Technical Design Document

Prepared By: Lerato Tlhako

TABLE OF CONTENTS

01. INTRODUCTION

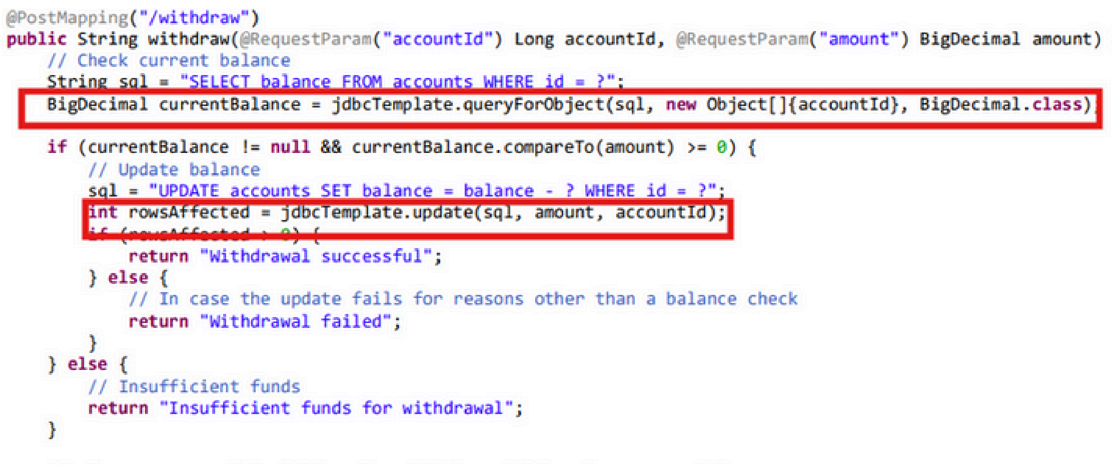
This document will:

- Outline the approach taken to enhance the code.
- Describe the reasoning behind refactoring choices made.
- Provide refactored code.

02. CODE SNIPPETS WITH IDENTIFIED PROBLEMS

TIGHTLY COUPLING OF BUSINESS LOGIC

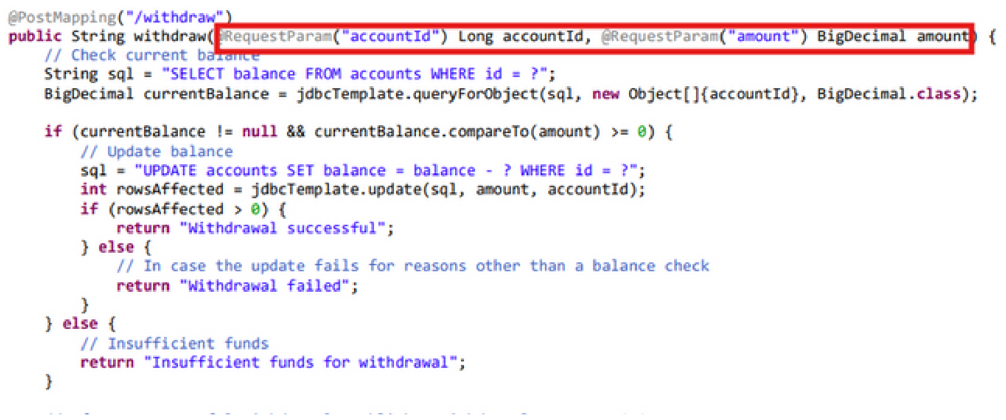
The business logic is currently embedded directly within the controller class. This violates the principle of separation of concerns and reduces testability and flexibility. Business logic should be encapsulated within a dedicated service layer, allowing the controller to focus solely on handling HTTP requests and responses.



```java
@PostMapping("/withdraw")
public String withdraw(@RequestParam("accountId") Long accountId, @RequestParam("amount") BigDecimal amount) {
    // Check current balance
    String sql = "SELECT balance FROM accounts WHERE id = ?";
    BigDecimal currentBalance = jdbcTemplate.queryForObject(sql, new Object[]{accountId}, BigDecimal.class);

    if (currentBalance != null && currentBalance.compareTo(amount) >= 0) {
        // Update balance
        sql = "UPDATE accounts SET balance = balance - ? WHERE id = ?";
        int rowsAffected = jdbcTemplate.update(sql, amount, accountId);
        if (rowsAffected > 0) {
            return "Withdrawal successful";
        } else {
            // In case the update fails for reasons other than a balance check
            return "Withdrawal failed";
        }
    } else {
        // Insufficient funds
        return "Insufficient funds for withdrawal";
    }
```

MISSING PARAMETER VALIDATIONS

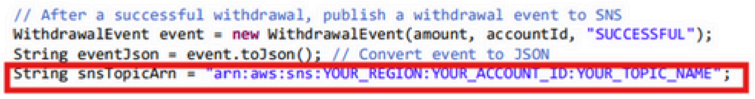
There is no validation of request parameters, such as ensuring that the withdrawal amount is greater than zero. This opens the system to potential inconsistencies and logical errors. Proper input validation should be implemented using frameworks like Java Bean Validation (@Valid, @Min, etc.) or within the service layer.



