Caching and Concurrent Transaction Management

1. Introduction to Caching

What is Caching?

Caching allows you to efficiently reuse previously retrieved or computed data. In computing, a cache is a high-speed data storage layer that stores a subset of data, typically transient in nature. This ensures that future requests for that data are served faster than by accessing the data's primary storage location.

How to Implement Caching in Spring Boot

Add Dependencies:

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

Enable Caching in Spring Boot:

```
1. @Configuration
2. @EnableCaching
3. public class CacheConfig {
4. }
```

2. Spring Boot Default Caching

Spring Boot provides built-in caching mechanisms using annotations:

Annotations Used in Caching:

- **@EnableCaching**: Enables caching in a Spring Boot application. It sets up a CacheManager and creates an in-memory cache using a concurrent HashMap.
- **@Cacheable**: Marks a method's response to be cached. Spring Boot manages the request and response of the method to the cache specified in the annotation.
- **@CachePut**: Updates the cache after invoking the method. It ensures the cache is updated with new data.
- **@CacheEvict**: Removes the data from the cache, ensuring that outdated data is cleared.

Example Implementation in Spring Boot

```
@Cacheable(cacheNames = "employees",key = "#id")
public EmployeeDTO getEmpById(int id) {
```

```
    log.info("Fetching employee with ID: {}", id);
    EmployeeEntity employeeEntity = employeeRepository.findById(id)
            .orElseThrow(() -> {
                log.error("No employee found with ID: {}", id);
                return new ResourceNotFoundException("No such Element Found by id: " + id);
            });
    log.info("Successfully fetched employee with ID: {}", id);
    return modelMapper.map(employeeEntity, EmployeeDTO.class);
}
```

3. Redis Cache in Spring Boot

How to Implement Redis Cache in Spring Boot

1.Add Dependencies:

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

2.Configure Redis:

```
#redis configuration
spring.cache.type=redis
spring.data.redis.host=${redis_host}
spring.data.redis.port=${redis_port}
spring.data.redis.password=${redis_password}
```

Enable Redis Caching:

```
@Bean
public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory){
    RedisCacheConfiguration
redisCacheConfiguration=RedisCacheConfiguration.defaultCacheConfig()
            .prefixCacheNameWith("Redis_Cache")
            .entryTtl(Duration.ofSeconds(60))
            .enableTimeToIdle()
            .serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(new
StringRedisSerializer()))
            .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(new
GenericJackson2JsonRedisSerializer()));

    return RedisCacheManager.builder(redisConnectionFactory)
            .cacheDefaults(redisCacheConfiguration)
            .build();
}
```

You can use offline redis cache or cloud redis mechanism

you can use offline redis cache or cloud redis mechanism

below links →

🌐 Redis – The Real-time Data Platform

4. Database Transactions and ACID Properties

A database transaction is a logical unit that includes several steps. If one step fails, the entire transaction fails.

ACID Properties:

- **Atomicity**: Ensures all operations succeed or none do (e.g., bank transfers should either debit and credit both accounts or fail entirely).
- **Consistency**: Guarantees the database transitions from one valid state to another.
- **Isolation**: Ensures concurrent transactions do not interfere with each other.
- **Durability**: Ensures that once a transaction is committed, it remains so even in case of system failure.

Implementing Transactions in Spring Boot

Ex.1 →

```
 @CachePut(cacheNames = "employees", key = "#result.empId")
@Transactional
public EmployeeDTO addEmp(EmployeeDTO employeeDTO) {
    log.info("Adding new employee: {}", employeeDTO);
    EmployeeEntity obj = modelMapper.map(employeeDTO, EmployeeEntity.class);
    EmployeeEntity employeeEntity = employeeRepository.save(obj);
    log.info("Successfully added employee with ID: {}", employeeEntity.getEmpId());
    salaryAccountServices.createAccount(employeeEntity);
    log.info("Successfully added employee in salary Account Table with ID: {}",
employeeEntity.getEmpId());
    return modelMapper.map(employeeEntity, EmployeeDTO.class);
}
```

Ex.2 →

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public SalaryAccount incrementSalary(Long accountId) {
    // A "Parallel API Load Testing Command
    // seq 100 | xargs -P 10 -I {} curl --location --request PUT
'http://127.0.0.1:9090/employees/incrementAccBalance/1'
    // "This command makes 100 API requests using cURL, running 10 requests at the same
time.
    //"It simulates multiple users updating an employee's account balance in parallel."
    // All the logging statements saved in logs/salary_account.log
    SalaryAccount
salaryAccount=salaryAccountRepository.findById(accountId).orElseThrow(() -> new
```

