Business Interruption Coverage

# Introduction

Hi (Greeting : GM, GA, GE) . Thank you for giving me this opportunity. This is Siddheshwar Jadhav. I have completed my BE from Pune University. I have been working in (Talentifiers Consulting pvt.ltd.) for the last 4.2 years as a **Backend Developer**.I am involved in 3 projects for several domains like **Insurance**, **Logistics and Transportation**.

Currently I have been working on the Policy Administration Software for the client (GuideOne Insurance).

In this, we have been implementing the Business Interruption Coverage Module for the last 15 months.

Technically I have used Java, Java 8 features, Spring Boot, microservices for developing the project. MySQL Database for Hibernate implementation.

I am having very good experience with Java & J2EE technologies, Unit Testing for application, 1.2 yrs of experience with Microservices architecture implementation & having good knowledge in application deployment process using Jenkins CI/CD pipeline and docker.

Junit and Mockito for Unit Testing, and other Tools like GitHub and for CI/CD we used Jenkins.

# Project Details:

In this project we implemented a new insurance module i.e. Business Interruption Coverage. Business interruption insurance is not sold as a separate policy but is either added to a property/casualty policy or included in a comprehensive package policy as an add-on or rider.

Business interruption insurance is a type of insurance that covers the loss of income that a business suffers after a disaster. The covered perils for business interruption insurance are generally theft, wind, fire, lightning and falling objects.

Covered operating expenses may include:

* The revenue your business would make if it were open for business
* Monthly mortgage, lease and rent payments for the business space
* Loan payments for the business
* Taxes
* Payroll
* Relocation costs if you have to move to a temporary location
* Training costs for employees to learn new equipment

To implement this solution, we utilised spring boot framework to create a structure architecture with Controllers, Services and Repositories, stored essential data into our database, perform necessary business logic and retrieve information as per application’s requirement.

For this project we created various entities like:

Property Coverage, Insured Property, Business Income Coverage, Additional Expenses, Employee Wages, Civil Authority Coverage, Loan Entity Coverage, Employee Theft Coverage, Contingent Business Coverage etc.

We created below list of Endpoints:

Policy Service API

* /api/policies: Endpoint to manage policies.
* /api/policies/{policyId}: Endpoint to get, update, or delete a specific policy.

Coverage Service API:

* /api/coverages: Endpoint to manage coverages.
* /api/coverages/{coverageId}: Endpoint to get, update, or delete a specific coverage.

Calculation Service API:

* Handles business logic related to calculating coverage amounts and costs.
* May be utilized by other services to perform calculations.
* @PostMapping("property/coverage/calculate")
* @PostMapping(“business-income/coverage/calculate”)
* @PostMapping("/calculate-temporary-accommodation-expenses")
* @PostMapping("/calculate-shipping-cost-expenses")
* @PostMapping("/calculate-extra-labor-expenses")
* GET /employees/{employeeId}/wages: Retrieve the wage details for a specific employee.
* POST /employees/{employeeId}/wages: Update the wage details for a specific employee.
* GET /employees/{employeeId}/wages/history: Retrieve the wage history for a specific employee, including past wage changes and adjustments.
* GET /employees/{employeeId}/wages/coverage: Check if employee wages are covered during a business interruption period.
* POST /employees/{employeeId}/wages/claim: File a claim for employee wages during a business interruption.
* GET /employees/wages/summary: Retrieve a summary of employee wages, including total wages paid, pending claims, and coverage status for all employees.
* POST /api/civil-authority-coverage/calculate-loss
* @PostMapping("/calculate-loan-payment")

Programming Logic:

First, *we* calculated Property coverage:

*Insured Amount = Property Value × Coverage Percentage*

Where:

Property Value: The estimated value of the property, including the cost of construction, market value, and any additional features or upgrades.

Coverage Percentage: A percentage typically determined based on factors such as the property's location, construction materials, age, and additional coverage needs.

*Maximum Covered Loss for Dwelling = Coverage Limit – Deductible*

*Maximum Covered Loss for Personal Property = Coverage Limit – Deductible*

**Property Value Calculation:**

1. Land Value Calculation:

*Land Value = (Base Land Value + Location Premium) × Size Adjustment*

1. Building Value Calculation:

*Building Value = (Base Building Value + Age Adjustment + Condition Adjustment) × Size Adjustment*

1. Site Improvements Cal:

*Site Improvements Value = ∑i=1n (Base Improvement Value(i) × Condition Adjustment(i))*

*Total Property Value = Land Value + Building Value + Site Improvements Value*

The actual calculation had more complex considerations and factors, including:

* Specific underwriting guidelines and methodologies used by the insurance provider.
* Adjustments for regional construction costs, market conditions, and regulatory requirements.
* Policy limits, deductibles, and optional endorsements that impact coverage and premiums.
* Individual property characteristics, such as size, age, construction materials, and special features.
* Risk assessment factors, such as location, occupancy type, loss history, and risk mitigation measures.

**Business Income Coverage:**

# Roles & Responsibilities:

I play a crucial role in designing, implementing, and maintaining the backend of our Business Interruption Coverage Module. My responsibilities involve creating the APIs, business logic, data management, and ensuring the security and performance of the server-side components

1. Requirement gathering and analysis
2. Developing and integrating the RESTful APIs and ensuring communication between diff. services and database using SB & spring data JPA
3. Implements core java concepts like oops, collection fw, multithreading, design patterns
4. Using advanced java concepts for developing application code
5. Used entity, controller, repositories to handle data from db
6. Writing unit tests and collaborating with QA engineers to ensure comprehensive test coverage and timely bug fixes
7. Collaborated with cross-functional teams to deliver high-quality software solutions on time.
8. And deliver code by using Git and documentation.

