CODING PRACTICES

1. Separation of Concerns (Model, View, and Controller)

Definition:

Separation of Concerns (SoC) is a **software design principle** that divides an application into distinct sections, where each section or component has a **specific responsibility**. This makes the system modular, organized, and easier to manage.

In the MVC Pattern:

- **Model:**
  - Represents the **data** and **business logic** of the application.
  - It is responsible for handling data storage, retrieval, and validation.
  - Example: A Student model class that interacts with the database to fetch and save student information.
- **View:**
  - Handles the **presentation layer** — how data is displayed to the user.
  - Views are typically made using HTML, CSS, or templating engines.
  - Example: A web page that displays the list of students retrieved by the model.
- **Controller:**
  - Serves as the **intermediary** between the Model and the View.
  - It processes user input, interacts with the Model, and updates the View accordingly.
  - Example: A StudentController that handles user requests like adding or editing student data.

Benefits:

- Enhances **code readability** and **modularity**.
- Makes maintenance and debugging easier since each concern is isolated.
- Allows **parallel development** — front-end developers can work on views while back-end developers handle logic.
- Improves **scalability** and **testability**.

2. Consistent Naming Conventions

Definition:

Naming conventions refer to a **standardized way of naming variables, methods, classes, and files** across a project. Following consistent conventions ensures that your codebase remains clear and understandable to all developers.

Common Naming Styles:

- **camelCase:** For variables and functions → studentName, calculateAverage()

- **PascalCase:** For classes, namespaces, and components → StudentController, DatabaseService
- **snake_case:** Commonly used in databases or file names → student_records, user_profile
- **UPPER_CASE:** For constants → MAX_ATTEMPTS, PI_VALUE

Best Practices:

- Use **meaningful and descriptive names** that reflect the purpose (e.g., getStudentList() is clearer than getData()).
- Maintain the **same convention throughout the project**.
- Avoid abbreviations unless they are universally understood (e.g., HTML, ID).
- Prefix private fields (in some languages like C#) with an underscore (e.g., _studentRepository).

Benefits:

- Improves **code readability and consistency**.
- Makes collaboration and code reviews easier.
- Reduces confusion and potential naming conflicts.
- Helps in maintaining **clean, professional-quality code**.

3. DRY (Don't Repeat Yourself) and Reusable Components

Definition:

The DRY principle emphasizes that **each piece of knowledge or logic in a system should exist in only one place**. Repetition leads to inconsistencies, errors, and makes maintenance more difficult.

How to Apply DRY:

- Create **reusable functions, classes, and modules** instead of copying code.
- Use **partial views** or **shared components** for repeating UI sections.
- Apply **inheritance or composition** to share common logic between classes.
- Store common configurations or constants in centralized files.

Example:

Instead of writing a form validation function multiple times across pages:

```
// Reusable validation method
public bool IsValidEmail(string email)
{
   return Regex.IsMatch(email, @"^[^@\s]+@[^@\s]+\.[^@\s]+$");
}
```

This can then be called anywhere in the application.

Benefits:

- **Reduces redundancy** and code duplication.
- Simplifies **maintenance and updates** — changes only need to be made once.
- Increases **code reliability** and decreases bugs.
- Encourages **modularity and reusability** across projects.

4. Folder Structure and Organization

Definition:

A well-organized folder structure groups files logically based on their **functionality and role** in the application. This improves project maintainability and clarity.

Typical MVC Folder Structure Example:

/Controllers → Contains controllers handling user requests.
/Models → Contains business logic and data classes.
/Views → Contains HTML/CSS templates for user interface.
/Services → Contains reusable logic or business services.
/wwwroot → Public folder for static assets like images, JS, and CSS.
/wwwroot/css → Stylesheets.
/wwwroot/js → JavaScript files.
/Data → Contains database context and migrations.
/wwwroot/images → Image files.

Best Practices:

- Group related files together.
- Follow consistent naming and casing for folders.
- Avoid placing all files in a single directory.
- Keep configuration files (like appsettings.json) in the project root for easy access.
- Use subfolders for modules or features in large projects.

