Transaction Management and Caching Lab Report

1. Introduction

This lab focuses on implementing transaction management and caching within a Spring Boot application. The goal is to ensure data consistency through proper transaction handling and to optimize performance using caching mechanisms.

2. Objectives

- Understand and implement both declarative and programmatic transaction management.
- Explore transaction propagation and isolation levels to manage data consistency in concurrent environments.
- Implement caching using Spring Data's `@Cacheable` annotation to enhance application performance.
- Configure cache eviction and expiration policies to ensure data freshness.

3. Implementation Details

3.1. Transaction Management

- **Declarative Transactions:**
    - Implemented using the `@Transactional` annotation on service layer methods.
    - Applied to methods handling critical business operations to ensure atomicity.

```java
@Transactional
public void processOrder(Order order) {
    // Business logic
}
```

Programmatic Transactions:

- Managed using `TransactionManager` for scenarios requiring finer control over transactions.
- Used for complex transactions that involve multiple service methods.

```java
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
TransactionStatus status = transactionManager.getTransaction(def);
try {
    // Business logic
    transactionManager.commit(status);
} catch (Exception ex) {
    transactionManager.rollback(status);
}
```

Transaction Propagation and Isolation Levels:

- **Propagation Levels:** Configured using options like `REQUIRED`, `REQUIRES_NEW`, etc., to control how transactions interact.
- **Isolation Levels:** Applied to handle concurrency issues, such as dirty reads and phantom reads, by using levels like `READ_COMMITTED` and `SERIALIZABLE`.
-

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

Caching with Spring Data

- **@Cacheable Annotation:**
    - Applied to repository methods to cache results and reduce database load.
    -

```java
@Cacheable("doctors")
public List<Doctor> findBySpecialty(String specialty) {
    return doctorRepository.findBySpecialty(specialty);
}
```

ache Eviction and Expiration Policies:

- Configured using `@CacheEvict` to remove outdated data and ensure the cache remains accurate.
- Set up expiration policies within Redis to automatically refresh cached data after a certain period.
-

```java
@CacheEvict(value = "doctors", allEntries = true)
public void deleteDoctor(int id) {
    doctorRepository.deleteById(id);
}
```

4. Results and Observations

- **Transaction Management:**
  - Successfully implemented both declarative and programmatic transaction management.
  - Observed how different propagation and isolation levels affect data consistency and concurrency.
- **Caching:**
  - Implemented caching using the `@Cacheable` annotation, which significantly reduced database load.
  - Configured cache eviction and expiration policies to maintain data accuracy.