# Project Architecture:

Architecture for your Business Interruption Coverage Module using microservices and Spring Boot:

1. **Gateway Service**:
   * Acts as the entry point for external clients and routes requests to appropriate microservices.
   * Utilizes Spring Cloud Gateway for routing and load balancing.
2. **Authentication and Authorization Service**:
   * Handles user authentication and authorization.
   * Utilizes Spring Security for authentication and JWT tokens for authorization.
3. **Config Service**:
   * Centralized configuration management using Spring Cloud Config Server.
   * Stores configurations for all microservices and allows for dynamic updates without requiring redeployment.
4. **Discovery Service (Eureka)**:
   * Service registry for registering and discovering microservices.
   * Microservices register themselves with Eureka server upon startup, allowing other services to discover and communicate with them.
5. **API Gateway**:
   * Exposes RESTful APIs for the Business Interruption Coverage Module.
   * Aggregates requests from clients and forwards them to the appropriate microservices.
   * Implements rate limiting, request logging, and other cross-cutting concerns.
6. **Business Interruption Service**:
   * Core service responsible for managing business interruption coverage.
   * Implements CRUD operations for policies, claims, and other related entities.
   * Utilizes Spring Boot for building RESTful APIs and Spring Data JPA for database interactions.

**Layered Architecture**:

* + Divide your application into layers, each responsible for a specific set of tasks. Common layers include:
    1. Controller Layer: Handles incoming HTTP requests and delegates processing to other layers.
    2. Service Layer: Contains business logic and performs operations such as data validation, calculation, and transformation.
    3. Repository Layer: Interacts with the database or external data sources to perform CRUD operations.

1. **Additional Microservices**:
   * **Taxes Service**: Manages taxation information related to business interruption coverage.
   * **Expenses Service**: Handles additional expenses such as rent, mortgage, lease, shipping charges, etc.
   * **Employee Service**: Manages employee-related data including wages, theft coverage, etc.
   * **Loan Service**: Manages loan payments and related information.
   * **Civil Authorities Service**: Handles civil authority charges and related notifications.
   * **Contingent Business Service**: Manages contingent business interruption coverage.
2. **Domain Models**:
   * Define domain models to represent the entities and data structures used in the application. These models should be simple POJOs (Plain Old Java Objects) with properties representing the attributes of the entities.
3. **Exception Handling**:
   * Implement exception handling to gracefully handle errors and exceptions that may occur during request processing. Use exception handling mechanisms provided by Spring Boot, such as **@ControllerAdvice** and **@ExceptionHandler**.
4. **Event Bus (Kafka or RabbitMQ)**:
   * Facilitates asynchronous communication and event-driven architecture between microservices.
   * Enables decoupling and scalability by allowing services to publish and subscribe to events.
5. **Database**:
   * Utilizes a relational database like MySQL or PostgreSQL for storing persistent data.
   * Each microservice may have its own database schema, promoting loose coupling.
6. **Monitoring and Logging**:
   * Implement centralized logging using tools like ELK stack (Elasticsearch, Logstash, Kibana) or Splunk.
   * Utilize Spring Boot Actuator for monitoring and managing microservices.
7. **Containerization**:
   * Dockerize each microservice for easy deployment and scalability.
   * Orchestrate containers using Kubernetes for automated scaling and management.
8. **Testing**:
   * Implement unit tests, integration tests, and end-to-end tests for each microservice.
   * Utilize tools like JUnit, Mockito, and Spring Boot Test for testing.
9. **CI/CD Pipeline**:
   * Set up a continuous integration and continuous deployment pipeline using Jenkins, GitLab CI, or similar tools.
   * Automate build, test, and deployment processes to ensure smooth delivery of updates.
10. **Documentation**:
    * Maintain comprehensive documentation for each microservice, API endpoints, and deployment procedures using tools like Swagger or OpenAPI.
11. **Security**:
    * Implement HTTPS for secure communication between services.
    * Secure sensitive data using encryption and proper access controls.
    * Regularly update dependencies to patch security vulnerabilities.

**How many modules are there in your project and what are those?**

**How many environments are there in your application?**

Local

Dev

SIT

UAT

Pre-Production

Production

# Challenges Faced

Business Interruption Coverage module:

# Technical

## OOPS Concepts & SOLID Principles

In a project like the Business Interruption Policy Module, Object-Oriented Programming (OOP) principles play a crucial role in designing maintainable, extensible, and efficient code. Here are some OOP principles that are likely to be implemented in this project, along with detailed examples:

Encapsulation:

Encapsulation involves bundling data (attributes) and methods (functions) that operate on that data within a single unit, typically a class.

Example: In the Policy class, encapsulation ensures that data such as policy ID, coverage start date, coverage end date, and premium are encapsulated within the class, and access to this data is controlled through getter and setter methods.

java

Copy code

public class Policy {

private String policyId;

private LocalDate coverageStartDate;

private LocalDate coverageEndDate;

private double premium;

// Getter and setter methods

public String getPolicyId() {

return policyId;

}

public void setPolicyId(String policyId) {

this.policyId = policyId;

}

// Other methods...

}

Abstraction:

Abstraction involves hiding the complex implementation details of a class and exposing only the essential features to the outside world.

Example: The PolicyService class abstracts away the internal logic for managing policies and exposes high-level methods for creating, updating, and retrieving policies.

java

Copy code

public class PolicyService {

public void createPolicy(Policy policy) {

// Implementation details...

}

public Policy getPolicyById(String policyId) {

// Implementation details...

return policy;

}

// Other methods...

