Amo Samuel

Report on Transaction Management and Caching Implementation

This document provides an overview of the implementation of transaction management and caching mechanisms in the Spring Data project. It covers the use of transaction management for ensuring data consistency and the application of caching to improve performance.

Overview

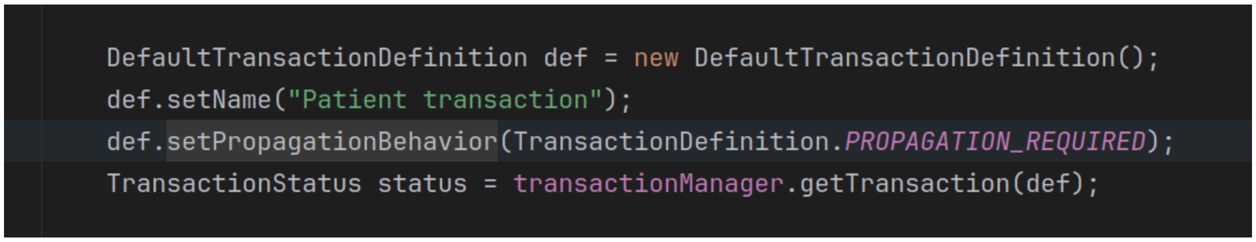
Transaction management is crucial for maintaining data integrity and consistency, especially when multiple operations need to be executed as a single unit of work. In this project, I use Spring's transaction management capabilities to handle transactions explicitly.

Implementation

PatientService Class

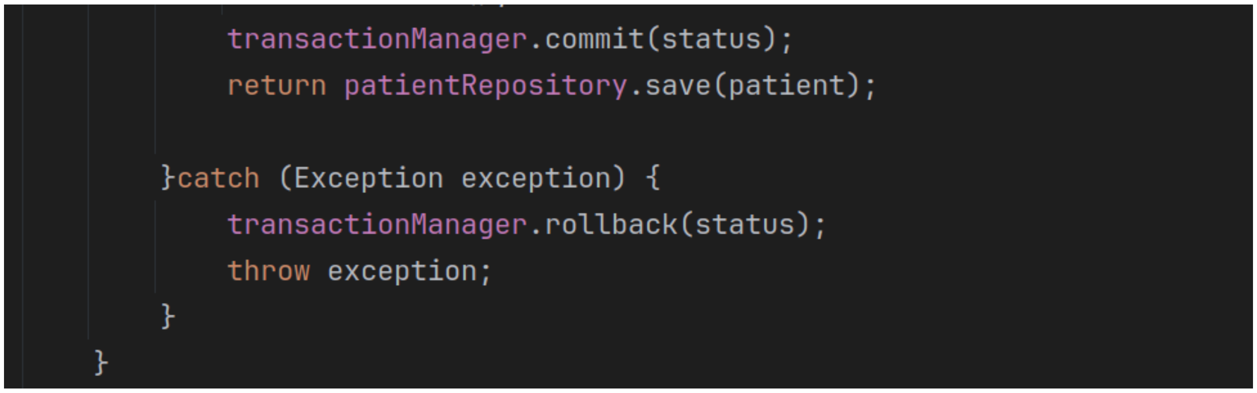
In the PatientService class, I have implemented explicit transaction management for creating a new Patient record.

Transaction Definition: I define a transaction using DefaultTransactionDefinition. This includes setting the transaction name and propagation behavior.



```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setName("Patient transaction");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
TransactionStatus status = transactionManager.getTransaction(def);
```

Transaction Management: I obtain the transaction status and perform operations within a try-catch block. If all operations succeed, we commit the transaction; otherwise, I roll back the transaction to ensure data consistency.



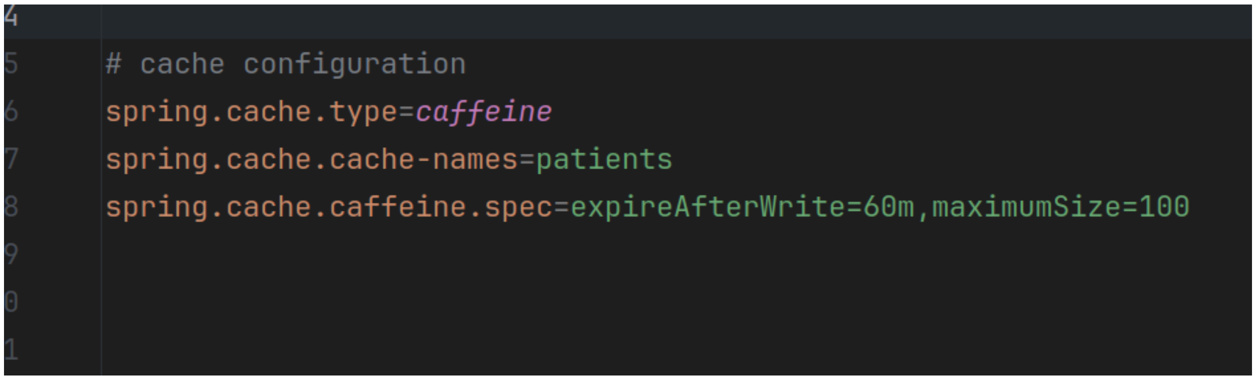
```
            transactionManager.commit(status);
            return patientRepository.save(patient);

        }catch (Exception exception) {
            transactionManager.rollback(status);
            throw exception;
        }
    }
```

Caching

Caching is used to store frequently accessed data in memory to improve performance and reduce database load. In this project, I use Caffeine as the caching solution.

