models
{
  public class Country
  {
    [Required]
    [StringLength(2, MinimumLength=2)]
    [RegularExpression(@"[A-Z]+", ErrorMessage="Only upper case characters are
allowed.")]
    [Display(Name = "Code")]
    public string ID { get; set; }
    [Required]
    [StringLength(2, MinmumLength=2)]
    [RegularExpression(@"[A-Z]+", ErrorMessage="Only upper case characters are
allowed.")]
    [Display(Name = "Continent")]
    public string ContinentID { get; set; }
    [Required]
    public string Name { get; set; }
    [Range(1, 10000000000)]
    [DisplayFormat(DataFormatString = "{0:N0}", ApplyFormatInEditMode = true)]
    public int? Population { get; set; }
    public int? Area { get; set; }
    [Display(Name = "UN Date")]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    [Range(typeof(DateTime), "10/24/1945", "1/1/2100", ErrorMessage = "MESSAGE")]
    [DataType(DataType.Date)]
    public DateTime? UnitedNationsDate { get; set; }
  }
}
```

- `text-danger` - make the message red

Markup Continent Field Level Validation

→ Server Code and the database now validate any data that is saved.

**User cannot see yet → Since records are not saved when the ModelState.IsValid is false.

1.User can get stuck in the edit page and not know why the Save button is failing.

→Apply validation controls and client-side JavaScript to the edit page.

1.Basic validation messages are displayed in a <span> tag using the tag helper asp-validation-for.

2.Will show a default message if any model annotations are violated. If a custom ErrorMessage will be displayed if specified in the annotation.

EX:

```
<div style "flex:1 0 auto;width:auto">
  <input asp-for "Continent.ID" class "form-control" />
  <span asp-validation-for "Continent.ID" class "text-danger"></span>
</div>
```

→Some client-side scripts are needed to handle the validation logic in the browser.

→Provided by asp.net but have to be added to each page that needs them.

1.@section

2.@RenderSection

EX:

```
@section OtherScripts {
  @{ await Html.RenderPartialAsync("_UsefulScriptsPartial"); }
}
```

```
@RenderSection("OtherScripts", required: false)
```

Edit Page Model Markup after adding validation using <span> tag wrapped in a div tag.

**Moved the input inside the div as well

```
@page "{id?}"
@model EditModel
@{
  ViewData["Title"] = "Continent Edit";
}

<div class="jumbotron p-3">
  <div class="d-flex align-items-center">
    <h1 class="display-4 flex-grow-1">Continent Edit</h1>
    <a class="btn btn-primary btn-sm" asp-page="./Index">Back to List</a>
  </div>
</div>

<form method="post">
  <div class="form-group d-flex">
    <label asp-for="Continent.ID" class="p-2 text-right" style="flex:0 0 15%"></label>
    <div style="flex:1 0 auto;width:auto">
      <input asp-for="Continent.ID" class="form-control" />
      <span asp-validation-for="Continent.ID" class="text-danger"></span>
    </div>
  </div>
    <div class="form-group d-flex">
    <label asp-for="Continent.Name" class="p-2 text-right" style="flex:0 0 15%"></label>
    <div style="flex:1 0 auto;width:auto">
      <input asp-for="Continent.Name" class="form-control" />
      <span asp-validation-for="Continent.Name" class="text-danger"></span>
    </div>
  </div>
  <div class="form-group d-flex flex-row-reverse">
```

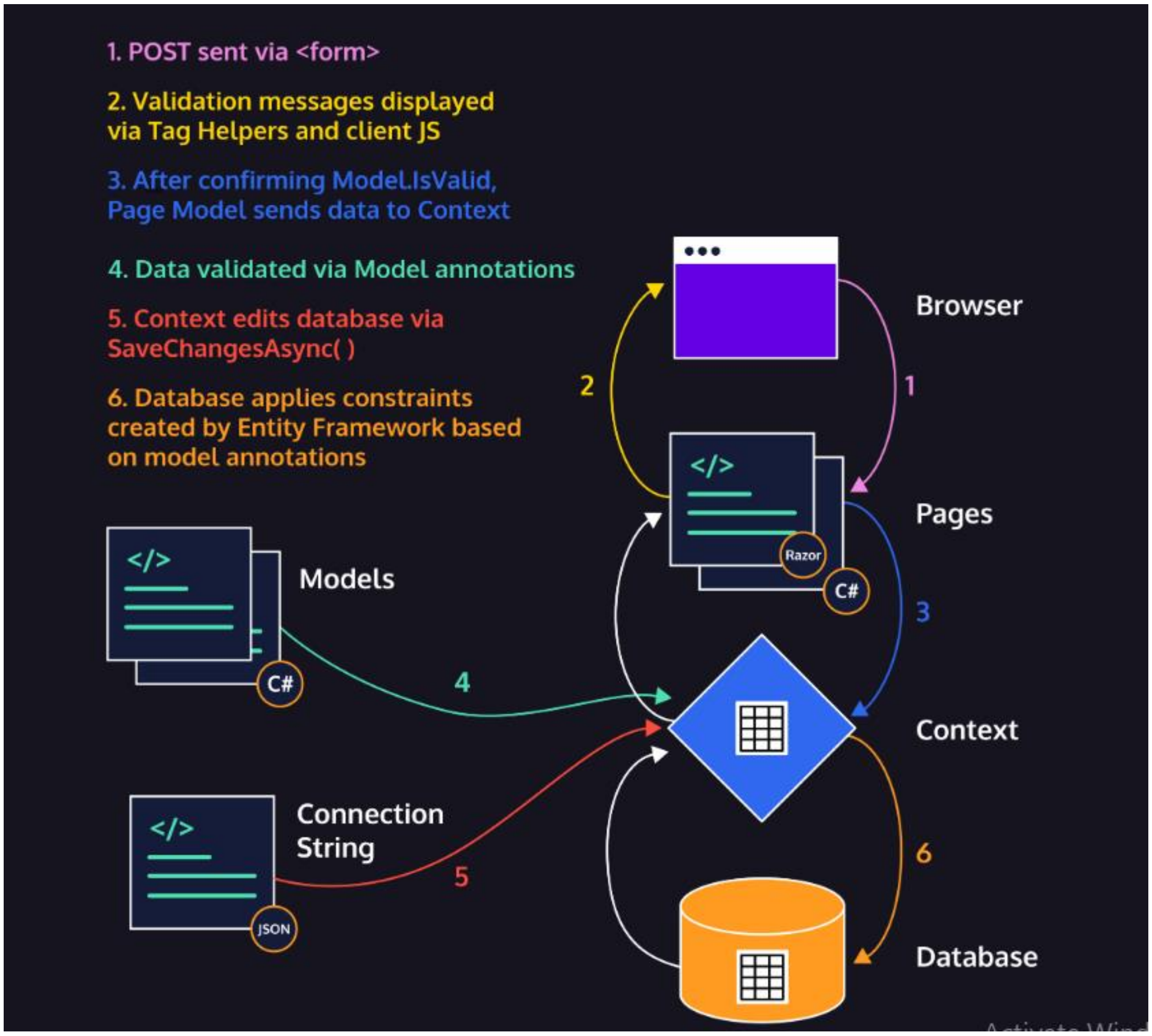
```
    <input type="submit" value="Save" class="btn btn-primary btn-sm" />
  </div>
</form>
@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