}

Inheritance:

Inheritance allows a class (subclass) to inherit properties and behavior from another class (superclass), promoting code reuse and facilitating polymorphism.

Example: A BusinessInterruptionPolicy class can inherit from a Policy class, inheriting common attributes and methods while adding specific attributes and methods related to business interruption coverage.

java

Copy code

public class BusinessInterruptionPolicy extends Policy {

private String interruptionCause;

// Additional attributes and methods specific to business interruption coverage

}

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and extensibility in method invocation.

Example: A calculatePremium() method can be polymorphic, with different implementations in subclasses such as BusinessInterruptionPolicy and PropertyDamagePolicy.

java

Copy code

public abstract class Policy {

// Common attributes and methods...

public abstract double calculatePremium();

}

public class BusinessInterruptionPolicy extends Policy {

// Implementation of calculatePremium() specific to business interruption coverage

}

public class PropertyDamagePolicy extends Policy {

// Implementation of calculatePremium() specific to property damage coverage

}

Composition:

Composition involves creating complex objects by combining simpler objects, allowing for better code organization and reusability.

Example: The Policy class may have a composition relationship with an InsuredBusiness class, where a policy is composed of an insured business entity.

java

Copy code

public class Policy {

private InsuredBusiness insuredBusiness;

// Other attributes and methods...

public Policy(InsuredBusiness insuredBusiness) {

this.insuredBusiness = insuredBusiness;

}

}

These OOP principles contribute to the overall design and structure of the project, promoting code modularity, maintainability, and scalability. By adhering to these principles, the Business Interruption Policy Module can be developed in a way that facilitates easy extension, modification, and collaboration among developers working on different parts of the system.

User

what other oops principles?

ChatGPT

Certainly! Here are a few more Object-Oriented Programming (OOP) principles commonly applied in software development:

Single Responsibility Principle (SRP):

SRP states that a class should have only one reason to change, meaning it should have only one responsibility or task.

Example: In the Business Interruption Policy Module, the PolicyService class should be responsible only for managing policies, not for handling claims or calculating premiums.

java

Copy code

public class PolicyService {

public void createPolicy(Policy policy) {

// Implementation details...

}

public Policy getPolicyById(String policyId) {

// Implementation details...

return policy;

}

// Other methods related to policy management...

}

Open/Closed Principle (OCP):

OCP states that software entities (classes, modules, functions) should be open for extension but closed for modification. This means that you should be able to extend the behavior of a system without modifying its existing code.

Example: Introducing a new subclass PremiumCalculator that extends an abstract Calculator class allows for adding new premium calculation methods without modifying existing code.

java

Copy code

public abstract class Calculator {

public abstract double calculate();

}

public class PremiumCalculator extends Calculator {

@Override

public double calculate() {

// Implementation of premium calculation...

return premiumAmount;

}

}

Liskov Substitution Principle (LSP):

LSP states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Example: If BusinessInterruptionPolicy is a subclass of Policy, instances of BusinessInterruptionPolicy should be substitutable for instances of Policy without breaking the expected behavior of the system.

java

Copy code

Policy policy = new BusinessInterruptionPolicy(); // Liskov substitution principle applied

Dependency Inversion Principle (DIP):

DIP states that high-level modules should not depend on low-level modules, but both should depend on abstractions. It promotes loose coupling between components by abstracting dependencies.

Example: Instead of directly instantiating a PolicyRepository in the PolicyService, dependency injection is used to inject an interface (PolicyRepository) into the PolicyService, allowing for easier swapping of implementations.

java

Copy code

public class PolicyService {

private PolicyRepository policyRepository;

public PolicyService(PolicyRepository policyRepository) {

this.policyRepository = policyRepository;

}

// Other methods...

}

Interface Segregation Principle (ISP):

ISP states that clients should not be forced to depend on interfaces they do not use. It promotes the idea of small, cohesive interfaces tailored to specific client needs, rather than large, monolithic interfaces.

Example: Instead of having a single monolithic PolicyService interface with all methods related to policy management, separate interfaces (PolicyCreationService, PolicyRetrievalService, etc.) can be defined, each catering to specific client requirements.

java

Copy code

public interface PolicyCreationService {

void createPolicy(Policy policy);

}

public interface PolicyRetrievalService {

Policy getPolicyById(String policyId);

}

// Other interfaces for policy management...

By applying these additional OOP principles, the design and implementation of the Business Interruption Policy Module can be further refined, leading to a more modular, maintainable, and extensible software system.

## Design Pattern

Which design patterns you used & why?

Singleton

Factory

Strategy / Chain of Responsibility

Saga

**In the context of developing the Business Interruption Policy Module within a Policy Administration System (PAS) based on microservices, Spring, and Spring Boot, several design patterns can be beneficial for achieving various objectives such as scalability, maintainability, and flexibility. Here are some design patterns commonly used in such projects and the reasons for their adoption:**

