Coding Standards

FrontEnd - Ionic/Angular

File and Folder Structure

- Files and folders are organised based on a feature or module.
- Used consistent naming conventions for files and folders.
- Separated components, services, and models into their respective folders.

Component Naming

- Component names follow the PascalCase convention.
- Component names are suffixed with Page, Component, or Dialog.

Module Organization

- Each feature or functionality has its own module.
- Related components, services, and models are grouped within the same module.
- Modules are defined and configured using the NgModule decorator.

Component Structure

- Components follow the Angular component structure, utilizing decorators such as @Component, @Input, @Output, and @ViewChild.
- Templates, styles, and logic are organized into their respective sections.
- Components adhere to the Single Responsibility Principle (SRP) by focusing on specific tasks.

Services

- Services encapsulate business logic and data manipulation.
- Services are registered either in the root module (AppModule) or feature modules based on their scope.
- Dependency injection is used to inject services into components.

Code Formatting

- Consistent indentation and formatting conventions are followed.
- Proper spacing and line breaks are used to enhance code readability.
- Variable and function names are descriptive, conveying their purpose.

TypeScript Best Practices

- Strict type checking is enabled by setting "strict": true in the tsconfig.json file.
- TypeScript features like interfaces, generics, and type annotations are utilized.
- The any type is avoided unless absolutely necessary.

Angular/Ionic Best Practices

- Angular's built-in directives and features are used whenever possible.
- Ionic's UI components are leveraged to achieve consistent and responsive design.
- Observables and reactive programming are employed for handling asynchronous operations.

Error Handling and Logging

- Error handling mechanisms are implemented to enhance the user experience.
- Angular's error handling mechanisms, such as ErrorHandler or global error interceptors, are utilized.
- Errors and exceptions are logged for debugging purposes.

Testing

- Unit tests are written using Jasmine.
- Components, services, and other application logic are tested.
- Karma and Cypresss are also used for testing.

Backend - Rust

File and Module Structure:

- Organized files and modules based on the application's features or functionality.
- Use a consistent naming convention for files and modules.
- Separate route handlers, models, and utility functions into their respective modules.

Endpoint Routing

- Endpoint routes are defined using attribute macros provided by Rocket, such as `#[get]`, `#[post]`, `#[put]`, `#[delete]`.
- Related routes are grouped within the same module.
- Dynamic routes are handled using route parameters and path variables.

Route Handlers

- Route handlers are implemented as functions with clear and descriptive names.
- Route handlers follow the single responsibility principle, focusing on specific tasks.
- Request data handling and validation are performed using Rocket's request guards and extractors.

Data Models

- Data models or request/response objects are defined using Rust structs.
- Traits such as `FromForm`, `FromData`, `Serialize`, and `Deserialize` are implemented for data serialization and deserialization.
- External crates like Serde can be used for advanced serialization and deserialization needs.

Error Handling

- Errors are handled and propagated using the `Result` type in route handlers.
- Rocket's error handling mechanisms, such as `Result` combinators (`?` operator) and the `#[catch]` attribute macro, are utilized.

- Custom error types are implemented, or existing crates like `anyhow` or `thiserror` are used for structured error handling.

Middleware

- Rocket's middleware feature is employed for cross-cutting concerns, such as logging, authentication, or request/response modification.
- Custom middleware functions are created or existing middleware crates compatible with Rocket are utilized.

Testing

- Unit tests are written to cover route handlers and other application logic.
- Rocket's `rocket::local::Client` is used to simulate HTTP requests in tests.
- Testing frameworks like `assert` or `expect` macros can be used for assertions.

Code Formatting

- Consistent code formatting conventions, including indentation and line length, are followed.
- Code is formatted using tools like Rustfmt to maintain consistency.
- Descriptive variable and function names enhance code readability.

Documentation

- Documentation comments (///) are included to provide clear explanations of the code.
- Route handlers, important functions, and modules are documented, highlighting their purpose, input, and output.

Security

- Security best practices, such as input validation, authentication, and authorization, are considered.
- User inputs are sanitized to prevent common web vulnerabilities like SQL injection and XSS.