CBILITY Technologies Co., Ltd.

Coding Standard

General Standards

1. **Formatting**

   1.1 Prioritize clarity and readability. Avoid unnecessary “innovative” approaches that result in hard-to-read codes.

   1.2 Avoid nesting too many if blocks or loop blocks

   1.3 Always use K&R bracketing style, not Allman style. This is because K&R is the official standard for all languages used in this project.

```
// GOOD EXAMPLE: K&R style
fn function() {
    // ...
}

// BAD EXAMPLE: Allman style
fn function()
{
    // ...
}
```

2. **File Organization**

   2.1 One file should contain one main purpose.

   2.2 Organize code files into modules/namespaces/packages logically and hierarchically.

3. Naming Conventions

3. 1 All namings must be meaningful, yet concise. If there are multiple variables with similar purposes, make sure the name differentiate them enough.

```
// Bad example
let a = 10;
let b = 20;
let c = 30;

// Better example
let maxUsers = 30;
let currentUsers = 20;
let minUsers = 10;
```

3.2 Struct (or class) names must be nouns or noun clauses.

```
// Bad example
class CalculateUser {}

// Good example
class User {}
class Product {}
class OrderHistory {}
```

3.3 Function names must be verbs or verb clauses.

```
// Bad example
function userData() {}

// Good example
function fetchUserData() {}
function calculateTotalPrice() {}
function sendEmailNotification() {}
```

3. 4 Variable names, in general, should be nouns or noun clauses except for boolean variables that act as states or flags. These can start with verbs “to be”, verbs “to have”, or modal verbs.

```
// General variables (nouns)
let userName = "Alice";
let productList: string[] = [];
let orderCount = 5;

// Boolean variables (state/flags)
let isUserLoggedIn = true;
let hasPaidSubscription = false;
let canEditProfile = true;
```

4. Comments

4.1 Every source file must have a header comment containing its description/purpose. Author names and dates can be omitted as those are already tracked by git.

```
/**
 * userManagement.ts
 *
 * This module handles user-related operations
 * such as registration,
 * login status checks, and profile updates.
 */
```

4.2 Every variable must have a comment clarifying its purpose.

```
// The maximum number of users allowed in the system
let maxUsers = 100;

// List of registered users in the system
let registeredUsers: string[] = [];
```

4.3 Every function must have a comment clarifying its purpose as well as the definitions of each of the input arguments and outputs.

```
/*
 * Checks if a user is currently active.
 *
 * @param userName - The username to check
```

```
 * @returns boolean - True if the user is active, false
otherwise
 */
function checkUserActive(userName: string): boolean {
    return activeUsers.includes(userName);
}
```

4.4 Keep comments updated when code changes.

5. Error Handling

5.1 Always handle exceptions gracefully. Minimize crashes or panics.

```
/**
 * Reads a user's profile data from a database.
 * @param userId - The ID of the user
 * @returns object | null - Returns user data if found, null
otherwise
 */
function getUserProfile(userId: string): object | null {
    try {
        return databaseFetch(userId);
    } catch (error) {
        console.error(`Failed to fetch user profile for ID
${userId}:`, error);
        return null; // Graceful fallback
    }
}

// Example usage
const profile = getUserProfile("12345");
if (!profile) {
    console.log("User profile could not be retrieved.");
}
```

5.2 Validate inputs where possible. Note that each input should only be validated once.

6. **Logging**

6.1 Always log with an appropriate log level. Common log levels include DEBUG, INFO, WARN, and ERROR.

6.2 DEBUG: Use for detailed technical info during development/troubleshooting. Example: function inputs/outputs, decision branches.

6.3 INFO: Use for normal but important application events. Example: user login, service started, background job completed.

6.4 WARN: Use when something unexpected happens, but the app continues running. Example: deprecated API call, retry attempt, slow query.

6.5 ERROR: Use when something fails and requires immediate investigation. Example: database connection failure, payment API error.

