SAMFMS

Coding Standards

Table of Contents

1. Introduction

This document establishes coding standards for the Safe And Modular Fleet Management System (SAMFMS), a microservices-based application built with Python, FastAPI, MongoDB, and Docker. These standards ensure consistency, maintainability, and readability across all services.

Reference: Based on Chapter 18 software engineering principles and adapted for SAMFMS microservices architecture.

2. General Principles

Core Principles

- **Readability**: Self-documenting code with clear naming
- **Consistency**: Follow established patterns throughout codebase
- **Modularity**: Small, focused functions and classes with single responsibility
- **DRY**: Avoid code duplication
- **SOLID**: Follow object-oriented design principles

Development Philosophy

- **Microservices-First**: Independently deployable services
- **API-First**: Design APIs before implementation
- **Security by Design**: Built-in security considerations
- **Observability**: Comprehensive logging and monitoring

3. Python Code Style

3.1 PEP 8 Compliance

Follow PEP 8 with these specifications:

- **Line length**: 88 characters (Black formatter)
- **Indentation**: 4 spaces (no tabs)

- **Type hints**: Required for all functions
- **Docstrings**: Required for all modules, classes, and public methods

3.2 Import Organization

```
# 1. Standard library
 import os, logging, asyncio
 from datetime import datetime
 from typing import Dict, List, Optional, Any

 # 2. Third-party
 from fastapi import FastAPI, HTTPException, Depends
 from pydantic import BaseModel, Field
 import motor.motor_asyncio

 # 3. Local application
 from config.settings import settings
 from services.auth_service import AuthService
 from logging_config import get_logger
```

3.3 Function Structure Example

```
async def process_vehicle_data(
        vehicle_id: str,
        location_data: Dict[str, Any],
        timestamp: Optional[datetime] = None
 ) -> Dict[str, Any]:
        """
        Process incoming vehicle location data.

        Args:
      vehicle_id: Unique vehicle identifier
      location_data: GPS coordinates and metadata
      timestamp: Recording time (defaults to now)

        Returns:
        Processed location information

        Raises:
      ValueError: If vehicle_id invalid
      HTTPException: If processing fails
    """
        if not vehicle_id:
        raise ValueError("Vehicle ID cannot be empty")

        timestamp = timestamp or datetime.utcnow()
```

```
try:
return {
"vehicle_id": vehicle_id,
"coordinates": location_data.get("coordinates"),
"timestamp": timestamp.isoformat(),
"processed_at": datetime.utcnow().isoformat()
}
except Exception as e:
logger.error(f"Failed to process vehicle data: {e}")
raise HTTPException(status_code=500, detail="Processing failed")
```

4. Project File Structure

4.1 Repository Structure

```
SAMFMS/
├── Core/              # Central API gateway
├── Sblocks/           # Service blocks (business logic)
│   ├── gps/           # GPS tracking service
│   ├── management/        # Vehicle management
│   ├── security/      # Authentication & authorization
│   └── utilities/     # Utility services
├── Dblocks/           # Data blocks (data access)
│   ├── users/, vehicles/, gps/
├── Frontend/          # React application
├── docs/              # Documentation
└── docker-compose.yml     # Service orchestration
```

4.2 Standard Service Structure

Each microservice must follow this layout:

```
service_name/
├── main.py            # FastAPI entry point
├── requirements.txt         # Dependencies
├── Dockerfile         # Container config
├── config/            # Settings and database
├── models/            # API and database models
├── routes/            # API endpoints
├── services/          # Business logic
├── middleware/        # Logging and security
├── utils/             # Helper functions
└── tests/             # Unit and integration tests
        ├── unit/
        └── integration/
```

5. Naming Conventions

5.1 General Rules

- **Files/Directories**: snake_case (e.g., auth_service.py, vehicle_management/)
- **Classes**: PascalCase (e.g., VehicleService, AuthenticationError)
- **Functions/Variables**: snake_case (e.g., get_vehicle_location, user_id)
- **Constants**: UPPER_SNAKE_CASE (e.g., MAX_RETRY_ATTEMPTS, DEFAULT_TIMEOUT)
- **Private**: Leading underscore (e.g., _connection, _validate_input)

