TRANSACTION MANAGEMENT IN SPRING

Sandra Kumi

REVIEWER Mr. Thomas

Transaction Management in Spring

1. Declarative Transaction Management

Spring provides a robust way to manage transactions declaratively using the @Transactional annotation. This approach allows you to define transaction boundaries declaratively in your service layer.

@Transactional Annotation:

Placed on methods or classes.

Propagation types define how transactions behave across methods.

Isolation levels control the visibility of data changes made by other transactions.

Rollback rules specify when to roll back the transaction.

Example:

```
@Service
public class EmployeeService {

    @Transactional
    public void saveEmployee(Employee employee) {
        // Code to save employee
    }
}
```

2. Programmatic Transaction Management

Sometimes, you may need finer control over transactions, which can be achieved programmatically using the PlatformTransactionManager interface.

Example:

```
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;
```

```
@Service
public class EmployeeService {

    private final PlatformTransactionManager transactionManager;

    public EmployeeService(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void saveEmployee(Employee employee) {
        TransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);

        try {
            // Code to save employee
            transactionManager.commit(status);
        } catch (Exception e) {
            transactionManager.rollback(status);
            throw e;
        }
    }
}
```

3. Transaction Propagation and Isolation

Propagation Types:

REQUIRED: Supports the current transaction or creates a new one.

REQUIRES_NEW: Always creates a new transaction.

MANDATORY, NESTED, etc.

Isolation Levels:

READ_COMMITTED: Default level; data cannot be read until committed.

SERIALIZABLE, REPEATABLE_READ, etc.

Example:

```
@Transactional(propagation = Propagation.REQUIRES_NEW, isolation =
Isolation.SERIALIZABLE)
public void processTransaction() {
    // Business logic
}
```

Caching in Spring

1. Enabling Caching

To enable caching in Spring Boot, add the following dependency:

xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

Enable caching with @EnableCaching in your configuration class:

java

```
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableCaching
public class CacheConfig {
    // Cache configuration
}
```

2. Basic Caching Annotations

@Cacheable: Caches the result of a method.

@CacheEvict: Removes an entry from the cache.

@CachePut: Updates the cache without skipping method execution.

Example:

java

```
@Service
public class ProductService {

    @Cacheable("products")
    public Product findProductById(Long id) {
        // Method to find product by ID
    }

    @CacheEvict(value = "products", key = "#id")
    public void deleteProduct(Long id) {
        // Method to delete product
    }
}
```

3. Cache Eviction and Expiration

Configure eviction and expiration policies to keep your cache data fresh and relevant.

@CacheEvict for manual eviction.

Cache Expiration configured within your caching provider (e.g., EhCache, Redis, Caffeine).

Example:

java

```
@CacheEvict(value = "products", allEntries = true)
public void clearCache() {
    // Clears the entire cache
}
```

4. Cache Providers

Spring supports multiple cache providers like EhCache, Caffeine, Redis, etc. Choose a provider based on your application's requirements.

Example (EhCache configuration):

```
<ehcache>
    <cache name="products" timeToLiveSeconds="600" maxEntriesLocalHeap="1000"/>
</ehcache>
```

Conclusion

Spring's transaction management and caching mechanisms are powerful tools for ensuring data integrity and improving application performance. By leveraging these features, you can build robust, scalable applications with minimal effort.