Minutes of Session: .NET Core, .NET Security & Reliability

Date: 29th July 2025
Topic:
Duration: Full Day

Participants : Batch C1

<table>
<tr><td rowspan="4">Day 10</td><td colspan="2">.NET Core</td></tr>
<tr><td rowspan="3">Introduction to .NET Core</td><td>DOT NET CORE
Free, open source crossplatform framework used for creating modern, cloud based , web enabled apps.</td></tr>
<tr><td>What is .NET Core</td></tr>
<tr><td>Benefits of .NET core
-Cross-platform
- High performance and scalability
- Modular and lightweight
-Improved CLI support</td></tr>
<tr><td>What is new in .NET Core
<u>-Built in Dependency Injection(DI)</u>: Ex1.Sandwich
Option 1: Make it yourself (Hard Coded approach)
Option 2: Order it from restaurant(DI)
Ex2. Building a Toy car :
Option 1: Glue all the parts together.
Option 2: Arranging them in a loosely coupled manner so that wheels, engine and colors can be changed later

Class Car {
Private Engine engine = new Engine() // hard coded dependency
}

When we are using DI:
Class car{
Private Engine engine;
Public Car(IEngine _engine) {
_engine = engine } // Injected dependency
}

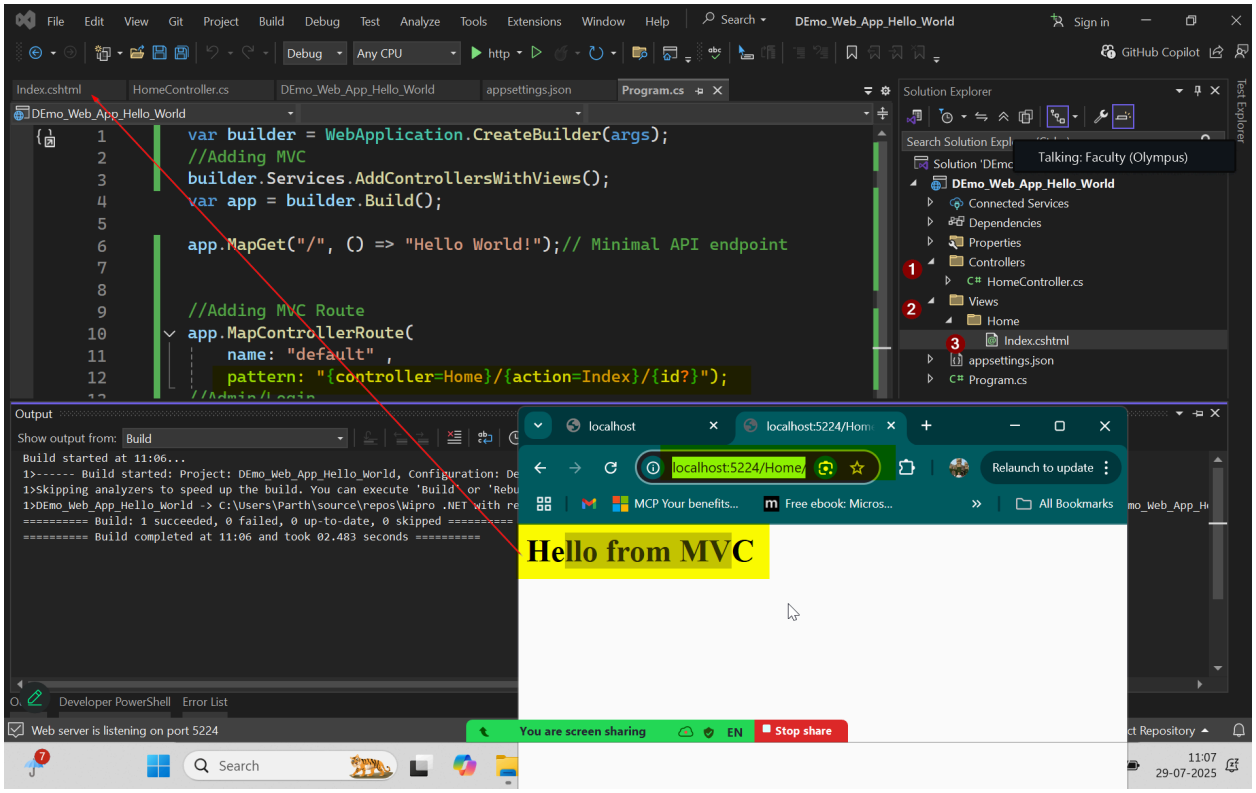
Why DI ?
Easier to change parts (Swap dependencies)
Easier testing(using fake/mock parts)</td></tr>
</table>