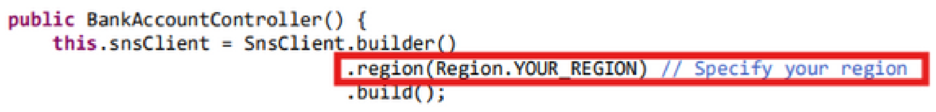
```java
@PostMapping("/withdraw")
public String withdraw(@RequestParam("accountId") Long accountId, @RequestParam("amount") BigDecimal amount) {
    // Check current balance
    String sql = "SELECT balance FROM accounts WHERE id = ?";
    BigDecimal currentBalance = jdbcTemplate.queryForObject(sql, new Object[]{accountId}, BigDecimal.class);

    if (currentBalance != null && currentBalance.compareTo(amount) >= 0) {
        // Update balance
        sql = "UPDATE accounts SET balance = balance - ? WHERE id = ?";
        int rowsAffected = jdbcTemplate.update(sql, amount, accountId);
        if (rowsAffected > 0) {
            return "Withdrawal successful";
        } else {
            // In case the update fails for reasons other than a balance check
            return "Withdrawal failed";
        }
    } else {
        // Insufficient funds
        return "Insufficient funds for withdrawal";
    }
```

HARDCODED VALUES

Critical configuration values such as the AWS region and SNS topic ARN are hardcoded within the codebase. This practice hinders flexibility and violates the twelve-factor app principles. These values should be externalized using environment variables or configuration files (e.g., application.properties or application.yml).



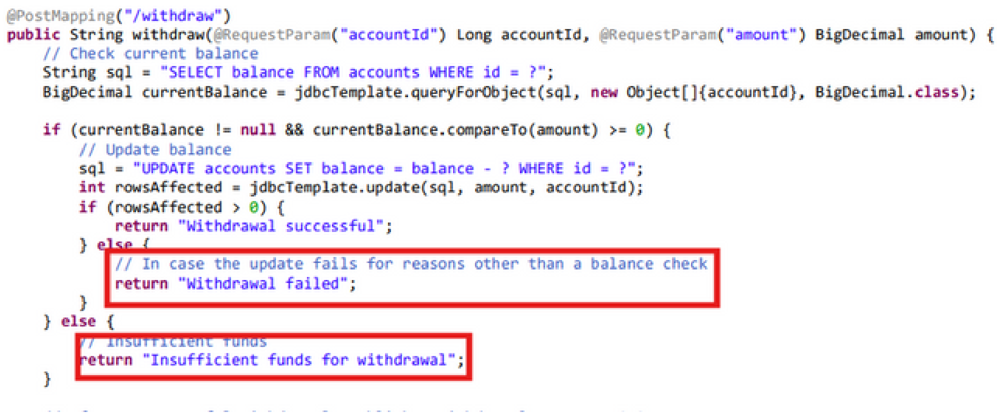
```
// After a successful withdrawal, publish a withdrawal event to SNS
WithdrawalEvent event = new WithdrawalEvent(amount, accountId, "SUCCESSFUL");
String eventJson = event.toJson(); // Convert event to JSON
String snsTopicArn = "arn:aws:sns:YOUR_REGION:YOUR_ACCOUNT_ID:YOUR_TOPIC_NAME";
```



```
public BankAccountController() {
    this.snsClient = SnsClient.builder()
                              .region(Region.YOUR_REGION) // Specify your region
                              .build();
```

LACK OF OBSERVABILITY

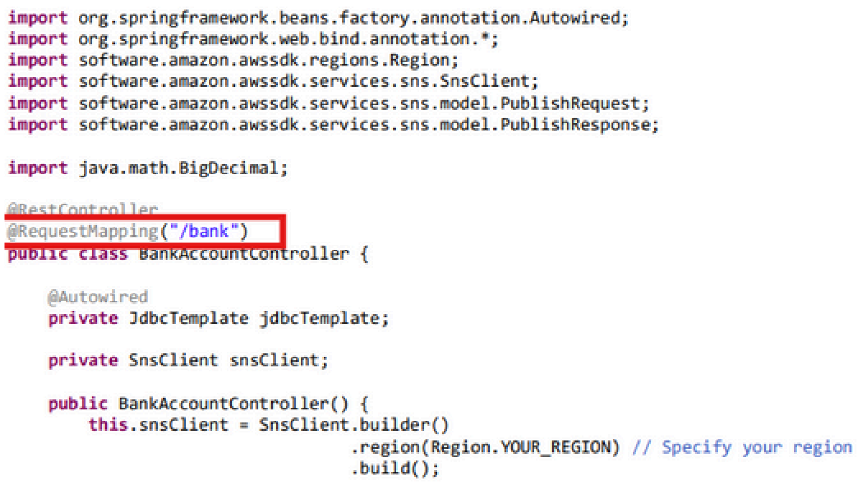
The current implementation lacks logging and tracing mechanisms. Without proper observability, diagnosing issues in production or during debugging becomes difficult. Implementing structured logging (e.g., using SLF4J) and integrating with observability tools enhances system reliability and maintainability.



```
@PostMapping("/withdraw")
public String withdraw(@RequestParam("accountId") Long accountId, @RequestParam("amount") BigDecimal amount) {
    // Check current balance
    String sql = "SELECT balance FROM accounts WHERE id = ?";
    BigDecimal currentBalance = jdbcTemplate.queryForObject(sql, new Object[]{accountId}, BigDecimal.class);

    if (currentBalance != null && currentBalance.compareTo(amount) >= 0) {
        // Update balance
        sql = "UPDATE accounts SET balance = balance - ? WHERE id = ?";
        int rowsAffected = jdbcTemplate.update(sql, amount, accountId);
        if (rowsAffected > 0) {
            return "Withdrawal successful";
        } else {
            // In case the update fails for reasons other than a balance check
            return "Withdrawal failed";
        }
    } else {
        // Insufficient funds
        return "Insufficient funds for withdrawal";
    }
```