DATA VALIDATION

EF Diagram

The EF diagram now includes additional steps to accommodate data validation:

- Notations are made in the model classes that tailor each field for type, display, length, and regex patterns
- The model hints alter the database schema so the data has proper constraints
- The model hints are also used by the context to determine if any data changes are valid
- The model hints also interact with JavaScript validation scripts that display helpful information on the browser page when data is changed by the user
- The JavaScript validation scripts will restrict a `<form>` POST if the data does not match the model hints
- Once the requested changes are marked as valid, the context method `SaveChangesAsync()` passes the data to the database by way of SQL statements
- The database constraints created by the context via the model hints further restrict inserts, updates, and deletes using SQL referential techniques


1. POST sent via <form>
2. Validation messages displayed via Tag Helpers and client JS
3. After confirming Model.IsValid, Page Model sends data to Context
4. Data validated via Model annotations
5. Context edits database via SaveChangesAsync( )
6. Database applies constraints created by Entity Framework based on model annotations
Browser
Pages
Context
Database
Models
Connection String
Razor
C#
JSON
1
2
3
4
5
6

<u>ToListAsync()</u> → returns all records in a table

<u>FindAsync()</u> → returns a single record by ID

<u>Entity Framework Context with LINQ</u>

LINQ FindAsync() Equivalent

```
Variable = await _context.SomeDbSet.FirstOrDefaultAsync(m => m.ID == id);
```

<u>FirstOrDeafaultAsync()</u> → returns a single record that matches the lambda expression in the parameter.

1. FindAsync() equivalent but not limited to searching by ID

Add a Search Filter to the Country List

→Add a text box for the search string

1.A button can post back a form with this string

2.SearchString must use [BindProperty(SupportsGet=true)]

EX:

```
var countries = from c in _context.Countries
        select c;

      if (!string.IsNullOrEmpty(SearchString))
      {
        countries = countries.Where(c => c.Name.Contains(SearchString));
      }
      Countries = await countries.ToListAsync();
```

HTML MARKUP

```
<div>
    <form class="form-group">
      <label asp-for="SearchString" class="control-label">
        Name:
      </label>
      <input class="p-2" type="text" asp-for="SearchString" />
      <input type="submit" value="Search" class="btn btn-primary btn-sm" />
    </form>
  </div>
</div>
```

Model Binding

Controllers and Razor pages work with data that comes HTTP requests.

EX: route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error prone.

- Retrieves data from various sources such as route data, form fields, and query strings

- Provides the data to controllers and Razor pages in method parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.

Add Column Sorting to the Country List

Now that we have the search up and running lets add sorting.

1. <select> tag will hold the column names in a series of <option> tags.
   → Will be linked to a member property that tracks the current sort column
2. Add an OnChange() event that posts the form each time achange is made.

EX:

```
<select class "p-2" asp-for "SortField" onchange "this.form.submit();">
  <option value "ID">Code</option>
  <option value "ContinentID">Cont</option>
  <option value "Name">Name</option>
</select>
```

3.The LINQ query can be modified multiple times before it is evaluated.

→ Sorting is managed by the OrderBy() clasue.

