

## Documentation for Developers

Coding standards for ATSPM v1.4

10/11/2017

Coding Standards

# Why implement coding standards

Best applications are coded property. What this means is that code not only does its job well, but is also easy to add to, maintain and debug. As new developers start working in a new application and its code base they start going through lines and lines of existing code. If original application developers make its purpose and intentions clear in the code, then the process of onboarding new developers is faster and requires less interaction from the members of the original team. It also has an effect of teaching beginner developer’s good practices. The goal of implementing coding standards is an overall application consistency that will allow developers to look at any part of the application and figure out what's going on in a reasonable amount of time.

Coding standards are not rule of the land, but collaborative effort to create better, more maintainable code and therefore are subject to change as the circumstances and needs change.

# Formatting

Formatting should be automatically handled by Visual Studio Editor settings or other editor tools (like Resharper). Use the Microsoft defaults for particular language used except when there is a mutually agreed exception to the defaults. This exception would be added to (Visual Studio or Resharper) formatting export that can be easily imported by any developer working on the code base.

# Commenting

Instead of commenting classes and functions try to follow the Clean Coders mantra that code should be self-explanatory and therefore self-documenting. Comments should be added when there is no obvious way to self-document complex functionality or when certain functionality is implemented for reasons that would otherwise not be obvious from code.

# Naming conventions

The names of functions and variables should be descriptive. Try to create short but descriptive variable names and function names.

Two types of capitalization rules are used.

#### LowerCamelcase

First letter lowercase, but after that all words capitalized and run together (“***thisIsAnExample”***).

#### UpperCamelcase

First letter is capitalized, and after that all words are capitalized and run together (“***ThisIsAnExample***”).

For variables:

* global variables are UpperCamelcase eg. ***NumberOfThreads***
* inputs are LowerCamelcase eg. ***deviceId***
* methods and classes are UpperCamelcase eg. ***GetNumberOfUser*s()**
* private static variable should start with \_ eg. ***\_userName***
* constants should be all capital letters eg. ***PRECISION***

Avoid these for variables:

* Variables names separate by \_ eg. user\_name
* All upper case eg. USERNAME
* Abbreviations (userGroup vs. usrGrp)

Grammatical rules for naming:

* Methods are verbs. The name of a class should always be a noun or a noun phrase like Account or MessageParser. And avoid noise words like Processor, Manager, Data, Info, etc.
* Variables hold instances of classes, so they should be nouns too. In fact, variables often have a name identical to or derived from their class. For example: account or depositAccount.
* Boolean variables should be written as predicates like isEmpty. Or isTerminated so that they read well in ‘if’ statements. Methods should be verbs or verb phrases like PostPayment, or GetPrice. If a function returns a boolean its name should be a predicate like isEmpty.
* Don’t use nouns for accessors like fuelRate or UirstName. Include the verb get instead as in getFuelRate and getFirstName.
* Properties, as in C#, are methods pretending to be variables, so they should be nouns. Boolean properties should be predicates.
* enums tend to be states or descriptors of an object, so they are often adjectives.
  + enum COLOR {RED, GREEN, BLUE};
  + enum Status {PENDING, CLOSED, CANCELLED};
  + enum Size {SMALL, MEDIUM, LARGE};

# Error Handling and Logging

ATSPM uses the Application Events table to log errors and other events that may need to be tracked. Events can be added with a critical level of High, Medium, Low, or Informational. The application events repository implements a quick add that you can utilize for adding events. See **Appendix A** for a code snippet to make it easier to apply the repository quick add.

Use try catch statements in your functions and let the central handling process log them and call other functions based on the error that has occurred.

Avoid having empty catch statements. Implementing empty catch statement without logging or pushing data to central error handler hides the exceptions that are occurring and can cause a significant challenge in debugging non-working process. The only time to use the empty catch statement is for noise suppression. For example, when handling a known exception that is considered noise (socket disconnect).

In JavaScript use the following console logging functions

* Console.info
* Console.log
* Console.error

Log to console should have if (DEBUG) set:

if (DEBUG) Console.error("we were not able to get user ID");

# Functions

Functions should be around 15 lines long. If they are longer consider refactoring the code to adjust the function length and encapsulate the functionality.