ABSENCE OF API VERSIONING

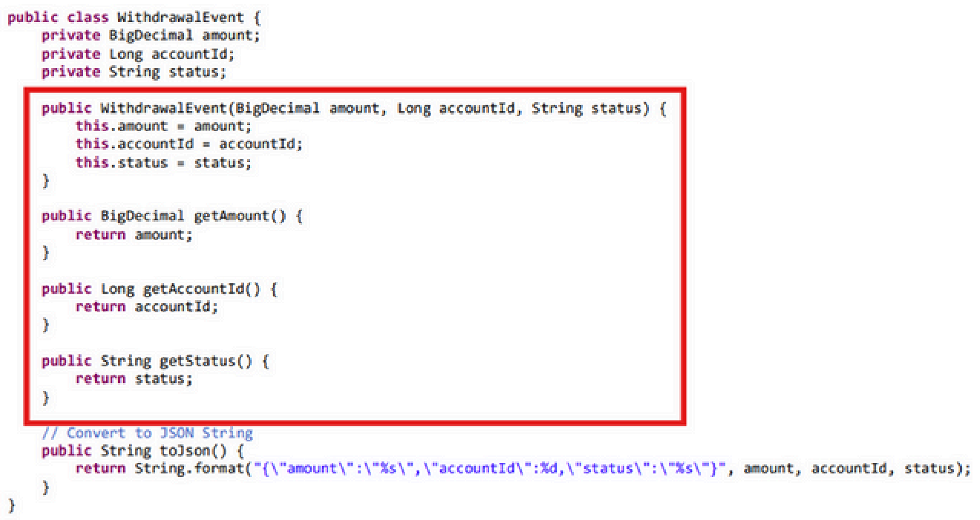
API versioning is not present. In larger systems, versioning is critical for maintaining backward compatibility when evolving or extending APIs. Adopting a versioning strategy (e.g., /api/v1/) ensures smoother transitions and minimizes disruptions for consumers of the API.



```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.sns.SnsClient;
import software.amazon.awssdk.services.sns.model.PublishRequest;
import software.amazon.awssdk.services.sns.model.PublishResponse;

import java.math.BigDecimal;

@RestController
@RequestMapping("/bank")
public class BankAccountController {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    private SnsClient snsClient;

    public BankAccountController() {
        this.snsClient = SnsClient.builder()
                                  .region(Region.YOUR_REGION) // Specify your region
                                  .build();
```

INEFFICIENT DEPENDENCY UTILIZATION

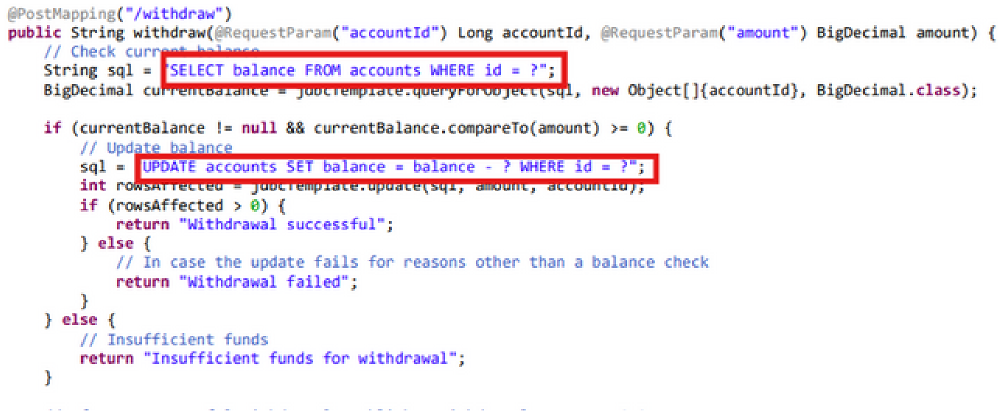
Manual creation of boilerplate code, such as getters and setters in classes like WithdrawalEvent, increases maintenance overhead. Leveraging tools like Project Lombok can significantly reduce verbosity and improve readability through annotations like @Getter, @Setter, and @Data.



```java
public class WithdrawalEvent {
    private BigDecimal amount;
    private Long accountId;
    private String status;

    public WithdrawalEvent(BigDecimal amount, Long accountId, String status) {
        this.amount = amount;
        this.accountId = accountId;
        this.status = status;
    }

    public BigDecimal getAmount() {
        return amount;
    }

    public Long getAccountId() {
        return accountId;
    }

    public String getStatus() {
        return status;
    }
    // Convert to JSON String
    public String toJson() {
        return String.format("{\"amount\":\"%s\",\"accountId\":%d,\"status\":\"%s\"}", amount, accountId, status);
    }
}
```