4.A switch statement determunes whuch column name to include in the LINQ query

EX:

```
switch (SortField)
{
  case "ID":
    countries = countries.OrderBy(c => c.ID);
    break;
  case "Name":
    countries = countries.OrderBy(c => c.Name);
    break;
  case "ContinentID":
    countries = countries.OrderBy(c => c.ContinentID);
    break;
}
```

<u>HTML Markup for colum sorting</u>

```
<label asp-for="SortField" class="control-label">
  Sort by:
</label>
 <select class="p-2" asp-for="SortField" onchange="this.form.submit();">
   <option value="ID">
     Code
   </option>
   <option value="ContinentID">
     Cont
   </option>
   <option value="Name">
     Name
   </option>
 </select>
```

NOTE:

1.OnChange() fires each time the dropdown value changes.

2.Short javascript statement causes the <form> to complete a post back.

3.This reloads the page with <u>OnGetAsync()</u> whin in turn adds the <u>OrderBy()</u> LINQ query modifier with the selected <u>SortField</u>.

<u>Add a Continent Select Box to the Filter</u>

What if we want to limit our seartches by Continent??

Using another select list we can sort alphabetucally:

EX:

```
IQueryable<string> continentQuery = from c in _context.Continents
      orderby c.ID
      select c.ID;
```

```
1.Result of the query is assigned to a SelectList component during the
OnGetAsync().

2.The component adds the All option as part of the markup.
```

Add a Content Select Box to the Filter <CONTINUED>

1.We use another LINQ query modifier that checks for a selected continent and adds that condition.

→ Multiple modifiers allow us to filter by a continent and include a country name search string:

EX:

```
        if (!string.IsNullOrEmpty(SelectedContinent))

       {
         countries = countries.Where(c => c.ContinentID == SelectedContinent);
       }


       IQueryable<string> continentQuery = from c in _context.Continents
           orderby c.ID
           select c.ID;
       Continents = new SelectList(await continentQuery.ToListAsync());
```

HTML Markup for the newly added Continent Select box filter

The following markup adds a drop down menu to select a continent to filterby

```
<label asp-for="SelectedContinent" class="control-label">
      Continent:
    </label>
    <select class="p-2" asp-for="SelectedContinent" asp-items="Model.Continents"
onchange="this.form.submit();">
      <option value="">All</option>
    </select>
```

Add a Select List for Contients

We just used a SelectList to filter Continents on the list page. Can use a similar approach on the Country Edit page

1.We replace the Continent <input> text box with a SelectList.

→ has the added value of ensuring that only a valid Continent is selected.

2.We fill the Continents select list from the EF context DBSet and specify the key and display value.

→ allows the user to see the full continent name but the database will properly store the Continent ID:

EX:

```
Continents = new SelectList(_context.Continents, nameof(Continent.ID),
nameof(Continent.Name));
```

3.The <select> tag uses the asp-for attribute the same way <input> does.

→has an additional asp-items attribute for the source of the dropdown list.

EX:

```
<select asp-for="Country.ContinentID" asp-items="Model.Continents" class="form-
control" ></select>
```

**No need for validation since the only values in the list are acceptable.

→So we remove the validation requirements from the Country model and rely on the SelectList.

→We add a member property called Continents to hold the SelectList

```
        public SelectList Continents { get; set; }
```

→In the OnGetAsync() method we fill the Continents select list from the EF context. We us new SelectList() to create the list of continents

Parameters for SelectList()

1.specify source DbSet(_context.Continents)

2.key(nameof(Continent.ID))

3.display value(nameof(Continent.Name))

EX:

```
Continents = new SelectList(_context.Continents, nameof(Continent.ID),
nameof(Continent.Name));
```

Show Related Records with Include

→LINQ takes advantage of navigational properties coded into the entity models.

1.Start by adding lines to our models so that Entity Framework knows that the two tables are related. Continents have a collection of Countries:

EX:

//navigation property

```
public ICollection<Country> Countries { get; set; }
```

and countries have a related Continent:

EX:

//adds a relation to the parent Continent

```
public Continent Continent { get; set; }
```

→We can enhance our Continent Detail page by including a list of related Countries.

- ➔ The LINQ record retrieval statement for a single Continent can add the Include() chain statement.
- ➔ EF loads all related Countries each time it loads a Continent. The AsNoTracking() is ahint that says we will not be updating the Countries. **AsNoTracking() improves performance.

EX:

```
Continent = await _context.Continents
// enhances Continent Detail page by including a list of related
countries

.Include(c => c.Countries)

 //an EF hint that says we will not be updating the countries each
time it loads a Continent

.AsNoTracking()
.FirstOrDefaultAsync(m => m.ID == id);
```

