Open in app ↗   (Sign up)   Sign in

◖◗ Medium   🔍 Search   ✎ Write   👤

Mastering @Transactional Annotations in Spring Boot

👤 Hiten Pratap Singh · Follow
Published in hprog99 · 6 min read · May 8, 2023

👏 224   💬 2   🔖 ▶ ⬆

The @Transactional annotation is used to define the transaction boundaries at the method level in your Spring Boot application. When you apply this annotation to a method, Spring will manage the transaction on your behalf, taking care of beginning and committing or rolling back the transaction as needed. This dramatically simplifies your code and reduces the chances of bugs related to transaction handling.

Let's start with a simple example. Suppose we have an application that manages users and their accounts. We will create a service class to handle creating and updating user accounts.

```java
@Service
public class AccountService {

    @Autowired
    private AccountRepository accountRepository;

    @Autowired
    private UserRepository userRepository;

    @Transactional
    public void createOrUpdateAccount(Account account, User user) {
        userRepository.save(user);
        accountRepository.save(account);
    }
}
```

In this example, we have annotated the `createOrUpdateAccount` method with @Transactional. This ensures that both the user and account are saved within the same transaction. If either operation fails, Spring will automatically roll back the transaction, ensuring data consistency.

Propagation

The @Transactional annotation provides different propagation behaviors that determine how transactions are managed when one method calls another. You can define the propagation behavior using the `propagation` attribute. Here are the available options:

- **REQUIRED:** This is the default behavior. If there's an existing transaction, the method will participate in it. If there's no transaction, a new one will be created.

- **REQUIRES_NEW:** The method will always run in a new transaction, suspending the existing one if available.

- **SUPPORTS:** The method will run in the existing transaction if available, otherwise it will run without a transaction.

- **NOT_SUPPORTED:** The method will always run without a transaction, suspending the existing one if available.

- **MANDATORY:** The method must run within an existing transaction, throwing an exception if there's no transaction.

- **NEVER:** The method must run without a transaction, throwing an exception if there's an existing transaction.

- **NESTED:** The method will run within a nested transaction if there's an existing transaction, otherwise, it will run in a new transaction.

Here's an example of using the `propagation` attribute:

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void createAccount(Account account) {
    accountRepository.save(account);
}
```

Isolation

The `isolation` attribute of the @Transactional annotation allows you to define the transaction isolation level, which determines how concurrent transactions interact with each other. Here are the available options:

- **DEFAULT:** Uses the default isolation level of the underlying database.

- **READ_UNCOMMITTED:** Allows dirty reads, non-repeatable reads, and phantom reads.

- **READ_COMMITTED:** Prevents dirty reads but allows non-repeatable reads and phantom reads.

- **REPEATABLE_READ:** Prevents dirty reads and non-repeatable reads but allows phantom reads.

- **SERIALIZABLE:** Prevents dirty reads, non-repeatable reads, and phantom reads, providing the highest level of isolation.

Example of using the `isolation` attribute:

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public void transferFunds(Account fromAccount, Account toAccount, BigDecimal amo
    fromAccount.debit(amount);
```

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

Timeout and Read-only

The `timeout` attribute of the @Transactional annotation allows you to specify the maximum number of seconds a transaction can take before it times out and gets rolled back. This can be useful in preventing long-running transactions from blocking resources indefinitely.

Example of using the `timeout` attribute:

```
    @Transactional(timeout = 10) // 10 seconds
    public void transferFunds(Account fromAccount, Account toAccount, BigDecimal amo
        fromAccount.debit(amount);
        toAccount.credit(amount);
        accountRepository.save(fromAccount);
        accountRepository.save(toAccount);
    }
```

The `readOnly` attribute of the @Transactional annotation indicates whether the transaction is read-only. If set to `true`, the transaction manager optimizes the transaction for read operations, potentially improving performance. It's important to note that marking a transaction as read-only when it involves write operations may result in unexpected behavior.