5.2 API and Database

```
# API endpoints: kebab-case
 @app.get("/api/vehicle-locations/{vehicle_id}")
 @app.post("/api/user-management/create-user")

 # Database collections: snake_case
 users_collection = db.users
 vehicle_locations_collection = db.vehicle_locations

 # Document fields: snake_case
 user_doc = {
         "user_id": "12345",
         "full_name": "John Doe",
         "created_at": datetime.utcnow()
 }

 # Pydantic models: snake_case with camelCase aliases for frontend
 class VehicleResponse(BaseModel):
         vehicle_id: str = Field(alias="vehicleId")
         license_plate: str = Field(alias="licensePlate")
```

6. Documentation Standards

6.1 Module Headers

```
"""
 GPS Service Module

 Provides GPS tracking and location management for SAMFMS.

 Features: Real-time tracking, geofence monitoring, location history
 Author: SAMFMS Development Team
 Version: 1.0.0
 """
```

6.2 Class and Function Documentation

```
class VehicleLocationService:
        """
```

```
    Manages vehicle location data and GPS tracking.

    Handles GPS processing, location history, geofence detection,
    and real-time updates via WebSocket.
"""

    async def calculate_distance(
    self, point1: Dict[str, float], point2: Dict[str, float]
    ) -> float:
    """
    Calculate distance between geographic points using Haversine formula.

    Args:
     point1: Coordinates with 'lat' and 'lng' keys
     point2: Coordinates with 'lat' and 'lng' keys

    Returns:
     Distance in kilometers

    Raises:
     ValueError: If coordinates are invalid
  """
    pass
```

7. Error Handling

7.1 Exception Hierarchy

```
class SAMFMSException(Exception):
    """Base exception for SAMFMS application errors."""
    pass

 class AuthenticationError(SAMFMSException):
    """Authentication failure."""
    pass

 class VehicleNotFoundError(SAMFMSException):
    """Vehicle not found."""
    pass
```

7.2 Error Handling Pattern

```
async def get_vehicle_by_id(vehicle_id: str) -> Dict[str, Any]:
    """Retrieve vehicle with proper error handling."""

    if not vehicle_id.strip():
    raise HTTPException(status_code=400, detail="Vehicle ID required")
```

```
        try:
        vehicle = await vehicles_collection.find_one({"_id": vehicle_id})
        if not vehicle:
        raise HTTPException(status_code=404, detail="Vehicle not found")
        return vehicle

        except HTTPException:
        raise
        except Exception as e:
        logger.error(f"Database error retrieving vehicle {vehicle_id}: {e}")
        raise HTTPException(status_code=500, detail="Retrieval failed")
```

7.3 Logging Standards

```
logger = get_logger(__name__)

 # Usage by level:
 logger.debug(f"Processing: {data}")           # Detailed diagnostics
 logger.info(f"Vehicle {id} updated")          # General information
 logger.warning(f"Vehicle {id} outside fence") # Unexpected but recoverable
 logger.error(f"Database connection failed")   # Serious problems
 logger.critical(f"Service startup failed")    # System-threatening errors

 # Structured logging with context
 logger.info("Authentication successful", extra={
         "user_id": user_id, "ip": request.client.host
 })
```

8. Testing Standards

8.1 Test Organization

```
tests/
 ├── unit/          # Individual functions/classes
 ├── integration/   # API endpoints and workflows
 └── fixtures/      # Test data and mocks
```

8.2 Test Structure Example

```
import pytest
 from unittest.mock import Mock
 from fastapi.testclient import TestClient

 class TestVehicleService:
         """Test suite for VehicleService."""

         @pytest.fixture
         def vehicle_service(self):
         return VehicleService(Mock())
```

```
    async def test_get_vehicle_success(self, vehicle_service):
    """Test successful vehicle retrieval."""
    # Arrange
  vehicle_service.db.find_one.return_value = {
    "_id": "123", "license_plate": "ABC123"
    }

    # Act
    result = await vehicle_service.get_vehicle_by_id("123")

    # Assert
    assert result["license_plate"] == "ABC123"

    async def test_get_vehicle_not_found(self, vehicle_service):
    """Test vehicle not found scenario."""
  vehicle_service.db.find_one.return_value = None

    with pytest.raises(HTTPException) as exc:
    await vehicle_service.get_vehicle_by_id("missing")

    assert exc.value.status_code == 404
```