1. **Microservices Architecture:**
   * **Microservices architecture patterns promote the decomposition of a complex system into smaller, loosely coupled services that can be developed, deployed, and scaled independently.**
   * **Each microservice focuses on a specific business capability, such as policy management, claims processing, or premium calculation, enabling agility and scalability in development and deployment.**
2. **Service Registry and Discovery (Service Discovery Pattern):**
   * **Service registry and discovery patterns, implemented using tools like Netflix Eureka or Spring Cloud Netflix, facilitate dynamic service registration and discovery within a microservices environment.**
   * **Services register themselves with a service registry upon startup, allowing other services to discover and communicate with them.**
   * **This pattern promotes flexibility and resilience in service-to-service communication, especially in dynamically changing environments where service instances may come and go.**
3. **API Gateway (Gateway Pattern):**
   * **An API Gateway acts as a single entry point for external clients to interact with the microservices ecosystem.**
   * **It handles request routing, load balancing, authentication, authorization, and other cross-cutting concerns, simplifying client access to the underlying microservices.**
   * **API Gateway patterns improve security, scalability, and manageability by centralizing common functionalities and enforcing consistent policies across microservices.**
4. **Circuit Breaker (Circuit Breaker Pattern):**
   * **The Circuit Breaker pattern, often implemented using libraries like Netflix Hystrix or resilience4j, helps manage and mitigate failures in distributed systems.**
   * **It monitors the availability and responsiveness of remote services and opens the circuit (stops forwarding requests) if the service fails or becomes unresponsive.**
   * **Circuit breakers prevent cascading failures, improve system resilience, and provide fallback mechanisms to handle failures gracefully.**
5. **Event-Driven Architecture (Event Sourcing, CQRS):**
   * **Event-driven architecture patterns, including Event Sourcing and Command Query Responsibility Segregation (CQRS), facilitate asynchronous communication and decoupling between microservices.**
   * **Events are used to represent state changes within the system, and message brokers like Apache Kafka or RabbitMQ facilitate event publishing, distribution, and consumption.**
   * **Event-driven patterns promote scalability, loose coupling, and real-time data processing, making them suitable for scenarios like claims processing, notifications, and audit logging.**
6. **Repository Pattern:**
   * **The Repository pattern abstracts away the data access layer and provides a standardized interface for accessing and managing data from various data sources.**
   * **It promotes separation of concerns and facilitates testability, maintainability, and flexibility in data access code.**
   * **Spring Data provides support for implementing the Repository pattern with ease, enabling developers to focus on business logic rather than low-level data access details.**
7. **Strategy Pattern:**
   * **The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.**
   * **It allows the algorithm to vary independently from the clients that use it, promoting flexibility and maintainability.**
   * **For example, different premium calculation strategies (e.g., fixed rate, dynamic rate based on risk factors) can be encapsulated as strategies, allowing the system to switch between them dynamically based on business requirements.**

**By leveraging these design patterns, the Business Interruption Policy Module can be architecturally sound, resilient, and adaptable to evolving business needs. Each pattern addresses specific concerns related to microservices architecture, communication, fault tolerance, data access, and flexibility, contributing to the overall effectiveness and success of the project.**

**Top of Form**

## Stream API

In the Business Interruption Policy Module and related services, you can leverage the Java Stream API in various scenarios where you need to process collections of data in a functional and efficient manner. Here are some applications of the Stream API along with detailed coding examples:

Filtering Policies Based on Criteria:

Use Case: Filtering policies based on specific criteria such as coverage start date, coverage end date, or premium amount.

Example:

java

Copy code

import java.util.List;

import java.util.stream.Collectors;

public class PolicyService {

private List<Policy> policies;

// Constructor and other methods...

public List<Policy> getPoliciesByStartDate(LocalDate startDate) {

return policies.stream()

.filter(policy -> policy.getCoverageStartDate().equals(startDate))

.collect(Collectors.toList());

}

}

Calculating Total Premium Amount:

Use Case: Calculating the total premium amount for all policies in the system.

Example:

java

Copy code

import java.util.List;

public class PolicyService {

private List<Policy> policies;

// Constructor and other methods...

public double getTotalPremiumAmount() {

return policies.stream()

.mapToDouble(Policy::getPremium)

.sum();

}

}

Finding Maximum Premium Policy:

Use Case: Identifying the policy with the highest premium amount.

Example:

java

Copy code

import java.util.Optional;

public class PolicyService {

private List<Policy> policies;

// Constructor and other methods...

public Optional<Policy> getPolicyWithMaxPremium() {

return policies.stream()

.max(Comparator.comparingDouble(Policy::getPremium));

}

}

Grouping Policies by Insured Business Type:

Use Case: Grouping policies based on the type of insured business.

Example:

java

Copy code

import java.util.Map;

import java.util.stream.Collectors;

public class PolicyService {

private List<Policy> policies;

// Constructor and other methods...

public Map<String, List<Policy>> groupPoliciesByBusinessType() {

return policies.stream()

.collect(Collectors.groupingBy(Policy::getBusinessType));

}

}

Processing Claims for Policies in Parallel:

Use Case: Processing claims for policies in parallel to improve performance.

Example:

java

Copy code

import java.util.List;

import java.util.concurrent.CompletableFuture;

public class ClaimService {

private List<Policy> policies;

// Constructor and other methods...

public List<Claim> processClaimsInParallel() {

List<CompletableFuture<Claim>> futures = policies.stream()

.map(policy -> CompletableFuture.supplyAsync(() -> processClaim(policy)))

.collect(Collectors.toList());

return futures.stream()

.map(CompletableFuture::join)

.collect(Collectors.toList());

}

private Claim processClaim(Policy policy) {

// Implementation details...

}

}

These are just a few examples of how you can apply the Stream API in the Business Interruption Policy Module and related services to perform common data processing tasks efficiently and concisely. By leveraging streams, you can write clean, expressive code that is easy to understand and maintain, while also taking advantage of modern Java features for functional programming.

## Abstract Class & Intefaces

Abstract classes and interfaces are valuable tools in the design and implementation of a Business Interruption Policy Module. They promote modularity, flexibility, and code reuse by defining common behaviors and contracts. Here's where you might use abstract classes and interfaces in this project and why:

Policy Abstract Class:

Why: An abstract class can define common attributes and methods shared by different types of policies, such as Business Interruption Policy, Property Damage Policy, etc. It provides a blueprint for concrete policy implementations while allowing specific policies to extend and customize functionality.