		Less Spaghetti code(no tight coupling) **-Minimal hosting model : A streamlined way to bootstrap .NET app with minimal boilerplate code** **_ No Startup.cs was present everything was inside Program.cs** **- Unified Web + API model:** **It was easy to merge WebApplication(MVC) with WEb API(REST) in to single programming model** **MVC : Model view controller is design pattern used in web development to separate Application logic into following components :** **1. Model: Data** **2. Views : Displaying data** **3. Controller : handles user input, process requests and return responses.** **Burger Truck : Cooking, Serving & Inventory etc | 100-200/Day** **Burger restaurant :** **Kitchen : Separate team for cooking** **Service: Separate ppl taking care of serving** **Owner: Inventory + feedback etc** **Burger chain of restaurants** **Old .NET Framework limitations :** **A Web API was separate from MVC** **API Controllers** **Routing was tricky** **Response generated was in terms of views(HTML) and API(JSON)**
		.NET Core vs .NET Framework
		First .NET Core Application.
		Building .NET Core Applications
	Middleware and Static Files	Understanding middleware in ASP.NET Core
		Configuring and using middleware components
		Serving static files (HTML, CSS, JavaScript)
		Security considerations with static files
	Introduction to Razor Pages	Overview of Razor Pages architecture
		Advantages of Razor Pages over traditional MVC - Structure : Combines logic and UI in Page Model(Code -Behind) - Routing : Here we use Page based routing ex Products/index.cshtml - MVC Uses Controller/Action routing - By Default pattern : Page driven where as MVC follows controller driven approach
		Creating and configuring Razor Pages in a project
		Folder structure and naming conventions
	Razor Syntax and Page	Razor syntax basics and directives

	Model	Mixing HTML and server-side code
		Understanding the PageModel class
		Property binding and handling requests
	.NET Security & Reliability from the perspective of Enterprise solution (Big Organisation solutions)	
	.NET Security & Reliability	Understanding security in .NET applications: Key pillars: 1) Authentication : Microsof.ASPNECORE.Authtication.JWTBearer 2) Authorisation : 3) Data protection : Microsoft.ASPNETCORE.Dataprotection 4) Secure Coding : System.Security.Cryptography
		Common security practices (authentication, authorization, encryption) Authentication : Verifying user identity ex password JWT Bearer Tokens(Fo API) External Providers(Google, Microsoft, Facebook) Authorization : Determines what user can access -Role based Authorisation: admin, guest -Policy based authorisation -Claim based Access Control Cross Site Request Forgery(CSRF) : Enabled with RAzor pages and MVC forms SQL INjections : We Have Entity Framework Core(EF) Which prevent Injection(Parameterised Queries) Data Encryption : (System.Security.Cryptography) This is used for custom encryption where user secret, environment variables can be safe gaurded.
		Secure coding practices
		Using .NET libraries for encryption, secure communication, and secure storage
		Building Reliable Applications
		Designing for reliability
		Exception handling strategies
		Error Handling & Logging
		Implementing error handling best practices
		Logging errors and monitoring application health
	Solid Principles	SRP OCP LSP ISP DIP
	Design patterns	Creational Structural Behavioral

Use Cases: **Recommended Approach**

Large , Complex Apps(Enterprises Systems) : MVC (Better Separation of Code)
Simple, page-focused apps(Blogs, admin panels): Razor pages(Faster in development)
API + Web UI(Angular/react)in one project : MVC(Unified controllers For API and Views)
Rapid Prototyping : Razor Pages(Less boilerplate code)



Demo: Secure Message Console App

Guided Implementation Plan

Phase 1: Project Setup

1. Create Project
    - Hint: Use `dotnet new` command with the template for console applications
    - Remember to navigate to your desired directory first
2. Add Required Package
    - Hint: You'll need the standard cryptography library
    - Use `dotnet add package` to include the necessary security package

Phase 2: Core Architecture

1. Plan Your Main Menu
    - Hint: Create a while loop with switch-case structure
    - Consider options for: Encrypt/Decrypt text, Encrypt/Decrypt files, Exit
2. User Input Handling
    - Hint: Use `Console.ReadLine()` for text input
    - For files, you'll need `File.ReadAllText()` and `File.WriteAllText()`

