**LAB1: First App**

1. Show proj in windows explorer
2. The important files/folders in MVC app are:
   * ROOT Directory: Solution file 🡪 open project (1 sol <=> 1 proj)
   * ROOT Directory: global.json 🡪 prj specific settings 🡪.NET SDK used 🡪src & test folders
   * Program.cs: Main() method
   * Web.config/appsettings.json: DB connections, logging settings, etc
   * launchsettings.json: Launch environment settings, set environment variable values + IIS in debug
     + IIS Settings
     + Port to run
   * bundleConfig.json: minify options and bundling options
   * project.json: Nuget packages used and the version used.
   * startup.cs: Serve each incoming web request and provide appropriate response
3. .csproj file and .sln file
4. StartUp.cs
   * ConfigureService:
     1. Add services (service is a component that is intended for common consumption)
     2. Mainly used to configure Dependency Injection.
     3. It is called before Configure() method
   * Configure:
     1. How MVC will respond to request, we actually build the HTTP request pipeline.
     2. Middlewares 🡪 What is middleware ?
     3. Order of Middleware
     4. EXAMPLE, if we want serve request for static files we will need to add some code here
5. appsettings.json
   * Add new item ‘appsettings.json’
   * Add following code in startup Configure()
   * Add following code in appsettings.json

|  |
| --- |
| {  "message": "Hello, World!"  } |

* + Add following code in Configure() method 🡪 PLACE IT BEFORE ANY ENDPOINT REGISTERATION

|  |
| --- |
| app.**Run**(async (context) => {  await context.Response.WriteAsync("<b>Hello World!</b>");  }); |

* + AllowedHosts filters the host which is allowed to access the application, the wild character ‘\*’ means everything is allowed.
  + The appsettings.json is a default configuration source for .NET Core

1. RUN DEBUG & BUILD <http://localhost:56789/>
2. ‘BIN’ folder
3. SWAP the position of Run() method with UseEndpoint() to show change in behaviour

**LAB2: Serving static Files**

1. Add HTML file **index.html** INSIDE wwwroot and add some message or HTML element inside it.

|  |
| --- |
| <html>  <body>  <h1>This is the index file</h1>  </body>  </html> |

1. Remove ~~app.UseStaticFiles();~~ if it is already present in StartUp.cs
2. Run and browse <http://localhost:56789/index.html> it will throw **ERROR**
3. Add Nuget package **Microsoft.AspNet.StaticFiles**
4. Register middleware in Configure() service using method .UseStaticFiles();

|  |
| --- |
| app.UseIISPlatformHandler();  app.UseDeveloperExceptionPage();  app.UseRuntimeInfoPage();  **app.UseStaticFiles()**; |

1. Run and browse <http://localhost:56789/index.html>
2. Make index.html your default landing page
3. Register middleware in Configure() service using method .UseDefaultFiles();

|  |  |
| --- | --- |
| app.UseIISPlatformHandler();  app.UseDeveloperExceptionPage();  **app.UseDefaultFiles()**;  **app.UseStaticFiles()**; | USE DEFAULT SHOULD BE FIRST |

1. Run and browse <http://localhost:56789/>
2. **ORDER of MIDDLEWARE**
3. 2 middlewares or 1 middleware

|  |  |
| --- | --- |
| app.UseDefaultFiles();  app.UseStaticFiles(); | app.UseFileServer(); |

**LAB2.1: Serving static Files (OPTIONAL)**

1. Add Nuget package **Microsoft.AspNet.Diagnostics**
2. Register middleware in Configure() service using method .UseRuntimeInfoPage();

|  |
| --- |
| app.UseIISPlatformHandler();  app.UseDeveloperExceptionPage();  app.**UseRuntimeInfoPage()**; |

1. Run and browse <http://localhost:56789/runtimeinfo>

**LAB3: My First MVC Application**

1. Create simple MVC project
2. Explain Controller code
3. Explain View
4. Newly installed packages appear in \*.csproj file the one which are by default do not appear
5. Show RedirectToAction(“ACTION”,”CONTROLLER”);
6. Native type response

|  |  |
| --- | --- |
| public string Index()  {  return “See string instead of view”;  } | DO NOT USE HTML |

