# Witherslack Group – Coding & Application Design Standards V1

Contents

[Witherslack Group – Coding & Application Design Standards V1 1](#_Toc8723196)

[1. Introduction 2](#_Toc8723197)

[2. Coding Standards 3](#_Toc8723198)

[2.1. Naming 3](#_Toc8723199)

[2.2. Commenting 5](#_Toc8723200)

[2.3. Indention, spacing and layout 6](#_Toc8723201)

[2.4. Error and exception handling 8](#_Toc8723202)

[2.5. Other programming practices 8](#_Toc8723203)

[3. Application architecture design standards 10](#_Toc8723204)

[3.1. Relational database design 10](#_Toc8723205)

[3.2. Event Store Design 10](#_Toc8723206)

[3.3. Application programming interface (API) design 11](#_Toc8723207)

[3.4. Model-View-Controller application design 11](#_Toc8723208)

[3.5. Single-Page Application (SPA) design 11](#_Toc8723209)

[3.6. Common project guidelines 12](#_Toc8723210)

[4. Language and framework-specific external guidelines 13](#_Toc8723211)

[4.1. SQL style guide 13](#_Toc8723212)

[4.2. Microsoft C# coding conventions 13](#_Toc8723213)

[4.3. JavaScript Conventions 13](#_Toc8723214)

[4.4. Angular Style Guide 13](#_Toc8723215)

## Introduction

The Witherslack Group develops and maintains internal software for managing organisational data. Its software, and the data it manages, is critical to the successful operation of the organisation, and the quality of the software is therefore of paramount importance to the development team and the overall group.

In developing consistent, reliable and maintainable applications, coding standards and common practices should be used by all developers. This document outlines those standards and practices to which software code should be written at the Witherslack Group in the form of guidelines on the expected structure, layout and syntax of code. It also outlines the expected naming, structure and design of some of the most commonly used architectural paradigms and frameworks used in the organisation’s internally developed applications.

These standards should be used at internal development team code reviews and serve as a reference to individual developers whilst writing application code. This document should also be regularly updated to reflect the latest coding standards and practices of the wider software development industry as well as the team.

## Coding Standards

Software written internally at the Witherslack Group has been, and is currently being, written using a number of different programming, query, scripting and markup languages. These include:

* Hyper Text Markup Language (HTML)
* Cascading Style Sheets (CSS)
* JavaScript (JS)
* TypeScript (TS)
* Structured Query Language (SQL)
* C#

The following coding standards relate to the programming languages used at the Witherslack Group and outline the expected structure, layout and style of individual files of code written in these languages, so as to ensure their comprehensibility and maintainability. They should not be taken as absolute rules in scenarios where there are sensible exceptions. Instead, these scenarios should be reviewed by the team and this document updated if necessary.

Most examples provided in this section use the C# programming language, as this is primary software development language used at the Witherslack Group for new developments. Where a difference in approach is required in another language, this is specified with examples.

---- Split into language sections and remove the coding standards from the application architecture standards.

### [Naming](https://en.wikipedia.org/wiki/Naming_convention_(programming))

* + 1. The choice of name for any object, whether a file, a class or a variable, should be as succinct as possible, whilst being meaningful, descriptive, explicit and free of acronyms or word-contractions.

**Recommended (C#):**

**string employeeAddress;**

**int staffAge;**

**float assessmentScore;**

**DateTime dateOfDeparture;**

**string youngPersonStandardTest;**

**Not Recommended (C#):**

**string eAddr;**

**int stfAge;**

**float assScr;**

**DateTime date\_depart;**

**string ypst;**

* + 1. Single character variable names should be avoided, such as **e** instead of **employee**, or **t** instead of **time**. One exception to this rule is the use of single character variables as iterators in loops, where it is common practice to use **i** as an index variable.

**Recommended (C#):**

**for ( int i = 0; i < count; i++ )**

**{}**

**Not Recommended (C#):**

**DateTime t;**

**string n;**

* + 1. Pascal Casing should be used for file, class and field names. Pascal casing is a capitalisation standard in which the first letter of every word in a name is capitalised.

**Recommended (C#):**

**public class Employee**

**{**

**public string FirstName;**

**public string LastName;**

**public int Age;**

**public string FullName()**

**{**

**Return FirstName + “ “ + LastName;**

**}**

**}**

* + 1. Camel Casing should be used for local variable and method parameter names. Camel Casing is a capitalisation standard in which the first letter of the first word in a name is lower cased and the first letters of all subsequent words are capitalised.