Markup added to page to display the list of member Countries

```
@page "{id}"
@model DetailModel
@{
  ViewData["Title"] = "Continent Detail";
}

<div class="jumbotron p-3">
  <div class="d-flex align-items-center">
    <h1 class="display-4 flex-grow-1">
      Continent Detail
    </h1>
    <a class="btn btn-primary btn-sm" asp-page="./Index">
      Back to List
    </a>
  </div>
</div>
```

```
<div class="d-flex">
  <div class="p-2 bg-primary text-white text-right" style="flex:0 0 15%">
    @Html.DisplayNameFor(model => model.Continent.ID)
  </div>
  <div class="p-2 border-top border-right border-bottom border-primary" style="flex:1 0
auto">
    @Html.DisplayFor(model => model.Continent.ID)
  </div>
</div>
<div class="d-flex">
  <div class="p-2 bg-primary text-white text-right" style="flex:0 0 15%">
    @Html.DisplayNameFor(model => model.Continent.Name)
  </div>
  <div class="p-2 border-right border-bottom border-primary" style="flex:1 0 auto">
    @Html.DisplayFor(model => model.Continent.Name)
  </div>
</div>
<div class="d-flex bg-success text-white mt-3">
    <div class="p-2" style="flex:0 0 10%">
        @Html.DisplayNameFor(model => model.Continent.Countries.FirstOrDefault().ID)
    </div>
    <div class="p-2" style="flex:0 0 10%">
        @Html.DisplayNameFor(model =>
model.Continent.Countries.FirstOrDefault().ContinentID)
    </div>
    <div class="p-2" style="flex:0 0 20%">
        @Html.DisplayNameFor(model => model.Continent.Countries.FirstOrDefault().Name)
    </div>
    <div class="p-2" style="flex:0 0 15%">
        @Html.DisplayNameFor(model =>
model.Continent.Countries.FirstOrDefault().Population)
    </div>
    <div class="p-2" style="flex:0 0 20%">
        @Html.DisplayNameFor(model =>
model.Continent.Countries.FirstOrDefault().UnitedNationsDate)
    </div>
    <div class="p-2" style="flex:0 0 25%">Options</div>
</div>
@foreach (var item in Model.Continent.Countries)
{
  <div class="d-flex border-left border-right border-bottom border-success">
        <div class="p-2" style="flex:0 0 10%">
            @Html.DisplayFor(modelItem => item.ID)
        </div>
        <div class="p-2" style="flex:0 0 10%">
            @Html.DisplayFor(modelItem => item.ContinentID)
        </div>
        <div class="p-2" style="flex:0 0 20%">
            @Html.DisplayFor(modelItem => item.Name)
        </div>
        <div class="p-2" style="flex:0 0 15%">
            @Html.DisplayFor(modelItem => item.Population)
        </div>
        <div class="p-2" style="flex:0 0 20%">
            @Html.DisplayFor(modelItem => item.UnitedNationsDate)
        </div>
        <div class="p-2 btn-group" style="flex:0 0 25%" role="group">
            <a class="btn btn-primary btn-sm" asp-page="/Countries/Detail" asp-route-
id="@item.ID">Details</a>
            <a class="btn btn-primary btn-sm" asp-page="/Countries/Edit" asp-route-
id="@item.ID">Edit</a>
```

```
            <a class="btn btn-danger btn-sm" asp-page="/Countries/Delete" asp-route-
id="@item.ID">Delete</a>
        </div>
    </div>
}
```

Add a Calculated Count Column

→We know that each Country has an Area and a Population.

→Density would be Population / Area to get people per square mile.

→this can be done client-side with JavaScript or server-side with some C#

1.We will make a new data model that inherits from Country.cs called CountryStats.cs

→Will have an additonal member called Density

EX:

```
using System.ComponentModel.DataAnnotations;

namespace RazorCountry.Models

{

  public class CountryStat : Country
  {
    [Display(Name = "Ppl/SqMi")]
    public float? Density { get; set; }
  }
}
```

2.Add a LINQ query at the top of OnGetAsync()

EX:

```
       var country - from c in _context.Countries

        where c.ID = id

        select new CountryStat {
          ID = c.ID,
          ContinentID = c.ContinentID,
          Name = c.Name,
          Population = c.Population,
          Area = c.Area,
          UnitedNationsDate = c.UnitedNationsDate,
          Density = c.Population / c.Area
        };
```

Markup added for the Density field

```
<div class="d-flex">
  <div class="p-2 bg-primary text-white text-right" style="flex:0 0 15%">
    @Html.DisplayNameFor(model => model.Country.Density)
  </div>
  <div class="p-2 border-right border-bottom border-primary" style="flex:1 0 auto">
    @Html.DisplayFor(model => model.Country.Density)
  </div>
</div>
```

What is Middleware?

→Everytime we perform an action on a web application, we are generating a request.

EX:putting items in a shoping cart on amazon.

→How do provide the appropriate response?

Middleware

→Middleware is essential to web applications as it is responsible for processing each request, routing each request, and responding accordingly.

Middleware Components

→Each middleware component is a request delegate(a function that can process our HTTP requests)

→Requests are sent via HTTP(Hypertext Transfer Protocol)

Start-Up Files

→Program.cs contains the main method and calls CreateHostBuilder() which is responsible for certain configuration settings for our application, but the focus of our attention will be pn the UseStartup().

EX:

```
public class Program
{
  public static void Main(string[] args)
  {
     CreateHostBuilder(args).Build().Run();
  }

  public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
       {
          webBuilder.UseStartup<Startup>();
       });
}
```

Examining the IapplicationBuilder Interface

Configure method accepts an IapplicationBuilder interface.

EX:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

→IapplicationBuilder provides the built-in middleware methods.