```
RuntimeException("Not found with salary Account "+accountId));
    BigDecimal prevBalance=salaryAccount.getBalance();
    BigDecimal newBalance=prevBalance.add(BigDecimal.valueOf(1L));

    salaryAccount.setBalance(newBalance);

    log.info("Incremented Salary Account: {} | Previous Balance: {} | New Balance: {}",
            accountId, prevBalance, newBalance);
    return  salaryAccountRepository.save(salaryAccount);

}
```

5. Concurrent Transaction Management

Common Transaction Problems:

- **Dirty Read**: Reads uncommitted data from another transaction.
- **Non-Repeatable Read**: Same row read twice but with different values.
- **Phantom Read**: Same query retrieves different rows when executed twice.

Isolation Levels in Spring Boot

```
@Transactional(isolation = Isolation.READ_COMMITTED)

public void processTransaction() {

    // Transaction code here

}
```

Others Isolations levels →
@Transactional(isolation = Isolation.**DEFAULT**) // Uses the default isolation level of the database.
@Transactional(isolation = Isolation.**READ_UNCOMMITTED**) // Allows reading uncommitted (dirty) data, improving performance but risking inconsistencies.
@Transactional(isolation = Isolation.**READ_COMMITTED**) // Ensures only committed data is read, preventing dirty reads.
@Transactional(isolation = Isolation.**REPEATABLE_READ**) // Ensures the same data is read consistently within a transaction, preventing non-repeatable reads.
@Transactional(isolation = Isolation.**SERIALIZABLE**) // Fully isolates transactions, preventing dirty reads, non-repeatable reads, and phantom reads.

6. Transaction Annotations in Spring Boot

Transaction Propagation:

- **REQUIRED**: Uses an existing transaction if available, otherwise creates a new one.

- **REQUIRES_NEW**: Always creates a new transaction.
- **MANDATORY**: Throws an exception if no transaction exists.
- **NOT_SUPPORTED**: Runs the method without a transaction.
- **SUPPORTS**: Uses an existing transaction if available, otherwise runs without one.
- **NEVER**: Throws an exception if a transaction exists.
- **NESTED**: Creates a sub-transaction within the parent transaction.

Example Implementation:

```
@Transactional(propagation = Propagation.REQUIRES_NEW)

public void saveUser(User user) {

    userRepository.save(user);

}
```

7. Transaction Locks in Spring Data JPA

1.Optimistic Locking

1. A new column called "version" is added to the database table.
2. Before a user modifies a database row, the application reads the version number of the row.
3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.
4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

```
@Version
private Long version;
```

2.Pessimistic Locking:

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
Optional<SalaryAccount> findById(Long id);

// @Lock(LockModeType.NONE)  // No locking, allows concurrent access.
// @Lock(LockModeType.READ)  // Acquires a shared lock, allows multiple reads but no writes.
// @Lock(LockModeType.OPTIMISTIC)  // Uses versioning to check for conflicts before committing.
// @Lock(LockModeType.OPTIMISTIC_FORCE_INCREMENT)  // Similar to OPTIMISTIC but forces a version increment.
// @Lock(LockModeType.PESSIMISTIC_READ)  // Blocks other transactions from writing but allows reading.
// @Lock(LockModeType.PESSIMISTIC_FORCE_INCREMENT)  // PESSIMISTIC_WRITE + forces version increment on lock acquisition.
```

8. Real-World Transaction Strategies

Strategy	Use Case Examples
Read Committed	E-commerce transactions (e.g., Amazon)
Read Uncommitted	Data analytics, non-critical reporting
Serializable	Financial transactions (e.g., banks)
Optimistic Locking	Collaborative editing (e.g., Google Docs)
Pessimistic Locking	Booking systems (e.g., airline seats)
No Transactions	Social media, large-scale systems (e.g., Twitter)

Conclusion

This documentation provides a comprehensive guide to caching and transaction management in Spring Boot, covering concepts, implementation, and real-world scenarios. By using the correct caching mechanisms and transaction management strategies, you can improve application performance, consistency, and concurrency handling.

Feel free to contribute to this repository to enhance transaction management techniques in Spring Boot applications!

By Aron...