**Recommended (C#):**

**int totalCount = 0;**

**void SayHello(string name)**

**{**

**string fullMessage = "Hello " + name;**

**...**

**}**

* + 1. Method names should use either Pascal or Camel Casing depending on the language. For JavaScript and TypeScript, Camel Casing is recommended, as this is the industry-wide standard practice. For C# Pascal Casing is recommended, as this is the industry-wide practice. Whichever approach is used, it should be used consistently across projects where that language is used.

**Recommended (C#):**

**public string SayHello(string name)**

**{**

**return "Hello " + name;**

**}**

**Recommended (JS):**

**function sayHello(name) {**

**return "Hello " + name;**

**}**

* + 1. Underscores (**\_**) should not be used in names, except for Unit Test names, where underscores should be used to separate words for readability in the test panel.

**Recommended (C#):**

**public void Test\_This()**

**Not Recommended (C#):**

**int total\_Count;**

**public interface Employee\_Service;**

* + 1. Variable names that resemble language keywords should not be used.

**Not Recommended (C#):**

**string** **class;**

**bool** **checked;**

* + 1. File names should match the contained primary class name.

**Recommended (C#):**

**HelloWorld.cs** should contain **public class HelloWorld{}**

* + 1. Boolean variables should be prefixed with an “**is**” or other similar prefix to represent the binary nature of the value.

**Recommended (C#):**

**bool isComplete;**

**bool hasPageAccess;**

### [Comment](https://en.wikipedia.org/wiki/Comment_(computer_programming))ing

* + 1. As a general rule, comments should not be required to explain what written code does. This should be clear from the appropriate naming and clear layout of the code. There may be acceptable exceptions to this guidance for complex code that the author wants to clarify.

**Not Recommended (C#):**

// Function adds two numbers together

**public string AddNumbers(int number1, int number2)**

**{**

**return number1 + number2;**

**}**

* + 1. Code comments should provide the reader with information that the author could not have otherwise communicated through their code, such as why the code does what it does, why the code has been written in the way it has, or the context for the code that has been written.

**Recommended (SQL):**

-- Written as a CTE instead of a temporary table and the temporary table implementation of the query was causing performance issues

* + 1. Comments should begin with a space between the comment initiating characters and the text to improve legibility.

**Recommended:**

**(C#)** // Code comment

**(SQL)** -- Code comment

* + 1. Comments should be added above the code that they relate to, not on the same line or below the code.

**Recommended (C#):**

// This comment relates to the below function

**public string AddNumbers(int number1, int number2)**

**{**

**return number1 + number2;**

**}**

**Not Recommended (C#):**

**public string AddNumbers(int number1, int number2)** // This comment relates to this function

**{**

**return number1 + number2;**

**}**

* + 1. Single-line comments should follow the individual line length guidance outline in 2.3.4.
    2. For comments longer than the suggested line-length in 2.3.4, multi-line comments shall be used, if the language in use supports them.

**Recommended (SQL):**

/\*

Functionality abstracted from main query body as it is required across multiple queries.

\*/

* + 1. Comments should be written using correct English-language spelling and grammar.
    2. Comments should only relate directly to code and not to other people.

**Not Recommended (C#):**

// Matt, this function could be written in 2 lines of code instead of 20!

**public string AddNumbers(int number1, int number2)**

### [Indention](https://en.wikipedia.org/wiki/Indent_style), spacing and layout

* + 1. One blank line should be used to separate logical groups of code to improve legibility.

**Recommended (C#):**

**private void AddClassYears(LinkDbContext context)**

**{**

**yearGroup1 = new YearGroup { Name = "Year 1", SortOrder = 1 };**

**yearGroup2 = new YearGroup { Name = "Year 2", SortOrder = 2 };**

**context.YearGroups.Add(yearGroup1);**

**context.YearGroups.Add(yearGroup2);**

**context.SaveChanges();**

**}**

**Not Recommended (C#):**

**private void AddClassYears(LinkDbContext context)**

**{**

**yearGroup1 = new YearGroup { Name = "Year 1", SortOrder = 1 };**

**yearGroup2 = new YearGroup { Name = "Year 2", SortOrder = 2 };**

**context.YearGroups.Add(yearGroup1);**

**context.YearGroups.Add(yearGroup2);**

**context.SaveChanges();**

**}**

* + 1. Indentations on new lines of code shall be created by using a single editor TAB, not using an editor SPACE. The default tab size should be 4.