DATABASE LOGIC IN CONTROLLER

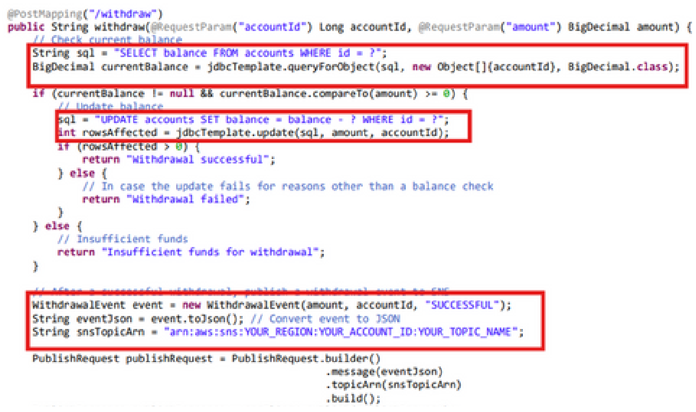
The controller currently executes raw SQL queries directly using JdbcTemplate. This violates architectural principles, as the controller should delegate data access to a dedicated repository or data access layer. Refactoring this logic promotes better abstraction and adherence to clean architecture.



```java
@PostMapping("/withdraw")
public String withdraw(@RequestParam("accountId") Long accountId, @RequestParam("amount") BigDecimal amount) {
    // Check current balance
    String sql = "SELECT balance FROM accounts WHERE id = ?";
    BigDecimal currentBalance = jdbcTemplate.queryForObject(sql, new Object[]{accountId}, BigDecimal.class);

    if (currentBalance != null && currentBalance.compareTo(amount) >= 0) {
        // Update balance
        sql = "UPDATE accounts SET balance = balance - ? WHERE id = ?";
        int rowsAffected = jdbcTemplate.update(sql, amount, accountId);
        if (rowsAffected > 0) {
            return "Withdrawal successful";
        } else {
            // In case the update fails for reasons other than a balance check
            return "Withdrawal failed";
        }
    } else {
        // Insufficient funds
        return "Insufficient funds for withdrawal";
    }
```

LACK OF FINAL KEYWORD USAGE

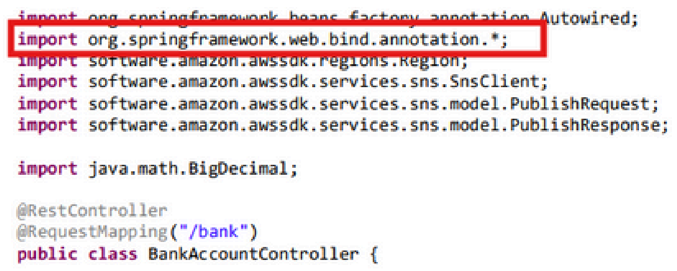
The given code does not utilize the final keyword and that means that the parameters, local variable initialized can be changed at a later stage, which may introduce a common issue in large codebases where the code is modified unintentionally.



```java
@PostMapping("/withdraw")
public String withdraw(@RequestParam("accountId") Long accountId, @RequestParam("amount") BigDecimal amount) {
    // Check current balance
    String sql = "SELECT balance FROM accounts WHERE id = ?";
    BigDecimal currentBalance = jdbcTemplate.queryForObject(sql, new Object[]{accountId}, BigDecimal.class);

    if (currentBalance != null && currentBalance.compareTo(amount) >= 0) {
        // Update balance
        sql = "UPDATE accounts SET balance = balance - ? WHERE id = ?";
        int rowsAffected = jdbcTemplate.update(sql, amount, accountId);
        if (rowsAffected > 0) {
            return "Withdrawal successful";
        } else {
            // In case the update fails for reasons other than a balance check
            return "Withdrawal failed";
        }
    } else {
        // Insufficient funds
        return "Insufficient funds for withdrawal";
    }

    // After a successful withdrawal, publish a withdrawal event to SNS
    WithdrawalEvent event = new WithdrawalEvent(amount, accountId, "SUCCESSFUL");
    String eventJson = event.toJson(); // Convert event to JSON
    String snsTopicArn = "arn:aws:sns:YOUR_REGION:YOUR_ACCOUNT_ID:YOUR_TOPIC_NAME";

    PublishRequest publishRequest = PublishRequest.builder()
                                                  .message(eventJson)
                                                  .topicArn(snsTopicArn)
                                                  .build();
```

UNOPTIMIZED IMPORTS

Wildcard imports such as org.springframework.web.bind.annotation.* are used, potentially introducing unnecessary classes into the compilation unit. It is recommended to explicitly import only the required classes to improve code clarity and avoid unintended dependencies.



```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.sns.SnsClient;
import software.amazon.awssdk.services.sns.model.PublishRequest;
import software.amazon.awssdk.services.sns.model.PublishResponse;

import java.math.BigDecimal;

@RestController
@RequestMapping("/bank")
public class BankAccountController {
```

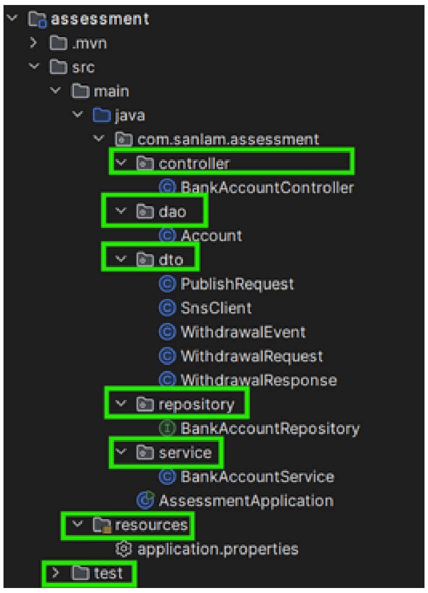
03. DESIGN GOALS

The goal is to refactor the existing withdrawal logic to transform it from tightly coupled and procedural codebase into a maintainable, uniform and production-ready service.

