Smart NFC Card Application

Coding Standards Document

Vast Expanse (Group 7)

COS 301 - 2019

Wian du Plooy	*u17237263*
Duncan Vodden	*u17037400*
Tjaart Booyens	*u17021775*
Savvas Panagiotou	*u17215286*
Jared O'Reilly	*u17051429*

Table of Contents

Introduction

Coding standards are a set of rules that serve as requirements and guidelines for writing programs for a project or within an organisation. This document will outline the various coding standards and coding conventions which will be used throughout the implementation and development of the Smart-NFC-Application project. Throughout the project, our documentation will occur between comments which follow the following standards, where information will be included in the lines with a single asterisk (*):

```
/**
*
*/
```

Coding Standards

File Headers

A file header will be located at the top of each file that is created, and it specifies various information about the file, including:

ITEM	DESCRIPTION
File Name	The name of the file that is being presented
Project Name	Name of the project for which the file was written for
Organisation Name	Name of the organisation which creates the program
Copyright	Copyright information
List of Classes	A List of the classes declared and implemented in the file
Related Documents	A list of the related documents (URLs included if possible)
Update History	A list of updates, specifying the date, author and the change made
Functional Description	Overall description of the functionality and behaviour of the program
Error Messages	A list of error messages that can be produced by the program specified in the file
Assumptions	A list of conditions that must be satisfied or may affect the operation of the program
Constraints	A list of restrictions on the use of the program including restrictions on the input, environment and various other variables

Here is an example of a file header located at the top of a file named helloWorld.js:

```
/**
*       File Name:      helloWorld.js
*       Project:        Smart-NFC-Application
*       Organization:   VastExpanse
*       Copyright:      ©  Copyright 2019 University of Pretoria
*       Classes:        HelloWorld
*       Related documents:      SRS Document - www.example.com
*
*       Update History:
*       Date            Author          Version         Changes
*       ---------------------------------------------------------------------------------
*       2019/05/18      Duncan          1.0             Original
*       2019/05/19      Tjaart          1.1             Added foo Function
*
*       Functional Description:         This class is to demonstrate the use of our Coding
*                                       standards that we will be using throughout our COS
*                                       301 Module
*       Error Messages:         “Error”
*       Assumptions:    None
*       Constraints:    None
*/
… rest of file
```

Description of Classes

A description of each class will be located before the declaration of the class, and this will include information such as:

ITEM	DESCRIPTION
Purpose of class	A statement of the purpose of the class
Usage Instructions	How the class will be used
Author	Specified by **“@author”**. The programmer who created the class
Version	Specified by **“@version”**. The version number of the class

Here is an example of a class description for a class called HelloWorld:

```
/**
*       Purpose:        This class is used demonstration purposes of coding conventions
*       Usage:          This class can be used to output “Hello World” to console by calling
*                       function foo
*       @author:        Duncan Vodden
*       @version:       1.1
*/
class HelloWorld{ … }
```

Description of Class Member Functions

Before a function is declared the following descriptions will be included for each function:

ITEM	DESCRIPTION
Description	Description of what the function will be used for/the function's purpose. The description will not have a label “Description”, instead, the first line will be the description
Parameters	Specified by **“@param paramName paramDataType paramDescription”** where paramName is the name of the parameter, paramDataType is the data type of the parameter and paramDescription is a description of what the parameter is in relation to the function
Return	Specified by **“@return returnDataType returnDescription”** where returnDataType is the data type of the returned object and returnDescription is a description of the object returned

Here are some examples of function descriptions for functions in the HelloWorld class:

```
/**
*       The constructor of the class is used to initialise the hello attribute of the class
*/
constructor(){
        this.hello = “Hello World”;
}

**
*       This function prints out a message to the console and then returns true
*       @param hello string This is a string passed into the function, printed first
*       @param bye string This is a string passed into the function, printed last
*       @return bool Return true after console logged output
*/
bar(hello, bye){
        console.log(hello);
        console.log(bye);
        return true;
}
```

Naming Conventions

The following naming conventions should help in the complete understanding and future maintenance of the program:

