CODING STANDARDS

BMW IT HUB

COS 301 | University of Pretoria

FIRE-FIGHTER ACCESS MANAGEMENT PLATFORM

Table of Contents

1. Introduction

This document outlines the coding standards and best practices for the FireFighter Access Management Platform developed by Team Apex. These standards ensure code consistency, maintainability, security, and quality across the entire project.

1.1 Goals

- **Consistency**: Uniform coding style across all team members
- **Maintainability**: Code that is easy to understand, modify, and extend
- **Security**: Secure coding practices and vulnerability prevention
- **Performance**: Optimized code for production environments
- **Testability**: Code designed for comprehensive testing
- **Documentation**: Well-documented code and APIs

2. Technology Stack Overview

2.1 Backend (FF-API)

- **Framework**: Spring Boot 3.3.5
- **Language**: Java 17
- **Database**: PostgreSQL (Production), H2 (Development/Testing)
- **Security**: Spring Security with JWT and Firebase Authentication
- **Build Tool**: Maven 3.8+
- **Testing**: JUnit 5, Mockito, Spring Boot Test
- **Documentation**: SpringDoc OpenAPI (Swagger)

2.2 Frontend (FF-Angular)

- **Framework**: Angular 19.0.0
- **Language**: TypeScript 5.6.3
- **Mobile**: Ionic 8.0.0 with Capacitor 7.2.0
- **UI Components**: Flowbite 3.1.2 (preferred over Ionic components)
- **Styling**: TailwindCSS 3.3.5 with custom FireFighter theme
- **State Management**: RxJS 7.8.0 with BehaviorSubjects
- **Testing**: Jasmine, Karma, ChromeHeadless
- **Internationalization**: @ngx-translate 16.0.4

2.3 DevOps and CI/CD

- **CI/CD**: Jenkins with GitHub integration
- **Containerization**: Docker with multi-stage builds
- **Code Quality**: ESLint, JaCoCo, Maven Surefire
- **Security Scanning**: OWASP Dependency Check, npm audit

3. Project Structure

3.1 Repository Structure

/Fire-Fighter

```
├── FF-API/                 # Spring Boot backend
├── FF-Angular/             # Angular frontend
├── Documentation/          # Project documentation
├── jenkins-config/         # Jenkins configuration
├── jmeter-testing/         # Performance testing
├── Jenkinsfile            # Main CI/CD pipeline
├── Jenkinsfile.develop    # Development pipeline
└── README.md              # Project overview
```

3.2 Backend Structure (FF-API)

src/main/java/com/apex/firefighter/

```
├── config/                 # Configuration classes
├── controller/             # REST controllers
├── dto/                    # Data Transfer Objects
├── model/                  # JPA entities
├── repository/             # Data access layer
├── security/               # Security configurations
└── service/                # Business logic layer
```

3.3 Frontend Structure (FF-Angular)

src/app/

```
├── components/          # Reusable UI components
├── guards/              # Route guards
├── interceptors/        # HTTP interceptors
├── pages/               # Page components
├── services/            # Business logic services
├── assets/              # Static assets
├── environments/        # Environment configurations
└── theme/               # Styling and themes
```

4. Backend Coding Standards (Java/Spring Boot)

4.1 Naming Conventions

- **Classes**: PascalCase (e.g., `UserService`, `TicketController`)
- **Methods/Variables**: camelCase (e.g., `getUserToken`, `isAuthorized`)
- **Constants**: UPPER_SNAKE_CASE (e.g., `JWT_EXPIRATION_TIME`)
- **Packages**: lowercase with dots (e.g., `com.apex.firefighter.service`)

4.2 Architecture Patterns

- **Controllers**: Handle HTTP requests, delegate to services
- **Services**: Business logic, marked with @Service and @Transactional
- **Repositories**: Data access using Spring Data JPA
- **DTOs**: Data transfer objects for API requests/responses
- **Entities**: JPA entities mapped to firefighter schema

4.3 Code Organization

- Use constructor injection over field injection
- Group imports: `java.*`, `javax.*`, `org.*`, `com.*`
- Document public methods with JavaDoc
- Use meaningful variable and method names
- Keep methods focused and under 50 lines

4.4 Error Handling

- Create custom exception classes for business logic errors
- Use `@ControllerAdvice` for global exception handling
- Log errors with appropriate levels (INFO, WARN, ERROR)
- Never expose stack traces to API responses

4.5 Security Practices

- Validate all input parameters with Bean Validation
- Use parameterized queries to prevent SQL injection
- Implement proper authentication and authorization
- Never log sensitive information (passwords, tokens)