Goal	Description
Separation of conerns	Decouple the business, data layer and handle API into distinct layers (Controller → Service → Repository). This will improve modularity, maintainability and testability.
Input Validation and Data Integrity	Enforce validation rules to ensure only legitimate operations are allowed such as rejecting negative amounts. This will assist in preventing invalid transition states and error propagation reduction. Add @Transactional annotation on the business logic method to ensure that all transactions succeed or fail at once.
Observability and Diagnostics	Introduce structured logging, error handling and traceability to enable easier debugging, monitoring and incidents response in different environments.

04. REFACTORED CODE

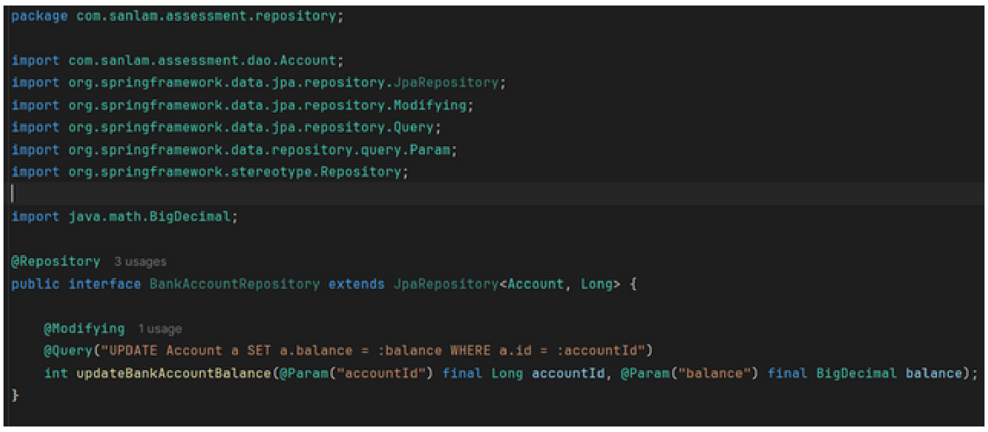
FOLDER STRUCTURE


```
assessment
  .mvn
  src
    main
      java
        com.sanlam.assessment
          controller
            BankAccountController
          dao
            Account
          dto
            PublishRequest
            SnsClient
            WithdrawalEvent
            WithdrawalRequest
            WithdrawalResponse
          repository
            BankAccountRepository
          service
            BankAccountService
          AssessmentApplication
      resources
        application.properties
    test
```

DATA ACCESS LAYER (REPOSITORY)

The data access layer has been streamlined by leveraging Spring Data JPA's built-in repository capabilities. Specifically, manual SELECT statements to retrieve account details by ID have been eliminated in favor of findById, a method automatically provided by the JpaRepository interface. This significantly reduces boilerplate code and enhances readability.

Additionally, the repository now only contains a single custom query—an annotated native UPDATE operation used to modify the account balance. This keeps the repository concise, focused, and aligned with clean architecture principles by separating read and write responsibilities appropriately.

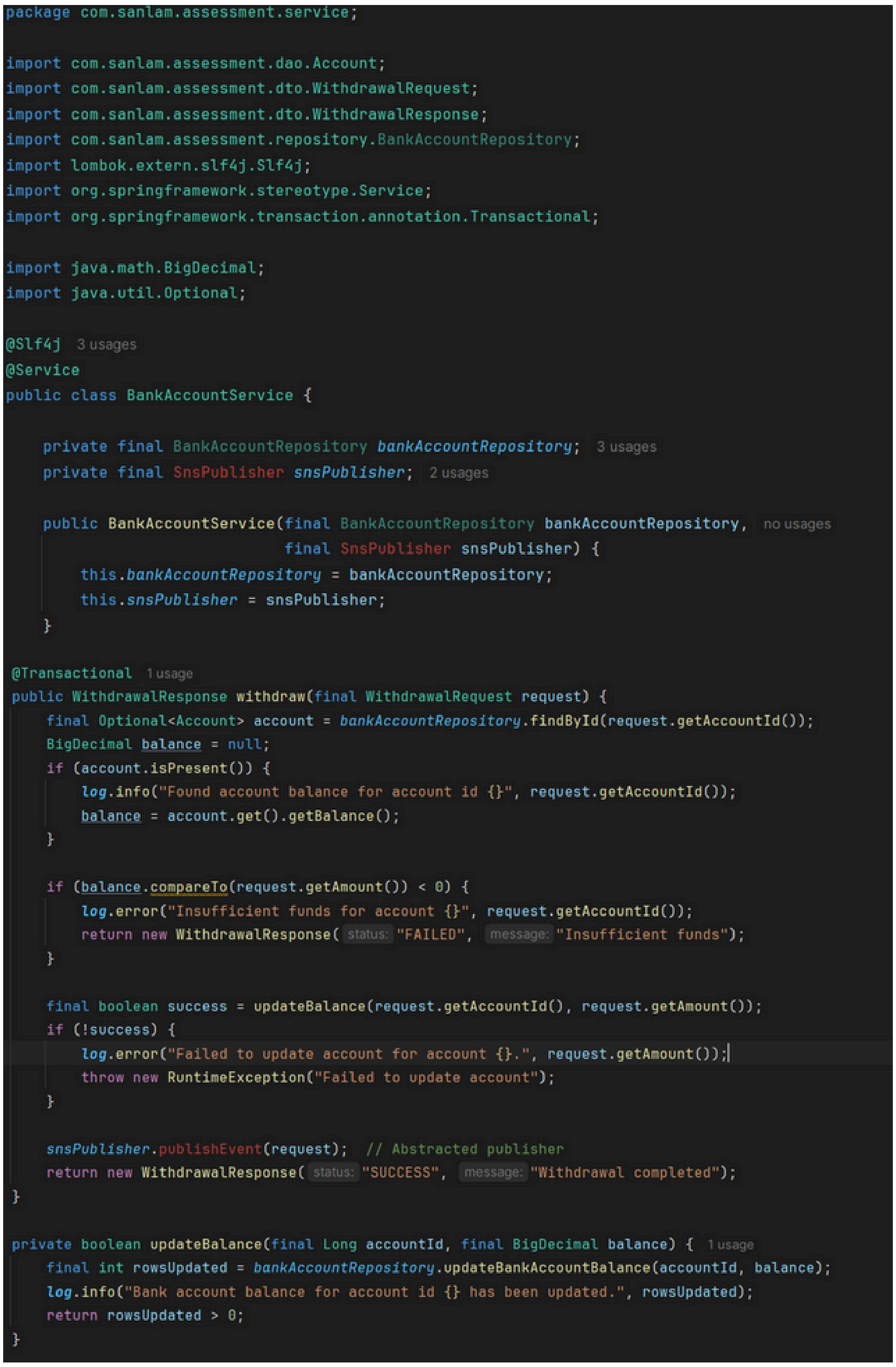

```java
package com.sanlam.assessment.repository;

import com.sanlam.assessment.dao.Account;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.math.BigDecimal;

@Repository  3 usages
public interface BankAccountRepository extends JpaRepository<Account, Long> {

    @Modifying  1 usage
    @Query("UPDATE Account a SET a.balance = :balance WHERE a.id = :accountId")
    int updateBankAccountBalance(@Param("accountId") final Long accountId, @Param("balance") final BigDecimal balance);
}
```

