Coding Standard Documentation

Coding Standards - Front-End (Flutter)

1. File naming

- **Standard:** Use snake_case for file names.
- **Description:** All file names should be in snake_case to follow Flutter conventions and maintain consistency.
- **Example:** `create_route.dart`, `location_provider.dart`

2. Class naming

- **Standard:** Use PascalCase for class names.
- **Description:** Class names should follow PascalCase as it is a common convention in Dart and Flutter for naming classes.
- **Example:** `class CreateRoute`, `class AddStopScreen`

3. Variable naming

- **Standard:** Use camelCase for variable names.
- **Description:** Variable names should be in camelCase for readability and to follow Dart conventions.
- **Example:** `int numberOfPeople`, `double rangeAlert`

4. Private variables

- **Standard:** Prefix private variables with an underscore (`_`).
- **Description:** Variable names should be in camelCase for readability and to follow Dart conventions.
- **Example:** `_stopNameController.text`, `_searchController`

5. Method naming

- **Standard:** Use camelCase for method names.
- **Description:** Method names should follow camelCase to maintain consistency and readability in Dart.
- **Example:** `void addStop(String name, LatLng location, TimeOfDay time)`

6. Import order

- **Standard:** Order imports by length from shortest to longest.
- **Description:** To maintain an organized and readable structure, import statements should be ordered from shortest to longest in terms of characters.
- **Example:**

```
 import 'package:flutter/material.dart';
import 'package:latlong2/latlong.dart';
import 'package:share_your_route_front/core/widgets/create_route_widgets.dart';
import 'package:share_your_route_front/modules/route_creation/presenters/add_stop_screen.dart';
```

7. Trailing comas

- **Standard:** Use trailing commas in multi-line collections and function parameters.
- **Description:** Trailing commas make it easier to add new elements, properties, or parameters without modifying the existing lines. This can result in cleaner diffs when changes are made.
- **Example:**

```
 List<String> items = [
  'item1',
  'item2',
  'item3',
];

// In function parameters
void myFunction({
  required String name,
  required int age,
  required String address,
}) {
  // Function body
}
```

8. Indentation

- **Standard:** Use consistent indentation to ensure readability.
- **Description:** The code should be correctly formatted and properly indented for readability and maintainability. The standard for the indentation is the use of two white spaces instead of a tab.
- **Example:**

```
return Scaffold(
  appBar: AppBar(
    title: const Text('Agregar Parada'),
  ),
  body: Column(
    children: [
      Padding(
        padding: const EdgeInsets.all(8.0),
        child: TextField(
          controller: _stopNameController,
          decoration: const InputDecoration(
            hintText: 'Nombre de la parada',
            border: OutlineInputBorder(),
          ),
        ),
      ),
    ],
  ),
);
```

9. Using `const` for Widgets that Don't Change

- **Standard:** Apply `const` modifier to immutable widgets.
- **Description:** Mark widgets as `const` if they do not change once assigned. This optimization prevents these widgets from rebuilding, thus improving performance by reducing memory usage and preventing unnecessary re-renders.
- **Example:**

```
Widget build(BuildContext context) {
  return const Scaffold(
    appBar: AppBar(
      title: Text('Title'),
    ),
    body: Center(
      child: Text('Hello, World!'),
    ),
  );
}
```

10. Consistent Naming for State and Widget Classes

- **Standard:** State classes should follow the Widget class names.
- **Description:** State class names should be prefixed with an underscore and match the widget class name to maintain a clear relationship.
- **Example:**

```
class CreateRoute extends StatefulWidget {
  @override
  _CreateRouteState createState() => _CreateRouteState();
}

class _CreateRouteState extends State<CreateRoute> {
  // State implementation
}
```

Use of the Flutter extension for coding standards enforcement

The Flutter extension for the Visual Studio Code IDE is highly recommended for several compelling reasons. By installing this extension, the development workflow is significantly enhanced, ensuring adherence to best practices, and minimizing errors. Some of the main benefits with this extension are the following:

Automatic Installation of the Dart Extension

The Flutter extension automatically installs the Dart extension, which provides comprehensive support for the Dart programming language. This ensures that all the necessary tools to write, debug, and optimize Flutter code are effectively installed.

Ensures Proper Code Formatting

The Flutter extension enforces consistent code formatting according to the Flutter style guide. It automatically formats code, helping to maintain a clean and readable codebase.

Detects and Highlights Best Practices

The extension provides real-time feedback and warnings for any code that does not adhere to Flutter's best practices. This includes highlighting potential performance issues, deprecated methods, and other anti-patterns.

Automated Closing Comments

The extension can automatically add closing comments for nested widgets, which is particularly useful when working with deeply nested widget trees. This feature helps keep track of which widget is being closed, enhancing readability.

Enhanced Debugging Tools

The Flutter extension provides powerful debugging tools, including a visual debugger, hot reload, and performance profiling. These tools help identify and fix issues quickly, optimizing the development workflow.

Integrated Testing Support

The extension offers integrated support for running and debugging tests within the IDE. It is easy to run unit tests, integration tests, and widget tests to ensure the reliability of the application.

Coding Standards - Back-End (Express)

1. File Naming Conventions

- **Standard:** Use `camelCase` for JavaScript files.
- **Description:** This is to maintain consistency across the project and to follow JavaScript community conventions.
- **Example:** `authController.js`, `locationRoutes.js`

2. Variable Naming Conventions

- **Standard:** Use `camelCase` for variable names.
- **Description:** `camelCase` is a widely adopted convention for variable names in JavaScript, enhancing readability.
- **Example:** `let firstName = "John"`

3. Constants Naming Conventions

- **Standard:** Use `UPPER_CASE` for constants.
- **Description:** Constants are named in `UPPER_CASE` to distinguish them from variables that can change.
- **Example:** `const PORT = process.env.PORT || 4001;`

4. Function Naming Conventions

- **Standard:** Use `camelCase` for function names.
- **Description:** `camelCase` is a widely adopted convention for function names in JavaScript, enhancing readability.
- **Example:**

```
function doSomething(){
  // Function body
}
```

5. Error Handling

- **Standard:** Use try-catch blocks for asynchronous operations.
- **Description:** Proper error handling ensures that the application can gracefully handle unexpected issues.
- **Example:**

```
try {
  const userRecord = await auth.createUser({ ... });
} catch (error) {
  return res.status(400).json({ error: error.message });
}
```

6. Response Standardization

- **Standard:** Standardize the structure of API responses.
- **Description:** Consistent response structures make it easier for clients to parse and understand the API.

- **Example:**

```
res.status(200).json({ message: "Login successful", token });
```

7. Middleware Usage

- **Standard:** Use middleware for repeated logic like authentication.
- **Description:** Middleware helps keep the code DRY (Don't Repeat Yourself) by centralizing repeated logic.
- **Example:**

```
 const { authenticate } = require("./middleware/authMiddleware");
app.use("/api", authenticate, locationRoutes);
```

8. Environment Variables

- **Standard:** Use environment variables for configuration.
- **Description:** Environment variables help keep configuration secure and flexible across different environments.
- **Example:**

```
 dotenv.config();
const PORT = process.env.PORT || 4001;
```