5. Frontend Coding Standards (Angular/TypeScript)

5.1 Naming Conventions

- **Components**: PascalCase with suffix (e.g., `DashboardPage`, `NavbarComponent`)
- **Services**: PascalCase with suffix (e.g., `AuthService`, `TicketService`)
- **Variables/Methods**: camelCase (e.g., `isLoading`, `getUserProfile`)
- **Files**: kebab-case (e.g., `user-profile.component.ts`)
- **Interfaces**: PascalCase with descriptive names (e.g., `UserProfile`, `TicketRequest`)

5.2 Component Architecture

- Use **standalone components** with explicit imports
- Implement `OnInit` and `OnDestroy` for lifecycle management
- Use **reactive patterns** with RxJS observables
- Manage subscriptions properly to prevent memory leaks
- Keep components focused on presentation logic

5.3 Service Patterns

- Use `@Injectable({ providedIn: 'root' })` for singleton services
- Implement state management with **BehaviorSubjects**
- Handle HTTP errors consistently across all services
- Use environment variables for API URLs
- Log service operations with consistent formatting

5.4 Template Standards

- Use **semantic HTML** with proper accessibility attributes
- Implement **translation pipes** for all user-facing text
- Use **async pipe** for observables in templates
- Apply **TailwindCSS classes** for styling
- Prefer **Flowbite components** over Ionic components

5.5 Routing and Security

- Use **functional route guards** (Angular 16+ style)
- Implement **lazy loading** for all page components
- Handle authentication and authorization at route level
- Use **HTTP interceptors** for JWT token management
- Implement automatic token refresh mechanisms

6. Database Standards

6.1 Schema Design

- **Schema**: All tables use firefighter schema
- **Naming**: snake_case for tables and columns (e.g., user_id, created_at)
- **Primary Keys**: Use BIGSERIAL for auto-incrementing IDs
- **Foreign Keys**: Explicit naming with fk_ prefix
- **Timestamps**: Use timestamp with time zone for all datetime fields

6.2 JPA Entity Standards

- Map entities to firefighter schema using `@Table(schema = "firefighter")`
- Use proper column definitions for PostgreSQL compatibility
- Implement default constructors and proper getters/setters
- Use `ZonedDateTime` for timestamp fields
- Follow Spring Data JPA naming conventions for repository methods

6.3 Repository Patterns

- Extend `JpaRepository<Entity, ID>` for basic CRUD operations
- Use method naming conventions for automatic query generation
- Implement custom queries with `@Query` annotation
- Use `@Param` for named parameters in custom queries
- Prefer JPQL over native SQL queries when possible

7. Security Standards

7.1 Authentication and Authorization

- **Firebase Authentication**: Primary authentication provider
- **JWT Tokens**: Custom JWT tokens for API authorization
- **Role-Based Access**: Admin and regular user roles
- **Token Refresh**: Automatic token refresh mechanism
- **CORS Configuration**: Proper cross-origin resource sharing setup

7.2 Security Best Practices

- Validate all input parameters using Bean Validation
- Use parameterized queries to prevent SQL injection
- Never log sensitive information (passwords, tokens)
- Implement proper error handling without exposing stack traces
- Use HTTPS in production environments
- Implement rate limiting for API endpoints

8. Testing Standards

8.1 Backend Testing

- **Unit Tests**: 80%+ line coverage using JUnit 5 and Mockito
- **Integration Tests**: Test all API endpoints with `@SpringBootTest`
- **Test Naming**: `methodName_WhenCondition_ShouldExpectedBehavior`
- **Test Data**: Use `@TestPropertySource` for test-specific configurations
- **Mocking**: Mock external dependencies and services

8.2 Frontend Testing

- **Unit Tests**: Test components and services using Jasmine and Karma
- **Component Tests**: Test component behavior and template rendering
- **Service Tests**: Mock HTTP calls using `HttpClientTestingModule`
- **E2E Tests**: Critical user journeys using Protractor or Cypress
- **Coverage**: Maintain 70%+ test coverage for critical business logic

9. Code Quality

9.1 Linting and Formatting

- **Java**: Use Google Java Style Guide with 4-space indentation
- **TypeScript**: ESLint with Angular rules, 2-space indentation
- **Auto-formatting**: Prettier for Angular, Google Format for Java
- **Line Length**: 120 characters for Java, 100 for TypeScript

9.2 Code Analysis Tools

- **Backend**: JaCoCo for coverage, Maven Surefire for testing
- **Frontend**: ESLint for linting, Karma for test execution
- **Security**: OWASP Dependency Check, npm audit
- **CI Integration**: All checks must pass before merge

10. Git Workflow

10.1 Branching Strategy

- **main**: Production-ready code (protected)
- **develop**: Integration branch for active development
- **feature/[name]**: Feature development branches
- **bugfix/[name]**: Bug fix branches
- **hotfix/[name]**: Emergency production fixes

11. Conclusion

These coding standards ensure the FireFighter Access Management Platform maintains high quality, security, and maintainability. All team members must follow these guidelines and contribute to their continuous improvement.