InControl

Code Quality

|  |  |  |
| --- | --- | --- |
| Version |  | Writer |
| 1.0 |  | Cedric Baetens |

|  |
| --- |
| Contents |

[1 Maintainability 4](#_Toc503862878)

[1.1 Application Architecture 4](#_Toc503862879)

[1.2 Code readability 5](#_Toc503862880)

[1.3 Complexity of algorithms 6](#_Toc503862881)

[1.3.1 Create 6](#_Toc503862882)

[1.3.2 Read 6](#_Toc503862883)

[1.3.3 Update 6](#_Toc503862884)

[1.3.4 Delete 7](#_Toc503862885)

[1.4 Complexity level of communication protocols 8](#_Toc503862886)

[1.4.1 SignalR 8](#_Toc503862887)

[1.4.2 REST Api 8](#_Toc503862888)

[1.5 Component or pattern re-use ratio 9](#_Toc503862889)

[1.6 Reduced duplicated code and functions 10](#_Toc503862890)

[1.7 Source code file organization cleanliness 11](#_Toc503862891)

[1.7.1 ASP.NET Core 11](#_Toc503862892)

[1.7.2 .NET 4.6.2 Console Application 13](#_Toc503862893)

[1.8 Portability 14](#_Toc503862894)

[1.9 Documentation 15](#_Toc503862895)

[2 Supplier independence 16](#_Toc503862896)

[2.1 .NET Core 16](#_Toc503862897)

[2.2 .NET Standard 16](#_Toc503862898)

[2.3 Microsoft SignalR 16](#_Toc503862899)

[2.4 External Open Source Libraries 16](#_Toc503862900)

[3 Testability 17](#_Toc503862901)

[3.1 Controllability 17](#_Toc503862902)

[3.2 Observability 17](#_Toc503862903)

[3.3 Isolateability 17](#_Toc503862904)

[3.3.1 Dependancy Injection 17](#_Toc503862905)

[3.4 Understandability 18](#_Toc503862906)

[3.4.1 Test Naming Scheme 18](#_Toc503862907)

[3.5 Automatability 19](#_Toc503862908)

[3.5.1 VSTS Build and Release 19](#_Toc503862909)

[4 Good practices 20](#_Toc503862910)

[5 Coding Standards 21](#_Toc503862911)

[5.1 C# 21](#_Toc503862912)

[5.1.1 PascalCasing 21](#_Toc503862913)

[5.1.2 CamelCasing 21](#_Toc503862914)

[5.1.3 Hungarian Notation 22](#_Toc503862915)

[5.1.4 Screaming Caps 22](#_Toc503862916)

[5.1.5 Abbreviations 23](#_Toc503862917)

[5.1.6 Underscores 24](#_Toc503862918)

[5.1.7 Predefined type names 24](#_Toc503862919)

[5.1.8 Var types 25](#_Toc503862920)

# Maintainability

## Application Architecture

Within **Intation** we try to use the same (if possible) application architecture for each of our projects.

Client Layer

* Chrome
* Firefox
* Internet Explorer
* Edge
* Browser on Mobile Devices

Presentation Layer

* HTML/CSS
  + Bootstrap
* JavaScript
  + jQuery
  + Moment.js
* Unity

Service Layer

* Razor Pages
* REST API

Business Layer

* Microsoft ASP.NET Core
* Microsoft .NET 4.6.2
* Mono 5.4

Database Tier

* Microsoft EntitiyFramworkCore
  + Microsoft SQL Express
  + Microsoft SQL

## Code readability

TODO

## Complexity of algorithms

To reduce the complexity of our software functionalities we try (if possible) to fulfill the functionality of the feature with CRUD (Create, read, update, delete) operations.

### Create

When a new Item needs to be added, to a database or some other form of storage, we define a ‘create’ function. This function will enable the user to create new items of certain type.

Example:

public Result Create(User user, string password, string role)

### Read

When data needs to be received, from a database or some other form of storage, we define ‘FindBy’ methods. These methods enable us to receive data based on a specific property.

Example:

public User FindById(string id)

public User FindByUsername(string username)

### Update

When data needs to be update, to a database or some other form of storage, we define an ‘update’ methods. This method accepts the object that needs to be updated.

Example:

public Result Update(User user, string role)

### Delete

When data needs to be deleted, from a database or some other form of storage, we define a ‘delete methods. This method accepts the object that needs to be deleted.

Example:

public Result Delete(User user)

## Complexity level of communication protocols

### SignalR

TODO

### REST Api

TODO

## Component or pattern re-use ratio

TODO

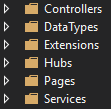
## Reduced duplicated code and functions

TODO

## Source code file organization cleanliness

### ASP.NET Core

For each ASP.NET Core application, we organize our sourcecode files in the following folder structure.



**Controllers**

This folder will contain all the Controller Classes used by the application. Controllers are needed when data needs to be accessible via REST end point.

Example:

[Route("[controller]/[action]")]

public class SimpleController : Controller

{

[HttpPost]

public IActionResult DoSomething()

{

return Ok();

}

}

**DataTypes**

This folder will contain all the Datatype Classes. A datatype class is the way to represent which object has wat properties.

Example:

public class User

{

public string Firstname { get; set; }

public string Lastname { get; set; }

}

**Extensions**

This folder will contain all the Extension methods. Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type

Example:

public static bool *HasPermission*(this ClaimsPrincipal user, string permission)

{

return user.HasClaim("permission", permission);

}

**Hubs**

This folder will contain all the different SignalR hubs used by the application.

