



CODING STANDARDS

WEB DEVELOPMENT  
  
COLLEGE FINDER INDIA

|  |  |  |  |
| --- | --- | --- | --- |
| **Created By:** | Prachi Jain | **Approved By:** | <Domain Lead Name> |
| **Created On:** | 19-07-2025 | **Approved On:** | DD-MMM-YYYY |

Page left blank intentionally

**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 TypeScript/React specific to College Finder India 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

* Use lowercase and hyphens for folder and file names (e.g., college-results.tsx, gemini-service.ts).
* Group related files in folders:
* components/ for React components
* services/ for API and utility logic
* assets/ for images and static files
* public for static files served directly
* Place the main entry point as main.tsx and the root component as App.tsx.
* Keep each file focused on a single responsibility.

## Markdown Files

* Use Markdown (.md) files for documentation, such as README.md (project overview), CONTRIBUTING.md (contribution guidelines), and CHANGELOG.md (release notes).
* Place documentation files in the project root.
* Use clear headings, bullet points, and code blocks for clarity.
* Document components and services with usage examples in Markdown when appropriate.

## Common Conventions

* Avoid large files; split code into smaller, reusable modules.
* Keep related files together (e.g., component .tsx and its .css in the same folder).
* Use relative imports within the src directory.
* Remove unused files and code regularly to keep the codebase clean.

# **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

* Use 2 spaces per indentation level (no tabs).
* Indent nested blocks, functions, and JSX elements consistently.

## Using Capitalization to Aid Readability

* Use PascalCase for React component and class names (e.g., CollegeResults, LocationSelector).
* Use camelCase for variables, functions, and object properties (e.g., searchLocation, handleSearch).
* Use UPPER\_CASE for constants (e.g., API\_URL).

## Formatting Single Statements

* Place single statements on their own line.
* Always use semicolons at the end of statements.
* Avoid unnecessary inline statements.

## Formatting Declarations

* Declare one variable, function, or class per line.
* Use const and let appropriately; prefer const by default.
* Type all variables and function parameters explicitly in TypeScript.

## Formatting Multi-line Statements

* Break long statements (over 100 characters) into multiple lines for readability.
* For JSX, place each prop on a new line if there are more than two.
* Align closing brackets with the opening element or statement.
* For object and array literals, put each item on a new line if the literal spans multiple lines.

# **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 – Component Table

|  |  |  |
| --- | --- | --- |
| Component Name | File Path | Description |
| App | src/App.tsx | Main application component, manages state & flow |
| main | src/main.tsx | Entry point, renders App |
| LocationSelector | src/components/LocationSelector.tsx | Location input and search UI |
| CollegeResults | src/components/CollegeResults.tsx | Displays list of colleges |
| geminiService | src/services/geminiService.ts | Handles Gemini AI API requests |
| DebugPanel | src/components/DebugPanel.tsx | (If present) Debugging UI/tools |
| style | src/style.css | Main custom styles |
| App CSS | src/App.css | App-specific styles |
| vite.config | vite.config.ts | Vite build configuration |
| assets | src/assets/ | Static assets (images, icons, etc.) |