# externalized configuration/properties design pattern in your microservices application

### External Configuration Pattern

What is it?

The External Configuration pattern involves separating configuration settings from the codebase and storing them externally, typically in configuration files or external configuration servers. This allows for dynamic configuration changes without redeploying the application.

Why we should use it?

Using the External Configuration pattern is crucial in microservices architectures due to the distributed and scalable nature of microservices. It allows for central management and dynamic updating of configuration settings across multiple microservices without the need for code changes or redeployment for defferent enviroments.

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources including Java properties files, YAML files, environment variables, and command-line arguments.
Property values can be injected directly into your beans by using the @Value annotation, accessed through Spring's Environment abstraction,

Purpose:

The purpose of the External Configuration pattern is to decouple configuration settings from the application code, making it easier to manage and update configurations across different environments (development, testing, production) and across multiple microservices.

Advantages:

1. Flexibility: Enables dynamic configuration changes without restarting or redeploying microservices.

2. Centralized Management: Facilitates centralized management of configurations across multiple microservices.

3. Environment-specific Configurations: Allows for environment-specific configurations, ensuring consistency and flexibility in different deployment environments.

4. Security: Helps in managing sensitive information (like database credentials, API keys) securely by externalizing them from the codebase.

5. Scalability: Supports scaling microservices independently without affecting configuration management.

### How it's used:

In your project, the External Configuration pattern is implemented using tools like Spring Cloud Config. Here's how it's used:

1. **Configuration Files:** Configuration settings are externalized into `application.properties` files, such as `application-dev.properties`, `application-test.properties`, and `application-prod.properties`, for each environment.
2. **Spring Boot Profiles:** Spring Boot automatically loads the appropriate `application.properties` file based on the active profile (e.g., `dev`, `test`, `prod`) specified during application startup.
3. **AWS Secret Manager:** Sensitive configuration properties, such as database credentials or API keys, are securely stored in AWS Secret Manager.

By using the External Configuration pattern, along with AWS Secret Manager for storing sensitive properties, your project achieves greater flexibility, scalability, security, and manageability of configuration settings in a microservices architecture, enhancing the overall agility and maintenance of the system.

---

---

etailed answer about the Circuit Breaker pattern, specifically focusing on using Resilience4j Circuit Breaker in your project for handling internal and external integrated APIs:

# Circuit Breaker Pattern

### What is it?

The Circuit Breaker pattern is a design pattern used in microservices architectures to handle failures and prevent cascading failures when calling remote services. It acts as a barrier to stop executing the function call if the failure rate exceeds a certain threshold, providing fault tolerance and improving system resilience.

### Why we should use it?

Using the Circuit Breaker pattern is crucial in microservices architectures where multiple services interact with each other over the network. It helps in maintaining system stability by isolating and handling failures gracefully, preventing them from propagating throughout the system.

### Purpose:

The purpose of the Circuit Breaker pattern is to protect the system from excessive retries and timeouts when calling unreliable services. It monitors the health of external or internal APIs and opens the

circuit (stops calling the service) if the failure rate or response times exceed predefined thresholds, thus preventing resource exhaustion and maintaining system availability.

### Advantages:

1. Fault Tolerance: Provides fault tolerance by isolating and handling failures at the service boundary.

2. Prevents Cascading Failures: Stops the propagation of failures throughout the system, preventing cascading failures.

3. Improves Resilience: Helps in building resilient systems that can gracefully handle failures and recover without impacting overall system performance.

4. Resource Management: Prevents resource exhaustion by reducing the number of failed retries and timeouts.

5. Dynamic Behavior: Allows for dynamic behavior adjustment based on the health of integrated services.

### How it's used:

In your project, the Circuit Breaker pattern is implemented using Resilience4j Circuit Breaker for handling both internal and external integrated APIs. Here's how it's used:

1. Integration with Resilience4j: Resilience4j is integrated into your microservices architecture as a dependency, providing Circuit Breaker capabilities.