**LAB4: Razor Views**

1. Use same project created in **LAB3**.
2. Add following razor syntax to the view

|  |
| --- |
| <ul>  @{string[] names = {"Rohit","Virat","Surya"}}  @foreach(var n in names)  {  <li>@n</li>  }  </ul> |

1. Run and show the view
2. Integrate IF-ELSE

|  |
| --- |
| <ul>  @{string[] names = {"Rohit","Virat","Surya"}}  @foreach(var n in names)  {  if (n == “Rohit”)  {  <li>**@**n<b>(C)</b></li>  }  else  {  <li>**@**n</li>  }  }  </ul> |

1. HTML helper

|  |
| --- |
| @Html.Label(“This is a label”) |

**LAB5: MVC Routing**

1. Use same project created in **LAB3**.
2. Remove any routing logic from Startup.cs [~~app.UseRouting()~~ & ~~app.UseEndpoints()~~]
3. Remove RouteConfig.cs if it exists.
4. Check the **ConfigureServices()** method in startup.cs if it has MVC reference.

|  |
| --- |
| **app.AddMvcCore(options => options.EnableEndpointRouting = false)**; |

1. Register middleware in **Configure()** service using method .UseStaticFiles();

|  |
| --- |
| **app.UseMvc(**ConfigureRoute**)**; |
| private void **ConfigureRoute**(IRouteBuilder routeBuilder)  {  routeBuilder.MapRoute("Default", "{controller = Home}/{action = Index}/{id?}");  } |

1. Run and browse <http://localhost:56789/home/index>
2. Demonstrate the purpose of ‘**?**’ in the pattern by adding DEBUG POINT and parameter in action method.

|  |
| --- |
| public class HomeController {  public ActionResult Index(int? id) {  return new View();  } |

1. In the same application remove the custom ConfigureRoute() method you added in Configure() method

|  |
| --- |
| **app.UseMvc(~~ConfigureRoute~~)**; |

1. Open home controller and decorate the class and action methods as follows:

|  |
| --- |
| [Route("MyHome")]  public class HomeController {  [Route ("MyIndex/{id?}")]  public ActionResult Index() {  return new View();  } |

1. Run and browse <http://localhost:56789/myhome/myindex>
2. Why attribute based routing?
   * Simple and readable
   * Support for COMPLEXITY OF ROUTE
3. **Route Constraints**

|  |  |
| --- | --- |
| [Route("MyHome")]  public class HomeController {  [Route("MyIndex/{ id:**int:min**(100)}")]  public ActionResult Index() {  return View();  } | max {x:max(10)}  minlength {x:minlength(10)}  maxlength {x:maxlength(10)}  length {x:length(10)}  range {x:range(40,50)}  alpha {x:alpha}  .  . |

1. BOTH ROUTINGS CAN BE COMBINED ONE (*ADD A NEW CONTROLLER DEMO*)

|  |  |
| --- | --- |
| ~~[Route("MyHome")]~~  public class HomeController {  ~~[Route("MyIndex")]~~  public ActionResult Index() {  return View();  } | [Route("MyDemo")]  public class **DemoController** {  [Route("MyIndex")]  public ActionResult Index() {  return Content(“DEMO”);  } |

**LAB6: Models, ViewData & ViewBag**

1. Create a new MVC project
2. Add a folder ‘Models’ and add Employee class as seen below:

|  |
| --- |
| public class Employee  {  public string FirstName { get;set; }  public string LastName { get;set; }  public int Salary { get;set; }  } |

1. Add 2 action methods for both ViewData & ViewBags

|  |  |
| --- | --- |
| public ActionResult **Data**()  {  Employee e = new Employee();  e.FirstName = “Bill”;  e.LastName = “Gates”;  e.Salary = 1000;  ViewData[“Employee”] = e;  return new View(“VDView”);  } | public ActionResult **Bag**  ()  {  Employee e = new Employee();  e.FirstName = “Tony”;  e.LastName = “Stark”;  e.Salary = 1000;  ViewBag.Employee = e;  return new View(“VBView”);  } |

