**ASP.Net Core Notes**

It is a cross-Platform, High Performance, Open-Source framework for building modern, cloud-based Internet-Connected applications.

ASP.Net core is a redesign of ASP.Net 4.x

**ASP.Net core benefits and features**

1. Cross platform
2. One programming model for MVC and Web API
3. Dependency Injection
4. Test Ability
5. Open Source
6. Modular

* No need to unload the project to edit the project file, just right click the project and select edit project.csproj.

**TFM - Target framework moniker**

**Name Abbreviation TFM**  
.Net Framework net net451 (if version is 4.5.1)

net472 (if version is 4.7.2)

.Net Core netcoreapp netcoreapp1.0 (if version is 1.0)

Netcoreapp2.2 (if version is 2.2)

**ASP.Net Core InProcess hosting**

Used kestrel server (process name: w3wp.exe or iisexpress.exe)

* Cross platform web server for ASP.Net Core
* Kestrel can be used by itself as an edge server
* The process used to host the app is dotnet.exe
* In process hosting uses only one server i.e. IIS
* InProcess hosting performance is good than OutOfProcess hosting

**ASP.Net Core OutOfProcess hosting**

* Process name is dotnet.exe
* Uses two web servers:
  + Internal web server – Kestral
  + External web server – Reverse Proxy Server (IIS, Nginx, Apache)

**Configurations sources in ASP.Net core**

Default order of configuration source is given below:

1. Files (appsettings.json, appsettings.{Envoirnment}.json)
2. User secrets
3. Environment variables (Launchsettings.ison)
4. Command-line Arguments (From cmd like **dotnet run MyKey="Hello from cmd"**)

**Note:** Later configuration sources will override the settings that are present in earlier configuration source.

Means appsettings.Development.json will override the settings in appsetting.json.

\*We can change the order of configurations also add the custom configuration

**Middleware in ASP.NET Core**

* Has access to both Request and Response.
* May simply pass the request to next Middleware
* May Process and then pass the request to next Middleware
* May handle the request and short-circuit the pipeline (Means avoids the next middleware in the pipeline)
* May process the outgoing response
* They are executed in the order they are added

Log -> Static Files -> MVC

<- <-

**Definition of Middleware**

It is a piece of software that can handle an HTTP Request or Response.

**Terminal Middleware**

It is a middleware that does not called next middleware in the request pipeline.

app.Run(async (context) =>

{

await context.Response.WriteAsync("From 2");

logger.LogInformation("From 2: Request and Response");

});

* If you want to use next middleware you need to use the “Use” method.

app.Use(async (context, next) =>

{

logger.LogInformation("MW1: IR");

await next();

logger.LogInformation("MW1: OR");

});

app.Run(async (context) =>

{

await context.Response.WriteAsync("From 2");

logger.LogInformation("MW2");

});

**Check the Output in output window in debut mode:** MW1: IR

MW2

MW1: OR

**How to use static files in .Net Core**

There are two requirements to use the static files

1. All static files should be in “wwwroot” folder.
2. Add Middleware app.UseStaticFiles();

**Note:** By default, default page name should be one of the following:

* default.htm
* default.html
* index.htm
* index.html
* To use the default files, we need to add middleware

app.UseDefaultFiles(); (this middleware should be above the UseStaticFiles();

app.UseStaticFiles();

It you want to use any other page as default page like “home.html” so we need to customize the behaviour of “UseDefaultFiles” middleware

DefaultFilesOptions defaultFilesOptions = new DefaultFilesOptions();

defaultFilesOptions.DefaultFileNames.Clear();

defaultFilesOptions.DefaultFileNames.Add("home.html");

app.UseDefaultFiles(defaultFilesOptions);

app.UseStaticFiles();

\* UseFileServer combines the functionality of UseDefaultFiles, UseStaticFiles and UseDirectoryBrowser middleware

FileServerOptions fileServerOptions = new FileServerOptions();

fileServerOptions.DefaultFilesOptions.DefaultFileNames.Clear();

fileServerOptions.DefaultFilesOptions.DefaultFileNames.Add("home.html");

app.UseFileServer(fileServerOptions);

**UseDeveloperExceptionPage**

* This is used to get the exception details.
* Must be plugin as early as possible.
* Contains Stack Trace, Query String, Cookies and HTTP headers.
* For customizing use DeveloperExceptionPageOptions object.
* If we don’t use this, we don’t get the proper exception details only HTTP ERROR 500.

if (env.IsDevelopment())

{

DeveloperExceptionPageOptions developerExceptionPageOptions = new DeveloperExceptionPageOptions();

developerExceptionPageOptions.SourceCodeLineCount = 2;

app.UseDeveloperExceptionPage(developerExceptionPageOptions);

}

**Environment Variables**

By default, Environment variable (ASPNETCORE\_ENVIRONMENT) is set to Production if not set explicitly.

We can also set the environment variable to operating system.

If we have environment variable set to both the places then launchSettings.json value will override the value in operating system environment variable.

By default, there are three environments:

* IsDevelopment
* IsStaging
* IsProduction

We can also set the custom environment like “UAT”, “QA” etc.

**Setup MVC in ASP.Net Core**

Step -1 Add the MVC services to the Dependency Injection Container

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

}

Step -2 Add MVC middleware to the Request Pipeline

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILogger<Startup> logger)

{

if (env.IsEnvironment("UAT"))

{

app.UseDeveloperExceptionPage();

}

app.UseStaticFiles();

app.UseMvcWithDefaultRoute();

}

**Difference between services.AddMvc() and services.AddMvcCore()**

AddMvcCore() method only adds the core MVC services

AddMvc() method adds all the required MVC services

AddMvc() method internally calls the AddMvcCore() method.

So if we are using AddMvc() method, then there is no need to explicitly call AddMvcCore() method again.

**Controller**

* Handles the incoming http request
* Builds the model AND
* Return the model data to the caller if we are building an API OR
* Select a View and pass the model data to the view
* The View then generates the required HTML to present the data.

**View**

* View() or View(object model): Looks for a view file with the same name as the action method
* View(string viewName)
  + Looks for a view file with your own custom name
  + You can specify a view name or a view file path
  + View file path can be absolute or relative
  + With absolute path .cshtml extension must be specified
  + With relative path do not specify the file extension .csthml
* View(string viewName, object model)

**Model**

Model => Employee + EmployeeRepository

**ViewData**

* Dictionary of weakly typed objects
* Use string keys to store and retrieve data
* Dynamically resolved at runtime
* No compile-time checking and intellisens

**ViewBag**

* Use dynamic property to store and retrieve data.
* Dynamically resolved at runtime
* No compile-time checking and intellisens

**Sections in Layout Page**

* A Section in Layout View provides a way to organize where certain page elements should be placed.
* A section can be optional or mandatory
* A section in the layout view is rendered at the location where RenderSection() method is called.
  + Code in Layout Page

@if (IsSectionDefined("Scripts"))

{

@RenderSection("Scripts", required: false)

}

* + Scripts in details page

@section Scripts{

<script src="~/js/Custome.js"></script>

}

**\_ViewStart.cshtml**

* Code in ViewStart is executed before the code in an individual view.
* Move the common code such as setting the Layout property to ViewStart
* ViewStart reduces code redundancy and improves maintainability
* We can have the multiple ViewStart file and it is hierarchical.
* ViewStart in child folder will override the ViewStart in parent folder.

**\_ViewImports**

* \_ViewImports file is places in the Views folder.
* Used to include the common namespaces
* To include the common namespaces, use @using directive
* Other supported directives

1. @addTagHelper
2. @removeTagHelper
3. @tagHelperPrefix
4. @model – To include the model(for e.g. Employee, Customer)
5. @inherits
6. @inject – To Inject the Service(for e.g. IEmployeeRepository) in a view

* ViewImports file is hierarchical.
* ViewImports in child folder will override the ViewImports in parent folder.

**Difference between [Route("~/")] and [Route("")]**

**[Route("~/")] or [Route("/")]** : Access the action from the root like <https://localhost:44394/>

**[Route("")]** : Access the action from the controller like <https://localhost:44394/>Employee.

**4 ways to generate hyperlinks**

<**a** **asp-controller**="employee" **asp-action**="details" **asp-route-id**="@item.Id" class="btn btn-primary">View @item.Id</**a**>

<a href="@Url.Action("details","employee", new { id = @item.Id })" class="btn btn-primary">show</a>

@Html.ActionLink("View","details","employee", new { id = @item.Id })

<a href="/Employee/Details/Details/@item.Id" class="btn btn-primary">display</a>

**TagHelpers**

* TagHelpers generates link based on application route templates.
* If we later change our templates the links generated by TagHelpers will automatically reflect those changes and generated links will work.