SvelteKit-specific Standards

1. **Casings**

1.1 Variables, attributes, and functions: camelCase.

```
const formattedName = formatUserName(newUser);
```

1.2 Types, interfaces, and component names: PascalCase.

```
export type UserData = { firstName: string; lastName: string };
```

1.3 SCSS class names: kebab-case.

```
.user-card { ... }
```

1.4 Constants and enum members: UPPER_SNAKE_CASE.

```
const MAX_USER_COUNT = 100;

export enum UserRole {
  ADMIN = 'ADMIN',
  MODERATOR = 'MODERATOR',
```

```
  USER = 'USER'
}
```

1.5 Generics: Upper single-letter

```
function wrapInArray<T>(value: T): T[] { ... }
```

2. Page Structure

2.1 Page components must follow SvelteKit file naming conventions:

"+page.svelte" : the Svelte component (HTML and minimal script)

"+page.ts" : data-loading functions (Page initialization).

2.2 Non-data-loading functions must not go into +page.ts. Instead, use a separate "<purpose>-utils.ts" file in the same page directory.

2.3 Page-specific styles must be in style.scss in the same folder. This can be imported to "+page.svelte" in the <script> or <style> tag.

2.4 <script> blocks in +page.svelte should only contain minimal Typescript. Minimal Typescript includes:

- Import statements
- Props declaration
- Reactive variables declaration
- Functions that require interacting with reactive variables such as event handlers
- Small amount of other functions/variables are allowed if offloading them makes the code less readable

Other codes should be offloaded to ./<purpose>-utils.ts or lib/utils/<purpose>-utils.ts

```
src/routes/users/   // in user page directory
  +page.svelte      // HTML and minimal TS
  +page.ts          // data-loading function
  user-utils.ts     // functions and classes related to user
  camera-utils.ts   // functions and classes related to camera
  style.scss        // page-specific styles
```

2.5 Utility TypeScript files export what to be imported and used by +page.svelte.

```
// in user-utils.ts
export type User = {
  firstName: string;
  lastName: string;
};

export function formatUserName(user: User): string {
    return `${user.firstName} ${user.lastName}`;
}


// in +page.svelte
<script lang="ts">
  import { formatUserName, type User } from './user-utils.ts';
  // Use this for transitive styles only
  import './style.scss'

  const newUser: User = {
    firstName: 'John',
    lastName: 'Doe'
  };

  const formattedName: string = formatUserName(newUser);
</script>

// Use this this for component-scoped styles (not transitive)
<style>
  @use './style.scss'
<style/>

<p>{formattedName}</p>
```

2.7 Reusable styling variables such as colors must be declared in lib/styles/ directory. They can be imported using the “@use” syntax by other scss files.

```
// in lib/styles/colors.scss
$grey-text: #6f6f6fff;
```

```scss
$grey-light: #8c8c8cff;
$grey-border: #3a3a3aff;

// in other scss files
@use "../lib/styles/colors.scss" as colors;

.auth-header h1 {
  font-size: 24px;
  font-weight: 600;
  color: colors.$grey-text; //variable is used here
  margin: 0;
}
```

3. **Reusable Component Structure**

3.1 All reusable components must reside in: src/lib/components/{ComponentName}/

3.2 File naming within the component folder:

```
src/lib/components/{ComponentName}/   // in component directory
  ComponentName.svelte      // HTML and minimal TS
  style.scss                 // component-specific styles
```

3.3 Any page can import these components from their +page.svelte

```
// In +page.svelte of a page
<script lang="ts">
  import UserCard from
'$lib/components/ComponentName/ComponentName.svelte';
</script>

<UserCard {user} />
```

4. **Reusable Functions Structure**

4.1 All globally reusable functions must reside in: src/lib/utils/

4.2 File naming within the component folder:

```
src/lib/utils/      // in utils directory
  date-utils.ts     // contains date-related functions
```

```
user-utils.ts     // contains user-related functions
database-utils.ts // contains database-related functions
```