ITEM	CODING CONVENTION	CORRECT	INCORRECT
Folders (Backend)	Folders on the backend system will follow the rules of camel casing and will start with uppercase characters.	BackEndServer	backEndServer backend_Server baCkEndServer
Folders (App)	Folders used in the app system will follow the Angular framework standard, since autogenerated folders are named that way.	nfc-service	Nfc-Service nfc_service nfcService
Files (Backend)	Files on the backend system will follow the rules of camel casing and start with lowercase characters.	appLogic.js	AppLogic.js applogic.js aPP_LoGic.js
Files (App)	Files used in the app system will follow the Angular framework standard, since autogenerated files are named that way.	nfc-controller.ts	nfc-Controller.ts nfc_controller.ts nfcController.ts
Classes	Classes will follow the rules of camel casing and will start with uppercase characters.	AdminLogic	adminLogic Adminlogic AdmiNLogiC
Attributes	Attributes will follow the rules of camel casing and start with lowercase characters.	obj.firstName	obj.First_Name obj.firstname obj.fIrStNaMe
Functions	Functions will follow the rules of camel casing and start with lowercase characters.	function getApiKey()	function GetApiKey() function getapikey() function getAPIKEY()
Constants	Constants will be declared in all uppercase characters with underscores separating words.	TIMEOUT_LIMIT	TimeoutLimit timeout_limit timeoutLIMIT
Variables	Variables will follow the rules of camel casing and start with lowercase characters.	var loopCount	var LoopCount var loopcount var loop_Count

Formatting Conventions

Formatting conventions specify formatting rules used to arrange program statements. The following formatting conventions should help in the structure of our program and help us write neater code:

ITEM	CODING CONVENTION	EXAMPLE

Line Break	Have a line break after every function to show separation clearly and in every function have a line break between different code blocks such as before and after loops.
	The first line of the function/loop/condition starts on the line after the function definition/loop/condition.
		function foo(){
		block1;
		xxxxxxx
		block2;
		}
		xxxxxxx
		function bar(){ … }

Indentation	Use tabs for indentation and indent separate coding blocks within each other with one more tab.	for(...){
		(tab) for(...){
		(tab)(tab) for(...){
		(tab)(tab) }
		(tab) }
		}

Alignment	After a coding block has started the '{' will be on the same line and the '}' will be aligned with the first character that started the coding block.	while(cond) {
		…
		}
		
		if(cond) {
		…
		}
		else {
		…
		}

Spacing	Add a space before and after every opening and closing parenthesis. For assignments and comparisons, a space is added before and after every operator. Spaces will be added for comments after the '//'.	… (...) … … x = y ... … // comment here ...

In-Code Comment Conventions

In-Code Comments will aid the understanding as well as the maintenance of the program.

These comments will be kept to a minimum to avoid having code that looks cluttered with comments. These comments should occur before a block of code to explain what that specific block of code achieves and are signalled with the // sequence of characters.

Code Quality

Code Quality through Standards

The quality of code can be ensured if the specified coding standards for the code are followed to the tee. In this instance of the term 'quality', we are not referring to the functionality of the code and if it meets user requirements, rather, we refer to the ability of the code to be understood, edited and continuously maintained by any software developer (including the original author of the code).

If the standards defined for the code are followed, the code will be more readable, understandable and clear. The use of naming and formatting conventions allow the code to be more readable and elements of the code to be more identifiable, whereas the use of file headers, class descriptors and function descriptors make the semantic role of what they describe clearer, and lead to a better general understanding of the workings of the code.

Therefore, we can see that enforcing these coding standards onto our code will lead to better code: indeed, code of a higher quality. However, to manually enforce these standards is tedious and time consuming - which is why we will use certain auto-formatting tools to ensure that any code written will be altered (in terms of appearance, not semantics) into code that follows the coding standards.

Linting Tools

There are various 'linting' tools available on the Internet. Linting tools are pieces of software which are configured to format code into a pre-defined format. For example, if we configured our linter to ensure all { after an if(…) should be on the following line, when we lint our code using the linter, it will find any if(…) and { sequences and ensure the required format is present - if not, it will change the code for you.

As our project is in Javascript, it is best to use linting tools designed for Javascript. These include JSLint, ESLint, JSHint, JSCS and Standard JS. All these linting tools can be downloaded from the Internet and the desired code formats can be fed into the linters as input. Our team will be using primarily ESLint, as it works very well as a plugin into VSCode, an IDE which most of us use.

Therefore, using these automatic linting tools, we can ensure that our code is always formatted in the correct way, as defined by our coding standards, before we push any code to our shared repository. This will ensure the quality of our code.

Repository File Structure

Below is an example of the file structure of our code repository, where all the code and documentation for our project exists and is organized: