



CODING STANDARDS

Blockchain Live Project

Blockchain Based Authentication System

|  |  |  |  |
| --- | --- | --- | --- |
| **Created By:** | Sheefa Jalali | **Approved By:** | Harshada Topale |
| **Created On:** | 16-09-2023 | **Approved On:** | 25-09-2023 |

**INDEX**

[**1** **PURPOSE** 2](#_Toc143202970)

[**2** **SCOPE** 2](#_Toc143202971)

[**3** **FILE STRUCTURE** 2](#_Toc143202972)

[3.1 Standard File Conventions 2](#_Toc143202973)

[3.2 Markdown Files 2](#_Toc143202974)

[3.3 Common Conventions 3](#_Toc143202975)

[**4** **FORMATTING CONVENTIONS** 3](#_Toc143202976)

[4.1 Indentation 3](#_Toc143202977)

[4.2 Using Capitalization to Aid Readability 3](#_Toc143202978)

[4.3 Formatting Single Statements 3](#_Toc143202979)

[4.4 Formatting Declarations 3](#_Toc143202980)

[4.5 Formatting Multi-line Statements 3](#_Toc143202981)

[**5** **NAMING CONVENTIONS** 3](#_Toc143202982)

[**6** **SCOPING CONVENTIONS** 4](#_Toc143202983)

[6.1 Lexical/Static Scoping 4](#_Toc143202984)

[6.2 Dynamic Scoping 4](#_Toc143202985)

[**7** **COMPILE ERRORS & WARNINGS** 4](#_Toc143202986)

[**8** **ENFORCING CODING STANDARD** 4](#_Toc143202987)

[**9** **APPENDICES** 1](#_Toc143202988)

[9.1 Appendix A – Detailed Description of Components 1](#_Toc143202989)

**General Instructions for using the Live Project Coding Standard Template**

* This template and the subsequent document created using this template is a confidential document and is the intellectual property of Cloud Counselage Pvt. Ltd. Circulating it outside of the organisation without the consent of Cloud Counselage Pvt. Ltd. is the breach of company policies and will lead to legal actions
* This template is a guideline and the Developers can define their own conventions as deemed appropriate for delivering this project based on the programming language/framework they are working on.
* The **text between inequality (< >) is to be replaced** by relevant text
* Please **remove the yellow highlight on the Text** between the inequality (< >). This is done to help you notice the text to be changed/replaced
* The text in *italics* highlighted in grey is just for reference and should be removed after adding the relevant text

# **PURPOSE**

The Coding Standards are the guidelines for software Developers to create uniform coding habits that eases the reading, checking, and maintaining code. The intent of these standards is to define a natural style and consistency, yet leave to the authors, the freedom to practice their craft without unnecessary burden.

The coding standards shall enable the following:

* **Improve Code Quality**: Coding standards ensure that code is written consistently, readably, and maintainable manner. This makes it easier for developers to understand and work with the code, leading to higher-quality software.
* **Increase Efficiency**: By following coding standards, developers can save time by avoiding common mistakes and implementing proven solutions.
* **Facilitate Collaboration**: It creates a common language that all developers can understand and allows teams to collaborate, share code, and communicate effectively.
* **Ensure Compatibility**: It ensures that code is compatible with different platforms, browsers, and OS-device combinations.
* **Reduce Maintenance Costs**: By following established standards, developers can avoid introducing new bugs and make changes to code more quickly and easily.

The coding standards should follow the below best practices:

1. Focus on code readability.
2. Enable Commenting
3. Formalising Exception Handling

# **SCOPE**

This document describes general software coding standards for code written for Solidity specific to Blockchain Based Authentication System and shall be implemented while developing the code for the said project.

# **FILE STRUCTURE**

The ‘File Structure’ allow developers to know where files are, when to use specific code, and locate associated results. Not only do file structures streamline productivity, but they also increase code consistency and shareability.

## Standard File Conventions

**Folder Structure:**

Organize by Functionality: Group related files into folders based on their functionality or purpose (e.g., contracts, tests, UI).

Follow a Logical Hierarchy: Maintain a logical folder hierarchy, making it easy to locate files.

**Documentation Files:**

Use Standard Extensions:

“.md” for Markdown files (e.g., README.md for project README).

“.txt” for plain text files.

“.pdf” for PDF documents.

**Source Code Files:**

Use Appropriate Extensions:

“.sol” for Solidity smart contract files.

“.js” for JavaScript files.

“.html” for HTML files.

“.css” for CSS files.

**Version Control:**

Follow Git Conventions:

Use “.gitignore” to specify files and directories to be ignored by version control. Commit messages should be concise, descriptive, and follow a consistent style.

## Markdown Files

**File Naming:**

Use descriptive names with the .md extension (e.g., README.md,

CONTRIBUTING.md).

