|  |
| --- |
| DAFTech Social ict Solution |
| Code Standard |
| .Net Core |
|  |
| **DAFTech** |
| **2/14/2024** |

****

Contents

[Introduction 2](#_Toc158811833)

[Naming Conventions 2](#_Toc158811834)

[Code Formatting 4](#_Toc158811835)

[Commenting and Documentation 6](#_Toc158811836)

[Error Handling 8](#_Toc158811837)

[Code Organization 10](#_Toc158811838)

[Code Reusability 13](#_Toc158811839)

[Security 14](#_Toc158811840)

[Other References 16](#_Toc158811841)

# Introduction

Welcome to the Code Standard Document for .NET Core at DAFTech Social ICT Solutions! This document serves as a comprehensive guide to establish consistent coding practices and standards for .Net Core development across our organization. By adhering to these standards we aim to enhance code quality, readability, maintainability, and overall software development efficiency.

At DAFTech, we recognize the importance of writing clean, well-structured, and optimized code in the .NET Core framework. Consistency in coding style and conventions not only improves collaboration among team members but also ensures that our codebase remains scalable, robust, and easy to maintain.

This document covers a wide range of topics specific to .NET Core development, including naming conventions, code formatting, commenting and documentation, error handling, code organization, code reusability, testing and quality assurance, performance optimization, security, version control, and collaboration. Each topic provides guidelines and best practices that should be followed when writing .NET Core code within our organization.

It is important to note that these standards are not set in stone and may evolve over time. We encourage all developers to actively contribute to the improvement of our coding practices by providing feedback, suggesting enhancements, and sharing their expertise.

By adhering to these code standards, we can ensure that our .NET Core codebase remains consistent, maintainable, and of high quality. This, in turn, will enable us to deliver reliable and efficient software solutions to our clients and stakeholders.

Thank you for your commitment to maintaining coding excellence in .NET Core at DAFTech Social ICT Solutions. Let's strive for code that is not only functional but also elegant and efficient!

# Naming Conventions

Naming Conventions are guidelines for naming various elements in the code to ensure consistency and readability.  Consistent naming conventions make it easier for you to understand and maintain the code.

1. **PascalCase:** Class names, struct names, and method names should use PascalCase. The first letter of each word should be capitalized, and there should be no underscores between words. For example: Customer, ProductService, GetCustomerById().
2. **CamelCase:** Local variables,method arguments and private fields should use camelCase. The first letter of the name should be in lowercase, and the first letter of each subsequent concatenated word should be capitalized. For example: customer, productName, GetCustomerById(int id).
3. **Plural Nouns:** Collection names, such as arrays or lists, should be named using plural nouns. For Example: customers , products
4. **Prefixes/Suffixes:** Use Prefixes of suffixes to provide additional context to your names. For example: IsReady, CustomerList.
5. **Namespace Naming:** Namespaces should follow a hierarchical naming pattern. Each segment of the namespace should be PascalCase.

For example: Company.Project.Module , Company.Project.Utilities.

1. **File Naming:** File names should match the class names, including the casing. For example, a class named CustomerService Should be defined in a file named CustomerService.cs.
2. **Abbreviations:** Use abbreviations sparingly and only for commonly understood terms. Avoid excessive abbreviation to maintain code readability but if the word is too long use comments to address the abbrivation.
3. **Boolean Names:** Boolean variables, properties, and methods should be named in a way that clearly indicates their meaning. Use a prefix like “is” , “has” or “should” to improve clarity. For example: isVisible, hasPermission, shouldProcess.
4. **Events and Delegates:** Events and delegate names should use pascalCase and should be named using verbs or verb phrases to indicate the action they perform. For example ButtonClicked, DataProcesed.
5. **Avoid Reserved Keywords:** Avoid using reserved keywords as names classes, methods, properties, variables, etc.
6. **Consistency:** Ensure consistent application of the naming convention throughout the codebase, including existing code and new code.

# Code Formatting

Code formatting standards are guidelines for how to format your code to improve readability and maintain consistency. Here are some code formatting standards for .NET Core, along with examples:

1. **Indentation:** Use tabs or spaces for indentation. Choose one and be consistent throughout your codebase. The recommended indentation size is typically four spaces.

// Good indentation example

public void MyMethod()

{

if (condition)

{

Console.WriteLine("Condition is true");

}

else

{

Console.WriteLine("Condition is false");

}

}

1. **Brace Placement:** Place opening braces on the same line as the statement or declaration, and place closing braces on a new line aligned with the corresponding opening brace.