Phase 3: Crypto Service Implementation

1. Create Encryption Helper Class
    - Hint: Make a static class with encrypt/decrypt methods
    - Consider what parameters each method will need
2. Implement AES Encryption
    - Hint: Use `Aes.Create()`
    - Remember to:
        - Generate initialization vector (IV)
        - Create encryptor/decryptor transforms
        - Use streams for the crypto operations
3. Password Handling
    - Hint: Use `Rfc2898DeriveBytes` for key derivation
    - You'll need to:
        - Convert password to secure key

        - Manage salt values appropriately

Phase 4: Error Handling

1. Add Basic Validation
    - Hint: Check for empty/null inputs
    - Consider what should happen if decryption fails
2. File Operations Safety
    - Hint: Wrap file operations in try-catch blocks
    - Check file existence before operations

Phase 5: Testing

1. Manual Testing Plan
    - Test cases to consider:
        - Encrypt short text → Decrypt → Compare results
        - Empty input handling
        - Wrong password during decryption
        - File operations
2. Automated Tests (Bonus)
    - Hint: Create a test project with `dotnet new xunit`
    - Test encryption/decryption roundtrip

Key Components to Implement

1. Main Program Flow
    - Menu display
    - User input collection
    - Operation routing
2. Crypto Service
    - AES initialization
    - Key derivation
    - Stream handling
3. File Operations
    - Reading input files

- Writing output files
- Path handling

Implementation Tips

- Start with text encryption only, then add file support
- Use helper methods to avoid code duplication
- Consider adding progress indicators for file operations
- Remember to dispose cryptographic objects properly

Checkpoint Questions

1. How will you handle the encryption IV?
2. What's your strategy for password-to-key conversion?
3. How will you structure your file operations?
4. What error cases should you handle?

Case Study: Product Management System with Razor Pages

User Story-Driven Implementation

User Story #1: Product Listing

As a store manager, I want to view all products in the system so I can monitor inventory.

Implementation Steps:

1. Create `Products/Index.cshtml` Razor Page
2. Define `IndexModel` with `OnGet()` handler
3. Initialize sample product list in PageModel
4. Display products in HTML table using Razor syntax

Key Razor Pages Features Demonstrated:

- Automatic routing to `/Products`
- Colocated view and logic
- Simple data binding

User Story #2: Add New Product

As a store employee, I want to add new products to the system so we can track new inventory items.

Implementation Steps:

1. Create `Products/Create.cshtml` Razor Page
2. Add `[BindProperty]` to product in PageModel
3. Implement `OnPost()` handler
4. Create form with tag helpers

Key Advantages Over MVC:

- No separate controller needed
- Automatic model binding
- Built-in anti-forgery tokens

User Story #3: Product Details

As a customer, I want to view product details so I can make purchasing decisions.

Implementation Steps:

1. Create `Products/Details.cshtml` with route parameter
2. Implement `OnGet(int id)` handler
3. Display product details using Razor syntax

Routing Example:

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

Automatically maps to `/Products/Details/5`

Technical Highlights

Folder Structure

```
Pages/
  Products/
    Index.cshtml
    Index.cshtml.cs
    Create.cshtml
    Create.cshtml.cs
    Details.cshtml
    Details.cshtml.cs
```

Code Samples (Partial Implementation)

PageModel (Details.cshtml.cs)

```
public class DetailsModel : PageModel
{
    public Product CurrentProduct { get; set; }

    public void OnGet(int id)
    {
        // Fetch product by ID
        CurrentProduct = _repository.GetProduct(id);
    }
}
```

Razor View (Details.cshtml)

```
@page "{id:int}"
@model DetailsModel

<h2>@Model.CurrentProduct.Name</h2>
<p>Price: @Model.CurrentProduct.Price.ToString("C")</p>
```

Benefits Realized

1. Faster Development

    - 30% fewer files than equivalent MVC implementation
    - Reduced context switching between controllers/views
2. Improved Maintainability
    - Related code colocated
    - Clear ownership of functionality
3. Enhanced Security
    - Automatic XSRF protection
    - Simplified authorization at page level
4. Better Performance
    - Lightweight page-focused architecture
    - Reduced middleware overhead

Evolution Path

1. Current: Basic CRUD operations
2. Next Phase: Add search functionality
3. Future: Implement user authentication
4. Advanced: Add reporting capabilities