**Headers:**

Use # for headers (headings), with one to six # symbols representing headings of different levels.

Use headers for titles, sections, and subsections to create a structured document.

**Text Formatting**:

Use \*\*bold text\*\* for bold formatting.

Use \*italic text\* for italic formatting.

Use ~~strikethrough~~ for strikethrough text.

**Lists:**

Use - or \* for unordered lists and numbers (e.g., 1., 2.) for ordered lists.

**Links:**

Use [<a href='https://www.linkedin.com/in/sheefajalali/' target='blank'>] to create hyperlinks.

**Images:**

Use ![ <Image

src='https://react.semantic-ui.com/images/avatar/large/steve.jpg'

wrapped ui={false}

/>](image URL) to embed images.

**Code Blocks:**

Use triple backticks (```) for code blocks. Specify the programming language for syntax highlighting (if applicable).

**Tables:**

Create tables using pipe | and hyphen - for headers and separator.

**Quotes:**

Use > for block quotes.

**Line Breaks:**

Use two spaces at the end of a line to create a line break.

**Comments:**

Use HTML comments <!-- comment --> for any additional comments.

**Escape Characters:**

Use \ as an escape character for characters that have special Markdown meanings (e.g., \\* for literal \*).

# Blockchain Based Authentication

This is a \*\*Markdown file\*\* for demonstration.

## Table of Contents

1. [Introduction](#introduction)

2. [Installation](#installation)

## Introduction

This is a brief introduction to the project.

## Installation

To install the project, follow these steps:

1. Clone the repository.

2. Run `npm install` to install dependencies.

3. Run the application using `node app.js`.

![Project Logo](https://example.com/project-logo.png)

> This is a block quote.

<!-- This is a comment -->

## Common Conventions

**Version Control:**

Use a version control system like Git.

Follow a branching strategy (e.g., Gitflow) to organize development efforts.

Ensure each commit is a logical and cohesive change.

**Coding Standards:**

Adhere to a consistent coding style guide.

Use meaningful and descriptive variable, function, and class names.

Comment complex or non-intuitive code to improve readability.

**Documentation:**

Maintain a project-wide README file that provides an overview of the project, its purpose, and how to set it up.

Document code using inline comments for functions, classes, and non-obvious logic.

Create and maintain a separate CONTRIBUTING.md file outlining contribution guidelines for the project.

**Testing:**

Write comprehensive unit tests to validate code functionality.

Aim for a high level of test coverage to ensure robustness.

Implement continuous integration (CI) and continuous deployment (CD) processes.

**Issue Tracking**:

Use an issue tracking system (e.g., GitHub Issues) to manage tasks, enhancements, and bugs.

Clearly define issue templates for various types of tasks to standardize information.

**Collaboration:**

Foster effective communication within the team through regular meetings and status updates.

Encourage collaboration and knowledge sharing among team members.

**Error Handling:**

Implement consistent and meaningful error handling throughout the codebase.

Log errors for effective debugging and monitoring.

**Security:**

Follow best practices for secure coding and data handling, especially since the project involves authentication.

Regularly update dependencies and libraries to patch security vulnerabilities.

**Build and Deployment:**

Automate the build process for the project to streamline development and deployment.

Use containerization tools (e.g., Docker) for consistent deployments across different environments.

**License:**

Clearly define and document the project's licensing terms to ensure compliance and proper usage.

**Code Reviews:**

Conduct code reviews for all changes to maintain code quality and consistency.

Provide constructive feedback and encourage peer learning.

**File Organization:**

Organize files in a clear and logical directory structure based on functionality and file type.

# **FORMATTING CONVENTIONS**

These conventions are all about the positions of line breaks, how many characters should go on a line, and everything in between.

## Indentation

**General Indentation**:

Use 2 or 4 spaces for each level of indentation.

Choose either spaces or tabs and be consistent across the entire codebase.

**JavaScript (Solidity) and HTML:**

Use 2 or 4 spaces for indentation.

Indent the content within a block (e.g., functions, loops, conditionals).

**Example (JavaScript):**

function exampleFunction() {

if (condition) {

console.log('Condition is true.');

} else {

console.log('Condition is false.');

}

}

**Example (Solidity):**

function exampleFunction() public view returns (uint256) {

if (condition) {

return 42;

} else {

return 0;

}

}

**Example (HTML):**

<html>

<body>

<div>

<p>Hello, world!</p>

</div>

</body>

</html>

**CSS:**

Use 2 or 4 spaces for indentation.

Indent nested selectors and rules.

**Example:**

body {

font-size: 16px;

}

.container {

width: 80%;

margin: 0 auto;

}

**JSON:**

Use 2 or 4 spaces for indentation.