**Recommended (C#):**

**public string AddNumbers(int number1, int number2)**

**{**

*tab space of 2* **return number1 + number2;**

**}**

* + 1. Indentations should be used only to indicate scope. Indentations, therefore, should follow after scope-denoting brackets in code.

**Recommended (C#):**

**public string AddNumbers(int number1, int number2)**

**{**

*tab space* **return number1 + number2;**

**}**

* + 1. It should be considered good practice to keep individual code line length under 120 columns. Longer lines than this warrant consideration of line breaks so as to be readable on a single line on a 1080p display with the Visual Studio (2017) code editor at half-screen width and at 100% zoom.
    2. In general, shorter line length is advised to improve the legibility of code. This should not be enforced in scenarios where it would sacrifice the readability and explicitness of the code.
    3. In general, avoid writing very long methods. A method should typically have 1-50 lines of code. If a method has more than 50 lines of code, you should consider re-factoring into separate methods.
    4. Opening curly braces for functions and control statements should be placed on either the same line of code or the subsequent line of code as the opening statement. The important thing is to be consistent in its placement throughout the code of an application. It is recommended that for C# code, the opening brace is placed on the subsequent line, and that for JS or TS code, the opening brace is placed on the same line as the opening statement. These different approaches for these languages serve to differentiate the main back and front-end languages used at the Witherslack Group, and are common approach taken in each language.

**Recommended (C#):**

**public int AddNumbers(int number1, int number2)**

**{**

**return number1 + number2;**

**}**

**Recommended (JS):**

**function addNumbers(number1, number2) {**

**return number1 + number2;**

**}**

### Error and exception handling

* + 1. Error messages should help the user to solve the problem. Do not give error messages like "**Error in Application**", "**There is an error**", etc. Instead give specific messages like "**Failed to login. Please make sure the login id and password are correct.**"
    2. When displaying error messages, in addition to telling the user what is wrong, the message should also tell the user what they should do to solve the problem. Instead of message like "**Failed to update database**", suggest what the user should do: "**Please specify Agency or Staff and submit form again.**"
    3. All application errors should be logged where possible.
    4. If an exception is caught, it should be handled. No exceptions should be hidden. At a minimum they should be logged for the development team.
    5. Exception handling should only be employed when there is a known, specific exception that could occur. Where it is possible to prevent an exception instead of handle it, this should be done instead of relying upon exception handling.

### Other [programming practices](https://en.wikipedia.org/wiki/Best_Coding_Practices)

* + 1. File includes should be maintained to only those includes that are used by the code within the file. Any excess, unused includes, should be removed from the file.
    2. In general, very long methods should be avoided. A method should typically have 1 to 50 lines of code. If a method has more than 50 lines of code, you should consider re-factoring into separate methods unless its length can be justified.
    3. In general, a method should perform only 'one task'. Combining multiple tasks into one method should be avoided, even if those tasks are very small.
    4. Application logic that is common in more than one area of an application or project should be abstracted to a shared area of the project, or another, separate shared project, and referenced where needed.
    5. In strongly typed languages that provide the option of weakly typed variable and object declarations, such as C#’s “var” keyword, the use of the weakly typed keyword should be context appropriate. For scenarios in which the type can be inferred from the object instantiation, weak typing is acceptable. For scenarios in which a type is returned from a method and the return type is unclear from the method name, declaring the type is recommended.
    6. Variables should be declared as close as possible to where they are first used. Only one variable should be declared on a single line.
    7. Assuming language support, Enums should be used in the place of numbers or strings to represent a discrete, immutable collection of values. Enums should not be used to represent mutable collections of values. Enums should not be used, however, to represent reference data pulled from a data source. Enums should only be used for internal application logic.
    8. Path or drive names should not be hard coded. Instead, applications should retrieve the application path programmatically and/or use relative paths where possible.
    9. In object-oriented languages, most files should contain no more than 1 single Class, Interface or Enum. Acceptable scenarios where this is not the case are those in which Classes, Interfaces or Enums are tightly coupled and dependant on each other.
    10. In general, very large files of code should be avoided. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Attempt to split them logically into two or more classes.