Example:

public class Chat : Hub

{

public Task Send(string message)

{

return Clients.All.*InvokeAsync*("Send", message);

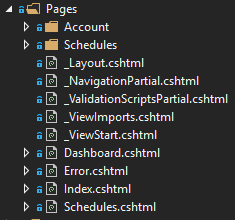
}

}

**Pages**

This folder will contain all the Razor Pages used by the application. These pages are used to create HTML pages that enable the user to view and update data from the application.

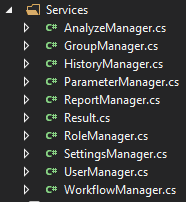
Example:



**Services**

This folder will contain all the different services used by the application. These services are called ‘managers’ and define the core functionality. These services are added using dependency injection so they can be added throughout the application.

Example:



### .NET 4.6.2 Console Application

TODO

## Portability

### ASP.NET Core

### .NET 4.6.2 & Mono 5.4

## Documentation

TODO

# Supplier independence

## .NET Core

Most of our project are developed using the .NET Core technology.

TODO

## .NET Standard

TODO

## Microsoft SignalR

TODO

## External Open Source Libraries

TODO

# Testability

The testability of software components (modules, classes) is determined by factors such as:

## Controllability

*The degree to which it is possible to control the state of the component under test (CUT) as required for testing.*

TODO

## Observability

*The degree to which it is possible to observe (intermediate and final) test results.*

TODO

## Isolateability

*The degree to which the component under test (CUT) can be tested in isolation.*

All of our projects are developed with testability in mind. This enables use to easily write multiple layer tests in order to guarantee each component works as intended.

### Dependancy Injection

By using the ASP.NET Core stack we have the ability to split all of our different functionalities in multiple services (Managers). These services all rely on dependency Injection.

* Database Context
* SignalR Hubs
* Other Services

Herby we have the ability to override some of these functions in order to write a working and isolated test.

## Understandability

*The degree to which the component under test is documented or self-explaining.*

### Test Naming Scheme

For **Unit Test** we use a naming scheme that makes test easily readable and define their function.

**Class Name**

As class name we use a combination of 2 words. The fist word is the function that is being tested followed by what parameter is passed to this function. These 2 words are separated by an underscore ‘\_’.

For Example:

public class Acknowledge\_ValidReport

// Acknowledge is the function that is being tested

// ValidReport is the parameter that is being passed

**Method Name**

As the method name of a test we use a combination of 3 words. The first word is what object we are testing. The second word is the property of that object and the last word is the expected result.

For Example:

public void Result\_IsSucceeded\_False()

// Result is the object that we want to test

// IsSucceeded is the propertie of ‘result’ that we test

// False is the expected result

## Automatability

*The degree to which it is possible to automate testing of the component under test.*

### VSTS Build and Release

Within Intation our project are build and tested on our own test agent with VSTS. This allows us to make sure each new change that is committed is rebuild and tested again.

When one of our tests fail the build will be failed, indicating that something has gone wrong and needs to be fixed.

This system allows us to make sure new developed features have no impact on already existing features.

# Good practices

TODO

# Coding Standards

## C#

### PascalCasing

Use **PascalCasing** for class names and methods names.

public class ClientActivity

{

public void ClearStatistics()

{

//...

}

public void CalculateStatistics()

{

//...

}

}

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

### CamelCasing

Use **camelCasing** for method arguments and local variables.

public class UserLog

{

public void Add(LogEvent logEvent)

{

int itemCount = logEvent.Items.Count;

// ...

}

}

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

### Hungarian Notation

Do not use **Hungarian** notation or any other type identification in identifiers

// Correct

int counter;

string name;

// Avoid

int iCounter;

string strName;

**Why**: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

### Screaming Caps

Do not use **Screaming Caps** for constants or readonly variables

// Correct

public static const string ShippingType = "DropShip";

// Avoid

public static const string SHIPPINGTYPE = "DropShip";

**Why**: consistent with the Microsoft's .NET Framework. Caps grap too much attention.

### Abbreviations

Avoid using **Abbreviations**.

Exceptions: abbreviations commonly used as names, such as **Id, Xml, Ftp, Uri**

// Correct

UserGroup userGroup;

Assignment employeeAssignment;

// Avoid

UserGroup usrGrp;

Assignment empAssignment;

// Exceptions

CustomerId customerId;

XmlDocument xmlDocument;

FtpHelper ftpHelper;

UriPart uriPart;

**Why**: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

Use **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

HtmlHelper htmlHelper;

FtpTransfer ftpTransfer;

UIControl uiControl;

**Why**: consistent with the Microsoft's .NET Framework. Caps would grap visually too much attention.

### Underscores

Do not use **Underscores** in identifiers.

Exception: you can prefix private variables with ‘m\_’.

// Correct

public DateTime clientAppointment;

public TimeSpan timeLeft;

// Avoid

public DateTime client\_Appointment;

public TimeSpan time\_Left;

// Exception

private DateTime m\_registrationDate;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

### Predefined type names

use **predefined type names** instead of system type names like Int16, Single, UInt64, etc

// Correct

string firstName;

int lastIndex;

bool isSaved;

// Avoid

String firstName;

Int32 lastIndex;

Boolean isSaved;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read.

### Var types

Use implicit type **var** for local variable declarations.

Exception: primitive types (int, string, double, etc) use predefined names.

var stream = File.Create(path);

var customers = new Dictionary();

// Exceptions

int index = 100;

string timeSheet;

bool isCompleted;

**Why**: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.