Example:

java

Copy code

public abstract class Policy {

private String policyId;

private LocalDate startDate;

private LocalDate endDate;

// Constructor, getters, and setters...

// Abstract method for calculating premium

public abstract double calculatePremium();

}

PolicyType Interface:

Why: An interface can define common behaviors related to policy types, such as retrieval of policy type name or code. Implementing classes can provide specific implementations for different policy types, ensuring consistency and flexibility.

Example:

java

Copy code

public interface PolicyType {

String getTypeName();

String getTypeCode();

}

PremiumCalculationStrategy Interface:

Why: An interface can define a contract for different premium calculation strategies, such as basic, advanced, or custom strategies. Implementing classes can encapsulate specific premium calculation algorithms while allowing the system to switch between strategies dynamically.

Example:

java

Copy code

public interface PremiumCalculationStrategy {

double calculatePremium(Policy policy);

}

PolicyRepository Interface:

Why: An interface can define methods for accessing and managing policy data, providing a common contract for different implementations (e.g., database repository, in-memory repository). This promotes loose coupling and facilitates dependency injection and testing.

Example:

java

Copy code

public interface PolicyRepository {

Policy findById(String policyId);

void save(Policy policy);

void update(Policy policy);

void delete(String policyId);

}

PolicyService Abstract Class:

Why: An abstract class can provide common functionality for policy management services, such as creation, retrieval, updating, and deletion of policies. It can define default implementations for common operations while allowing subclasses to override specific behaviors if needed.

Example:

java

Copy code

public abstract class PolicyService {

public abstract Policy findById(String policyId);

public abstract void createPolicy(Policy policy);

public abstract void updatePolicy(Policy policy);

public abstract void deletePolicy(String policyId);

}

By using abstract classes and interfaces in these areas of the project, you can achieve a more modular, flexible, and maintainable design. Abstract classes provide a foundation for common behavior and allow for code reuse, while interfaces define contracts for interactions between different components, promoting interoperability and flexibility.

## Functional Interface

Functional interfaces can be utilized in various scenarios within the Business Interruption Policy Module, particularly where behavior needs to be passed as a parameter or where a single abstract method is required. Here's a detailed example of where and why you might use a functional interface:

Scenario: Premium Calculation Strategy

Why Functional Interface: Different premium calculation strategies might be employed based on policy attributes, such as business type, location, or coverage level. By using a functional interface, you can encapsulate each calculation strategy as a separate implementation, allowing flexibility and easy switching between strategies without modifying existing code.

Example Implementation:

Functional Interface:

Define a functional interface PremiumCalculationStrategy with a single abstract method calculatePremium().

java

Copy code

@FunctionalInterface

public interface PremiumCalculationStrategy {

double calculatePremium(Policy policy);

}

Concrete Strategy Implementations:

Implement specific premium calculation strategies by providing concrete implementations of the calculatePremium() method.

java

Copy code

public class BasicPremiumCalculationStrategy implements PremiumCalculationStrategy {

@Override

public double calculatePremium(Policy policy) {

// Basic premium calculation logic based on policy attributes

return policy.getBasePremium();

}

}

public class AdvancedPremiumCalculationStrategy implements PremiumCalculationStrategy {

@Override

public double calculatePremium(Policy policy) {

// Advanced premium calculation logic based on policy attributes

return policy.getBasePremium() \* policy.getRiskFactor();

}

}

Usage:

Inject the appropriate premium calculation strategy based on policy attributes, and invoke the calculatePremium() method to determine the premium amount.

java

Copy code

public class PremiumCalculationService {

private PremiumCalculationStrategy calculationStrategy;

// Inject the strategy (can be done via constructor or setter injection)

public PremiumCalculationService(PremiumCalculationStrategy calculationStrategy) {

this.calculationStrategy = calculationStrategy;

}

public double calculatePremium(Policy policy) {

// Delegate premium calculation to the strategy implementation

return calculationStrategy.calculatePremium(policy);

}

}

Explanation:

By using a functional interface PremiumCalculationStrategy, you encapsulate the behavior of premium calculation into separate strategy implementations (BasicPremiumCalculationStrategy, AdvancedPremiumCalculationStrategy).

The PremiumCalculationService class can accept any implementation of PremiumCalculationStrategy, allowing it to calculate premiums using different strategies without being tightly coupled to specific implementations.

This approach promotes flexibility, maintainability, and extensibility, as new premium calculation strategies can be added easily by implementing the PremiumCalculationStrategy interface. It also follows the principles of abstraction and separation of concerns.

## Collections

For optimal performance in the Business Interruption Policy Module, the following collection objects are typically preferred:

HashMap:

Suitable for key-value pair storage and retrieval with O(1) time complexity for average-case operations.

Provides efficient performance for mapping policy IDs to policy objects.

java

Copy code

Map<String, Policy> policyMap = new HashMap<>();

ArrayList:

Provides constant-time positional access and efficient iteration with O(1) time complexity for average-case operations.

Ideal for storing a list of policies when sequential access and batch processing are common requirements.

java

Copy code

List<Policy> policyList = new ArrayList<>();

HashSet:

Ensures uniqueness of elements with O(1) time complexity for average-case operations such as add, remove, and contains.

Useful for storing unique policy IDs for quick lookup and to avoid duplicates.

java

Copy code

Set<String> policyIds = new HashSet<>();

These collection objects offer efficient performance characteristics, making them suitable for various data storage and manipulation tasks within the Business Interruption Policy Module. They provide fast access, insertion, and removal operations, ensuring optimal performance in typical usage scenarios.

Concurrent Collections:

Concurrent collections are thread-safe versions of standard collections, suitable for concurrent access in multi-threaded environments.