EX:

```
1.        `UseHttpsRedirection()` redirects HTTP requests to the more secure
   HTTPS protocol

2.  `UseStaticFiles()` allows static files, like CSS and images, to be served

3.  `UseRouting()` allows route matching

4.  `UseAuthorization()` ensures requests are authorized
```

```
5.  `UseEndpoints()` adds endpoint execution to the middleware pipeline. It runs
   the code associated with the selected endpoint.
```

Adding Custom Middleware Components

→A custom middleware component that writes a pass through the pipeline.

EX:

```
await context.Response.WriteAsync("A pass through the pipeline!");
```

→We know we can use IapplicationBuilder to add built-in middleware components to the request pipeline.

1.UseX() → wide variety of middleware methods available via the interface.

2.there is a generic Use() method that can be used to process custom middleware components and call the next component in the pipeline.

→Use() - method accepts 2 parameters, the HTTP context and the next middleware component to be called.

→Performing actions inside of a web application correspond with sending requests,]

i.e. HTTP requests

→Each HTTP request and its specific data are encapsulated in the HTTP context object(ofType HttpContext).

1.When the HTTP context object is passed inside of the Use() method, we are just passing along the information about an individual HTTP rquest.

→Each middleware component is responsible for making a decision.

1.Pass the request to the next component for processing

2.Short-circuit the pipeline by returning a response.

→If the component passes the request to the next component in the sequence, the second parameter(next), provides this fuctionality.

Adding Terminal Middleware with IapplicationBuilder.Run()

Run() - Used when passing the request is no longer desired. Another way to begin returning the response to the user.

→considered terminal middleware because it terminates the pipeline.

1.may be at the end of th emiddleware pipeline and need to start returning a response.

**run()/Use does not have to be included in the pipeline.

1.Run() - the pipeline will just continue processing until it reaches the last component in the sequence.

2.Use() - if it doesn't invoke the next component the pipeline will be short-circuited and the response will be returned.

**ANY COMPONENTS AFTER RUN() will not be executed

Nesting Components

→Within the lambda expression passed to app.Use() and app.Run(), we can add a next parameter.

1.Allows us to call the "next" middleware in the pipeline.

EX: This middleware does nothing but call the next middleware

```
app.Use(async (context, next) =>
{
  await next();
});
```

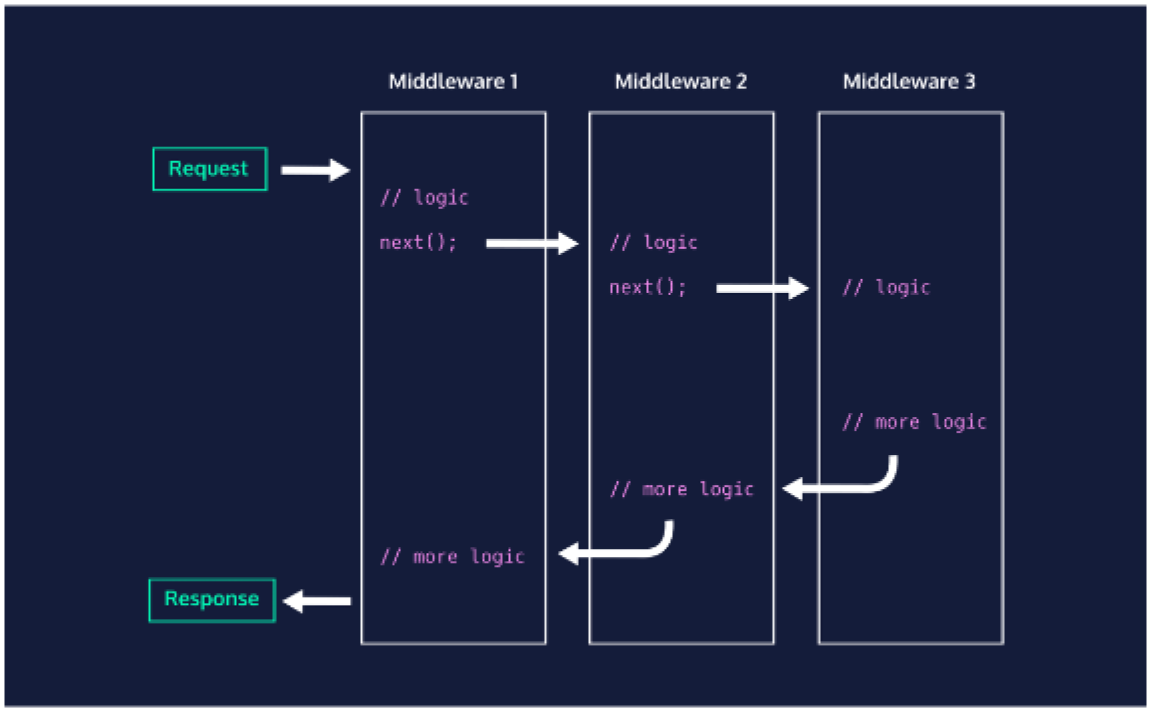
**In a typical app.Use() expression, there are 3 steps:

```
app.Use(async (context, next) =>
{
  // i. Do stuff before next middleware
  await context.Response.WriteAsync("Hello from 1st middleware\n");

  // ii. Execute next middleware
  await next();

  // iii. Do stuff after next middleware
  await context.Response.WriteAsync("Hello again from 1st middleware\n");
});
```