1. Add 2 Views for above action methods:

|  |
| --- |
| using ApplicationXYZ.Models |
| <div>  @{  ApplicationXYZ.Models.Employee emp = (ApplicationXYZ.Models.Employee)ViewData[“Employee”];  }  </div> |
| <div>  @{  ApplicationXYZ.Models.Employee emp = (ApplicationXYZ.Models.Employee)ViewBag.Employee;  }  </div> |
| <div>  <b>Employee Details</b><br/>  <span>Employee Name: **@emp**.FirstName **@emp**.LastName</span><br/>  <span>Employee Salary: @emp.Salary.ToString(“C”)</span><br/>  </div> |

1. Show that ViewData or ViewBag cannot be used to send data to Controller with debugging.

|  |
| --- |
| @{  ViewData[“Title”] = “My Home Page”;  } |

**LAB7: Strongly typed View**

1. Create simple MVC project
2. Add a folder ‘Models’ and add Employee class as seen below:

|  |
| --- |
| public class Employee  {  public string FirstName { get;set; }  public string LastName { get;set; }  public int Salary { get;set; }  } |

1. Modify the Index.cshtml add following HTML:

|  |
| --- |
| @using StrongTypeViewDemo.Models  **@model** Employee |
| <div>  <b>@Html.LabelFor(e => e.,”First Name:”)</b>**@Model**.FirstName <br/>  <b>@Html.LabelFor(e => e.,”Last Name:”) </b>**@Model**.LastName <br/>  <b>@Html.LabelFor(e => e.,”Salary:”)</b>**@Model**.Salary  </div> |

1. In the Index() action method for the view add following code:

|  |
| --- |
| public ActionResult Index()  {  Employee e = new Employee();  e.FirstName = “Bruce”;  e.LastName = “Banner”;  e.Salary = 1000;  return new View(e);  } |

**LAB8: ModelBinder**

1. Create simple MVC project ‘ModelBinderDemo’
2. Add a folder ‘Models’ and add Employee class as seen below:

|  |
| --- |
| public class Employee  {  public string FirstName { get;set; }  public string LastName { get;set; }  public int Salary { get;set; }  } |

1. Modify the Index.cshtml add following HTML:

|  |
| --- |
| @using ModelBinderDemo.Models  **@model** Employee |
| <form method=”post” action=”**/Employee/SaveEmployee**”>  <span>First Name: </span><input type=”text” name=”FirstName” value=”**@Model**.FirstName” /><br/>  <span>Last Name: </span><input type=”text” name=”LastName” value=”**@Model**.LastName” /><br/>  <span>Salary: </span><input type=”text” name=”Salary” value=”**@Model**.Salary” /><br/>  <input type=”submit” name=”**btnSubmit**” value=”Save” />&nbsp;<input type=”submit” name=”btnReset” value=”Reset” />  </form> |

1. In the Index() action method for the view add following code:

|  |
| --- |
| public ActionResult Index()  {  Employee e = new Employee();  e.FirstName = “”;  e.LastName = “”;  e.Salary = 0;  return new View(e);  } |

1. Add a new action method SaveEmploye() as follow:

|  |
| --- |
| ActionResult SaveEmployee(int id,string fName,string lName,int sal,string btnSubmit)  {  if(btnSubmit.Equals(“Save”))  {  return Content(id+ “|” + fName + “|” + lName + “|”+ sal);  }  else  (  return RedirectToAction(“Index”);  }  } |
| public ActionResult SaveEmployee(Employee emp, string btnSubmit)  {  if(btnSubmit.Equals(“Save”))  {  return Content(emp.FirstName + “|” + “emp. LastName” + “|” + emp. Salary);  }  else  (  return RedirectToAction(“Index”);  }  } |

1. Run the application

**LAB9: EFCore**

1. Check connectivity with SQL database (server name, credentials, etc)
2. Create simple MVC project ‘EFCoreDemo’
3. Search and add following Nuget packages in it
   * Microsoft.EntityFrameworkCore
   * Microsoft.EntityFrameworkCore.SqlServer
   * Microsoft.EntityFrameworkCore.Tools
4. Check if the application has ‘appsettings.json’ or else add file to application
5. Add following code to ‘appsettings.json’

|  |
| --- |
| “ConnectionStrings”: {  “DefaultConnection”: “server=ABC;database=OrganizationDB;user id=sa;password=\*\*\*;trustservercertificate=true;”  } |

1. Add a folder ‘Models’ and add Employee class as seen below:

|  |
| --- |
| public class Employee  {  public int EmployeeId { get;set; }  public string FirstName { get;set; }  public string LastName { get;set; }  public int Salary { get;set; }  } |

1. Add a new using statement to this model class

|  |
| --- |
| using System.ComponentModel.DataAnnotations; |

1. Now add a new annotation to the Employee model class

|  |
| --- |
| public class Employee  {  **[Key]**  public int EmployeeId { get;set; }  public string FirstName { get;set; }  public string LastName { get;set; }  public int Salary { get;set; }  } |

1. Since you want to show a list of employees in the folder ‘Models’ add another class ListOfEmployees as seen below:

|  |
| --- |
| public class ListOfEmployees  {  public List<Employee> EmployeeList { get;set; }  } |

1. Create new folders “Infrastructure”, “BusinessLayer” & “DataAccessLayer” in the root folder
2. Add a new class **MyDbContext** in folder ‘Infrastructure’ and add the following code:

|  |
| --- |
| **using Microsoft.EntityFrameworkCore;**  **using EFCoreDemo.Models;**  public class MyDBContext : **DbContext**  {  public **MyDbContext**(DbContextOptions<MyDbContext> options) : base(options)  {  }  public DbSet<Employee> **Employees**;  } |

1. Add a new class **OrganisationDAL** in the folder ‘DataAccessLayer’ and add the following code:

|  |
| --- |
| **using EFCoreDemo.Infrastructure;**  **using EFCoreDemo.Models;**  public class OrganisationDAL  {  private readonly MyDbContext context;  public OrganisationDAL(**MyDbContext** mydbContext)  {  this.context = mydbContext;  }  public List<Employee> GetEmployees()  {  return this.context.Employees.ToList();  }  } |

1. Add a new class **OrganisationBL** in the folder ‘BusinessLayer’ and add the following code:

|  |
| --- |
| **using EFCoreDemo.DataAccessLayer;**  **using EFCoreDemo.Infrastructure;**  **using EFCoreDemo.Models;**  public class OrganisationBL  {  private readonly MyDbContext context;  public OrganisationBL(**MyDbContext** mydbContext)  {  this.context = mydbContext;  }  public List<Employee> GetEmployees()  {  return new OrganisationDAL(this.context).GetEmployees();  }  } |

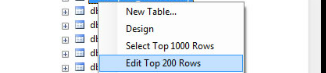
1. Modify the ‘Startup’ class with following code:

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using MyMvcApp.Data;    public void **ConfigureServices**(IServiceCollection services)  {  services.AddDbContext<MyDbContext>(options =>  options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));  services.AddControllersWithViews();  } |

1. Use the **package manager console** to run the migration command:

|  |
| --- |
| > **Add-Migration** IntialCreate **-OutputDir** Infrastructure\Migrations  > **Update-Database** |

1. Check the SQL server to see if DB is created, edit the table and add employee records



1. Modify the controller class **HomeController.cs** and add new action method GetEmployeeList()

|  |
| --- |
| public class EmployeeListController : Controller  {  private **MyDbContext context**;  public HomeController(MyDbContext myDbContext)  {  this.context = myDbContext;  }  public ListOfEmployees GetEmployeeList()  {  OrganisationBL empBL = new OrganisationBL(this.context);  ListOfEmployees lstEmp = empBL.GetEmployees();  return View(“MyList”,lstEmp);  }  } |

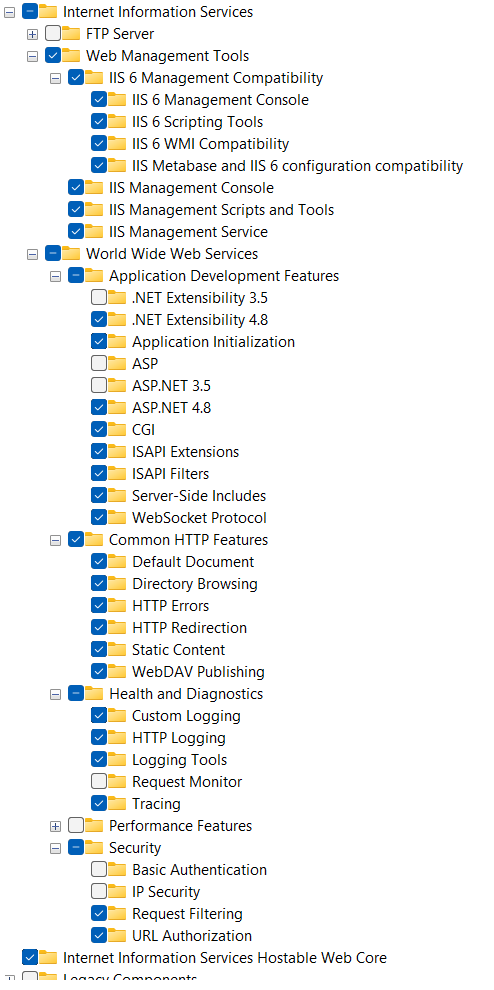
1. Add a new view **MyList.cshtml** add following HTML:

|  |
| --- |
| @using EFCoreDemo.Models  **@model** ListOfEmployees |
| <div>  @foreach(var emp in **@Model**)  {  <b>First Name: </b>**@emp**.FirstName <br/>  <b>Last Name: </b>**@emp**.LastName <br/>  <b>Salary: </b>**@emp**.Salary  <br/>  }  </div> |

1. RUN the application and visit <http://localhost:56789/home/getemployeelist> to check if employee list is visible

**LAB10: Deployment & Hosting**

1. Check to see ‘http://localhost:80’ is opening, if not then IIS is not enabled.
2. Search for ‘Turn Windows features on or off’ and open the **Windows Feature** window
3. In side the window search for ‘*Internet Information Services*’ feature
4. Select following features:



1. Windows will searh the features, download and install them on the machine
2. Check to see ‘http://localhost:80’ is opening, if not then restart machine
3. Build the ASP.NET Core MVC application & right-click the MVC application and select the option to ‘publish’
4. Browse and select appropriate location on physical drive to publish the application and click ‘Publish’
5. Open IIS and expand to ‘Sites’, right-click **Sites** and select ‘Add Website’ to open the dialog
6. Add the website values as seen below:

|  |  |  |  |
| --- | --- | --- | --- |
| Site Name | **app.deploydemo.com** | ApplicationPool | app.deploydemo.com |
| Physical Path |  | | |
| Type | http | IP Address | All Unassigned |
|  |  | Port | **81** |
| Host name | **app.deploydemo.com** | |  |

1. See the right side menu to browse the website, if it does not open change the DNS settings
2. Browse & open hosts file ‘C:\Windows\System32\drivers\etc\hosts’
3. Add the following un-commented line inside the file

|  |
| --- |
| 127.0.0.1 **app.deploydemo.com** |

1. Browse [http://**app.deploydemo.com:81**](http://app.deploydemo.com:81) to open your application

XSRF, or Cross-Site Request Forgery, is a type of cyber attack where an attacker tricks a user into performing unintended actions on a web application that the user is authenticated to. This is typically done by getting the user to click on a malicious link or submit a malicious form that makes a request to the web application, which then performs the action as if it was initiated by the user.

Here's a simple example to illustrate this:

Let's say you're logged into your online banking account and an attacker sends you an email with a link that says "Click here to win a prize!" The link actually points to a malicious website that makes a request to your bank's website to transfer money from your account to the attacker's account. If you click on the link, the request will be sent to your bank's website, and since you're already logged in, the bank's website will think the request came from you and perform the transfer.

To prevent XSRF attacks, web applications can use techniques such as:

1. Token-based validation: The web application generates a unique token for each user session and includes it in every request. The token is verified on the server-side to ensure the request came from the user's session.
2. Header-based validation: The web application sets a custom header in every request, and the server verifies the presence and value of the header to ensure the request came from the user's session.
3. Double-submit cookie: The web application sets a cookie with a unique value and includes the same value in every request. The server verifies that the value in the request matches the value in the cookie to ensure the request came from the user's session.