4.3 Any page can import these functions just like how they import components

TypeScript-specific Standards

1. **Casings**

   Follow casings standards defined in SvelteKit-specific Standards section

2. **Formatting**

   2.1 Use semi-colon consistently

   2.2 Use trailing commas in objects/arrays

```
const user = {
  id: 1,
  name: 'Alice',
};
```

   2.3 Always explicitly specify input/output types of every function

```
const add = (a: number, b: number): number => a + b;
```

   2.4 Always explicitly specify the types for all variables

```
let name: string = 'Alice';
const age: number = 30;
```

   2.5 Avoid using “any” type. Use Union types instead.

```
let value: string | number;
```

Rust-specific Standards

1. **Casings**

   Follow the official "rustfmt" standard casings.

   1.1 Variables and functions: snake_case

```rust
let user_name: &str = "Alice";
fn get_user_name() -> &str { "Alice" }
```

   1.2 Structs, Enums, and Traits: PascalCase

```rust
struct UserAccount { id: u32, name: String }
enum Role { Admin, User }
trait Greetable { fn greet(&self); }
```

   1.3 Constants and statics: UPPER_SNAKE_CASE

```rust
const MAX_RETRY: u32 = 5;
```

2. **File organization (Generic Rust)**

   2.1 Each module must be defined under src/{module_named}

   2.2 Inside each module folder, there must be at least 2 files: mod.rs and test.rs.

   2.3 Other files may be added to offload codes from mod.rs and keep mod.rs clean

   2.4 Unit tests module (test.rs) must cover all functions defined in mod.rs

```
src/database_interface/  // in module directory
  mod.rs    // contains main functionality for the module
  types.rs  // contains user structs and impl blocks
  test.rs   // Unit tests for all functions defined in mod.rs
```

3. **File organization (Tauri-specific)**

   3.1 main.rs must be directly under "src" or "src-tauri/src". This file only runs the application. It does not do anything else.

   3.2 lib.rs must be directly under "src" or "src-tauri/src". This file

   - Defines all functions that can be called by SvelteKit and TypeScript.
   - Defines and maintains app states.

- Define how the application is initialized and run.

4. Formatting

4.1 Since rustfmt (Rust official formatter) is already very strict. The code is automatically considered well-formatted as long as it follows rustfmt standard. The standard can be applied by running “cargo fmt” command.

4.2 The code must not contain any warnings when compiled, including non-functional warnings such as unused variables, unneeded mutability for variables, and so on.

5. Best practises

5.1 “unsafe” flag is not allowed to maintain Rust’s memory safety nature.

5.2 Use Enums such as Option and Result instead of “special value”

```
// BAD EXAMPLE: Return an empty user if user not found
fn find_user_by_id_bad(id: u32) -> User {
    // Some logic...
    if found {
        // ...found
        User { id, name}
    } else {
        // ...not found, returning a "special" empty user
        User { id: 0, name: "".to_string() }
    }
}

// GOOD EXAMPLE: Return Option::Some(User) if found,
Option::None otherwise.
fn find_user_by_id(id: u32) -> Option<User> {
    // Some logic...
    if found {
        // The user was found, wrap it in Some
        Some(User { id, name})
    } else {
        // The user was not found, return None
        None
```

```
    }
}
```

5.3 Handle errors gracefully. Never use “unwrap”, “expect”, or similar handlers which might lead to panics.

```
// BAD EXAMPLE: panics if file opening fails
let file = File::open("file.txt").unwrap()
// do something with "file" variable

// GOOD EXAMPLE: check file opening status before using the file
let file = File::open("non_existent.txt");

match file {
    Ok(f) => {
    // do something with "f" variable
    },
    Err(e) => eprintln!("Error opening file: {}", e),
}
```