Coding Standard and Best Programming Practices

Version Description

Version	**Date (DD-MM-YYY)**	**Updated By**	**Comment**
v0.0	22-11-2025	Carl Mervyn Marinas	Document Created
v1.1	24-11-2025	Carl Mervyn Marinas	Document updated to adhere to document standards.
v.1.2	25-11-2025	Christian Timothy Araño	Document updated to restructure file structure section.

Contents of this Document

Introduction

This document establishes the coding standards and the best practices for the Reports Management system developed for the organization “Unbound Manila”. The purpose of this document is to ensure consistency, readability, and maintainability of the code across all modules of the system.

This also serves as a guide for current and future developers who seek to further progress the application. It outlines the naming conventions, formatting rules, and documentation requirements that should be followed to maintain the overall quality and scalability of the application.

Scope

The Report Management System’s main programming language is *JavaScript* for the backend connections and functionalities, with *Embedded JavaScript* and *CSS* for the frontend. The application utilizes *Supabase PostgreSQL* service to build the relations for the database, and the main information management system. Additionally, the application utilizes the *Dropbox* API for storing reports. And finally, the application utilizes the *Render* web hosting service for deployment.

The standards outlined in this document encompasses all layers of the system, that being the user interfaces, data handling and its integration with backend services.

Coding Standards

A. Project Framework

Auth: The auth folder handles all authentication-related logic and UI for the application, including user login, registration, and access control.

Core: The core folder includes widgets or functions that might be common across all pages, regardless of their role.

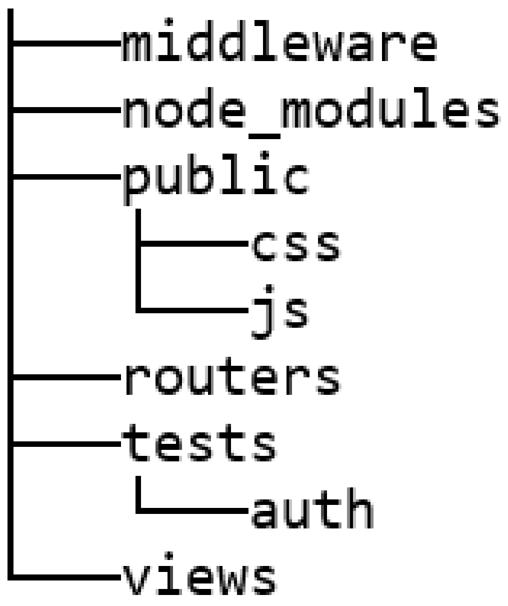
Data: The data folder contains .json files with dummy data, mostly used during the development of the interface’s layout. This may be removed later upon deployment.

Features: The features folder houses all pages for all 3 types of roles, with each role having their own respective package.

Assets: The assets folder is located outside of the lib folder and contains the resources for building the layout such as icons and fonts.

B. File Structure

The project directory would typically follow this structure:



```
├───middleware
├───node_modules
├───public
│   ├───css
│   └───js
├───routers
├───tests
│   └───auth
└───views
```

1. **Root:** Top level of the project. This directory level contains the `package.json/package-lock.json`, configuration files (e.g., `.env`, `.gitignore`), README file, and the main server file (e.g., `app.js`).

2. **Middleware:** Contains custom middleware functions for server frameworks (e.g., *Express*), such as authentication middleware.

3. **Node Modules:** This directory is an automatically generated/adjusted folder containing all installed *NPM* packages. Developers do not need to manually edit the contents of this folder, and it is recommended to be excluded from *Git* via the `.gitignore` file.

4. **Public:** Contains static assets/resources served directly to the website browser. This directory generally contains these sub-directories:
    a. **CSS**: This sub-directory includes front-end stylesheets (e.g., `styles.css`).
    b. **JS:** This sub-directory includes client-side *JavaScript* files.

5. **Routers:** Contains server-side functionalities such as route modules that define API endpoints or page/file routes (e.g., `home.js`, `login.js`).

6. **Tests:** Contains files for unit, or integration testing.
    a. **Auth:** Contains tests specifically targeting authentication flow.

7. **Views:** Contains templates for server-rendered pages (e.g., `login.ejs`, `register.ejs`), which are rendered using `res.render()`.