Nested aspect of middleware:



1.Middleware 1 writes to the response(i)

2.Middleware 1 invokes middleware 2(ii)

2.Middleware 2 and 3 execute, then control is passed back to middleware 1(ii)

3.Middleware 1 writes again to the response.(iii)

EX:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
      // Middleware 1
      app.Use(async (context, next) =>
      {
        await context.Response.WriteAsync("Hello from 1st middleware\n");

        await next();

        // Write to response again:
        await context.Response.WriteAsync("Hello again from 1st middlewar\n");
      });

      // Middleware 2
      app.Use(async (context, next) =>
      {
        await context.Response.WriteAsync("Hello from 2nd middleware\n");

        await next();

        // Write to response again:
        await context.Response.WriteAsync("Hello afain from 2nd middleware\n");      });

      // Middleware 3
      app.Run(async (context) =>
      {
        await context.Response.WriteAsync("Hello from 3rd middleware\n");
      });
```

Terminal response:

```
Hello from 1st middleware
Hello from 2nd middleware
Hello from 3rd middleware
Hello afain from 2nd middleware
Hello again from 1st middleware
```

HTTP **requests pass through middleware components in **sequential order**.

HTTP **responses pass through middleware components in **reverse order.**

UseDeveloperExceptionPage() → provides an exception page specifically designed for developers.

1.Should be placed before middleware components that are catching exceptions.

→includes stack trace, query string parameters, cookies, headers, and routing information.

If application is not running in development and an exception is thrown, an alternate error handling pathway will be followed

IwebHostEnvironment → Interface that provides information about the web hosting environment in which the application is running.

→an object that implements this interface (env) is passed into the Configure() method.

1.Inside the Configure() we can use this object to obtain more information about the environment.

2.Information can then be used to determine if the developer exception page should be shown.

UserExceptionHandler() → component can be used in production environments.

1.Catches log errors and route ysers to a general error page.

→error page can be customized but the significant detail is that this is a way to notify users that an issue has occurred while still hiding the internal details of our application.

2.Has several overloads, simplest is to pass in the name of the page(string) that should display if an exception is thrown.

EX:

```
if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }
      else
      {
        app.UseExceptionHandler("/Error");
      }
```

UseHttpsRedirection() → is a middleware compoonent used to capture HTTP requests and redirect them to the more secure HTTPS.

1.HTTPS is an extension of HTTP for securely transmitting communication between the web server and the browser.

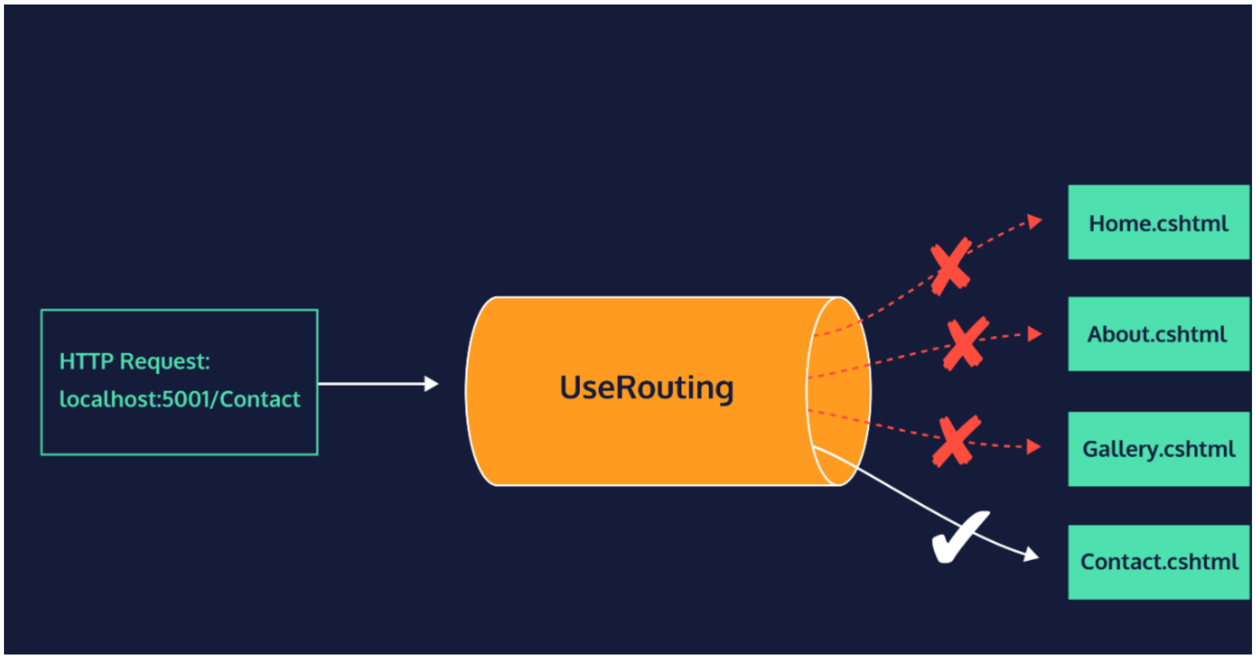
UseStaticFiles() → is available to ensure static file content is rendered alongside the HTML for our web applications.

<u>UseRouting()</u> → when configure() starts UseRouting() must be called to compare the HTTP request with the available endpoints and decide which is the best match.

1.Prior to UseRouting() being executed, the endpoint for all HTTP requests is null and it continues being null unless an appropriate match is found.

EX:

If requesting the Contact page, UseRouting() would take our HTTP request and match it with the /Contact endpoint.



<u>Useauthorization()</u> → checks the user's request with their authorization status.

**Importance of components being in the right order.

1.After UseRouting() has established the destination for the request, UseAuthorization() checks to see if the destination requires authorization and whether or not the user is authorized.

→If the authorization check passes, this component will pass the request to the next component in the pipeline.

→If authorization check does not pass, the pipeline will be short-circuited and either present the user with a login page or an error.

**authorization can also be implemented through the use of data annotations.

EX:

[Authorize]

Added to the page model for which we want users to obe authorized.

Enabling Razor Pages Endpoints

If the pipeline is not short-circuited, typically the last component in the pipeline is UseEndpoints().

UseEndpoints() → calls MapRazorPages() with a lambda expression to register the endpoint, and then executes the selected delegate that matched our HTTP request.

→Once the appropriate delegate for our HTTP request has been executed, a response is enerated.

→The response travels back through the middleware pipeline traversing each compooonent in reverse order until the response is returned to the user.

**Any code placed after UseEndpoints() will onlu be executed if no endpoint was found to match the request.

EX:

```
       app.UseEndpoints(endpoints =>
      {
        endpoints.MapRazorPages();
      });
```

FULL EX:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Http;

namespace CCsCoffeeShop
{
  public class Startup
  {
    public Startup(IConfiguration configuration)
    {
      Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
      services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
      env.EnvironmentName = "Production";
      if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }
      else
      {
        app.UseExceptionHandler("/Error");
      }
      app.UseHttpsRedirection();
      app.UseRouting();
      app.UseAuthorization();
      app.UseStaticFiles();
      app.UseEndpoints(endpoints =>
      {
        endpoints.MapRazorPages();
       });
      app.Run(async context =>
      {
        await context.Response.WriteAsync("The requested page is unavailable");
      });
    }
  }
}
```

Dependency Injection

→Problems solved by dependency injection:

1.if dependency changes any dependent class has to change.

2.code base is harder to maintain

3.Can’t test classes individually(behavior will change even if code doesn’t change within a class)

4.System isn’t flexible enough to allow us to swap the dependency for something similar

Dependency EX:

```
namespace DependencyInjection
{
  public class LoudSpeaker
  {
    public void Speak(string message)
    {
      Console.WriteLine(message.ToUpper() + "?");
    }
  }
}
namespace DependencyInjection
{
  public class Trainer
  {
    private readonly QuietSpeaker speaker;

    public Trainer()
    {
      speaker = new QuietSpeaker();
    }

    public void BeginTraining()
    {
      speaker.Speak("Time to sweat");
    }
  }
}
```

```
namespace DependencyInjection
{
  public class QuietSpeaker : ISpeaker
  {
    public void Speak(string message)
    {
      Console.WriteLine("..." + message.ToLower() + "...");
    }
  }
}
```

Principles to Solve Problems

→**Dependency Inversion Principle**

1.High-level modules should not depend on low-level modules. Both should depend on ` abstraction(i.e interfaces)

2.Abstractions should not depend on details. Details(concrete implementations) should depend on abstractons.

→In the example above, the Trainer class depended on the LoudSpeaker class. DIP states that the Trainer class should depend on some Ispeaker interface. LoudSpeaker should be designed to implement that interface.

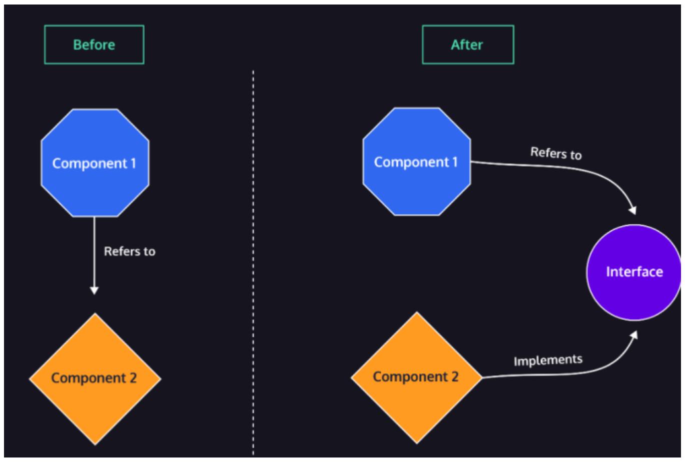
→**Inversion of Control Principle**

1.Methods defined by the user should be called from within the framework itself, rather than from the user's applicatuin code.

→AKA the "Hollywood Principle"

→Don't call us, we'll call you

→In the previous example the Trainer class should not instantiate another class. It may declare a certain dependency and some other part of the program will provide and manage that dependency.

Visualize the EX: 

Abstract the Dependency

→Implementing the Dependency Inversion Principle:

1.class should depend on abstraction.

2.Interface doesn't define an method bodies, but forces any class implementing the interface to Define them.

→making the class depend on the interface, allows the method body to change without Breaking the class.