9. Database and API Conventions

9.1 MongoDB Document Structure

```
# User document
{
        "_id": ObjectId(),
        "user_id": str(uuid.uuid4()),
        "email": "user@example.com",
        "full_name": "John Doe",
        "role": "driver",
        "permissions": ["view_vehicles", "update_location"],
        "profile": {"phone": "+27123456789"},
        "metadata": {
        "created_at": datetime.utcnow(),
        "updated_at": datetime.utcnow(),
        "is_active": True
        }
}

# Vehicle document
{
        "_id": ObjectId(),
        "vehicle_id": str(uuid.uuid4()),
        "license_plate": "ABC123GP",
        "make": "Toyota", "model": "Hilux", "year": 2023,
        "current_status": {
        "status": "active",
        "location": {"coordinates": [-26.2041, 28.0473]},
        "driver_id": "driver_uuid"
        },
        "metadata": {"created_at": datetime.utcnow(), "is_active": True}
}
```

9.2 Pydantic Models

```
class VehicleStatus(str, Enum):
        ACTIVE = "active"
        MAINTENANCE = "maintenance"
        OUT_OF_SERVICE = "out_of_service"

class VehicleBase(BaseModel):
        license_plate: str = Field(..., min_length=6, max_length=10)
        make: str = Field(..., min_length=1, max_length=50)
        year: int = Field(..., ge=1900, le=2030)

        @validator('license_plate')
        def validate_license_plate(cls, v):
        pattern = r'^[A-Z]{2,3}[0-9]{3,4}[A-Z]{0,2}$'
        if not re.match(pattern, v.upper()):
        raise ValueError('Invalid SA license plate format')
        return v.upper()

class VehicleResponse(VehicleBase):
        vehicle_id: str
        status: VehicleStatus
        created_at: datetime
```

9.3 API Response Formats

```
# Success response
{
        "success": true,
        "data": {"vehicle_id": "123", "license_plate": "ABC123"},
        "message": "Vehicle retrieved successfully",
        "timestamp": "2024-01-15T10:30:00Z"
}

# Error response
```

```
{
        "success": false,
        "error": {
        "code": "VEHICLE_NOT_FOUND",
        "message": "Vehicle not found",
        "details": {"requested_id": "123"}
        },
        "timestamp": "2024-01-15T10:30:00Z"
}
```

10. Configuration Management

Environment Variables

```
# config/settings.py
 class Settings:
        """Application settings from environment variables."""

        # Database
        MONGODB_URL: str = os.getenv("MONGODB_URL", "mongodb://localhost:27017")
        DATABASE_NAME: str = os.getenv("DATABASE_NAME", "samfms")

        # Message Queue
        RABBITMQ_URL: str = os.getenv("RABBITMQ_URL",
"amqp://guest:guest@localhost:5672/")
        REDIS_URL: str = os.getenv("REDIS_URL", "redis://localhost:6379")

        # Authentication
        JWT_SECRET_KEY: str = os.getenv("JWT_SECRET_KEY", "your-secret-key")
        ACCESS_TOKEN_EXPIRE_MINUTES: int =
int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "30"))

        # Service Configuration
        SERVICE_NAME: str = os.getenv("SERVICE_NAME", "samfms-core")
        API_PORT: int = int(os.getenv("API_PORT", "8000"))
        DEBUG: bool = os.getenv("DEBUG", "False").lower() == "true"

 settings = Settings()
```