2. Configuration: Each API call to internal or external services is wrapped in a Resilience4j Circuit Breaker configuration, specifying failure thresholds, timeout settings, and fallback mechanisms.

3. Monitoring Health: Resilience4j monitors the health of API calls by tracking response times, error rates, and timeouts.

4. Circuit State Management: When the failure rate or response times exceed configured thresholds, the Circuit Breaker transitions from a closed state (normal operation) to an open state (circuit broken), preventing further calls to the failing service.

5. Fallback Mechanisms: Resilience4j provides fallback mechanisms (e.g., returning cached data, providing default responses) when the Circuit Breaker is in the open state, ensuring graceful degradation of functionality.

6. Automatic Recovery: Resilience4j periodically checks the health of the failed service and attempts to close the circuit (transition to closed state) if the service becomes healthy again, allowing normal operation to resume.

By using the Circuit Breaker pattern with Resilience4j Circuit Breaker in your project, you ensure fault tolerance, prevent cascading failures, improve system resilience, and maintain overall system stability, especially when dealing with internal and external integrated APIs in a microservices environment.

using PostgreSQL as the database in your project, let's modify the Database per Service pattern answer accordingly:

# Database per Service Pattern with PostgreSQL

## What is it?

The Database per Service pattern is a design pattern used in microservices architectures where each microservice has its dedicated database. In your project using PostgreSQL, this pattern ensures that each microservice manages its data independently within separate PostgreSQL database instances or schemas.

## Why we should use it?

Using the Database per Service pattern with PostgreSQL offers several advantages:

1. Service Isolation: Each microservice operates within its dedicated PostgreSQL database, promoting service isolation and minimizing dependencies between services.

2. Data Autonomy: Microservices teams have control over their data models, schema designs, and database configurations, enabling flexibility and autonomy in data management.

3. Scalability: PostgreSQL supports horizontal scalability, allowing each microservice's database to scale independently to meet varying workload demands.

4. Flexibility: Rapid development, testing, and deployment cycles are facilitated as changes to one microservice's database schema or data do not impact other services.

5. Security: PostgreSQL provides robust security features, ensuring data integrity and access control within each microservice's database.

6. Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch.

## Purpose:

The Database per Service pattern with PostgreSQL serves the following purposes:

1. Data Autonomy: Each microservice team can choose appropriate PostgreSQL features, such as JSONB data types, advanced indexing, or stored procedures, to optimize data storage and retrieval.

2. Service Encapsulation: Encapsulation of data within microservices enhances maintainability, as changes to one service's database schema or queries do not affect other services.

3. Horizontal Scalability: PostgreSQL's support for horizontal scaling via partitioning, replication, and clustering enables efficient scaling of microservice databases based on workload demands.

4. Data Consistency: Transactional consistency within each microservice's PostgreSQL database ensures data integrity and atomicity for critical operations.

Advantages:

1. Service Isolation: Each microservice's PostgreSQL database operates independently, minimizing the impact of failures or changes on other services.

2. Data Autonomy: Microservice teams can optimize database schemas, indexes, and queries based on specific service requirements, promoting efficient data management.

3. Scalability: PostgreSQL's scalability features enable horizontal scaling of microservice databases, ensuring performance under varying workloads.

4. Flexibility: Independent database management allows for rapid development, testing, and deployment cycles without affecting other services.

5. Security: PostgreSQL's robust security mechanisms, including role-based access control (RBAC) and data encryption, enhance data protection and access control within microservices.

How it's used:

1. Database Creation: Separate PostgreSQL database instances or schemas are created for each microservice, ensuring data isolation and autonomy.

2. Configuration: Microservices are configured to connect to their respective PostgreSQL databases using connection properties (host, port, username, password).

3. Data Migration: Initial data migration scripts or tools are used to populate and initialize microservice databases with required schema structures and data.

4. Transactional Boundaries: PostgreSQL's support for transactions ensures data consistency and atomicity within each microservice's database operations.

5. Schema Evolution: Microservice teams can evolve database schemas independently, performing schema migrations and data migrations as needed.