Indent nested objects and arrays.

**Example:**

{

"key1": "value1",

"key2": {

"nestedKey": "nestedValue"

},

"key3": ["item1", "item2"]

}

## Using Capitalization to Aid Readability

**File and Folder Names:**

Use lowercase letters for file and folder names.

Separate words with underscores (\_) or hyphens (-) for improved readability.

Example: user\_profile.js, contract\_abi.json, css\_styles.

**Variables and Function Names:**

Use camelCase for variables and function names.

Start with a lowercase letter and capitalize the first letter of subsequent concatenated words.

Example: accessToken, verifyUserCredentials, contractInstance.

**Constants:**

Use uppercase letters for constant variables.

Separate words with underscores (\_) for clarity.

Example: MAX\_ATTEMPTS, API\_BASE\_URL, DEFAULT\_TIMEOUT.

**Classes and Constructor Functions:**

Use PascalCase for class names and constructor functions.

Capitalize the first letter of each concatenated word.

Example: AuthenticationService, UserCredentials.

**Enums:**

Use uppercase letters for enum values.

Separate words with underscores (\_) for clarity.

Example: StatusEnum, PermissionTypes.

**Acronyms and Abbreviations:**

Capitalize all letters in acronyms or abbreviations.

Example: API\_URL, HTTPRequest.

**HTML Elements and Attributes:**

Use lowercase letters for HTML elements and attributes.

Example: <div>, <input type="text">.

**CSS Classes and IDs:**

Use lowercase letters for class names and IDs.

Separate words with hyphens (-) for better readability.

Example: .user-profile, #main-content.

**Comments:**

Write comments in sentence case with appropriate punctuation.

Use capitalization rules as per regular text.

## Formatting Single Statements

**JavaScript, Solidity and HTML:**

Use a newline after the statement for clarity.

Indent the statement to align with the surrounding code for readability.

**Example (JavaScript):**

function exampleFunction() {

let result = performOperation();

console.log(result);

}

**Example (Solidity):**

function exampleFunction() public view returns (uint256) {

uint256 result = performOperation();

return result;

}

**Example (HTML):**

<script>

let result = performOperation();

console.log(result);

</script>

**CSS:**

Use a newline after the statement for clarity.

Indent the statement to align with the surrounding code for readability.

**Example:**

body {

font-size: 16px;

}

**JSON:**

Use a newline after the statement for clarity.

Indent the statement to align with the surrounding code for readability.

**Example:**

{

"key": "value",

"another\_key": "another\_value"

}

## Formatting Declarations

JavaScript (Solidity):

Use one variable declaration per line.

Use meaningful variable names and initialize variables at the point of declaration whenever possible.

Align variable declarations for better readability.

**Example (JavaScript):**

let username = "john\_doe";

let isLoggedIn = true;

const MAX\_ATTEMPTS = 5;

const API\_BASE\_URL = "https://api.example.com";

**Example (Solidity):**

uint256 public totalUsers;

string private contractName;

string private contractName;

**HTML:**

Separate attributes onto new lines, each indented for clarity and readability.

Use double quotes for attribute values.

**Example:**

<input

type="text"

id="username"

placeholder="Enter your username"

>

**CSS:**

Use one declaration per line.

Indent declarations to align with the property for readability.

**Example:**

body {

font-family: Arial, sans-serif;

font-size: 16px;

}

**JSON:**

Use one property per line, indented for readability.

Align property names and values for better readability.

**Example:**

{

"key1": "value1",

"key2": "value2",

"key3": "value3"

}

## Formatting Multi-line Statements

**JavaScript (Solidity):**

For multi-line expressions or function calls, break lines after commas to separate arguments.

Indent continuation lines to align with the opening parenthesis or function call.

If a line is too long, break it into multiple lines, aligning with the first argument.

**Example (JavaScript):**

function exampleFunction(

arg1,

arg2,

arg3

) {

return arg1 + arg2 + arg3;

}

let result = someLongFunctionCall(

argument1,

argument2,

argument3

);

**Example (Solidity):**

function exampleFunction(

uint256 arg1,

uint256 arg2

) public pure returns (uint256) {

return arg1 + arg2;

}

**HTML:**

For multi-line HTML elements, place each attribute on a new line, indented for clarity and readability.

Break lines after the opening tag and before the closing tag.

**Example:**

<input

type="text"

id="username"

placeholder="Enter your username"

>

**CSS:**

For long CSS properties or property values, break the line after each semicolon (;).

Indent continuation lines to align with the property.

**Example:**

.example-element {

font-family: Arial,

sans-serif;

font-size: 16px;

margin: 10px;

}

# **NAMING CONVENTIONS**

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Naming conventions result in improvements in terms of "four Cs": communication, code integration, consistency, and clarity. The idea is that "code should explain itself"

Naming convention is applicable to constants, variables, functions, modules, packages and files. In object-oriented languages, it's applicable to classes, objects, methods and instance variables.