// Good brace placement example

public void MyMethod()

{

if (condition)

{

Console.WriteLine("Condition is true");

}

else

{

Console.WriteLine("Condition is false");

}

}

1. **Spacing:** Use spaces to separate operators, commas, and other elements to improve readability.

// Good spacing example

int result = 10 + 5;

string message = "Hello, " + name;

1. **Line Length:** Limit the line length to a reasonable number of characters, typically around 80-120 characters. Break long lines into multiple lines for improved readability.

// Good line length example

string longMessage = "This is a long message that exceeds the recommended line length, so it is broken into multiple lines for better readability.";

1. **Naming Conventions**: Follow the previously mentioned naming conventions for classes, methods, variables, etc., to ensure consistency and readability.
2. **Vertical Alignment:** Vertically align related code elements to improve readability. For example, align the parameters of a method call or align the properties of a class.

// Good vertical alignment example

public void MyMethod(

int parameter1,

int parameter2,

string parameter3)

{

// Method body

}

1. **Blank Lines:** Use blank lines to separate logical sections or blocks of code. This improves readability and helps to clearly define different parts of your code.

// Good use of blank lines

public void MyMethod()

{

// Code block 1

// Code block 2

// Code block 3

}

# Commenting and Documentation

Commenting and documentation are essential for understanding code and promoting maintainability. Here are some guidelines for commenting and documentation in .NET Core:

1. **XML Documentation Comments:** Use XML documentation comments to provide meaningful documentation for classes, methods, properties, and other code elements. These comments can be automatically extracted by tools to generate API documentation.

/// <summary>

/// Represents a customer in the system.

/// </summary>

public class Customer

{

/// <summary>

/// Gets or sets the customer's unique identifier.

/// </summary>

public int Id { get; set; }

/// <summary>

/// Retrieves a customer by their unique identifier.

/// </summary>

/// <param name="id">The customer's unique identifier.</param>

/// <returns>The customer object.</returns>

public Customer GetCustomerById(int id)

{

// Method implementation

}

}

1. **Method-level Comments:** Include comments at the beginning of methods to describe their purpose, input parameters, return values, and any exceptions they may throw. Use this comments for method that have complex and non-obvious codes.

/// <summary>

/// Calculates the sum of two numbers.

/// </summary>

/// <param name="a">The first number.</param>

/// <param name="b">The second number.</param>

/// <returns>The sum of the two numbers.</returns>

public int CalculateSum(int a, int b)

{

// Method implementation

}

1. **Avoid Redundant Comments:** Focus on providing valuable information in comments, rather than restating what the code already expresses. Comments should enhance understanding, not repeat what is already clear from the code itself.
2. **Regularly Review and Update Comments:** Keep comments up to date as code changes. Outdated or incorrect comments can be misleading.
3. **README Files:** Include a README file in your project's root directory to provide an overview, installation instructions, usage examples, and any other necessary information for developers working with the project.

# Error Handling

Error handling is crucial for creating robust and reliable applications. Here are some guidelines for error handling in .NET Core:

1. **Use Exception Handling**: Wrap code that may throw exceptions in try-catch blocks to handle and manage exceptions gracefully.

try

{

// Code that might throw an exception

}

catch (Exception ex)

{

// Exception handling code

}

1. **Catch Specific Exceptions:** Catch specific exceptions whenever possible, rather than catching the general Exception type. This allows for more targeted handling and avoids masking other potential issues.

try

{

// Code that might throw a specific exception

}

catch (SpecificException ex)

{

// Specific exception handling code

}

1. **Handle Exceptions at the Appropriate Level:** Handle exceptions at the appropriate level of your application. This can be at the method level, UI level, or even at the global level.
2. **Log Exceptions:** Log exceptions to a centralized logging system or log files to track and investigate errors. Include relevant information such as the exception type, message, stack trace, and any additional details that may be helpful for debugging.

try

{

// Code that might throw an exception

}

catch (Exception ex)

{

// Log the exception

logger.LogError(ex, "An error occurred while processing XYZ.");

}

1. **Throw Custom Exceptions:** When appropriate, throw custom exceptions to provide more meaningful error messages and to encapsulate specific application logic or domain-specific errors.

public class CustomException : Exception

{

public CustomException(string message) : base(message)