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "jeet/{controller=Employee}/{action=Index}/{id?}");

});

* <a href="/Employee/Details/Details/@item.Id" >display</a>

will generate :

<a href="/Employee/Details/1">display</a>

* <a asp-controller="employee" asp-action="details" asp-route-id="@item.Id" >View @item.Id</a> will generate:

<a href="/jeet/Employee/Details/1">View</a>

**Image Tag Helper**

* **Cache bursting behaviour:** It generates the hash code corresponding the image, with this hash code browser detects weather the content of image has been changed, if the content has changes then it loads the image from the server otherwise loads it from the browser cache.
* <**img** class="card-img-top" **src**="~/images/B1.jpeg" **asp-append-version**="true" />
* In the above image tag we have asp-append-version, this append the hash code corresponding to the image.

**ASP.NET Core Model Binding**

To bind the request data to the controller action method parameters, model binding looks for data in the HTTP request in given order:

* Form Values
* Route Values
* Query Strings

**Service Type In the scope of a given http request Across different http requests**

AddSingleton Same Instance Same Instance

AddScoped Same Instance New Instance

AddTransient New Instance New Instance

**Entity Framework Core**

* ORM (Object-Relational Mapper)
  + Generates the SQL that the underline database understand
* Lightweight, Extensible and Open Source
* Like Dot Net Core EF Core also works Cross Platform
* Microsoft’s Official Data Access Platform

**AddDbContextPool(Introduced in ASP.NET Core 2.0)** – When every time an instance of this AppDbContext class is requested, instead of creating a brand new instance ASP.NET Core checks if there is an instance available in the DbContextPool then that instance is returned instead of creating a brand new instance of this class.

It is similar to database connection pooling in ADO.NET.

From performance point of view AddDbContextPool is better then AddDbContext method

services.AddDbContextPool<AppDBContext>(

option => option.UseSqlServer(\_config.GetConnectionString("EmployeeDBConnection")));

**Migration Command in Entity Framework Core**

Add-Migration – To add new migration

*Add-Migration InitialMigration*

Update-Database – To update database with latest migration

Remove-Migration – To remove the latest migration that is not yet applied to database.

**“\_\_EFMigrationsHistory”** table is used to keep track of the migrations that are applied to database

ModelSnapshot.cs file contains the snapshot of the current model and is used to determine what has changed when adding the next migration

**Note:** To remove a migration that is already applied to database

* First use the Update-Database command to undo the database changes applied by the migration
* Next, use Remove-Migration command to remove the migration code file.

To get the physical path of wwwroot folder we use IHostingServiceEnvironment

GUID- Global Unique Identifier

We can’t use “href” attributes starting with the tag helpers like 'asp-route-' or an 'asp-action', 'asp-controller', 'asp-area', 'asp-route', 'asp-protocol', 'asp-host', 'asp-fragment', 'asp-page' or 'asp-page-handler' attribute

**Difference between “UseStatusCodePagesWithRedirects” and “UseStatusCodePagesWithReExecute”**

**UseStatusCodePagesWithRedirects:**

* Because a redirect is issued, the URL in the address bar changes.
* Returns a success status code(200) when actually an error occurred which isn’t semantically correct.

**UseStatusCodePagesWithReExecute:**

* Re-execute the pipeline and returns the original status code (404 for example)
* As it re-execute the pipeline and not issue a redirect request, we also preserve the original URL (/foo/bar) in the address bar

302 – URI is temporarily changed or not available

**Global Exception Handling in ASP.NET Core MVC**

* Add Exception handling middleware to the request processing pipeline

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

}

* Implement Error Controller

[Route("Error")]

[AllowAnonymous]

public IActionResult Error()

{

var exceptionDetails = HttpContext.Features.Get<IExceptionHandlerPathFeature>();

ViewBag.ExceptionPath= exceptionDetails.Path;

ViewBag.ExceptionMessage = exceptionDetails.Error.Message;

ViewBag.StackTrace = exceptionDetails.Error.StackTrace;

return View("Error");

}

* Implement Error View

<h3>Exception Details:</h3>

<div class="alert alert-danger">

<h5>Exception Path</h5>

<hr />

<p>@ViewBag.ExceptionPath</p>

</div>

<div class="alert alert-danger">

<h5>Exception Message</h5>

<hr />

<p>@ViewBag.ExceptionMessage</p>

</div>

<div class="alert alert-danger">

<h5>Exception Stack Trace</h5>

<hr />

<p>@ViewBag.StackTrace</p>

</div>

**ASP.NET Core Logging Provider:** Logging provider stores or displays logs

* **Console Logging Provider:** Displays log on the console
* **Debug Logging Provider:** Displays logs on the Debug window in visual Studio

