**1. Objective**

This document outlines the requirements for creating reusable Spring Boot templates that will accelerate the development of spring boot based microservice development. This accelerator is designed to simplify the implementation of microservice for policy submission, claim notification etc., while adhering to clean architecture and 12-factor methodology principles to ensure maintainable, scalable, and cloud-native solutions.

The goal is to reduce development time and effort by providing a robust foundation that can be reused across multiple projects.

**2. Scope and Use Cases**

The scope of this accelerator will cover the following primary insurance use case:

1. **Policy Submission**
   * A system where users can submit new policy applications.
   * Integration with policy management systems (e.g., Guidewire, Socotra).
   * Support for validating customer data, underwriting rules, and policy binding.

**3. Architecture Overview**

The template will follow **clean architecture** principles, ensuring a separation of concerns and making the codebase easy to test, maintain, and scale. The architecture will be divided into four primary layers:

1. **Domain Layer**:
   * Contains business logic and domain entities.
   * Should be agnostic of frameworks and technology.
2. **Application Layer**:
   * Coordinates use cases and business rules.
   * Application services (e.g., handling a policy submission).
3. **Interface Layer (API Layer)**:
   * Exposes APIs for external communication (for the scope of our requirement we choose REST).
   * Facilitates input validation, authentication, and authorization.
4. **Infrastructure Layer**:
   * Handles technical concerns like persistence, messaging (e.g., Kafka, RabbitMQ), and integrations with third-party systems.

**12-Factor Methodology**: The following 12-factor principles will be incorporated into the templates to ensure cloud-native and scalable applications:

* **Codebase**: Version-controlled, one codebase per service.
* **Dependencies**: Use dependency management via Maven or Gradle.
* **Config**: Store environment-specific configurations (e.g., AWS keys, DB URIs) in environment variables.
* **Backing services**: Treat external services (databases, queues) as attached resources.
* **Build, release, run**: Separate build and run stages with Docker and CI/CD pipelines.
* **Processes**: Stateless and share-nothing architecture.
* **Port binding**: Expose services via port binding (HTTP, HTTPS).
* **Concurrency**: Scale out via process concurrency (auto-scaling on AWS).
* **Disposability**: Ensure fast startup and graceful shutdown.
* **Dev/prod parity**: Use similar environments (Docker, localstack for AWS emulation).
* **Logs**: Treat logs as event streams, integrate with logging solutions (CloudWatch, ELK).
* **Admin processes**: Use task runners for admin/maintenance tasks.

**4. Technical Design**

**Template Structure**

* **Base package structure**:
  + com.vm.ec.accelerator (Root package)
  + /domain
  + /application
  + /api
  + /infrastructure
* Include standardized configurations for:
  + **API structure**: Using OpenAPI/Swagger for API documentation.
  + **Spring Profiles**: For environment-specific configurations (dev, prod, test).
  + **Error Handling**: Common error-handling mechanism using global exception handlers.
  + **Security**: Integrated with Spring Security for basic authentication and authorization.

**Layer Breakdown**

* **Domain Layer**:
  + Defines core domain entities (e.g., Policy) and business rules.
* **Application Layer**:
  + Handles application services (e.g., PolicyService) to manage use cases.
* **Interface Layer (API)**:
  + Contains controllers to expose REST endpoints (e.g., POST /policy, POST /claim).
  + Data Transfer Objects (DTOs) will be used to transfer data between layers.
* **Infrastructure Layer**:
  + Handles external services like databases, third-party APIs, and messaging.
  + Integration with AWS SDKs for S3, RDS, SQS, etc.

**5. Non-Functional Requirements**

* **Scalability**: Each service must be stateless and horizontally scalable using any cloud service, example AWS ECS or AWS EKS.
* **Security**: The templates must include provisions for secure communication (HTTPS), data encryption, and identity management (JWT, OAuth).
* **Performance**: The applications should support high throughput with minimal latency, ensuring response times under 600ms.
* **Monitoring & Logging**: Integration with services like AWS CloudWatch for centralized logging and metrics, providing a clear view of performance and system health.

**6. Tools & Frameworks**

* **Spring Boot 3.x**
  + Spring Data JPA for persistence.
  + Spring Security for authentication/authorization.
  + Spring Cloud for service discovery and configuration.
* **AWS SDK** for:
  + S3: Document storage (for policy or claim-related files).
  + SQS: Messaging between services.
  + RDS: Data storage (MySQL/PostgreSQL).
* **JUnit & Mockito** for unit testing.
* **WireMock** for API mocking.
* **SonarQube** for code quality checks.

**7. Developer Guidelines**

* **Usage of Templates**:
  + Clone or fork the base repository for each new project.
  + Modify the domain and application logic as per the specific use case (policy or claims).
  + Use pre-built patterns for common operations like CRUD operations, external API calls, etc.
  + Follow the 12-factor app principles during development to ensure consistent cloud-native behavior.
* **Best Practices**:
  + Ensure that the domain logic remains isolated from the framework-specific code.
  + Always write unit tests for application services and business logic.
  + Use Spring Boot profiles to manage configurations for different environments (e.g., dev, staging, prod).

**8. Appendix**

* **Reference Diagrams**:
  + Include diagrams illustrating the clean architecture pattern applied to the insurance use cases (policy submission, claim notification).