BUSINESS (SERVICE) LOGIC

The business logic has been extracted into a dedicated service class, organized under its own package to promote separation of concerns and improve maintainability. The class is annotated with @Service, allowing Spring to recognize and manage it as a service-level bean within the application context.

Rather than executing raw SQL queries to retrieve account details, the service now delegates this responsibility to the repository layer by invoking findById. This approach enhances code readability and reduces redundancy by utilizing Spring Data JPA's abstraction.

Furthermore, relevant log statements have been introduced to improve observability and traceability across key operations. The overall logic has been streamlined to ensure clarity, reduce complexity, and align with industry-standard service-layer design principles.

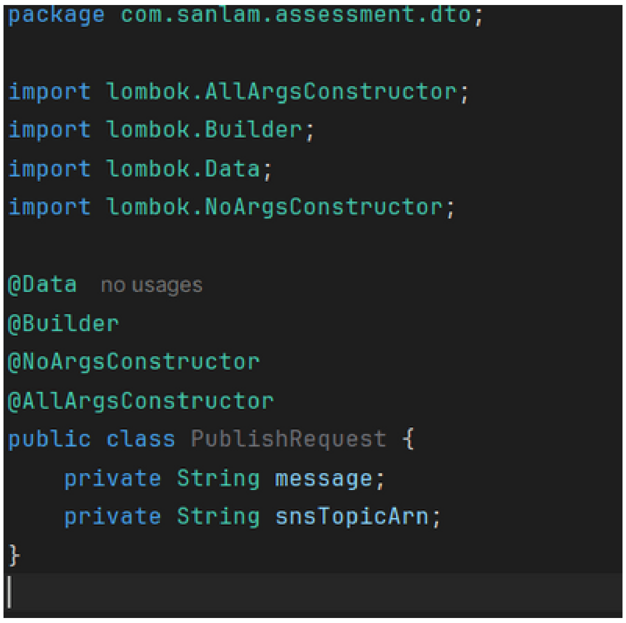


```java
package com.sanlam.assessment.service;

import com.sanlam.assessment.dao.Account;
import com.sanlam.assessment.dto.WithdrawalRequest;
import com.sanlam.assessment.dto.WithdrawalResponse;
import com.sanlam.assessment.repository.BankAccountRepository;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.math.BigDecimal;
import java.util.Optional;

@Slf4j  3 usages
@Service
public class BankAccountService {

    private final BankAccountRepository bankAccountRepository;  3 usages
    private final SnsPublisher snsPublisher;  2 usages

    public BankAccountService(final BankAccountRepository bankAccountRepository,  no usages
                              final SnsPublisher snsPublisher) {
        this.bankAccountRepository = bankAccountRepository;
        this.snsPublisher = snsPublisher;
    }

    @Transactional  1 usage
    public WithdrawalResponse withdraw(final WithdrawalRequest request) {
        final Optional<Account> account = bankAccountRepository.findById(request.getAccountId());
        BigDecimal balance = null;
        if (account.isPresent()) {
            log.info("Found account balance for account id {}", request.getAccountId());
            balance = account.get().getBalance();
        }

        if (balance.compareTo(request.getAmount()) < 0) {
            log.error("Insufficient funds for account {}", request.getAccountId());
            return new WithdrawalResponse( status: "FAILED",  message: "Insufficient funds");
        }

        final boolean success = updateBalance(request.getAccountId(), request.getAmount());
        if (!success) {
            log.error("Failed to update account for account {}.", request.getAmount());
            throw new RuntimeException("Failed to update account");
        }

        snsPublisher.publishEvent(request);  // Abstracted publisher
        return new WithdrawalResponse( status: "SUCCESS",  message: "Withdrawal completed");
    }

    private boolean updateBalance(final Long accountId, final BigDecimal balance) {  1 usage
        final int rowsUpdated = bankAccountRepository.updateBankAccountBalance(accountId, balance);
        log.info("Bank account balance for account id {} has been updated.", rowsUpdated);
        return rowsUpdated > 0;
    }
```

