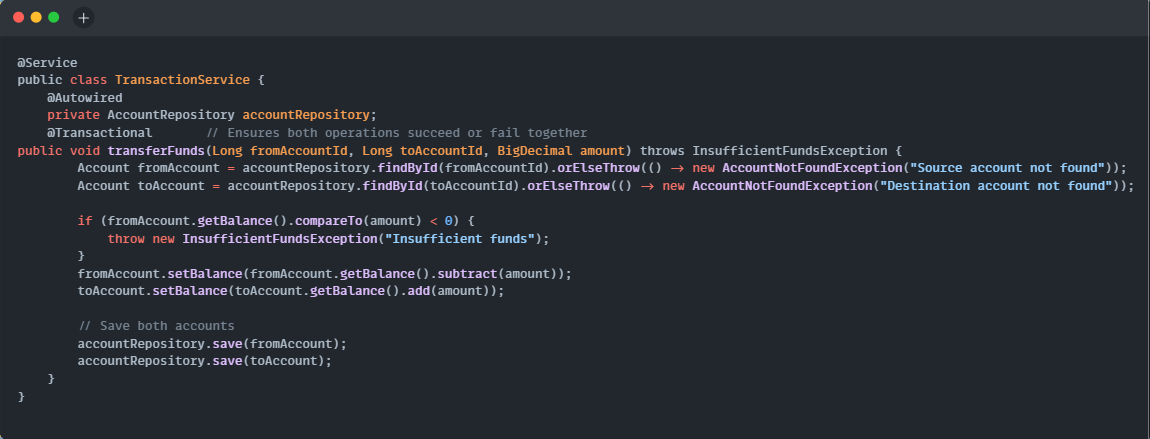
**Spring Boot Transactions**

* In Spring Boot, transactions ensure that a series of operations on a database either complete fully or do not happen at all, maintaining data integrity. In a financial system, this is critical to ensure the accuracy of operations like money transfers, loan processing, or portfolio management.
* Transactions are typically managed using the @Transactional annotation. Spring Boot provides declarative transaction management that works with multiple databases, including those in distributed environments.
* Example: Consider a scenario in a finance application where a user transfers money between two accounts.



**Explanation**:

* The @Transactional annotation ensures that if any part of the transaction fails (e.g., insufficient funds), the entire operation is rolled back, meaning no money is deducted or added.
* The balance is checked, and funds are moved between accounts. If an exception is thrown, Spring will automatically roll back the changes.

**Important points**:

* Propagation: Spring supports different propagation settings like REQUIRED, REQUIRES\_NEW, etc., which determine how transactions interact with one another.
* Isolation Levels: This control how transaction changes are visible to other transactions, important in financial systems to avoid dirty reads or phantom reads.
* Rollback: You can specify exceptions to roll back on (e.g., @Transactional(rollbackFor = InsufficientFundsException.class)).

**TYPES**:

1. **Transaction Propagation Types**: Propagation defines how a method should participate in a transaction, especially when called from a method that already has an active transaction. Spring provides multiple propagation options that you can configure with the @Transactional annotation.

|  |  |
| --- | --- |
| **REQUIRED** (default):   * If there is an existing transaction, the method will join it. * If there is no existing transaction, a new one will be created. | Example: A financial service that needs to debit and credit two accounts can use REQUIRED to ensure both operations happen in a single transaction.  @Transactional(propagation = Propagation.REQUIRED)  public void transferFunds() {  debitAccount(); // Uses the same transaction  creditAccount(); // Uses the same transaction  } |
| **REQUIRES\_NEW**:   * Always creates a new transaction, suspending the current one if it exists. | Example: A financial service that needs to debit and credit two accounts can use REQUIRED to ensure both operations happen in a single transaction.  @Transactional(propagation = Propagation.REQUIRED)  public void transferFunds() {  debitAccount(); // Uses the same transaction  creditAccount(); // Uses the same transaction  } |
| **SUPPORTS**:   * If an existing transaction exists, the method will join it. * If there’s no active transaction, it will execute non-transactionally. | Example: Fetching read-only account balance information may use SUPPORTS because it does not require a transaction unless it's part of a larger process that is already within a transaction.  @Transactional(propagation = Propagation.SUPPORTS, readOnly = true)  public Account getAccountBalance(Long accountId) {  // No transaction needed for a read-only operation  } |
| **NOT\_SUPPORTED**:   * Always runs outside of a transaction, suspending any active transaction. | Example: In a finance app, some non-critical processes like generating reports may not require transactions.  @Transactional(propagation = Propagation.NOT\_SUPPORTED)  public void generateReport() {  // Runs without any transaction  } |
| **MANDATORY**:   * Requires an existing transaction, and if there’s none, an exception is thrown. | Example: In a financial system, you can mandate that certain sensitive operations (e.g., updating account limits) must only occur within an ongoing transaction.    @Transactional(propagation = Propagation.MANDATORY)  public void updateAccountLimit() {  // Throws exception if called outside of an existing transaction  } |
| **NEVER**:   * Ensures that the method is never executed within a transaction. If a transaction exists, an exception will be thrown. | Example: A financial service might use this when executing highly asynchronous or external calls that shouldn’t be tied to transaction management.    @Transactional(propagation = Propagation.NEVER)  public void externalServiceCall() {  // Must not be inside a transaction  } |
| **NESTED**:   * Executes within a nested transaction. * If the nested transaction fails, only the nested part is rolled back, but the outer transaction may still succeed. | Example: In a batch financial operation, you may have a nested transaction to process individual transactions. If one fails, you may still want to proceed with the rest.    @Transactional(propagation = Propagation.NESTED)  public void processSingleTransaction() {  // Nested transaction that can be rolled back independently  } |

1. **Transaction Isolation Levels**: Isolation levels determine how transaction changes are visible to other transactions and how concurrent transactions are handled.

|  |  |
| --- | --- |
| **READ\_UNCOMMITTED**:   * No isolation: dirty reads are allowed (one transaction can read uncommitted data from another). | Example: In financial apps, this level is rarely used due to potential issues of reading uncommitted, potentially incorrect data.    @Transactional(isolation = Isolation.READ\_UNCOMMITTED)  public void fetchMarketPrices() {  // Can see uncommitted data from other transactions  } |
| **READ\_COMMITTED (default in most databases):**   * Ensures that a transaction only reads committed data. Dirty reads are not allowed. | Example: A bank’s loan processing system might use this to ensure that it only processes data that has been committed (no half-completed updates).    @Transactional(isolation = Isolation.READ\_COMMITTED)  public void processLoanApplication() {  // Only reads data that has been committed  } |
| **REPEATABLE\_READ:**   * Ensures that if a transaction reads a row, the data will not change until the transaction is complete. * This prevents non-repeatable reads. | Example: A system calculating a user’s portfolio value may use this level to ensure that the value remains stable while processing.    @Transactional(isolation = Isolation.REPEATABLE\_READ)  public BigDecimal calculatePortfolioValue(Long userId) {  // Data won't change during this transaction  } |
| **SERIALIZABLE:**   * The strictest isolation level, it ensures that transactions are executed one after another (as if they were serialized). * This prevents phantom reads. | Example: In high-value transactions, like large transfers between financial institutions, you may want the strictest isolation to prevent any race conditions.    @Transactional(isolation = Isolation.SERIALIZABLE)  public void processHighValueTransfer() {  // No other transactions can interfere while this is in progress  } |

1. **Other Important Attributes of Transactions**
2. **Timeout**: Sets the timeout period (in seconds) for a transaction before it’s rolled back.

@Transactional(timeout = 30) // Transaction will be rolled back if it exceeds 30 seconds

public void processTransaction() {

// Logic here

}

1. **Rollback Rules**: You can specify the exceptions that will trigger a rollback or prevent one.

@Transactional(rollbackFor = CustomException.class) // Rollback on specific exception

public void processWithRollback() throws CustomException {

// Logic here

}

1. **Read-Only Transactions**: For queries, a transaction can be marked as read-only to optimize performance by hinting to the database that no data will be modified.

@Transactional(readOnly = true)

public List<Account> getAccounts() {

// No data modification allowed in this transaction

}

1. **Programmatic Transaction Management**

* While the declarative approach using @Transactional is common, you can also manage transactions programmatically with the TransactionTemplate or PlatformTransactionManager if you need more control.
* Example:

@Service

public class ProgrammaticTransactionService {

@Autowired

private PlatformTransactionManager transactionManager;

public void processTransaction() {

TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);

transactionTemplate.execute(status -> {

// Your transactional code here

return null;

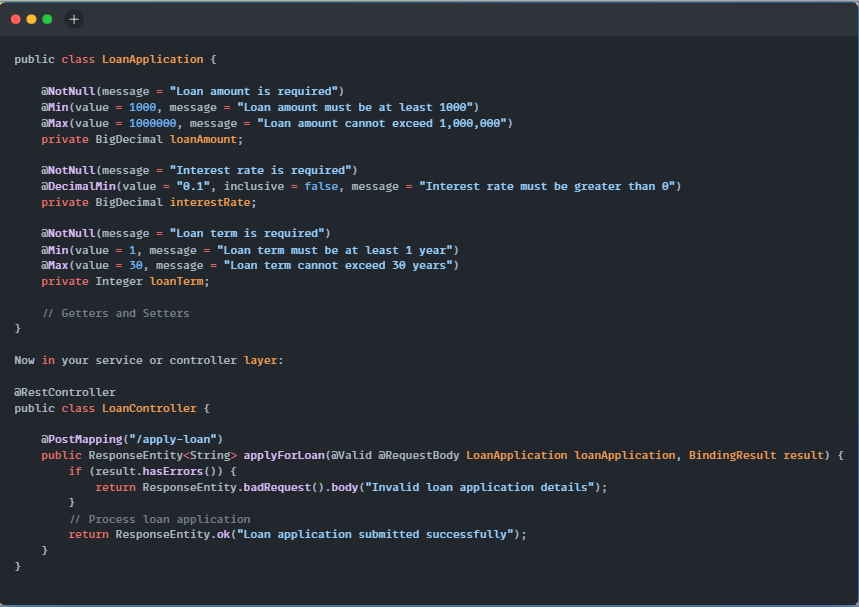
});

}

}

**Spring Boot Validation**

* Validation is crucial in a finance application to ensure the correctness and completeness of data before processing. Spring Boot provides robust validation support using JSR-303/JSR-380 (Bean Validation). You can annotate your model classes with validation constraints like @NotNull, @Min, @Max, etc.
* Example: Imagine a scenario where a user creates a new loan application. The system needs to ensure that the loan amount is within a valid range, the interest rate is positive, and other fields are properly filled.



**Explanation**:

* The LoanApplication class is annotated with validation constraints.
* @NotNull ensures critical fields like loanAmount, interestRate, and loanTerm are present.
* @Min and @Max enforce valid ranges for loan amounts and terms.
* @DecimalMin ensures that the interest rate is positive.
* The @Valid annotation in the controller ensures that the incoming request is validated against these rules before proceeding. If the validation fails, the BindingResult captures the errors, and a bad request is returned.

**Important points:**

* Custom Validation: You can create custom validation annotations (e.g., validating a loan’s eligibility based on credit score or existing loans).
* Group Validation: Different validation groups can be applied based on the state of the object (e.g., new loan application vs. loan modification).

**Types:**

1. **Bean Validation (JSR-303/JSR-380)**
2. **Basic Validation Annotations:**
3. **@NotNull**:

* Ensures that a field is not null.
* Example: A loanAmount field should never be null in a loan application.

@NotNull(message = "Loan amount is required")

private BigDecimal loanAmount;

1. **@Min / @Max:**

* Ensures that numeric values fall within a specified range.
* Example: Ensure that the loan amount is at least $1,000 and does not exceed $1,000,000.

@Min(value = 1000, message = "Loan amount must be at least 1000")

@Max(value = 1000000, message = "Loan amount cannot exceed 1,000,000")

private BigDecimal loanAmount;

1. **@Size**:

* Validates the size of strings or collections.
* Example: Ensure the account holder’s name is between 2 and 50 characters.

@Size(min = 2, max = 50, message = "Name must be between 2 and 50 characters")

private String accountHolderName;

1. **@DecimalMin / @DecimalMax:**

* Ensures that floating-point numbers or BigDecimal values are within a range.
* Example: Ensure that the interest rate for a loan is greater than 0 but less than or equal to 20%.

@DecimalMin(value = "0.1", inclusive = false, message = "Interest rate must be greater than 0")

@DecimalMax(value = "20.0", message = "Interest rate cannot exceed 20%")

private BigDecimal interestRate;

1. **@Email:**

* Ensures that a field contains a valid email address.
* Example: Validate the email address of a customer when creating an account.

@Email(message = "Invalid email format")

private String email;

1. **@Pattern:**

* Ensures that a string matches a regular expression.
* Example: Validate that an IBAN (International Bank Account Number) follows a specific format.

@Pattern(regexp = "[A-Z0-9]{15,34}", message = "Invalid IBAN format")

private String iban;

1. **Custom Validation**

* Business logic can be complex, and the basic annotations may not cover all use cases. You can create custom validators to meet specific business needs.
* Example: Let's say that in a finance application, a loan can only be granted if the applicant’s credit score is above 650. You can create a custom validator for this.
* Step 1: Define the custom annotation.