With regard to scope, global names may have a different convention compared to local names; such as, Pascal Case for globals: Optind rather than optind in gawk. Private or protected attributes may be named differently: \_secret or \_\_secret rather than secret. Some may want to distinguish local variables from method arguments using prefixes.

For naming conventions, please refer to <https://www.pluralsight.com/blog/software-development/programming-naming-conventions-explained>

# **SCOPING CONVENTIONS**

Scoping is generally divided into two types:

## Lexical/Static Scoping

A variable in this scope always refers to its top-level environment. This characteristic of the program text has nothing to do with the call stack at runtime. Static scoping makes it considerably easier to write modular code because a programmer can find out the scope by looking at the code.

## Dynamic Scoping

With dynamic scope, a global identifier directs to the identifier associated with the most current environment and is unusual in modern languages. In technical terms, each identifier has a global stack of bindings, and the most current binding is explored for events of the identifier.

In another way, the Compiler successfully explores the current block and all calling functions first in dynamic scoping.

# **COMPILE ERRORS & WARNINGS**

## Errors

Errors report problems that make it impossible to compile your program.

When developing programs there are three types of error that can occur:

* **Syntax error** occurs when the code given does not follow the syntax rules of the programming language. A program cannot run if it has syntax errors. Examples include:
  + misspelling a statement, e.g. writing pint instead of print
  + using a variable before it has been declared
  + missing brackets, eg opening a bracket, but not closing it

Any such errors must be fixed first. A good integrated development environment (IDE) usually points out any syntax errors to the programmer.

* **Logic error** is an error in the way a program works. The program can run but does not do what it is expected to do. Logic errors can be caused by the programmer:
  + incorrectly using logical operators, eg expecting a program to stop when the value of a variable reaches 5, but using <5 instead of <=5
  + incorrectly using Boolean operators
  + unintentionally creating a situation where an infinite loop may occur
  + incorrectly using brackets in calculations
  + unintentionally using the same variable name at different points in the program for different purposes
  + using incorrect program design
* **Runtime error** is an error that takes place during the running of a program. An example is writing a program that tries to access the sixth item in an array that only contains five items. A runtime error is likely to crash the program.

## Warnings

Warnings report other unusual conditions in your code that may indicate a danger points where you should check to make sure that your program really does what you intend**.**

Compiler warnings are useful, but they are highly unreliable. In addition, they are no substitute for language subsetting.

Please refer [this article for understanding working with compiler warnings.](https://www.linkedin.com/pulse/compiler-warnings-use-them-dont-trust-roberto-bagnara/)

# **ENFORCING CODING STANDARD**

Please refer [this article](https://www.linkedin.com/advice/3/how-do-you-enforce-coding-standard-across-different?src=go-pa&trk=sem-ga_campid.20316911727_asid.154319842041_crid.663989285736_kw._d.c_tid.dsa-2085021268780_n.g_mt._geo.9300016&mcid=7080236969011671041&cid=&gclid=Cj0KCQjwuNemBhCBARIsADp74QTSZVIz_1ypt1X9-S2GDXhfwONgO7hnaHP_IPfLJbKcgG3v2gU4Zp0aAkAnEALw_wcB&gclsrc=aw.ds) to enforce coding standard across different toolsand platform.

# **APPENDICES**

## Appendix A: Blockchain-Based Authentication Objectives

This appendix elaborates on the objectives achieved through the implementation of blockchain-based authentication, focusing on enhancing security, privacy, and user experience in the authentication process.

## Appendix B: FILE STRUCTURE

In this appendix, the file structure of the blockchain-based authentication project is outlined, detailing the organization of files, directories, and their respective purposes.

## Appendix C: FORMATTING CONVENTIONS

This appendix provides a comprehensive overview of the formatting conventions applied in the project, including indentation, line breaks, and other formatting practices for ensuring consistent code styling.

## Appendix D: NAMING CONVENTIONS

This appendix outlines the naming conventions followed in the project, encompassing variables, functions, classes, and other identifiers, ensuring meaningful and standardized names.

## Appendix E: SCOPING CONVENTIONS

In this appendix, the scoping conventions used in the project are elucidated, defining the scope of variables and functions to ensure appropriate access and visibility.

## Appendix F: COMPILE ERRORS & WARNINGS

This appendix catalogues common compile errors and warnings encountered during the development process, offering solutions and explanations for resolving them effectively.

## Appendix G: ENFORCING CODING STANDARD

## This appendix describes the methods and tools used to enforce coding standards in the project, emphasizing consistency and adherence to established conventions throughout the development lifecycle.