Functions should follow the guidelines for naming and error handling and should also have the following comment that explains the purpose of the function. This can be automatically generated by typing /// above the function declaration.

/// <summary>

/// generate a customProtocol session and return the visible access code

/// </summary>

/// <param name="userId">unique user id</param>

/// <param name="deviceId">unique device id</param>

/// <param name="usersRole">role type to use for this session</param>

/// <param name="cacheRedis">redis cache for customProtocol sessions</param>

/// <param name="cacheRetryPolicy">global cache retry policy</param>

/// <param name="errorCatcher">error catcher</param>

/// <returns>session string on success or null on any failure</returns>

# Architecture

For interaction with the database use a repository factory architecture. This allows for injection of test data anywhere in the system when repositories are used. The repositories use Entity Framework. The following general principles for Entity Framework apply:

1. No underscores in table names
2. Pluralize Table names
3. Models should be singular

There are certain exceptions to using Entity Framework and the Repository Factory architecture. Namely Entity framework does not handle bulk inserts well. Also on the front end if you depend on Lazy Loading across various models you will need to either pass in the context or directly access the context.

See **Appendix B** for an example of the Repository Factory classes.

# Visual Studio and Resharper Exports

There is an export of Visual Studio and Resharper settings that can be loaded to each developer’s workstation. The export is located at the following location “xxx”.

# General guidelines

Add general guidelines here

# Appendix A - Application Events Quick Add Code Snippet

<?xml version="1.0" encoding="utf-8"?>

<CodeSnippet Format="1.0.0" xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">

<Header>

<Title>Error Log</Title>

<Author>Derek Lowe</Author>

<Shortcut>errL</Shortcut>

<Description>Inserts an entry into the application event log table</Description>

<SnippetTypes>

<SnippetType>SurroundsWith</SnippetType>

<SnippetType>Expansion</SnippetType>

</SnippetTypes>

</Header>

<Snippet>

<Declarations>

<Literal>

<ID>error</ID>

<Default>error function</Default>

</Literal>

</Declarations>

<Code Language="XML">

<![CDATA[var errorLog = ApplicationEventRepositoryFactory.Create();

errorLog.QuickAdd(System.Reflection.Assembly.GetExecutingAssembly().GetName().ToString(),

this.GetType().DisplayName (), e.TargetSite.ToString(), ApplicationEvent.SeverityLevels.High, e.Message);

throw new Exception("");]]>

</Code>

</Snippet>

</CodeSnippet>

# Appendix B – Repository Factory Architecture Example

**Interface:**

namespace MOE.Common.Models.Repositories

{

public interface IFAQRepository

{

List<Models.FAQ> GetAll();

Models.FAQ GetbyID(int id);

void Add(Models.FAQ item);

void Remove(int id);

void Update(Models.FAQ item);

}

}

**Implementation:**

namespace MOE.Common.Models.Repositories

{

public class FAQRepository : IFAQRepository

{

Models.SPM db = new SPM();

public List<Models.FAQ> GetAll()

{

return db.FAQs.OrderBy(f => f.OrderNumber).ToList();

}

public Models.FAQ GetbyID(int id)

{

return db.FAQs.Where(m => m.FAQID == id).First();

}

public void Add(Models.FAQ item)

{

db.FAQs.Add(item);

db.SaveChanges();

}

public void Remove(int id)

{

FAQ faq = GetbyID(id);

if (faq != null)

{

db.FAQs.Remove(faq);

db.SaveChanges();

}

else

{

throw new Exception("FAQ Not Found");

}

}

public void Update(Models.FAQ item)

{

FAQ faqFromDatabase = GetbyID(item.FAQID);

if (faqFromDatabase != null)

{

db.Entry(faqFromDatabase).CurrentValues.SetValues(item);

db.SaveChanges();

}

else

{

throw new Exception("FAQ Not Found");

}

}

}

}

**Factory:**

namespace MOE.Common.Models.Repositories

{

public class FAQsRepositoryFactory

{

private static IFAQRepository faqRepository;

public static IFAQRepository Create()

{

if (faqRepository != null)

{

return faqRepository;

}

return new FAQRepository();

}

public static void SetMenuRepository(IFAQRepository newRepository)

{

faqRepository = newRepository;

}

}

}