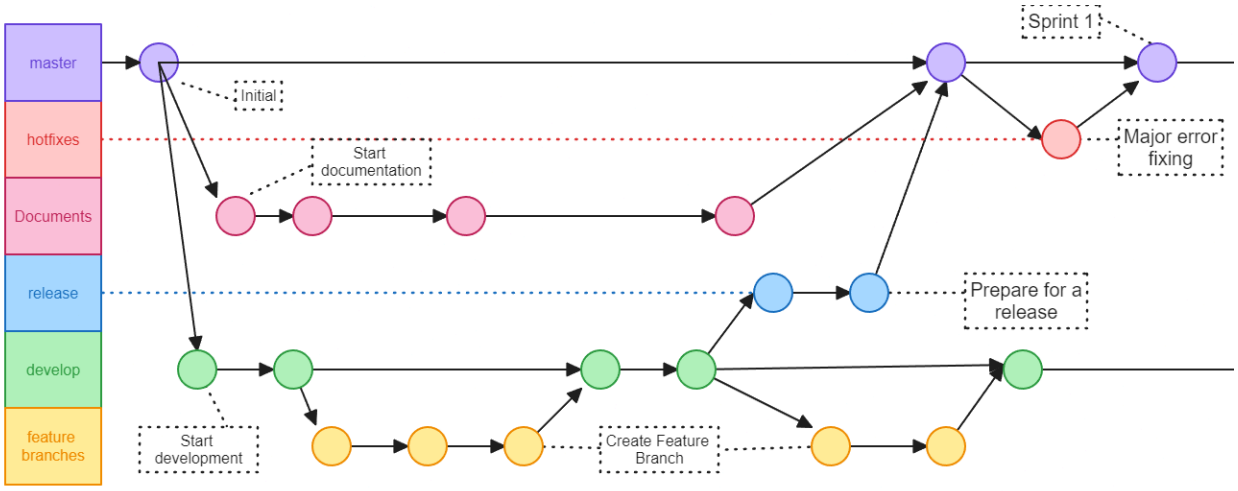

# Coding Standards

29.09.2024

# Introduction

This document outlines coding standards for our project, aiming to ensure consistency, maintainability, reliability, scalability, efficiency, and collaboration.

# Git Strategy




Use Git Flow with main branches (main, develop) and supporting branches (feature, release, hotfix, bugfix).

- Main Branches:
  - `main`: Production-ready code
  - `develop`: Integration branch for feature development
- Supporting Branches:
  - `feature/*`: New features or improvements
  - `bugfix/*`: Non-critical bug fixes
  - `hotfix/*`: Critical production fixes
  - `release/*`: Preparation for new production release

## Commit Messages

- Commit messages structure:
  - type(scope): description
- Example:
  - `feat(user-auth): implement OAuth2 login`

## Branch Management

- Create feature branches from develop
- Merge via Pull Requests with at least one approval
- Squash commits when merging to keep history clean
- Tag releases using semantic versioning.

## Code Review

- Enforce code reviews for all PRs
- Use linters and automated checks in CI pipeline
- Ensure test coverage meets defined thresholds

# File Structure

Monorepo approach with client, config, docs, services and scripts directories.

## Root level

- amplify:
    - Contains AWS Amplify configuration and backend resources
- angular:
    - Angular-specific configurations
- github:
    - Contains GitHub Actions and other GitHub-related configs
- vscode:
    - VS Code editor configurations

## Project level

- amplify:
    - AWS Amplify project files
- cypress:
    - End-to-end testing framework
- documents:

    - Project documentation
- media:
    - Media assets
- node_modules:
    - Node.js dependencies

### Inside src:

- app:
    - Main application code
- services:
    - Angular services
- assets:
    - Static assets like images
- components:
    - Reusable Angular components
- pages:
    - Components representing different pages/routes
- environments:
    - Environment-specific configuration files

### Key files in app:

- api-key.interceptor.ts:
    - HTTP interceptor for API key handling

# Naming Conventions

## General

- Use descriptive, self-explanatory names
- Variables/functions:
    - camelCase (e.g., getUserData)
- Classes/Types/Interfaces:

    - PascalCase (e.g., UserProfile)
- Constants:
    - UPPER_SNAKE_CASE (e.g., MAX_RETRY_ATTEMPTS)
- Boolean variables:
    - prefix with "is", "has", or "should" (e.g., isLoading, hasPermission)

## Angular-specific

- Components:
    - PascalCase with Component suffix (e.g., UserProfileComponent)
- Services:
    - PascalCase with Service suffix (e.g., AuthenticationService)
- Modules:
    - PascalCase with Module suffix (e.g., UserManagementModule)

## DynamoDB

- Tables:
    - snake_case (e.g., user_profiles)
- Attributes:
    - snake_case (e.g., first_name, last_login)

## Git

- Branch names:
    - kebab-case (e.g., feature/user-authentication, bugfix/login-error)