Example of using the `readOnly` attribute:

```
@Transactional(readOnly = true)
public List<Account> getAllAccounts() {
    return accountRepository.findAll();
}
```

Rollback

By default, the @Transactional annotation rolls back the transaction only when a runtime exception is thrown. You can customize this behavior using the `rollbackFor` and `noRollbackFor` attributes. The `rollbackFor` attribute defines a list of exception classes for which the transaction should be rolled back, while the `noRollbackFor` attribute defines a list of exception classes for which the transaction should not be rolled back.

Example of using the `rollbackFor` and `noRollbackFor` attributes:

```
@Transactional(rollbackFor = CustomException.class, noRollbackFor = MinorExcepti
public void processAccount(Account account) throws CustomException, MinorExcepti
    // Code that might throw CustomException or MinorException
}
```

Advantages of @Transactional

Using the @Transactional annotation in Spring Boot applications provides several advantages, making it easier to manage and maintain transactional behavior in your code. Some of the key benefits include:

1. **Declarative transaction management:** With the @Transactional annotation, you can manage transactions at the method level without

writing any explicit code for transaction handling. This declarative approach simplifies your code and makes it more readable, reducing the chances of bugs related to transaction management.

2. **Consistent transaction handling:** By using the @Transactional annotation, you can ensure consistent transaction handling across your application. This allows you to follow a uniform approach to transaction management, which makes the code easier to understand and maintain.

3. **Flexibility and configurability:** The @Transactional annotation provides several attributes that allow you to customize transaction behavior, such as propagation, isolation, timeout, read-only, and rollback rules. This flexibility enables you to optimize transaction handling for different use cases and requirements.

4. **Improved performance:** By marking transactions as read-only when appropriate, the transaction manager can optimize the transaction for read operations, potentially improving performance. Additionally, specifying a timeout for transactions helps prevent long-running transactions from blocking resources indefinitely.

5. **Automatic rollback:** With the @Transactional annotation, Spring automatically rolls back the transaction when an exception is thrown, ensuring data consistency. You can also customize the rollback behavior using the `rollbackFor` and `noRollbackFor` attributes, giving you fine-grained control over transaction handling.

6. **Integration with Spring ecosystem:** The @Transactional annotation is well-integrated with the Spring ecosystem, including Spring Boot, Spring Data, and Spring Security. This seamless integration ensures that transaction management works consistently across various components of your application.

7. **Improved testability:** Using the @Transactional annotation makes it easier to test transactional behavior in your code. You can create test methods with specific transaction configurations and easily verify the expected behavior in different scenarios.

Do's and Don'ts of @Transactional

When using the @Transactional annotation in Spring Boot applications, it's essential to follow certain best practices and avoid common pitfalls. Here are some do's and don'ts to help you use the @Transactional annotation effectively:

Do's

1. **Apply @Transactional at the service layer:** Transactions typically involve multiple operations that need to be executed together. Applying the @Transactional annotation at the service layer ensures that these operations are performed within a single transaction, maintaining data consistency.

2. **Use the appropriate propagation level:** Understand the different propagation behaviors and use the one that best suits your use case. Using the wrong propagation level can lead to undesired transaction behavior and resource management issues.

3. **Set the appropriate isolation level:** Choose the right isolation level based on your application requirements. Higher isolation levels provide better data consistency but may come at the cost of performance. Understand the trade-offs and choose the level that meets your needs.

4. **Use read-only transactions when applicable:** When a transaction only involves read operations, mark it as read-only using the `readOnly`

attribute. This can optimize performance by allowing the transaction manager to apply optimizations specific to read operations.

5. **Define rollback rules:** Customize the rollback behavior using the `rollbackFor` and `noRollbackFor` attributes. Specify the exception classes for which the transaction should be rolled back or not rolled back, providing finer control over transaction handling.