Example: Use ConcurrentHashMap for concurrent access to policy data in a multi-threaded application.

java

Copy code

Map<String, Policy> policyMap = new ConcurrentHashMap<>();

policyMap.put("POL001", new Policy("POL001", ...));

// Add more policies...

Policy policy = policyMap.get("POL001");

## Exception Handling

**How you ever handled exception in your project? How?**

In a Business Interruption Policy Module project, you would typically encounter various exceptions, both built-in and custom, to handle different error conditions and exceptional scenarios. Here's a list of commonly used built-in exceptions along with examples of potential custom exceptions that could be defined in the project:

Built-in Exceptions:

NullPointerException:

Thrown when attempting to access or manipulate a null object reference.

Example: Occurs when trying to access attributes of a null policy object.

IllegalArgumentException:

Thrown when an illegal or inappropriate argument is passed to a method.

Example: Raised if an invalid policy ID is provided as an argument.

IllegalStateException:

Thrown when a method is invoked at an inappropriate time or in an inappropriate state.

Example: Thrown if attempting to update a policy that is already canceled.

NumberFormatException:

Thrown when attempting to convert a string to a numeric format, but the string does not have the appropriate format.

Example: Raised when parsing a non-numeric input as a premium amount.

IndexOutOfBoundsException:

Thrown when attempting to access an index outside the bounds of an array, collection, or string.

Example: Occurs if trying to access an invalid index in a list of policies.

Custom Exceptions:

PolicyNotFoundException:

Custom exception to indicate that a requested policy could not be found.

Example: Thrown when attempting to retrieve a policy with a non-existent policy ID.

java

Copy code

public class PolicyNotFoundException extends RuntimeException {

public PolicyNotFoundException(String message) {

super(message);

}

}

InvalidPolicyDataException:

Custom exception to signify that the data provided for a policy is invalid or incomplete.

Example: Raised when attempting to create a policy with missing or invalid attributes.

java

Copy code

public class InvalidPolicyDataException extends RuntimeException {

public InvalidPolicyDataException(String message) {

super(message);

}

}

PremiumCalculationException:

Custom exception to handle errors during premium calculation.

Example: Thrown when encountering issues during the calculation of policy premiums.

java

Copy code

public class PremiumCalculationException extends RuntimeException {

public PremiumCalculationException(String message) {

super(message);

}

}

ClaimProcessingException:

Custom exception to handle errors during claim processing.

Example: Raised when encountering errors while processing a claim.

java

Copy code

public class ClaimProcessingException extends RuntimeException {

public ClaimProcessingException(String message) {

super(message);

}

}

These custom exceptions can be used to provide meaningful error messages and to handle specific error conditions within the Business Interruption Policy Module. By combining built-in and custom exceptions effectively, you can ensure robust error handling and fault tolerance in your application.

## JWT Token

Get the jwt based token from the authentication

Extract token from the authentication result

Set the Http header authorization value as bearer jwt token

Then send a request to access the protected resources

If the requested resource is protected spring security will use our custom filter to validate the jwt token

If the jwt token is valid it will return the requested resource to client

Structure of jwt token

Header: name of algo type of token

Payload :the claims which are statements about entity(public ,private)

Signature: verify the sender of jwt is who it says it is & to ensure that the massage was’nt change along the way

## Testing Framework

**JUnit:** JUnit is a widely used testing framework for Java applications. It provides annotations and assertions for writing unit tests to verify individual components of our code.

**Mockito:** Mockito is a mocking framework that can be used in conjunction with JUnit. It helps you create mock objects to isolate the components you are testing from their dependencies.

**Junit annotations:**

**@Test**: This annotation is used to mark a method as a test method.

// Coding example

**What is mockup**

**Mockup :**A mockup typically refers to a prototype or a rough design representation of a software application or user interface. Mockups are used to visualize the layout, structure, and functionality of an application before it is fully developed.

**UNIT TESTING DIFFERENCE INTEGRATION TESTING**

**How do you handle exception in junit**

**assertThrows** is used to ensure that the **MyException** is thrown when the code block within the lambda expression is executed. Also we can use try catch block within our test method to catch and handle exceptions

**how much code covreage u got** ?

Code coverage is an important aspect of software quality assurance, but it's not something I can measure or report on directly.

**where do u store your report ?**

**spy method in junit?**

Is used to create a partial mock of a real object. It allow you to spy object that retains the behavior of the original object while still allowing you to stub and verify sprcific method calls.

EX. RealObject realObj=new RealObject();

RealObject realObj=Spy( realObj);

When (spyObject.methodToStub()).thenReturn (“mocked value ”)

spyObject.MethodToStub();

**can we create junit class for private and static class ?**  
In JUnit, you generally create test classes for public methods of public classes. Private methods and static methods are typically not tested directly.

**How we test this classes then ?**

**Private class**

**1.Reflection-** You can use Java reflection to access and test private methods.

***to inspect and change the behavior of a program at runtime.With*** the help of this reflection API,

**2. Testing Through Public Methods**:

* Instead of directly testing private methods, you can test them indirectly through the public methods that use them

**Static Methods-**

Mockito that allows you to mock static methods and classes.

**What is Junit Lifecycle ?**

1.Initialization (@BeforeClass and @Before)

**2.Test** Execution **(@Test)**

**3**. Cleanup (@After and @AfterClass)

4. Test Result Reporting

5. Test Suite (@RunWith and Test Suites)

**Which are log 4 issues ?**

1. Configuration Errors
2. Memory Leaks
3. Security Concerns
4. Performance Impact

**What is diff between Junit and mockito ?**

While JUnit focuses on testing individual units of code, Mockito specializes in managing dependencies and mocking external interactions.

