Assignment: Understanding Spring Boot Data JPA Transaction Propagation Techniques with Use Case Scenarios

Objective: To understand the various transaction propagation behaviors provided by Spring Boot Data JPA, how they are implemented, and where they are applicable by demonstrating them through practical use case scenarios.

Assignment Structure:

1. **Introduction to Spring Boot Data JPA Transaction Propagation**
    - Briefly explain Spring Boot and Spring Data JPA.
    - Discuss the need for transaction management.
    - Introduce the concept of transaction propagation and explain its significance.
2. **Understanding Transaction Propagation**
    - Explain each propagation type in detail:
        - PROPAGATION_REQUIRED
        - PROPAGATION_SUPPORTS
        - PROPAGATION_MANDATORY
        - PROPAGATION_REQUIRES_NEW
        - PROPAGATION_NOT_SUPPORTED
        - PROPAGATION_NEVER
        - PROPAGATION_NESTED
3. **Use Case Scenarios**
    - **PROPAGATION_REQUIRED**
        - Scenario: A Banking Application where fund transfer between accounts requires multiple operations to be successful.
        - Explanation: Discuss why PROPAGATION_REQUIRED is needed and how it ensures atomicity.
    - **PROPAGATION_SUPPORTS**
        - Scenario: A Content Management System where reading operations can occur within or outside a transaction.
        - Explanation: Discuss how PROPAGATION_SUPPORTS is useful for read operations that don't necessarily need a transaction but can still occur within one if it exists.
    - **PROPAGATION_MANDATORY**
        - Scenario: An Order Management System where every operation related to an order must happen within a transaction.
        - Explanation: Explain how PROPAGATION_MANDATORY ensures that there's always a transaction for critical operations.
    - **PROPAGATION_REQUIRES_NEW**
        - Scenario: A Reward System where points calculation and addition should not affect the main transaction of purchasing.
        - Explanation: Discuss why PROPAGATION_REQUIRES_NEW is beneficial to isolate specific operations within their own transactions.
    - **PROPAGATION_NOT_SUPPORTED**
        - Scenario: A Logging System where logging operations should not interfere with the main business operations.
        - Explanation: Show how PROPAGATION_NOT_SUPPORTED allows operations to be performed outside any existing transaction.

- **PROPAGATION_NEVER**
    - Scenario: An Analytics System where data aggregation and analysis should always happen outside a transaction.
    - Explanation: Discuss how PROPAGATION_NEVER ensures operations always execute non-transactionally.
- **PROPAGATION_NESTED**
    - Scenario: A Booking System where main reservation and auxiliary operations are related but can be rolled back separately.
    - Explanation: Explain how PROPAGATION_NESTED provides granularity in transaction management by allowing nested transactions.

4. **Implementation and Testing**
    - Choose one of the scenarios and implement it using Spring Boot Data JPA.
    - Show how different propagation behaviors work by making changes in the application and observing the results.
5. **Conclusion**
    - Recap the understanding of different transaction propagation techniques and their use-cases.
    - Discuss some best practices and things to consider while choosing the appropriate propagation technique.

Expected Outcome:

- Thorough understanding of each transaction propagation type in Spring Boot Data JPA.
- Ability to select the appropriate propagation type depending upon the use case.
- Ability to implement and test a practical scenario using Spring Boot Data JPA transaction propagation.

Resources:

- Spring Boot Documentation
- Spring Data JPA Documentation
- Spring Transaction Management

Assessment Criteria:

- Clarity of explanation of each propagation type.
- Relevance and clarity of the use case scenarios.
- Correctness and completeness of the chosen implementation.
- Ability to explain the observed results in the implementation.

Solutions

Let's use the Banking Application use case where we will be implementing **PROPAGATION_REQUIRED** propagation behavior using Spring Boot and Spring Data JPA. This solution will demonstrate a fund transfer operation between two accounts that requires multiple operations to be successful. The operations are atomic, meaning that if one operation fails, the entire transaction should be rolled back.

Step 1: Setup the Project

First, we need to set up a new Spring Boot project. You can use Spring Initializr to bootstrap your application. Choose the following dependencies:

- Spring Web
- Spring Data JPA

- H2 Database (or any other database you're comfortable with)

Step 2: Define the Account Entity
The **Account** entity will represent a bank account in the application.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Account {

   @Id
   @GeneratedValue
   private Long id;

   private Double balance;

   // getters and setters
}
```

Step 3: Create the Account Repository

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface AccountRepository extends JpaRepository<Account, Long> {
}
```

Step 4: Create the BankService
The **BankService** class will contain the **transferFunds** method, which will be responsible for the fund transfer operation.

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Service
public class BankService {

   private final AccountRepository accountRepository;

   public BankService(AccountRepository accountRepository) {
      this.accountRepository = accountRepository;
   }

   @Transactional(propagation = Propagation.REQUIRED)
   public void transferFunds(Long fromAccountId, Long toAccountId, Double amount) {
      Account fromAccount = accountRepository.findById(fromAccountId)
            .orElseThrow(() -> new RuntimeException("Account not found"));
      Account toAccount = accountRepository.findById(toAccountId)
            .orElseThrow(() -> new RuntimeException("Account not found"));
```

```
        fromAccount.setBalance(fromAccount.getBalance() - amount);
        toAccount.setBalance(toAccount.getBalance() + amount);

        accountRepository.save(fromAccount);
        accountRepository.save(toAccount);
    }
}
```

In this **transferFunds** method, we're setting the propagation behavior to **Propagation.REQUIRED** using the **@Transactional** annotation. This makes the entire method execute within a single transaction. If any exception occurs during the execution of this method (e.g., due to insufficient funds), the transaction will be rolled back, and no changes will be made to the database.

Step 5: Test the Application

You can test the application by creating some accounts and then trying to transfer funds between them. If the **transferFunds** method is able to complete successfully, the balance of the **fromAccount** should decrease by the transferred amount and the balance of the **toAccount** should increase by the same amount. If there's an exception (like insufficient funds), none of the accounts' balances should change.

This use case demonstrates how **PROPAGATION_REQUIRED** can be used in Spring Boot Data JPA to ensure atomicity of transactions. Similar implementations can be created for the other propagation behaviors by modifying the **@Transactional** annotation and the operations within the **transferFunds** method.

PROPAGATION_SUPPORTS: This setting means that the current method should run within a transaction if one is already running. Otherwise, it can run without a transaction.

In the context of our banking application, consider a scenario where you want to log the fund transfer but only if there's already a transaction running:

```
@Transactional(propagation = Propagation.SUPPORTS)
public void logTransfer(Long fromAccountId, Long toAccountId, Double amount) {
    // Logging code here
}
```

PROPAGATION_MANDATORY: This setting means that the current method must run within a transaction. If there isn't one running, an exception will be thrown.

In our banking application, this could be used for a method that's verifying the transfer:

```
@Transactional(propagation = Propagation.MANDATORY)
public void verifyTransfer(Long fromAccountId, Long toAccountId, Double amount) {
    // Verification code here
}
```

PROPAGATION_REQUIRES_NEW: This setting creates a new transaction, suspending the current one if it exists. This can be used for operations that should not be influenced by the current transaction.

In our banking application, we could use this propagation for an audit log that we want to commit regardless of the main transaction result:

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void auditLog(Long fromAccountId, Long toAccountId, Double amount) {
   // Auditing code here
}
```

PROPAGATION_NOT_SUPPORTED: This means that the current method should not run within a transaction. If there's an existing transaction, it's suspended.
In our banking application, we might have a method for sending an email notification. We'd want this to happen regardless of the transaction status:

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void sendNotification(Long fromAccountId, Long toAccountId, Double amount) {
   // Notification code here
}
```

PROPAGATION_NEVER: This means the current method should not run within a transaction. If there's a transaction already running, an exception is thrown.
For our banking application, we could imagine a method to check some external API for currency exchange rates. This method should not run within a transaction:

```
@Transactional(propagation = Propagation.NEVER)
public Double getExchangeRate(String currencyFrom, String currencyTo) {
   // API check code here
}
```

PROPAGATION_NESTED: This creates a nested transaction if one is already running, or behaves like PROPAGATION_REQUIRED otherwise.
In our banking application, this could be useful if we want to make multiple transfers as part of a larger, outer transaction. Each transfer would be a nested transaction:

```
@Transactional(propagation = Propagation.NESTED)
public void makeTransfer(Long fromAccountId, Long toAccountId, Double amount) {
   // Transfer code here
}
```

A complex scenario where an E-commerce website allows customers to place orders. For simplicity, the process will involve:

1. Creating a new order (requires a new transaction each time)
2. Updating product inventory (should be part of the order transaction)

3. Sending a confirmation email (should not be part of the transaction)
4. Auditing order placement (should be part of its own transaction)

Step 1: Define Entities and Repositories

Create **Order**, **Product**, and **Audit** entities along with their respective repositories. For brevity, I am not providing full code for these classes. Your entities would include necessary fields such as **id**, **quantity**, etc., and repositories would extend **JpaRepository**.

Step 2: Create Order Service

