**CODING STANDARD AND GUIDELINES**

1. **Indentation**

As we know proper and consistent indentation is important in producing easy to read and maintainable programs.We have used 4 space (1 tab) for indentation.

**Example for node/javascript -**

Test.distinct("tag", function(err, listTest){  
 if (err){  
 console.log(err);  
 }else{  
 console.log("Test command executed Successful!!!");  
 console.log(listTest);  
 contestList = listTest;   
 }  
});

1. **Inline Comments**

Inline comments explaining the functioning of the subroutine or key aspects of the algorithm shall be frequently used.

**Example-**

//post requests  
 app.post('/next',function (req, res) {e-

1. **Structured Programming**

JavaScript is asynchronous in nature and so is Node. Asynchronous programming is a design pattern which ensures the **non-blocking code execution** i.e. at compile time code is not executed sequentially as written.

Also, **GO TO** statements shall not be used as they lead to “spaghetti” code, which is hard to read and maintain.

1. **Classes, Subroutines, Functions and Methods**

In javascript we have used, **Arrow functions** (also called “fat arrow functions”) are undoubtedly one of the more popular features of **ES6**. They introduced a new way of writing concise functions.

**Example-**

Without ES6 (old method)

*function timesTwo(params) {  
 return params \* 2  
 }  
 timesTwo(4); // 8*

With ES6 (used by us)

*var timesTwo = params => params \* 2  
 timesTwo(4); // 8*

1. **Source Files**

The name of the source file or script shall represent its function. All of the routines in a file shall have a common purpose.

1. **Variable Names**Ourvariable have mnemonic or meaningful names that convey to a casual observer, the intent of its use. All the variables have been initialized prior to its first use. In our scenario we have used **CamelCase** variable name throughout the project.
2. **Use of Braces**

In some languages, braces are used to delimit the bodies of conditional statements,

control constructs, and blocks of scope. We have used either of the following

bracing styles.

**Example :**

for (var j = 0 ; j < max\_iterations ; ++j)

{

/\* Some work is done here. \*/

}

or the Kernighan and Ritchie style:

for ( var j = 0 ; j < max\_iterations ; ++j ) {

/\* Some work is done here. \*/

}

It is felt that the former brace style is more readable and leads to neater-looking code than

the latter style, but either use is acceptable.

Braces has been used even when there is only one statement in the control block.

**For example:**

if (j == 0)

{

console.log(“j is zero”);

}

1. **Compiler Warning**

Compilers often issue two types of messages: warnings and errors. Compiler warnings normally do not stop the compilation process. However, compiler errors do stop the compilation process, forcing the developer to fix the problem and recompile. Compiler and linker warnings shall be treated as errors and fixed. Even though the program will continue to compile in the presence of warnings, they often indicate problems which may affect the behavior, reliability and portability of the code.

Some compilers have options to suppress or enhance compile-time warning messages.

Developers study the documentation and/or man pages associated with a compiler and choose the options which fully enable the compiler’s code-checking features.

For example the –Wall option fully enables the gcc code checking features and should always be used:

**nodemon** index.js

1. **Line length**

It is considered good practice to keep the lengths of source code lines at or below 80 characters. Lines longer than this may not be displayed properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

1. **Spacing**

The proper use of spaces within a line of code can enhance readability. Good rules of thumb are as follows:

* A keyword followed by a parenthesis should be separated by a space.
* A blank space should appear after each comma in an argument list.
* All binary operators except “.” should be separated from their operands by spaces.
* Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement(“—“) from their operands.
* Casts should be made followed by a blank space.

**Bad example :**

cost=price+(price\*sales\_tax);

**Better example:**

cost = price + ( price \* sales\_tax );

1. **Wrapping lines**

When an expression will not fit on a single line, break it according to these following principles:

* Break after a comma

**Example:**

var questionSchema = new mongoose.Schema({

question : String,

op1 : String,

op2 : String,

op3 : String,

op4 : String,

ans : String

});

* Break after an operator

**Example:**

Console.log(“This is my ”+

name);

* Prefer higher-level breaks to lower-level breaks

**Example:**

Bad:

longName1 = longName2 \* (longName3 + LongName4 – longName5) + 4 \* longName6 ;

Better:

longName1 = longName2 \* (longName3 + LongName4 – LongName5) + 4 \* longName6 ;

* Align the new line with the beginning of the expression at the same level on the previous line.

**Example:**

total\_windows = number\_attic\_windows + number\_second\_floor\_windows + number\_first\_floor\_windows ;

1. **Variable Declarations**

Variable declarations that span multiple lines should always be preceded by a type.

**Example:**

Acceptable:

var name, author;

Not Acceptable:

var name,

author;

1. **Program Statements**

Program statements should be limited to one per line. Also, nested statements should be avoided when possible.

**Example:**

Bad:

UserSchema.plugin(passportLocalMongoose); Console.log(“Bad Stmt”);

Better:

UserSchema.plugin(passportLocalMongoose);

Console.log(“Good Stmt”);

1. **Use of Parentheses**

It is better to use parentheses liberally. Even in cases where operator precedence unambiguously dictates the order of evaluation of an expression, often it is beneficial from a readability point of view to include parentheses anyway.

**Example:**

Acceptable: total = 3 – 4 \* 3 ;

Better: total = 3 – ( 4 \* 3 ) ;

Even better: total = ( -4 \* 3 ) + 3 ;

1. **Coding for Efficiency vs. Coding for Readability**

There are many aspects to programming. These include writing software that runs efficiently and writing software that is easy to maintain. These two goals often collide with each other. Creating code that runs as efficiently as possible often means writing code that uses tricky logic and complex algorithms, code that can be hard to follow and maintain even with ample inline comments. The programmer needs to carefully weigh efficiency gains versus program complexity and readability. If a more complicated algorithm offers only small gains in the speed of a program, the programmer should consider using a simpler algorithm. Although slower, the simpler algorithm will be easier for other programmers to understand.

1. **Meaningful Error Messages**

Error handling is an important aspect of computer programming. This not only includes adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

Error messages should be meaningful. When possible, they should indicate what the problem is, where the problem occurred, and when the problem occurred. A useful NodeJS Error handling feature is the option to show a stack trace, which shows the sequence of method calls which led up to the exception.

Code which attempts to acquire system resources such as dynamic memory or files should always be tested for failure.

Interactive applications can either send error messages to a log file, standard output, or standard error. Interactive applications can also use popup windows to display error messages.

1. **Reasonably Sized Functions and Methods**

Software modules and methods should not contain an excessively large number of lines of code. They should be written to perform one specific task. If they become too long, then chances are the task being performed can be broken down into subtasks which can be handled by new routines or methods. A reasonable number of lines of code for routine or a method is 200. This does not include documentation, either in the function/method header or inline comments.