@Target({ElementType.FIELD})

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy = CreditScoreValidator.class)

public @interface ValidCreditScore {

String message() default "Invalid credit score";

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

}

* Step 2: Implement the validation logic.

public class CreditScoreValidator implements ConstraintValidator<ValidCreditScore, Integer> {

@Override

public boolean isValid(Integer value, ConstraintValidatorContext context) {

return value != null && value >= 650; // Credit score must be at least 650

}

}

* Step 3: Use the custom validation in the model.

public class LoanApplication {

@ValidCreditScore

private Integer creditScore;

// Other fields

}

1. **Group Validation**

* Group validation allows you to apply different validation rules depending on the context, such as when an entity is being created or updated.
* Example: In a finance application, certain fields may be mandatory during account creation but optional during account updates.

public class Account {

@NotNull(groups = OnCreate.class, message = "Account holder's name is required for account creation") private String accountHolderName;

@NotNull(groups = {OnCreate.class, OnUpdate.class}, message = "Email is required") private String email; // Validation groups public interface OnCreate {}

public interface OnUpdate {}

}

* In the controller, you specify which group to validate against:

@PostMapping("/create-account")

public ResponseEntity<String> createAccount(@Validated(OnCreate.class) @RequestBody Account account) {

// Handle account creation

}

1. **Method-Level Validation**

* Spring Boot supports method-level validation, which allows you to validate method parameters or return values using annotations.
* Example: Validate the loan application parameters in a service layer.

@Service

public class LoanService {

public LoanApplication applyForLoan(

@NotNull(message = "Loan amount is required") @Min(value = 1000, message = "Loan amount must be at least 1000") BigDecimal loanAmount,

@NotNull(message = "Interest rate is required") @DecimalMin(value = "0.1", inclusive = false, message = "Interest rate must be greater than 0") BigDecimal interestRate

) {

// Process loan application

}

}

* To enable method validation, you must annotate the service class with @Validated.

@Service

@Validated

public class LoanService {

// Method with validation

}

1. **Cross-Field Validation**

* Sometimes, the validation of one field depends on the value of another field. For example, in a finance project, if the loan amount exceeds a certain threshold, additional documentation may be required.
* You can create a custom cross-field validator to handle such cases.
* Example: If the loan amount is greater than $500,000, the highValueLoanDocumentation field must not be null.

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy = HighValueLoanValidator.class)

public @interface ValidHighValueLoan {

String message() default "High-value loans require additional documentation";

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

}

* Step 2: Implement the cross-field validation logic.

public class HighValueLoanValidator implements ConstraintValidator<ValidHighValueLoan, LoanApplication>{

@Override

public boolean isValid(LoanApplication loanApplication, ConstraintValidatorContext context) {

if (loanApplication.getLoanAmount().compareTo(new BigDecimal(500000)) > 0) {

return loanApplication.getHighValueLoanDocumentation() != null;

}

return true;

}

}

* Step 3: Apply the cross-field validation to the LoanApplication class.

@ValidHighValueLoan

public class LoanApplication {

private BigDecimal loanAmount;

private String highValueLoanDocumentation;

// Other fields

}

1. **Validation Groups for Different Scenarios**

* You can create different validation groups for different operations, such as validating a loan application in different stages like pre-approval, approval, and disbursement.
* Pre-Approval Validation: Validates basic loan criteria such as loan amount and credit score.
* Approval Validation: Validates additional fields such as interest rate and collateral.
* Disbursement Validation: Validates the account details and legal documentation.

public class LoanApplication {

@NotNull(groups = PreApproval.class, message = "Loan amount is required for pre-approval")

@Min(groups = PreApproval.class, value = 1000, message = "Loan amount must be at least 1000")

private BigDecimal loanAmount;

@NotNull(groups = Approval.class, message = "Interest rate is required for approval")

private BigDecimal interestRate;

@NotNull(groups = Disbursement.class, message = "Disbursement account details are required")

private String disbursementAccount;

public interface PreApproval {}

public interface Approval {}

public interface Disbursement {}

}

1. **Programmatic Validation**

* Spring Boot also allows for programmatic validation, where you can manually trigger validation logic.
* Example: Validate a loan application manually before processing it.

@Autowired

private Validator validator;

public void processLoanApplication(LoanApplication loanApplication) {

Set<ConstraintViolation<LoanApplication>> violations = validator.validate(loanApplication);

if (!violations.isEmpty()) {

throw new ValidationException("Loan application validation failed: " + violations);

}

// Proceed with processing

}