C. Naming Conventions

Package Names: Package names will take on the Snake Case naming convention with a limited number of 30 characters, not including the extension.

```
e.g. admin_payment_review_details.dart
```

Class Names: Concrete classes should use natural descriptive names, begin with a capital letter, and utilize the Pascal Case naming convention.

```
e.g. AdminPaymentReviewDetails
```

Function Names: Function names should use natural descriptive names and make use of the Camel Case naming convention.

```
e.g. function getAccountDetails(<parameters>) { }
```

Variable Names: Variables of any parent should use natural descriptive names, begin with a capital letter, and utilize the Camel Case naming convention. The purpose of the variable name must be made clear especially at the end of name.

```
e.g.
/// For holding temporary data
final List<Map<String, dynamic>> activeLoansData = activeLoans;

/// Controller that takes user input for back end processing
final TextEditingController loanAmtController = TextEditingController();

/// Takes user input for back end processing
int? selectedMemberId;

/// Errors to be displayed
String? staffSearchError;
```

D. Declarations

When declaring variables in a state class, be sure to observe the following order:

1. **Controllers:** Because controllers handle user input, these are declared as final to prevent reassignment. They must also be disposed of in the `dispose()` method to prevent memory leaks

```
e.g.
final TextEditingController loanAmtController = TextEditingController();

@override
void dispose() {
  loanAmtController.dispose();
  super.dispose();
}
```

2. **State Initializers:** These are variables that set default values within forms and other movable parts which may also be passed on to the back end for further processing.

```
e.g.
// Sets its table in ascending order by default
int? sortColumnIndex;
bool isAscending = true;

// waits for user input which controls the interface
String? selectedReportType;
```

3. **Layout Constants:** Layout constants are used to create a more uniform interface.

```
e.g.   double buttonHeight = 28;
```

4. **Test Data:** Lastly, test data are variables that take imported dummy data from the data folder which will later be used in layout assessments. This is placed at the end of other more important functions for easier deletion when it is no longer needed.

```
e.g.   final List<Map<String, dynamic>> activeLoansData = activeLoans;
```

E. **Parenthesis:** For multi-block statements, open parentheses are placed on the same line as its statement after which is followed by a new line and an indentation of where its logic should be placed.

```
e.g.
Container(
    child: Row(
        // logic is placed here
    )
)
```

F. **Braces:** Braces take on the same rules as parentheses.

```
e.g.
Widget activeLoans() {
    return Container(
        child: Row(
            children: [
                 // other widgets are placed here
            ]
        )
    );
}
```

G. **Line Spacing and Indentation:** Make sure to leave 2 blank lines between logical code blanks. To ensure uniformity, make certain that there are no trailing blank lines.

```
e.g.
Container(
    child: Row(
        children: [
            Text("Active Loans"),
            <more code for active loans>,


            Text("Overdue Loans"),
            <more code for overdue loans>,
        ]
    )
)
```

H. **Commenting Code:** Use `///` for documentation comments and `//` for inline or single-line notes.

`///` Documentation comments

```
e.g.
/// A widget that displays the list of active loans for a member.
///
/// This widget retrieves loan data from the database and
/// shows it in a paginated table format.
class MemberLoanList extends StatefulWidget {
  const MemberLoanList({super.key});

  @override
  State<MemberLoanList> createState() => _MemberLoanListState();
}
```

`//` Documentation comments

```
E.g.

// Sign in with email and password
Future<AuthResponse> signInWithEmailPassword(
    String input_credential, String password) async {
    <more function logic>
    }
)
```

I. **Error Handling:** Catch errors gracefully using try/catch blocks and user-friendly messages.

```
e.g.
Future<void> submitForm() async {
  try {
    await supabase.from('loans').insert({
      'member_id': selectedMemberId,
      'amount': double.parse(loanAmtController.text),
    });

    // Notify user of success
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Loan submitted successfully!')),
    );
  } catch (error) {
    // Log and display the error
    debugPrint('Error submitting loan: $error');
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Failed to submit loan. Try again.')),
    );
  }
}
```

Coding Documentation Template:
https://www.docsity.com/en/docs/coding-standard-software-engineering-i-eel-5881/6757935/