**Which server is used for local testing ?**

depending on the type of application we are developing,

**Apache Tomcat** is one of the most popular choices for running Java web applications locally.

Jetty is another lightweight and embeddable web server and servlet container. It's often used for development and testing purposes. Jetty is known for its simplicity and fast startup times.

**How to resolve bug if come ,at production environment ?**

The first step to handle any performance issue or bug in production is **to identify the root cause** **of the problem**. This can be done by using various tools and techniques, such as logging, monitoring, profiling, debugging, testing, and tracing.

**If any bug in a UI layer, What to do ?**

**Identify the Specific Issues and report it to frontend developer.**

**Sure, let's consider an example where we want to test the PremiumCalculationService class, which calculates premiums for policies. We'll use Mockito to mock dependencies such as a PolicyRepository and demonstrate how to write unit tests for the PremiumCalculationService class.**

**Assuming the PremiumCalculationService class has a method calculatePremium() that takes a policy ID as input and returns the calculated premium amount, and it internally uses a PolicyRepository to retrieve policy information for calculation.**

**Here's how you can write a JUnit test using Mockito:**

**java**

**Copy code**

**import static org.junit.Assert.assertEquals;**

**import static org.mockito.Mockito.\*;**

**import org.junit.Before;**

**import org.junit.Test;**

**import org.mockito.InjectMocks;**

**import org.mockito.Mock;**

**import org.mockito.MockitoAnnotations;**

**public class PremiumCalculationServiceTest {**

**@Mock**

**private PolicyRepository policyRepository; // Mocked dependency**

**@InjectMocks**

**private PremiumCalculationService premiumCalculationService; // Class under test**

**@Before**

**public void setUp() {**

**MockitoAnnotations.initMocks(this); // Initialize Mockito annotations**

**}**

**@Test**

**public void testCalculatePremium() {**

**// Given**

**String policyId = "POL001";**

**Policy policy = new Policy(policyId, /\* Other policy attributes \*/);**

**when(policyRepository.findById(policyId)).thenReturn(policy); // Mock policy repository to return a policy**

**// When**

**double premium = premiumCalculationService.calculatePremium(policyId);**

**// Then**

**assertEquals(expectedPremium, premium, 0.001); // Verify that the calculated premium matches the expected value**

**}**

**}**

**Explanation:**

**@Mock: This annotation creates a mock object for the PolicyRepository, which is a dependency of the PremiumCalculationService.**

**@InjectMocks: This annotation injects mocked dependencies into the PremiumCalculationService.**

**@Before: This method initializes Mockito annotations before each test method.**

**MockitoAnnotations.initMocks(this): This line initializes the Mockito annotations within the test class.**

**when(policyRepository.findById(policyId)).thenReturn(policy): This line mocks the behavior of the findById() method of the PolicyRepository to return a predefined policy when called with a specific policy ID.**

**assertEquals(expectedPremium, premium, 0.001): This assertion verifies that the calculated premium matches the expected value with a tolerance of 0.001.**

**Make sure to replace Policy with the actual policy class and PolicyRepository with the appropriate repository class used in your project. This example demonstrates how to write a JUnit test using Mockito to test the calculatePremium() method of the PremiumCalculationService class.**

## Annotations

**For Entity Classes:**

1. **@Entity**: This annotation is used to mark a class as a JPA entity, representing a table in the database.
2. **@Id**: Marks a field as the primary key of the entity.
3. **@GeneratedValue**: Specifies the strategy for generating primary key values.
4. **@ManyToOne** and **@JoinColumn**: Define a many-to-one relationship between entities.

**For Repository Interfaces:**

1. **@Repository**: Marks an interface as a Spring Data repository, enabling data access and manipulation.
2. **JpaRepository<T, ID>**: A generic interface provided by Spring Data JPA that extends **CrudRepository** and provides additional methods for data manipulation.

**For Controllers:**

1. **@RestController**: Combines **@Controller** and **@ResponseBody**. Marks a class as a controller that handles HTTP requests and returns JSON responses.
2. **@RequestMapping**: Specifies the base URL path for mapping controller methods.
3. **@PostMapping**: The POST method creates a new resource on the backend
4. **@GetMapping**: If we want to retrieve data from a resource
5. **@PutMapping:** he PUT method updates a resource by replacing its entire content completely.
6. **@PatchMapping :** PATCH is another HTTP method that is not commonly used. Similar to PUT, PATCH updates a resource, but it updates data partially and not entirely.
7. **@PathVariable**: Extracts values from the URL path and maps them to method parameters.
8. **@RequestBody**: Binds the request body to a method parameter, useful for handling JSON or XML payloads.
9. **m**:

**For Dependency Injection:**

1. **@Autowired**: Autowires beans by type. Injects instances of required beans into other beans.

**For Data Persistence:**

1. **Transactional**: it’s represent unit of work with the database.In such case if one step is failed the whole transaction is failed .Most of the RDBMS support transaction functionality .It is used the ACID property

* **Atomocity**:the entire transaction takes place at once or does’nt happen at all Eg. Money send should be received by the receiver If money is send but not received than the entire transaction fails is retracted.
* **Consistency**:It’s measure of correctness .Eg A=50+ B=50 ans=100,A=60+B=40 ans=100
* **Isolation**:multiple transaction independently without interference Eg. T1----,t2-----
* **Durability**:the changes of a successful transaction occurs even if the system is failure Eg.
* **@Trasaction**:The transactional annotation itself defines the scope of a single database transaction

## Generic Questions

**What is bug reporting tool in your project---**Jira

**How task will be assigned to u---**Task is assign through Jira

**Application is lot of time to respond in production,how will you identify the root cause?**

Application perform slow can be multiple reason like Network Speed,Execution time,DB query execution time.