## Application architecture design standards

Software written internally at the Witherslack Group has been, and is currently being, written using a number of different architectural paradigms. These include:

* Relational Databases (Data stores)
* Event Stores (Data stores)
* Application programming Interfaces (APIs)
* Model-View-Controller (MVC) Applications
* Single-Page Applications (SPAs)

The following standards relate to these architectural paradigms used at the Witherslack Group and outline the expected design of projects built using these paradigms to ensure project quality and maintainability. They should not be taken as absolute rules in scenarios where there are sensible exceptions, but these scenarios should be reviewed by the team and this document updated if necessary.

* Routing (lowercase)

### Relational database design

* + 1. Relational database tables designed to act purely as data stores should follow the design rules of normalisation (<https://support.microsoft.com/en-gb/help/s283878/description-of-the-database-normalization-basics>).
    2. Relational database tables intended to provide reporting data are not expected to follow the rules of normalisation for the purposes of improving data structuring and read speeds for reporting.

* + 1. Each table must have only one primary key field that contains a unique value within the field.

* + 1. The naming of a Primary Key field in any table should simply be “Id”.
    2. The naming of a Foreign Key field in any table should be the Foreign Key reference table name “**AssessmentCategory**”, followed by “**Id**”. In this example, the Foreign Key name would be “**AssessmentCategoryId**”.
    3. Each field within a data storage table should contain a single, discrete value. No field should contain concatenated values.
    4. Each field within a data storage table should represent a distinct characteristic of the subject of the table.

### Event Store Design

* + 1. tbc

### Application programming interface (API) design

* + 1. Web application APIs should be written as REpresentation State-Transfer (REST) APIs, and use the HTTP data transfer protocol for communication.
    2. JSON should be the data transfer medium of choice for an API, unless there is a specific requirement for another data transfer medium to be used.
    3. Web APIs should only use two HTTP methods – GET and POST. GET methods should be employed for reading data. POST methods should be employed for Creating, Updating or Deleting data.
    4. API endpoints should conform to the following structure: **“ApplicationURI/PrimaryEndPoint/?SecondaryEndpoint/?MethodName/?{QuerystringValues}”**. Examples of this include: “**/employees/{id}**” and “**/absence/monitoring/changedatestarted**”.

### Model-View-Controller application design

* + 1. The user-interface, business logic and data access elements of an MVC application should be split into separate projects.
    2. Application services that are used in more than one area of an MVC application should be extracted to a separate “**ApplicationServices**” project.
    3. Application services should be injected into controllers using Dependency Injection.
    4. Common functionality, such as logging, data exporting, chart rendering, etc., should be extracted to a separate “**Common**” project.

### Single-Page Application (SPA) design

* + 1. SPA applications should always provide user feedback on Asynchronous JavaScript and XML (AJAX) server request successes and failures. Without this, uncommunicated errors can occur in SPA applications. Refer to section **2.3.4** for guidance on how to present error or exception information to users.
    2. When writing an Angular-based SPA application, a component’s TypeScript class, HTML and CSS should be maintained in separate files and referenced in the class’ “**@Component**” decorator. HTML and CSS should not be written inline in the class.
    3. When writing and Angular-based SPA application, common components and services should be extracted into a “**shared**” that exists under the “**app**” folder for reference throughout the application.

### Common project guidelines

* + 1. All Link database related development projects should use the following naming convention: “**Link.ProjectName**”. Examples of this include “**Link.HR**” and “**Link.IncidentReporting**”.
    2. Non-Link database related development projects should use the following naming convention: “**ProjectName.WitherslackGroup”**. Examples of this include “**Directory.WitherslackGroup**” and “**AccessRequest.WitherslackGroup**”.
    3. Where possible, test-driven development should be employed, with re-usable unit test being written to ensure that functionality can be easily tested following code changes.

## Language and framework-specific external guidelines

Some of the programming languages, frameworks and architectural paradigms used at the Witherslack Group have conventions that are maintained by their originating organisations or interested parties. Topics not covered in this document may be covered in the following documentation for these subjects:

### SQL style guide

* <https://www.sqlstyle.guide/>

### Microsoft C# coding conventions

* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>

### JavaScript Conventions

* <https://www.w3schools.com/js/js_conventions.asp>

### Angular Style Guide

* <https://angular.io/guide/styleguide>