6. **Set a transaction timeout:** Specify a timeout value for transactions using the `timeout` attribute. This helps prevent long-running transactions from blocking resources indefinitely and can improve overall application performance.

Don'ts

1. **Don't use @Transactional on private, protected, or package-private methods:** The @Transactional annotation relies on Spring's AOP proxies, which can only be created for public methods. If you apply @Transactional to non-public methods, it will be silently ignored, and no transaction will be created or managed.

2. **Don't mix transaction management approaches:** Mixing different transaction management approaches, such as programmatic and declarative, can lead to confusion and inconsistencies. Stick to one approach (preferably declarative using @Transactional) for uniformity and easier maintainability.

3. **Don't use @Transactional for non-transactional methods:** Avoid using the @Transactional annotation for methods that don't need transaction management. This can lead to unnecessary overhead and potential performance degradation.

4. **Don't ignore the impact of nested transactions:** Be cautious when using the NESTED propagation level, as it can have performance implications

and may cause deadlocks in certain situations. Understand the implications of nested transactions and use them judiciously.

5. **Don't mark transactions as read-only when they involve write operations:** Marking a transaction as read-only when it involves write operations can lead to unexpected behavior and data inconsistencies. Only use the `readOnly` attribute when a transaction exclusively involves read operations.

By following these do's and don'ts, you can effectively use the @Transactional annotation to manage transactions in your Spring Boot application, ensuring consistent transaction handling and maintainable code.

The @Transactional annotation in Spring Boot is a powerful feature that simplifies transaction management and improves code maintainability. By understanding its various attributes and their implications, you can effectively control transaction behavior in your application, ensuring data consistency and proper resource management.

Spring   Transaction Spring   Transactions   Spring Boot   Java

Written by Hiten Pratap Singh

429 Followers · Editor for hprog99

Passionate programmer with a gamer's soul

More from Hiten Pratap Singh and hprog99

Events in Spring

Hiten Pratap Singh in hprog99

Mastering Events in Spring Boot: A Comprehensive Guide

Spring Boot Events are a part of the Spring Framework's context module. These events...

6 min read · May 31, 2023

246   2

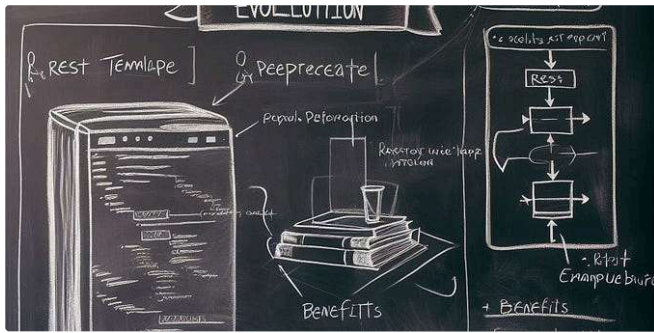Hiten Pratap Singh in hprog99

Set-up Flyway with Spring Boot

Flyway is a database migration tool that helps you manage and apply database changes in ...

3 min read · Dec 27, 2022

158

Hiten Pratap Singh in hprog99

Hiten Pratap Singh in hprog99

Transitioning from RestTemplate to WebClient in Spring Boot: A...

Mastering Kotlin Coroutines with Practical Examples

For many years, Spring Framework's RestTemplate has been the go-to solution fo...

Before we dive into examples, it's crucial to grasp the concept of coroutines. In the...