**Logging to ASP.NET Core using NLOG**

* Install NLog.Web.AspNetCore nuget package
  + For configuration NLog looks for a file named “nlog.config” in the root project folder
* Create nlog.config file as below

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<!-- the targets to write to -->

<targets>

<!-- write logs to file -->

<target name="allfile" xsi:type="File"

fileName="G:\DemoLogs\nlog-all-${shortdate}.log"/>

</targets>

<!-- rules to map from logger name to target -->

<rules>

<!--All logs, including from Microsoft-->

<logger name="\*" minlevel="Trace" writeTo="allfile" />

</rules>

</nlog>

* Enable copy to “bin” folder
* Add NLog as one of the Logging providers

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.ConfigureLogging((hostingContext, logging) =>

{

logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));

logging.AddConsole();

logging.AddDebug();

logging.AddEventSourceLogger();

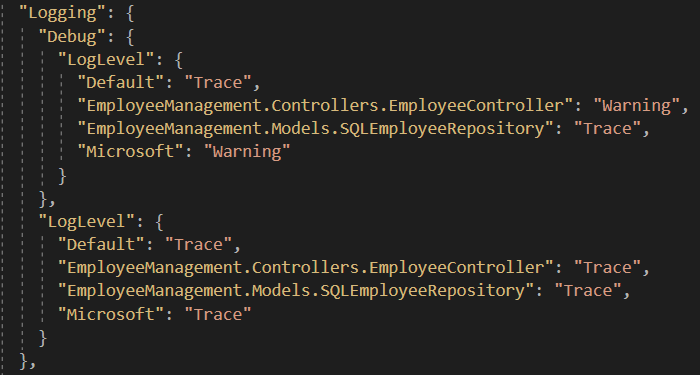
logging.AddNLog();

})

.UseStartup<Startup>();

**Logs can be filtered by:**

* Log Category
* Logging Provider
* Even Both



**How to setup ASP NET Core Identity**

1. Your application “DbContext” class must inherit from “IdentityDbContext”
   * public class AppDBContext : IdentityDbContext
2. Add ASP.NET Core Identity Services.
   * services.AddIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<AppDBContext>();

1. Add Authentication middleware

app.UseStaticFiles();

**app.UseAuthentication();**

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "jeet/{controller=Employee}/{action=Index}/{id?}");

});

1. Generate ASP.NET Core Identity Tables
   * Add-Migration
   * Update-Database

**isPersistent:** false => When we want to use **Session Cookie**. A Session Cookie will automatically be lost after we close the browser window

**isPersistent:** true => When we want to use **Permanent Cookie**. A Permanent cookie is retained on the client machine even after the browser window is closed

**How to prevent Open Redirect Attacks**

* Use LocalRedirect(returnUrl)
  + OR
* Url.IsLocalUrl(returnUrl)

**To perform client side validation we need below files in the given order**

<script src="jquery.js"></script>

<script src="jquery.validate.js"></script>

<script src="jquery.validate.unobtrusive.js"></script>

**Remote Attribute is used to make AjaxCall to controller action**

First decorate the property with [Remote] Attribute

For e.g.

[Remote(action: "IsEmailInUse", controller: "Account")]

public string Email { get; set; }

Also make sure we need below jquery files in the given order.

<script src="jquery.js"></script>

<script src="jquery.validate.js"></script>

<script src="jquery.validate.unobtrusive.js"></script>

**Way to Extend IdentityUser Class**

Create a class like below:

public class ApplicationUser : IdentityUser

{

public string City { get; set; }

}

Then update the signature of AppDbContext like below:

public class AppDBContext : IdentityDbContext<ApplicationUser>

**Variation in Authorize Attribute**

1. Member of either **Admin or HR** role have access to “Administration” controller

[Authorize(Roles = "Admin,HR")]

public class AdministrationController : Controller

{

}

1. Users have to be a member of both **Admin and HR** role to have access of “Administration” controller

[Authorize(Roles = "Admin")]

[Authorize(Roles = "HR")]

public class AdministrationController : Controller

{

}

By default in Entityframework Core, if we delete the parent row the child row will automatically deleted, by default Cascading referential integrity constraint is set to “Cascade” option.