{

// Additional initialization logic

}

}

// Usage

if (condition)

{

throw new CustomException("Invalid input data.");

}

1. **Handle Exceptions Gracefully:** Provide user-friendly error messages and handle exceptions in a way that gracefully recovers from errors or provides appropriate feedback to the user.
2. **Avoid Swallowing Exceptions**: Avoid swallowing exceptions without appropriate handling. If an exception occurs, it should be either handled or propagated to higher levels for handling.
3. **Use Finally Blocks**: Use finally blocks to ensure cleanup code and resource disposal, regardless of whether an exception occurred or not.

try

{

// Code that might throw an exception

}

catch (Exception ex)

{

// Exception handling code

}

finally

{

// Cleanup code or resource disposal

}

1. **Validate Inputs:** Validate inputs and enforce business rules to prevent exceptions caused by invalid or unexpected data.

public void ProcessOrder(Order order)

{

if (order == null)

{

throw new ArgumentNullException(nameof(order), "Order cannot be null.");

}

// Process the order

}

# Code Organization

Code organization is crucial for maintaining a clean and maintainable codebase. Here are some guidelines for organizing code in .NET Core:

1. **Use Namespaces:** Use namespaces to organize related classes and provide logical grouping. Namespaces help prevent naming conflicts and make it easier to locate and understand code.

namespace MyProject.Data

{

// Data-related classes

}

namespace MyProject.Services

{

// Service-related classes

}

1. **Group Related Code:** Group related classes, interfaces, and enums together within a file or folder. This promotes cohesion and makes it easier to find and navigate related code.
2. **Follow Single Responsibility Principle (SRP):** Each class or module should have a single responsibility. This helps to keep code focused, maintainable, and easier to understand.
3. **Organize Files by Feature or Module:** Organize files into folders based on the feature or module they belong to. This helps keep related code together and makes it easier to locate and modify code.

MyProject

├── Controllers

│ ├── HomeController.cs

│ ├── AccountController.cs

├── Models

│ ├── User.cs

│ ├── Order.cs

├── Services

│ ├── UserService.cs

│ ├── OrderService.cs

1. **Use Regions Sparingly:** Avoid excessive use of #region directives as they can hide code complexity and make it harder to navigate. Reserve their usage for genuinely large or complex code sections.
2. **Follow Consistent Naming Conventions**: Use consistent naming conventions for classes, methods, properties, and other code elements. This improves code readability and helps other developers understand your codebase.
3. **Separate Concerns:** Separate concerns by applying architectural patterns such as Model-View-Controller (MVC) or clean architecture. This promotes loose coupling and improves code maintainability.
4. Use Partial Classes: Utilize partial classes to split a large class into multiple files. This can help organize code and make it more manageable.

// File 1: UserService.cs

public partial class UserService

{

// Part of UserService class

}

// File 2: UserService.Validation.cs

public partial class UserService

{

// Validation-related methods for UserService

}

1. **Apply Folder Conventions**: Organize code files into folders based on their type (e.g., controllers, models, services) or architectural layer. This makes it easier to locate specific types of code.
2. **Use Dependency Injection:** Implement dependency injection to decouple components and simplify code organization. This allows for easier testing, maintainability, and scalability.
3. **Consistent Formatting:** Apply consistent code formatting throughout the codebase. Consistent indentation, spacing, and code layout make the code easier to read and understand.
4. **Keep Code Files Concise:** Avoid creating excessively long code files. If a file becomes too large, consider refactoring it into smaller, more focused files.

# Code Reusability

Code reusability is an important aspect of software development that aims to maximize the utilization of existing code to minimize redundancy and improve efficiency. Here are some guidelines for achieving code reusability in .NET Core:

1. **Create Modular and Independent Components**: Design your codebase with modular components that have well-defined responsibilities and minimal dependencies. This allows individual components to be reused in different contexts without affecting other parts of the system.
2. **Extract Reusable Functions and Methods**: Identify common functionality that can be extracted into standalone functions or methods. These can be placed in utility classes or libraries and reused across multiple projects.
3. **Use Class Inheritance and Polymorphism**: Utilize inheritance and polymorphism to create base classes or interfaces that define common behavior. This promotes code reuse by allowing derived classes to inherit and extend the functionality of the base class.
4. **Leverage Libraries and Frameworks**: Take advantage of existing libraries and frameworks, such as NuGet packages, that provide reusable functionality. These libraries often provide well-tested and optimized code for common tasks, saving development time and effort.
5. **Create Generic Classes and Methods**: Use generics to create flexible and reusable code that can work with different types. This allows you to write code that can handle a wide range of data types without duplicating logic.
6. **Extract Configuration and Constants**: Separate configuration settings and constants into dedicated files or classes. This allows for easy modification and reuse of values across the application.
7. **Encapsulate Code in Libraries or Packages**: Wrap reusable code into libraries or packages that can be shared and distributed across multiple projects. This enables teams to reuse code effectively and maintain a single source of truth for common functionality.
8. **Document and Share Code**: Document your reusable code with clear and concise documentation, including usage instructions and examples. Share the documentation internally with your team or consider contributing your code to open-source projects to benefit a wider audience.
9. **Test and Validate Reusable Components**: Ensure that your reusable code is thoroughly tested to verify its correctness and reliability. This provides confidence to developers who reuse the code and minimizes the risk of introducing bugs into different projects.
10. **Refactor and Improve**: Regularly review your codebase and identify opportunities for refactoring and improving code reusability. As you gain more experience with your codebase and requirements evolve, you may discover new ways to make your code more reusable and efficient.

# Security

When developing applications in .NET Core, it is essential to follow security best practices to protect your application and its data from potential vulnerabilities and threats. Here are some important security standards and practices to consider:

1. **Secure Coding Practices**: Follow secure coding practices to minimize common vulnerabilities, such as injection attacks (e.g., SQL injection, cross-site scripting) and buffer overflows. This includes input validation, parameterized queries, output encoding, and avoiding the use of insecure functions.
2. **Authentication and Authorization**: Implement proper authentication and authorization mechanisms to ensure that only authenticated and authorized users can access sensitive functionality and data. Use strong authentication methods, such as multi-factor authentication (MFA), and leverage frameworks like ASP.NET Core Identity for user management.
3. **Secure Communication**: Use secure communication protocols, such as HTTPS, for transmitting sensitive data over networks. Ensure that appropriate encryption algorithms and secure cipher suites are in place to protect data confidentiality and integrity.
4. **Password Security**: Enforce strong password policies, including password complexity requirements, hashing and salting passwords, and implementing secure password storage mechanisms. Avoid storing passwords in plain text or using weak hashing algorithms.
5. **Input Validation**: Validate and sanitize all user inputs to prevent common security vulnerabilities, such as SQL injection and cross-site scripting (XSS). Utilize input validation techniques, such as whitelisting, input parameter validation, and output encoding, to mitigate these risks.
6. **Secure Configuration Management**: Safely manage sensitive information, such as database connection strings, API keys, and encryption keys, in configuration files or secure key management systems. Avoid hard-coding sensitive information in the source code, and use environment-specific configuration settings for different deployment environments.
7. **Error Handling and Logging**: Implement proper error handling and logging mechanisms to prevent sensitive information leakage and aid in identifying potential security issues. Avoid displaying detailed error messages to end users and log security-related events for monitoring and auditing purposes.
8. **Secure Session Management**: Implement secure session management practices, such as generating strong session tokens, enforcing session expiration, and protecting against session fixation and session hijacking attacks. Consider using frameworks like ASP.NET Core's session middleware and cookie policies for secure session handling.
9. **Secure File Management**: Implement secure file handling practices to prevent arbitrary file uploads, path traversal attacks, and unauthorized access to sensitive files. Validate file types, restrict file upload locations, and ensure appropriate file permissions and access controls are in place.
10. **Regular Updates and Patching**: Keep your .NET Core framework, libraries, and dependencies up to date by applying security patches and updates promptly. This helps protect your application against known vulnerabilities and security issues.
11. **Security Testing**: Conduct regular security assessments, including penetration testing and vulnerability scanning, to identify and address potential security weaknesses in your application. Utilize security tools and frameworks, such as OWASP ZAP and SonarQube, to automate security testing processes.
12. **Secure Deployment and Hosting**: Follow secure deployment practices by ensuring secure server configurations, using secure protocols for deployment (e.g., SSH), and utilizing secure hosting environments that provide protection against common attacks and vulnerabilities.

# Other References

1. <https://www.dofactory.com/csharp-coding-standards>
2. <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>
3. <https://www.geeksforgeeks.org/c-sharp-coding-standards/>