11. Security Standards

Authentication and Authorization

```
from fastapi import Depends, HTTPException, status
 from fastapi.security import HTTPBearer
 import jwt

 security = HTTPBearer()

 async def get_current_user(credentials = Depends(security)):
        """Extract and validate user from JWT token."""
        try:
        payload = jwt.decode(credentials.credentials, settings.JWT_SECRET_KEY,
                        algorithms=[settings.JWT_ALGORITHM])
        user_id = payload.get("user_id")
        if not user_id:
        raise HTTPException(status_code=401, detail="Invalid token")
        return user_id
        except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token expired")
        except jwt.JWTError:
        raise HTTPException(status_code=401, detail="Invalid credentials")

 def require_permission(permission: str):
        """Decorator requiring specific permission."""
        def decorator(func):
        async def wrapper(*args, user_id: str = Depends(get_current_user), **kwargs):
```

```
    if permission not in await get_user_permissions(user_id):
        raise HTTPException(status_code=403, detail=f"Permission required:
{permission}")
    return await func(*args, user_id=user_id, **kwargs)
    return wrapper
    return decorator
```

Input Validation

```
class VehicleUpdateRequest(BaseModel):
    license_plate: Optional[str] = Field(None, min_length=6, max_length=10)

    @validator('license_plate')
    def validate_license_plate(cls, v):
    if v:
    v = re.sub(r'[^A-Z0-9]', '', v.upper())  # Sanitize
    if not re.match(r'^[A-Z]{2,3}[0-9]{3,4}[A-Z]{0,2}$', v):
        raise ValueError('Invalid license plate format')
    return v
```

12. Performance Guidelines

Database Optimization

```
# Create indexes for frequently queried fields
 await collection.create_index("user_id")
 await collection.create_index([("location.coordinates", "2dsphere")])  # Geospatial
 await collection.create_index([("created_at", -1)])

 # Use projections to limit data transfer
 async def get_vehicle_summary(vehicle_id: str):
    return await vehicles_collection.find_one(
    {"vehicle_id": vehicle_id},
    {"vehicle_id": 1, "license_plate": 1, "status": 1, "_id": 0}
    )

 # Aggregation for complex queries
 async def get_vehicle_stats():
    pipeline = [
    {"$match": {"is_active": True}},
    {"$group": {"_id": "$status", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}}
    ]
    return await vehicles_collection.aggregate(pipeline).to_list(None)
```

Caching Strategy

```
class CacheManager:
    def __init__(self, redis_client):
    self.redis = redis_client
    self.default_ttl = 300
```

```
async def get_cached_data(self, key: str):
try:
cached = self.redis.get(key)
return json.loads(cached) if cached else None
except Exception:
return None

async def set_cached_data(self, key: str, data: Dict, ttl: int = None):
try:
self.redis.setex(key, ttl or self.default_ttl, json.dumps(data, default=str))
return True
except Exception:
return False
```

Async Best Practices

```
# Concurrent operations
 async def update_multiple_vehicles(updates: List[Dict]):
     tasks = [update_single_vehicle(update) for update in updates]
     return await asyncio.gather(*tasks, return_exceptions=True)

 # Connection management
 @asynccontextmanager
 async def get_database_connection():
     connection = None
     try:
     connection = await AsyncIOMotorClient(settings.MONGODB_URL)
     yield connection[settings.DATABASE_NAME]
     finally:
     if connection:
     connection.close()
```

13. Version Control Standards

Branch Naming

- **Features**: feature/vehicle-tracking-improvements
- **Bug fixes**: bugfix/authentication-token-expiry
- **Hot fixes**: hotfix/critical-security-patch
- **Releases**: release/v1.2.0

Commit Messages

```
type(scope): short description

 Detailed description if needed
 - List specific changes
 - Include breaking changes
 - Reference issue numbers
```

```
Closes #123
```

Examples:

```
feat(auth): add JWT refresh functionality
 - Implement refresh endpoint
 - Add token blacklisting
 - Update middleware
 Closes #45

 fix(vehicles): resolve location race condition
 - Add database transactions
 - Implement error handling
 - Add concurrent update tests
 Closes #67
```

Code Review Guidelines

- **Required**: All code reviewed by ≥1 developer
- **Focus Areas**: Quality, security, performance, tests, documentation
- **Pull Requests**: Descriptive titles, link issues, include screenshots for UI
- **Standards**: Compliance with this document

Document Version: 1.0
Last Updated: July 2025
Next Review: October 2025