Benefits:

- Makes the project **easier to navigate** for any developer.
- Improves **scalability** — easy to add new features without clutter.
- Enhances **collaboration** by standardizing where each type of file belongs.
- Supports **clean architecture** and maintainable code structure.

Deployment of ASP.NET Applications

1. Preparing the Application for Deployment

Definition

Before deploying, the application must be **production-ready**, meaning it should be **optimized, tested, and configured** to run outside the development environment.

Steps

- **Code Review & Testing:** Ensure there are no compilation errors, broken links, or runtime exceptions.
- **Remove Debug Code:** Disable or remove debugging statements to improve performance.
- **Optimize Resources:** Minify CSS and JS files, compress images, and bundle assets if necessary.
- **Check Dependencies:** Ensure all required packages, DLLs, and NuGet packages are included.
- **Set Production Configuration:** Use production app settings (appsettings.Production.json) for databases, APIs, and other services.

Benefit

Reduces runtime errors, improves performance, and ensures smooth operation in the production environment.

2. Publishing ASP.NET Apps in Visual Studio

Definition

Publishing is the process of **building and packaging** your application so that it can be deployed to a server.

Steps in Visual Studio

1. Right-click on the project → **Publish**.
2. Select **Target**:
    - **Folder** → Generates files to a local folder.
    - **IIS, FTP, Azure, or Web Server** → Direct deployment options.
3. Configure **Settings**:
    - Choose **Release** build configuration.
    - Set target **framework** and environment variables.
4. **Validate and Publish** → Visual Studio builds the project and copies necessary files.

Benefit

Simplifies deployment by creating a structured folder with all required binaries, configs, and assets.

3. Installing and Configuring IIS on Windows

Definition

IIS (Internet Information Services) is a web server used to **host ASP.NET applications** on Windows environments.

Steps

1. Open **Control Panel → Programs → Turn Windows Features On or Off**.
2. Enable **Internet Information Services** and necessary components:
    - Web Management Tools
    - World Wide Web Services
    - Application Development Features (.NET Extensibility, ASP.NET, etc.)
3. Launch **IIS Manager** and verify the server is running.
4. Configure **Application Pool**:
    - Set the correct **.NET CLR version**.
    - Enable **Integrated Pipeline** mode for modern ASP.NET apps.

Benefit

Provides a stable and configurable server environment for hosting web applications.

4. Deploying to a Local Server or LAN-Accessible Environment

Definition

This involves **moving the application from your development environment to a server** that can be accessed locally or within a network.

Steps

- **Copy Published Files:** Copy files from the Visual Studio publish folder to the target server folder.
- **Set up IIS Site:**
    1. Open IIS Manager → **Add Website**.
    2. Specify **Site Name**, **Physical Path**, and **Port/Host Name**.
    3. Assign the correct **Application Pool**.
- **Test Access:** Verify the app runs correctly using http://localhost:port or LAN IP.
- **Configure Firewall/Network:** Ensure ports are open for LAN users.

Benefit

Allows multiple users within a network to access the application safely.

5. Managing Connection Strings Securely in Production

Definition

Connection strings define how your application connects to databases or external services. In production, they must be **secured** to prevent unauthorized access.

Best Practices

- **Use Configuration Files:** Store connection strings in appsettings.Production.json.
- **Environment Variables:** Load sensitive values from environment variables instead of hardcoding.
- **Encrypt Connection Strings:** Use ASP.NET Core's **User Secrets**, **Azure Key Vault**, or Windows **DPAPI**.
- **Least Privilege Principle:** Ensure database accounts used have only necessary permissions.

Benefit

Protects sensitive data, reduces risk of security breaches, and ensures compliance with best practices.

Summary of Deployment Workflow

1. **Prepare application** → Test, optimize, configure for production.
2. **Publish** → Build and package application using Visual Studio.
3. **Set up IIS** → Install and configure web server with appropriate application pools.
4. **Deploy to server** → Copy files, configure site, and test LAN access.
5. **Secure connection strings** → Use encrypted or environment-based configuration for sensitive data.