```
@Service
public class OrderService {

   private final OrderRepository orderRepository;
   private final ProductService productService;
   private final EmailService emailService;
   private final AuditService auditService;

   // Constructor with necessary autowiring

   @Transactional(propagation = Propagation.REQUIRES_NEW)
   public void placeOrder(Order order) {
      // Step 1: Create a new order
      orderRepository.save(order);

      // Step 2: Update product inventory
      productService.updateInventory(order.getProduct(), order.getQuantity());

      // Step 3: Send a confirmation email (should not be part of the transaction)
      emailService.sendOrderConfirmation(order);

      // Step 4: Audit order placement (should be part of its own transaction)
      auditService.auditOrderPlacement(order);
   }
}
```

Step 3: Create Product Service

```
@Service
public class ProductService {

   private final ProductRepository productRepository;

   // Constructor with necessary autowiring

   @Transactional(propagation = Propagation.MANDATORY)
   public void updateInventory(Product product, int quantity) {
      product.setInventory(product.getInventory() - quantity);
      productRepository.save(product);
   }
}
```

Step 4: Create Email Service

```
@Service
public class EmailService {

   @Transactional(propagation = Propagation.NOT_SUPPORTED)
   public void sendOrderConfirmation(Order order) {
      // Email sending code here
   }
}
```

Step 5: Create Audit Service

```
@Service
public class AuditService {

   private final AuditRepository auditRepository;

   // Constructor with necessary autowiring

   @Transactional(propagation = Propagation.REQUIRES_NEW)
   public void auditOrderPlacement(Order order) {
      Audit audit = new Audit(order.getId(), "Order Placed");
      auditRepository.save(audit);
   }
}
```

In this setup:

- **placeOrder** starts a new transaction (using **Propagation.REQUIRES_NEW**) for each order placement.
- **updateInventory** is part of the current order transaction (using **Propagation.MANDATORY**) so if anything goes wrong with the order placement, the inventory won't be updated.
- **sendOrderConfirmation** sends an email and is independent of the transaction (using **Propagation.NOT_SUPPORTED**). This is because sending or failing to send an email should not interfere with the order transaction.
- **auditOrderPlacement** records an audit log for order placement, and it runs in its own separate transaction (using **Propagation.REQUIRES_NEW**). This way, even if the order placement fails for some reason, the audit record about an attempt being made will be saved.

Remember to handle exceptions appropriately in your real application to make sure that the service behaves as expected. This example demonstrates how different propagation types can be combined in a complex transactional operation.

Detailed explanation

n the given example, we've used four propagation behaviors in the context of an e-commerce system, where a customer places an order, the inventory of the product gets updated, a

confirmation email is sent to the customer, and finally an audit log is generated for the order placement.
Here's an in-depth explanation for each propagation behavior:

1. **Propagation.REQUIRES_NEW** in **placeOrder()** method: When the **placeOrder()** method is called, regardless of whether a transaction is already in progress or not, a new transaction starts. This is because of the **Propagation.REQUIRES_NEW** setting. This ensures that the order placement is always handled in a new, independent transaction.
2. **Propagation.MANDATORY** in **updateInventory()** method: After the order has been created, the system needs to update the product inventory. The **updateInventory()** method is marked with **Propagation.MANDATORY** which means it must be called within an existing transaction context, or else it will throw an exception. In our scenario, it's called after **placeOrder()**, so it uses the transaction started by **placeOrder()**. If everything goes fine, the inventory gets updated, otherwise (for example, if there isn't enough inventory), the order placement and inventory update would both roll back, leaving the system in a consistent state.
3. **Propagation.NOT_SUPPORTED** in **sendOrderConfirmation()** method: This method is responsible for sending a confirmation email. This operation doesn't need to be transactional as it doesn't change any shared application data that would require rollback on failure. Therefore, it's marked with **Propagation.NOT_SUPPORTED**. This propagation setting will suspend any existing transaction, execute this method, and then resume the transaction. Even if sending the email fails, it won't affect the order placement transaction.
4. **Propagation.REQUIRES_NEW** in **auditOrderPlacement()** method: The auditing operation is marked with **Propagation.REQUIRES_NEW**, meaning it will always run in a new transaction. We want to keep auditing separate from the main transaction because, even if the main transaction fails and rolls back (maybe there wasn't enough inventory), we still want to record that an attempt to place an order was made.

This example showcases how different propagation behaviors can be combined to manage complex transaction scenarios effectively. You'll choose propagation types based on your requirements and the level of independence or interaction you want between methods involved in the transaction. It's all about ensuring data integrity and consistency in your application.