Bank Account Application

Overview

This application simulates a fundamental banking operation (withdrawals) and event notification. The event notification forms an integral part of the banking transaction, ensuring consistency and reliability in financial operations.

Approach

In reevaluating the provided code, the primary goal is to enhance its structure, maintainability, scalability, and overall quality while preserving its core business functionality.

The following improvements have been implemented across various facets of the application:

Configuration Improvements

- **Centralised Configuration**: YAML was adopted for representing the properties used in the application. The choice to use YAML (over a text properties file) is that it enhances readability and maintainability.
- **Logical Organization**: Configuration settings have been logically grouped into sections for improved clarity and organization.

Retry Mechanism

- **Utilization of RetryTemplate**:
    - Spring's RetryTemplate has been used to implement a retry mechanism (specifically around the event publishing) with configurable max attempts and backoff period.
    - This could be extended to the withdraw process in its entirety and is a possible further improvement that can be made.
- **Enhanced Fault Tolerance**: The retry mechanism enhances fault tolerance by automatically retrying failed withdrawal event publising operations, thereby improving system robustness.

Code Structure Enhancements

- **Modularization**: The codebase has been modularized into separate classes for configuration, controller, service, repository, and exception handling, promoting a separation of concerns.
- **Lombok Integration**: Lombok annotations have been utilized to minimize boilerplate code and enhance code readability.

Exception Handling

- **Global Exception Handler**: A global exception handler has been implemented to centralize exception handling logic and provide consistent error responses across the application.
- **Observability and Logging**: Detailed error messages are logged throughout the application for debugging and auditing purposes, enhancing observability.

Documentation and Logging

- **Comprehensive Logging**:
  - Logging statements have been strategically placed throughout the codebase to provide comprehensive visibility into application events and behaviors.
- **Logback Configuration**:
  - Logback has been configured as the logging framework with a simple console appender, ensuring effective logging of application events.
  - Logback is easy to implement and flexible. Being lightweight and efficient, application performance should not be adversely affected.
  - Of course, as with any logging, it needs to be used judiciously.

Observability and Monitoring

- **Actuator Integration**: Spring Boot Actuator has been integrated to expose health and info endpoints for monitoring and management purposes.

Implementation Choices

- **Configuration Format**: Used YAML for improved readability and maintainability of property values and application config.
- **Retry Mechanism**: Utilized RetryTemplate for its simplicity and seamless integration with Spring applications.
- **Exception Handling**: Employed global exception handling using ControllerAdvice for consistent error responses.
- **Logging Framework**: Configured Logback for its flexibility, performance, and widespread adoption in Spring Boot applications.
- **Actuator Configuration**: Integrated Actuator for observability and monitoring, providing health and info endpoints.

Areas Covered

- **Structure and Maintainability**: Modularized code into separate components for enhanced maintainability and scalability.
- **Fault Tolerance**: Implemented a retry mechanism to improve resilience against transient failures.
- **Observability**: Integrated Actuator for monitoring application health and providing additional information.
- **Exception Handling**: Centralized exception handling logic for consistent error responses and improved reliability.
- **Logging and Auditing**: Enhanced logging for debugging, auditing, and observability purposes.

Further improvements that can be made

- **Performance and optimisations**
  - The base tables may be indexed to improve database querying.
  - By pooling and reusing database connections, we can significantly reduce overhead associated with connection establishment and teardown, leading to improved performance and resource utilization.

    - Implementing caching (at the database level) can be used to help improve performance and reduce response times.
    - Caching account balances reduce database load and enhance response times.

- **Scalability, redundancy and resilience**

    - As the volume of data increases and to mitigate a single point of failure, we may consider duplication, partitioning and sharding the data across instances.
        - Keeping in mind of course, that this increases the complexity of the system as all datasources need to be kept up to date and consistent
    - We can further decompose the BankAccount service into smaller services (e.g., a separate publishing service).
    - Implement a load balancer to distribute the load across different instances can be done at different layers (database, services).

- **Consistency and Availability**

    - In my opinion, given this is a financial component, I would favor Consistency over Availability. We need the transactional data to be consistent irrespective of the node.
    - When considering caching policies, we need to weigh the benefits of read-through and write-back strategies against their potential impact on data integrity and system complexity.
    - Given the nature of this service, we need to prioritize consistency. As such, read-through caching on a cache miss would be best. We want the balance retrieval and update to be as accurate as possible and maintain integrity.
    - I don't believe that the alternative - write-back caching - would be advisable in a bank transactional service such as this. We cannot risk stale or inconsistent data over a performance gain.

- **Log optimisations and space consideration**

    - The system will generate logs as its running and these log files can grow over time. In this simple implementation, the logging is set to log to the console.
    - However, in a production system, this would most likely be to a file.
    - There should be a mechanism in place for log rotation of these files - and possibly moving the archived log files to a store outside of the application (eg. object storage bucket)

- **Containerisation**

    - To further improve on the scalability, deployment and consistency, the services in the application could be containerised
    - We could then use Kubernetes (or similar cloud based services like GKE, OKE, ECS) as an orchestration tool to manage the containers

General

Some useful Actuator Endpoints to ensure health of the system

Health Check

- **Endpoint:** `/api/v1/actuator/health`
- **Description:** Provides information about the health status of the application, including details about database connectivity, disk space, and other dependencies.

Application Info

- **Endpoint:** `/api/v1/actuator/info`
- **Description:** Displays general information about the application, such as name, version, and description.

Metrics

- **Endpoint:** `/api/v1/actuator/metrics`
- **Description:** Offers various metrics about the application's performance, including request rates, response times, and error counts.

Swagger

- **Endpoint** `/api/v1/swagger-ui/index.html`
- **Description:** Swagger endpoint