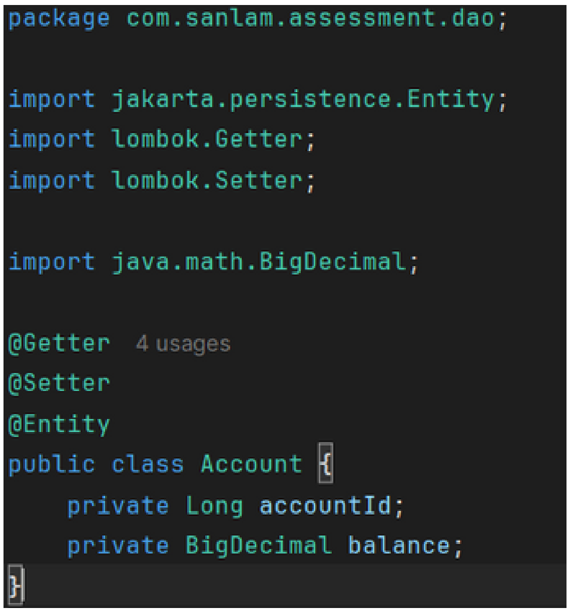
MODELS (DTO AND DAO CLASSES)

The domain and data transfer models have been simplified to reduce boilerplate code and enhance maintainability. Rather than manually implementing constructors, getters, setters, and other utility methods, we have integrated Lombok annotations (such as @Data, @AllArgsConstructor, @NoArgsConstructor, and @Builder) to automate their generation at compile time.

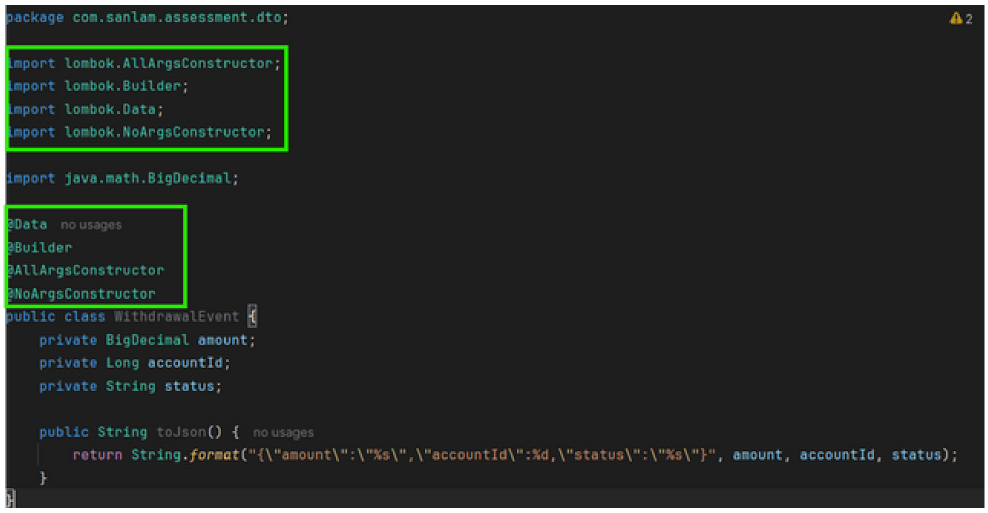
This approach significantly improves code readability, reduces the likelihood of human error, and accelerates development by minimizing repetitive code patterns. The result is a leaner, cleaner model layer that is easier to extend and maintain while remaining fully compliant with Java best practices.



```java
package com.sanlam.assessment.dto;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data  no usages
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class PublishRequest {
    private String message;
    private String snsTopicArn;
}
```



```java
package com.sanlam.assessment.dao;

import jakarta.persistence.Entity;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;

@Getter  4 usages
@Setter
@Entity
public class Account {
    private Long accountId;
    private BigDecimal balance;
}
```