6. Monitoring and Management: PostgreSQL's monitoring tools and management utilities facilitate performance monitoring, backup, and recovery for microservice databases.

By leveraging the Database per Service pattern with PostgreSQL in your project, you achieve enhanced data autonomy, service encapsulation, scalability, flexibility, and security within your microservices architecture.

Using a database per service has the following drawbacks:

- Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them.
- Implementing queries that join data that is now in multiple databases is challenging.
- Complexity of managing multiple SQL and NoSQL databases

---

---

detailed answer about the Event Sourcing pattern (Pub-Sub pattern) using Apache Kafka as the message broker in your project:

# Event Sourcing (Pub-Sub Pattern) with Apache Kafka

## Few of the use case in the project as

1. For sending email, sms or push notification to users for their appointment booking, cancel, or reschedule confirmation
    a. Here this type of events is consume by multiple services for example: if patient did appointment booking then the appointment service will publish the event in the apache kafka topic and this topic is subscribed by multiple services such as notification service (for sharing notification), survey service ( for doing appointment survey related processing and rating calculation etc.)
2. for sending early slot available notification to user who have subscribed for this service.
3. Internal events consumption

## What is it?

The Event Sourcing pattern, also known as the Pub-Sub (Publish-Subscribe) pattern, is a design pattern used in distributed systems where publishers publish events to topics, and subscribers consume these events asynchronously. In your project using Apache Kafka as the message broker, this pattern facilitates event-driven communication and data processing.

## Why we should use it?

Using the Event Sourcing pattern with Apache Kafka offers several advantages:

1. Asynchronous Communication: Event-driven communication allows microservices to communicate asynchronously, promoting loose coupling and scalability.

2. Scalability: Apache Kafka's distributed architecture supports horizontal scaling and high-throughput event processing, making it suitable for handling large volumes of events.

3. Fault Tolerance: Kafka's replication and fault tolerance mechanisms ensure reliable event delivery even in the presence of failures.

4. Real-time Processing: Pub-Sub enables real-time data processing and analytics by consuming events as they are published.

5. Event Logging: Event sourcing allows capturing and storing a history of events, enabling auditing, analytics, and data replayability.

## Purpose:

The Event Sourcing pattern with Apache Kafka serves the following purposes:

1. Event-driven Architecture: Enables building event-driven architectures where microservices communicate and react to events asynchronously.

2. Real-time Data Processing: Facilitates real-time processing of events, allowing microservices to respond quickly to changes and events in the system.

3. Fault Isolation: Decouples services by using events as an intermediary, reducing dependencies and isolating failures within individual microservices.

4. Scalable Message Broker: Apache Kafka acts as a scalable and reliable message broker, handling the distribution, storage, and consumption of events across services.

## Advantages:

1. Asynchronous Communication: Facilitates loose coupling between microservices, promoting scalability and fault tolerance.

2. Real-time Processing: Enables real-time data processing and analytics by consuming events as they occur.

3. Fault Tolerance: Apache Kafka's replication and partitioning ensure reliable event delivery and fault tolerance.

4. Event Replayability: Events can be replayed for auditing, debugging, or rebuilding state, providing resilience and flexibility.

5. Scalability: Kafka's distributed nature supports horizontal scaling and high-throughput event processing.

## How it's used:

1. Event Publishing: Microservices publish events to Kafka topics using Kafka producers, specifying the topic and event data.

2. Topic Subscription: Microservices subscribe to Kafka topics using Kafka consumers, receiving events asynchronously as they are published.

3. Event Processing: Subscribed microservices process events based on business logic, updating their state or triggering actions.

4. Fault Tolerance: Kafka's replication and partitioning ensure fault tolerance and high availability, allowing services to recover from failures.

5. Event Logging: Kafka's log-based storage enables capturing a history of events, supporting auditing, analytics, and event replayability.

By leveraging the Event Sourcing pattern (Pub-Sub pattern) with Apache Kafka in your project, you enable asynchronous communication, real-time data processing, fault tolerance, and scalability, enhancing the overall agility and responsiveness of your microservices architecture.