Configuration

Caching is configured in the application.properties file. I specified Caffeine as the cache provider and configured cache properties such as expiration and maximum size.



```
# cache configuration
spring.cache.type=caffeine
spring.cache.cache-names=patients
spring.cache.caffeine.spec=expireAfterWrite=60m,maximumSize=100
```

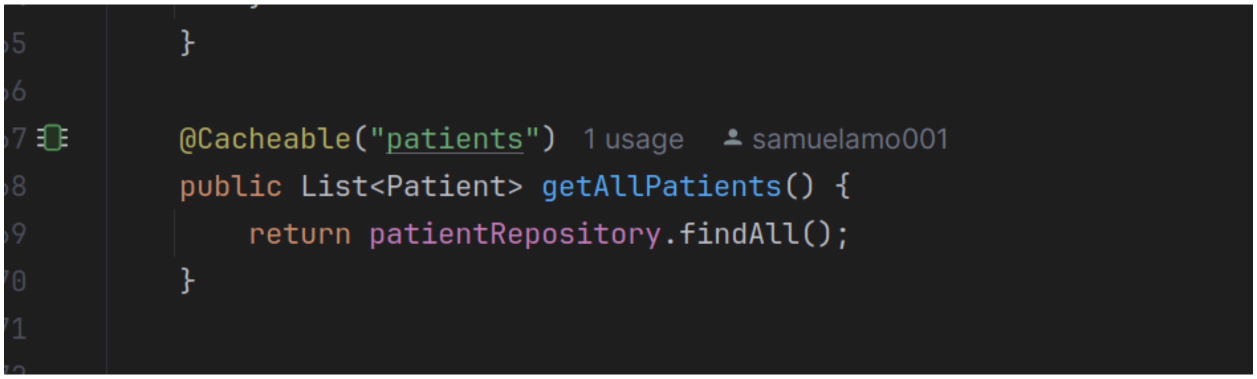
expireAfterWrite=60m: Specifies that cache entries will expire 60 minutes after being written.

maximumSize=100: Limits the cache to 100 entries to prevent excessive memory usage.

PatientService Class

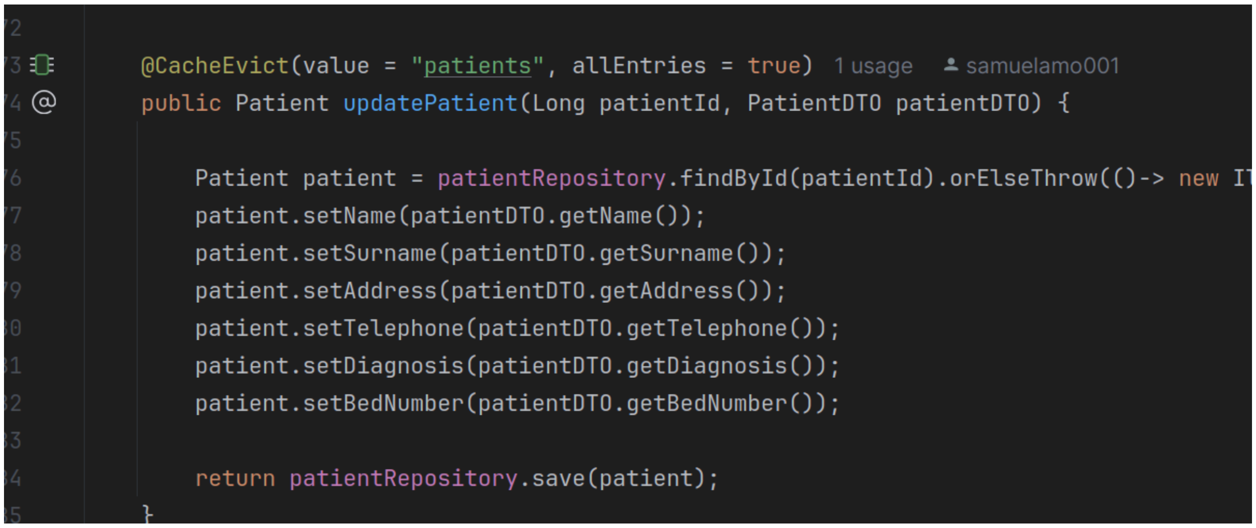
Caching Implementation:
In the PatientService class, caching is applied to methods to reduce database calls:

Caching All Patients: The getAllPatients() method is annotated with @Cacheable, which caches the result of the method call.



```java
    }

    @Cacheable("patients")  1 usage   samuelamo001
    public List<Patient> getAllPatients() {
        return patientRepository.findAll();
    }
```

Evicting Cache on Update: The updatePatient() method is annotated with @CacheEvict, which clears the cache entries when a patient record is updated.



```java
    @CacheEvict(value = "patients", allEntries = true)  1 usage   samuelamo001
    public Patient updatePatient(Long patientId, PatientDTO patientDTO) {

        Patient patient = patientRepository.findById(patientId).orElseThrow(()-> new I
        patient.setName(patientDTO.getName());
        patient.setSurname(patientDTO.getSurname());
        patient.setAddress(patientDTO.getAddress());
        patient.setTelephone(patientDTO.getTelephone());
        patient.setDiagnosis(patientDTO.getDiagnosis());
        patient.setBedNumber(patientDTO.getBedNumber());

        return patientRepository.save(patient);
    }
```

Conclusion

The combination of transaction management and caching in this Spring Data project enhances data consistency, integrity, and performance. By explicitly managing transactions and leveraging Caffeine for caching, I ensured that the application operates efficiently and reliably.