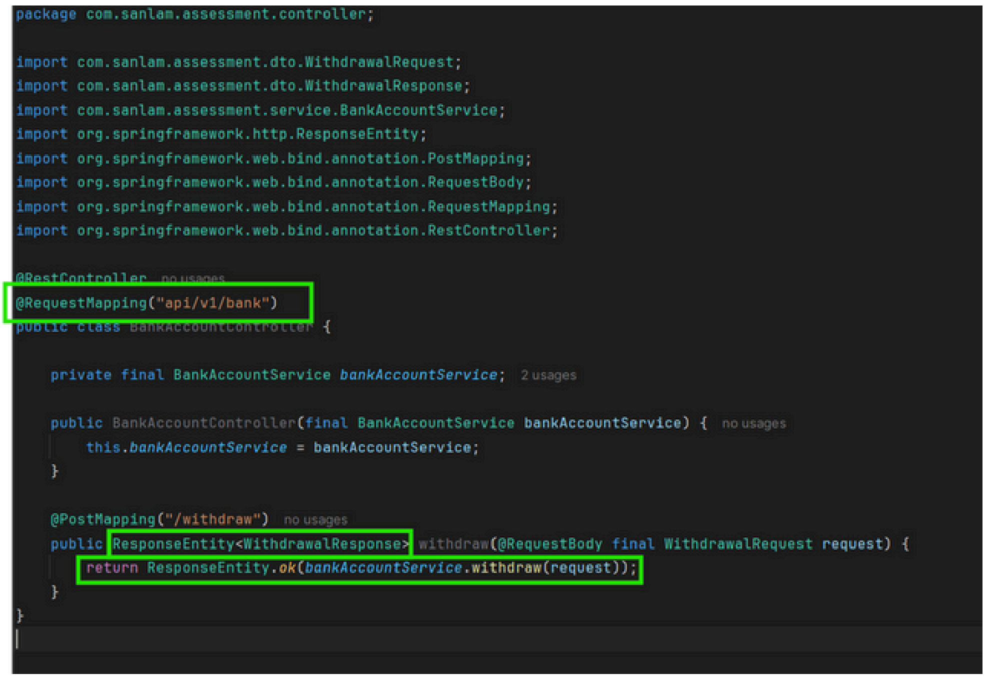
```java
package com.sanlam.assessment.dto;                                    ⚠2

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data  no usages
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class WithdrawalEvent {
    private BigDecimal amount;
    private Long accountId;
    private String status;

    public String toJson() {  no usages
        return String.format("{\"amount\":\"%s\",\"accountId\":%d,\"status\":\"%s\"}", amount, accountId, status);
    }
}
```

CONTROLLER

The controller layer has been restructured to follow RESTful best practices and ensure a clear separation of concerns. API versioning has been introduced directly in the URL path (e.g., /api/v1/withdraw), enabling backward compatibility and future expansion of the API.

The controller now delegates business logic and data access responsibilities to the service and repository layers, respectively. This improves code modularity and testability by ensuring that the controller is solely responsible for handling HTTP requests and responses.

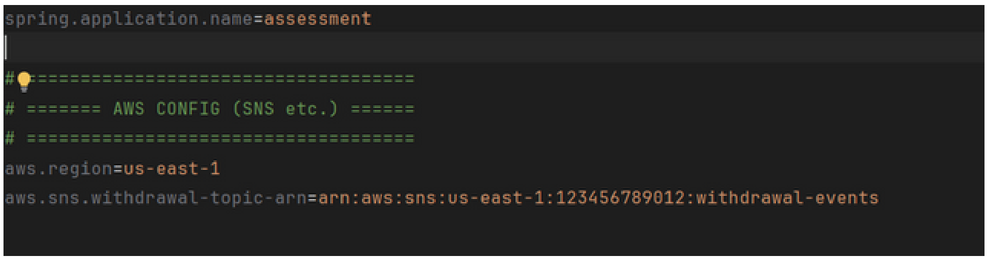
Additionally, the response has been formalized. Instead of returning a raw string, the endpoint now returns a ResponseEntity wrapping a structured WithdrawalResponse DTO. This change provides better HTTP semantics (e.g., status codes, headers) and aligns with modern API design standards.



```java
package com.sanlam.assessment.controller;

import com.sanlam.assessment.dto.WithdrawalRequest;
import com.sanlam.assessment.dto.WithdrawalResponse;
import com.sanlam.assessment.service.BankAccountService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController  no usages
@RequestMapping("api/v1/bank")
public class BankAccountController {

    private final BankAccountService bankAccountService;  2 usages

    public BankAccountController(final BankAccountService bankAccountService) {  no usages
        this.bankAccountService = bankAccountService;
    }

    @PostMapping("/withdraw")  no usages
    public ResponseEntity<WithdrawalResponse> withdraw(@RequestBody final WithdrawalRequest request) {
        return ResponseEntity.ok(bankAccountService.withdraw(request));
    }
}
```

AWS SNS AND REGION ENHANCEMENT

To support integration with AWS services—such as Amazon SNS for publishing withdrawal-related events—dedicated configuration properties have been added to the application.properties file. These configurations ensure that sensitive cloud settings are externalized, allowing the application to remain flexible and cloud-ready.



```properties
spring.application.name=assessment

#==================================
# ======= AWS CONFIG (SNS etc.) ======
# ==================================
aws.region=us-east-1
aws.sns.withdrawal-topic-arn=arn:aws:sns:us-east-1:123456789012:withdrawal-events
```

05. CONCLUSION

In conclusion, the refactoring effort has successfully transformed the withdrawal feature into a cleaner, modular, and production-ready implementation. By adopting established design patterns and leveraging Spring Boot's ecosystem—such as service separation, Spring Data JPA, and Lombok—the codebase is now more maintainable, testable, and scalable.

Key improvements include the decoupling of business logic from the controller, externalization of configuration, standardized logging for observability, and the elimination of boilerplate code through modern tooling. These changes align with best practices in software engineering and set a strong foundation for future enhancements and team collaboration.

As a result, the system is now better positioned to support growth, handle complexity, and meet evolving business requirements with greater confidence and stability.

Goal	Description	Addressed
Separation of conerns	Decouple the business, data layer and handle API into distinct layers (Controller → Service → Repository). This will improve modularity, maintainability and testability.	✓
Input Validation and Data Integrity	Enforce validation rules to ensure only legitimate operations are allowed such as rejecting negative amounts. This will assist in preventing invalid transition states and error propagation reduction. Add @Transactional annotation on the business logic method to ensure that all transactions succeed or fail at once.	✓
Observability and Diagnostics	Introduce structured logging, error handling and traceability to enable easier debugging, monitoring and incidents response in different environments.	✓