## Interface and Type Definitions

- Use PascalCase for interface and type names
- Use camelCase for properties
- Group related properties
- Example:

```
interface UserProfile {
```

```
  readonly id: string;
  firstName: string;
  lastName: string;
  email: string;
  preferences: {
    theme: 'light' | 'dark';
    notifications: boolean;
  };
}
```

### Function Conventions

- Use camelCase for function names
- Follow the Single Responsibility Principle
- Limit function length to 30 lines; extract longer logic into separate functions
- Use async/await for asynchronous operations

# Code Layout and Formatting

### Indentation and Spaces

| Aspect | JS/TS (Angular) | CSS/SCSS | JSON/YAML |
|---|---|---|---|
| Indentation | Spaces 2 | Spaces 2 | Spaces 2 |
| Line Length | 100 characters | 100 characters | 80 characters |
| Braces | Same line opening new line closing | Same as JS/TS | N/A |

### General Rules

- Use braces for all control structures, even single-line blocks

- Add spaces around operators and after commas
- Use semicolons at the end of statements
- Use single quotes for strings, except when avoiding escapes
- Prefer template literals for string interpolation

### Comments

- Use inline comments sparingly, preferring self-explanatory code
- Keep comments up-to-date with code changes
- Use single-line comments for brief notes, block comments for longer explanations.

### Whitespace

- Use blank lines to separate logical blocks of code
- No trailing whitespace at the end of lines
- End files with a single newline character

# Testing and Debugging

- Use describe/it blocks for test structure.
- Use Cypress for component and end-to-end tests.
- Follow the Arrange-Act-Assert (AAA) pattern
- Write tests that mimic user behavior
- Use data-testid attributes for element selection
- Implement page object models for complex workflows

# Error Handling

### Exception Handling

- Use try-catch blocks for synchronous code
- Use .catch() for promise chains or try-catch with async/await
- Throw specific, custom exceptions for better error identification

## Global Error Handling

- Implement a global error handler using Angular's ErrorHandler
- Log errors to a centralized error logging service

## HTTP Error Handling

- Use HttpInterceptor for global HTTP error handling
- Implement retry logic for transient failures

## Graceful Degradation

- Provide user-friendly error messages
- Implement fallback UI for component failures

## Logging

- Use a centralized logging service
- Log different levels: debug, info, warn, error
- Include contextual information in log messages

# Linting Rules

We use Prettier as our code formatter to maintain consistent code styles across our projects. By adhering to these formatting rules, we maintain clean, consistent, and readable code across our codebase. All team members should integrate this Prettier configuration into their development environment to ensure uniformity in code style.

## ESLint Configuration

- Extend from recommended configs (e.g., eslint:recommended, @typescript-eslint/recommended)
- Enforce strict TypeScript checks
- Prohibit use of any type
- Enforce consistent naming conventions
- Detect and prevent common mistakes

- This configuration
    - A maximum line width of 80 characters
    - Indentation using 4 spaces (not tabs)
    - Semicolons at the end of statements
    - Single quotes for strings
    - Spaces inside object literal braces
    - Parentheses around arrow function parameters
    - Unix-style line endings (LF)

### Additional Formatting Rules:

- Use parentheses to make operator precedence explicit
- Align multiline parameters
- Use meaningful variable names (avoid abbreviations)

### Integration

- Configure IDE extensions (ESLint, Prettier) for real-time feedback

### Custom Rules

- Enforce consistent import ordering
- Enforce use of TypeScript's strict mode

# Code Review Process

All code changes must go through a pull request and be reviewed by at least one other team member.

Reviewers should focus on code quality, adherence to standards, potential bugs, and areas for improvement.

Use constructive feedback and discuss alternative approaches when necessary.

### Regular Review and Updates

Encourage team members to propose improvements and discuss them during review sessions.

Update the document as needed and communicate changes to the entire team.

### Performance and Optimization

Consider performance implications when writing code, especially in critical paths.

Avoid premature optimization; focus on writing clean, maintainable code first.

### Security Best Practices

Validate and sanitize all user inputs to prevent injection attacks.

Use parameterized queries or prepared statements to avoid SQL injection.

- Store sensitive data securely and follow encryption best practices.
- Keep dependencies up-to-date and monitor for security vulnerabilities.

# Documentation

- Maintain a README file for the project, outlining its purpose, setup instructions, and key information.
- Use inline comments to explain complex code segments or non-obvious functionality.
- Keep documentation up-to-date as the codebase evolves.

# Continuous Integration and Deployment (CI/CD)

- Implement a CI/CD pipeline to automate build, test, and deployment processes.
- Run automated tests on each commit to catch regressions early.
- Use a staging environment for testing before deploying to production.
- Include code examples demonstrating good and bad practices for each programming language used in the project.
- Highlight common pitfalls and how to avoid them.