8 min read · Oct 18, 2023

10 min read · May 23, 2023

See all from Hiten Pratap Singh

See all from hprog99

Recommended from Medium
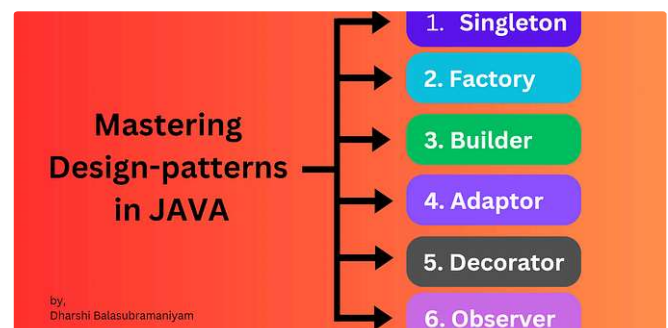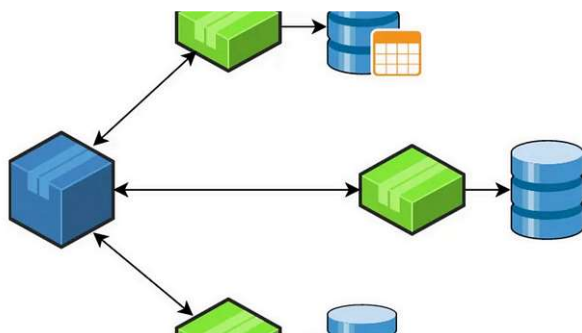
Dharshi Balasubramaniyam in Javarevisited

Distributed Transactions in Spring Boot Microservices: Simplified...

Mastering Design Patterns in Java

Have you ever wondered how complex applications like e-commerce websites or...

Figure 1: Design patterns

10 min read · Dec 20, 2023   16 min read · Feb 26, 2024

👏 81 💬   📑   👏 321 💬 6   📑

Lists

General Coding Knowledge
20 stories · 1153 saves

data science and AI
40 stories · 139 saves
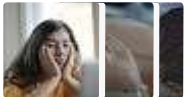
Staff Picks
630 stories · 922 saves

Shubham Yadav

Autowiring vs Constructor Injection in Spring Boot

When it comes to dependency injection in Spring Boot, two popular approaches are...

3 min read · Jan 1, 2024
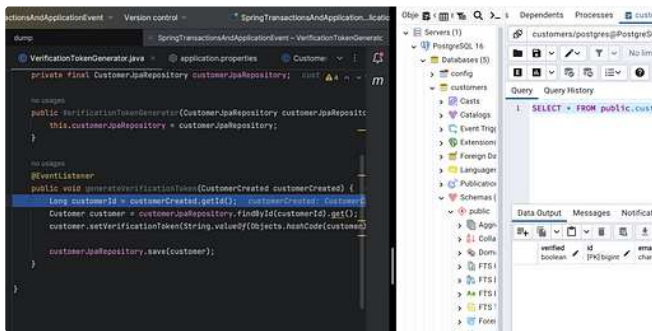
👏 12   💬 1

Nayana Weligalla

Applying @PrePersist and @PreUpdate in JPA

JPA Also Known as Jakarta Persistence API (formerly Java Persistence API) is a...

4 min read · Nov 28, 2023

👏 8   💬

Ishan Soni

Using Spring Application Events within Transactional Contexts

This post is inspired by this talk given by Bartłomiej Słota. I highly recommend...

4 min read · Dec 20, 2023

👏 8   💬

Ajay Rathod

Top 60 Spring-Framework Interview Questions for Java...

Hey there! If you're a Java Developer, mastering the Spring framework is crucial fo...
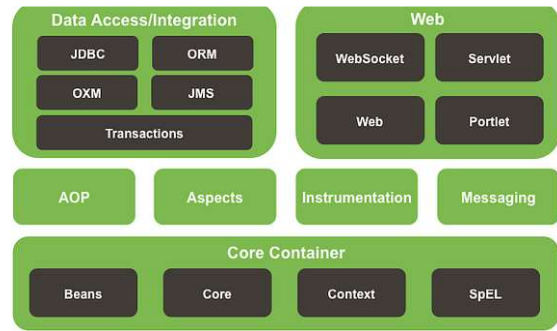
39 min read · Jan 6, 2024

👏 447   💬 5

See more recommendations