EX:

```
namespace DependencyInjection
{
  public class Trainer
  {
    private readonly ISpeaker speaker;

    public Trainer()
    {
      speaker = new QuietSpeaker();
    }

    public void BeginTraining()
    {
      speaker.Speak("Time to sweat");
    }
  }
}
```

Inject the Dependency

→Implementing the Inversion of Control Principles:

1.Trainer shouldn't instantiate any dependency: LoudSpeaker or QuietSpeaker.

2.Should only declare a dependency, where the actual class will be instantiated elsewhere and inserted into the Trainer class.

→Dependency Injection is the actual insertion of the object.

→Many ways to do this, the example below uses constructor injection.

1.most commonly used in ASP.NET framework.

Common EX:

```
private static readonly ILogger _logger;
public void IndexModel(ILogger logger)
{
  _logger = logger;
}
```

1.In this example, IndexModel depends on some logging class.

2.Declares dependency in the form of an interface, Ilogger.

3.Never instantiates an actual class though.

**somewhere else a Logger object is constructed and passed(injected) into the IndexModel:

```
IndexModel i = new IndexModel(new Logger());
```

EX:

```
public class Trainer
  {
    private readonly ISpeaker _speaker;

    public Trainer(ISpeaker speaker)
    {
      _speaker = speaker;
    }

    public void BeginTraining()
    {
      _speaker.Speak("Time to sweat");
    }
  }
```

```
}
class Program
  {
    static void Main()
    {
      Console.WriteLine("Using trainer:");
      Trainer trainer = new Trainer(new LoudSpeaker());
      trainer.BeginTraining();
    }
  }
```

TERMS REVIEW

- The *Dependency Inversion Principle* (DIP) attempts to define and resolve the dependency problem through use of abstraction (interfaces).
- The *Inversion of Control* principle attempts to resolve the dependency problem by moving control of dependencies to a separate class.
- *Dependency Injection* is one of many design patterns that implements the IoC principle.

Applying DI to ASP.NET

→Anytime you log information or add a database to an ASP.NET application, these principles are used.

1.**Razor Pages** → each model depends on tools such as logger, database, and authentication.

→Also known as services

→Instead of instantiating the relevant objects and injecting them ourselves, ASP.NET gives us built-in features called IOC/DI containers.

→Each page model "requests" and accesses servuces by listing them within its constructor.

EX:

```
public class IndexModel : PageModel
{
 private readonly IService service;

 public IndexModel IService _service
 {
  service   _service;
 }
}
```

Injection in ASP.NET

→How it happens:

→Instead of : `new IndexModel(new Logger());`

→There is no need to type this because it is handled by ASP.NET's IoC container.

→Actual:

```
ConfigureServices(IServiceProvider services)
{
  services.AddScoped<ISomething, Something>();
}
```

→the code above basically say whenever a class asks an ISomething service, inject Something Instance into it.

→then request that service in our page models:

```
private readonly ISomething _something;
public IndexModel(ISomething something)
{
  _something = something;
}
```

→this approach to dependency injection makes the Dependency Inversion and Inversion of Control principles more obvious:

1.Modules should depend on abstractions: page model only references an interface, i.e abstraction.

2.Don't call us, we'll call you: we never see the servuce object instantiated or passed into a constructor. All handled by the framework.

Registering other services

→AddScoped() – each service has a "lifetime". In the previous example we wanted services to have a lifetime "scoped to each user".

→Three lifetime options:

1.AddScoped<T1, T2>() – sevice class is instantiated once per user, across the web app.

2.AddTransient<T1, T2>() – service class is instantiated every time it is requested.

3.AddSingleton<T1, T2>() – service class is instantiated once. That object is used throughout the web app, across users.

General Example:

```
public void ConfigureServices(IServiceProvider services)
{
  services.AddScoped<IAccount, CustomerAccount>();
  services.AddTransient<IResponder, Responder>();
  services.AddSingleton<ILogger, Logger>();
}
```

**register – register the abstraction and its implementation with the IoC Container so that other classes can access the implementation via dependency injection.

Built-in Services

→There are some common services that the ASP.NET framework has mad easy-to-register by providing unique AddX() methods.

EX:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddRazorPages();

  services.AddDbContext<CountryContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("CountryContext")));
}
```

→The AddRazorPages() method allows us to use page model features and the AddDbContext<T>() method connects our app to a database.

→We access database services:

```
public class IndexModel : PageModel
{
  private readonly CountryContext _context;

  public IndexModel(CountryContext context)
  {
    _context = context;
  }

  public IList<Country> Country { get;set; }

  public async Task OnGetAsync()
  {
    Country = await _context.Country.ToListAsync();
  }
}
```

→One exception to the Dependency Inversion Principle:

1.we depend on the class CountryContext instead of an interface because our app relies specifically on this type.

**More common services ASP.NET just assumes you need them:

→ILogger services

1.No need to register these in ConfigureServices(), they are already

Registered automatically.

Just request in the page model:

EX:

```
public class PrivacyModel : PageModel
{
  private readonly ILogger<PrivacyModel> _logger;

  public PrivacyModel(ILogger<PrivacyModel> logger)
  {
    _logger = logger;
  }
}
```

```
**Automatically registered services are determined by the host typem
which is defined in Program.cs
```