# Non Technical

## Agile Methodology

It is a methodology which tells you what we can do to achieve continuous development and continuous production means never ending process.

We follow agile methodology or process and use work in sprint is of two weeks and two sprint is one **Release.** And any task is come then ticket is allocated to us in Jira.

(1 Release cycle => 1 Month. In a year we have 12 Release cycles. Currently this one is the third Release cycle.)

Ticket are four types .CR [Change Request], SR [Service Request], PSR [Paid service Request], PCR [Paid change Request].

These tickets are classified based on priorities p1, p2, p3, p4. P1 is called critical, p2 is called high priority, p3 is called medium and p4 is called low priority. Once our team lead (scrum master) assigns the ticket to us then we acknowledge that ticket.

Ticket consists of ticket number, user story, priority, severity, sprint version, creation date, assigned to, **date file**.

We are assigned story points by our team lead. 1 day = 3 story points, 2 days = 5 story points & 3 days = 8 story points.

We are start doing analysis, and mark as analysis in progress and after that we do business analysis and technical analysis (Impact analysis). For more clarification we communicate & discuss with the business analyst and if it is not satisfied then we can talk to the client along with the team lead and manager. Meeting with client can happen once in a release (1 month). For clarification on technical analysis, we discuss with the team lead. It its not feasible then we keep it ON HOLD.

After that we start development and we do coding in development environment. Once the coding is done. we do unit testing and Integration testing. Then its deployed-on testing environment. i.e. QA Environment. We also have to do the documentation

If any problem occurs then we fix it and deploy again on QA Environment.

When it’s ok then it goes to UAT [User Acceptance Testing] Environment. If code breaks here then we need to do reanalysis and fix it. If issue resolve then it will go to the production environment. In production environment if any issue occurs then we have to fix it.

All this communication is done in the JIRA.

Daily work report communication against tickets. All sorts of record will be stored in Jira

Code Review

Peer Review

Types of Testing

* Unit Testing
* System Integration testing (SIT)
* Sanity testing – conducted every weekend

How many team members in your team?

Total = 13

* 5 Developers
* 3 QA
* 2 DevOps
* 1 BA
* 1 Team Lead
* 1 Product Owner

What type of work do you get?

* Implementation / development (PCR)
* Enhancement (PCR/ PSR)
* Change/modify (PSR)
* Maintenance (PSR)
* Support (PSR)

What type of analysis do you perform?

* Impact Analysis – we predict probable positive & negative impacts that might arise in the future.
* 5 why Analysis

Spring Planning

Backlog Grooming/ Backlog Refinement

Sprint review

Sprint Retrospective

Sprint Brainstorming

Dev work cycle begins in the last week of the month. QA cycle begins in the first week. Dev Week 1 (month actual week 4) is called the regression week, (Entire app is tested). In the week developer works on new Release tickets.

PIP – Performance Improvement Program

Code Freeze

Patch Fix – Fix issues at required server

Hot Fix – Fix issues at production server. We do not have access to the production server. I talk to architect and that code is used to fix production issue by the architect. We have production replica. I work in the shadow.

When do we face client?

* Along with BA while requirement gathering, or for clarification or for demo.

## Daily Routine

My day start at 9.30Am. First, I have login to my system with credentials, I check all my mails & check calendar, if suppose in mails same work regarding the hot issue, accordingly will work on that probably.

Then I attend my daily stand-up call at 11am with team and team leader (Scrum master). In that meeting we all discuss what we need to do in whole day. In this standup call we discuss about if any query raised at analysis side and development side. After the standup call. I start my actual work on user story implementation.

Suppose in my user story I get a challenging task then I collaborate with my colleague’s. If I want suggestions on any approach related to that challenge then I asked to my team members opinions and work on that & get appropriate solution.

After lunch I attend daily scrum meeting call, this call happens to check the progress towards the sprint goals to produce actionable plan for the next 24 hrs.

Scrum call improve communication between team it improves transparency among us.

Then again move back to my work like that all my day spend with my work & my work hours end by 6.30pm

Project 3 - 15 - 18 months

Project 2 – 14 months

Project 1 – 13 months

Training – 3 months

## Scenarios

Did you ever face merge conflict? How did you resolve it?

If you were given a requirement A & while working on it, you were asked to work in requirement B. How will you handle it?

## Sprint workflow and process

The sprint workflow is intended to help team members evaluate their work and communicate with each other throughout the entire process. The workflow is followed for each sprint. The process includes:

* **Backlog** - A list of set tasks that must be completed before the product is released. The backlog is built by the product owner. The product owner gives a backlog of prioritized items to the scrum master and scrum team. The backlog is based on user stories, which focus on features that consider the type of end user, what they want and why.
* **Sprint planning** - The team discusses top priority user stories and decides what can be delivered in the sprint.
* **Sprint backlog** - Agreed upon by the entire team, this list finalizes and defines what the development team will complete during the sprint.
* **Sprint** – The time frame in which the work must be completed – often 30 days.
* **Daily scrum** – Lead by the scrum master, the team comes together for short daily meetings, in which they discuss what they have completed, what they are working on and any issues that are blocking the work.
* **Outcome** - The outcome of a sprint is a hypothetically usable product. The product owner can decide if the product is ready or if additional features are needed.
* **Sprint end** - At the end of a sprint, two meetings are held:
  + **Sprint review** – The team shows their work to the product owner.
  + **Sprint retrospective** – The team discusses what they can do to improve processes. An important goal is [continuous improvement](https://www.techtarget.com/searcherp/definition/kaizen-or-continuous-improvement).