Spring Data for NoSQL Databases with MongoDB and Redis Integration

Introduction

Spring Data offers powerful and easy-to-use integrations with NoSQL databases such as MongoDB and Redis. With the rise of NoSQL databases, applications can now leverage more flexible data models, scalability, and performance enhancements that traditional relational databases may struggle to provide. This document provides a detailed overview of using Spring Data for MongoDB and Redis, along with code examples that demonstrate integration and CRUD operations.

Differences Between Relational and NoSQL Databases

Relational Databases:

- **Structured Schema**: Relational databases (e.g., MySQL, PostgreSQL) rely on structured schema designs where data is stored in tables with predefined columns.
- **SQL**: Data is accessed and manipulated through SQL (Structured Query Language), which ensures ACID (Atomicity, Consistency, Isolation, Durability) properties.
- **Joins**: Relationships between data entities are managed using foreign keys, and complex joins are often required to retrieve related data across tables.

NoSQL Databases:

- **Flexible Schema**: NoSQL databases like MongoDB do not enforce a fixed schema, allowing for dynamic and hierarchical data structures (e.g., documents, key-value pairs).
- **Horizontal Scalability**: NoSQL databases are built to scale out easily by distributing data across multiple nodes.
- **BASE**: Many NoSQL systems operate under BASE (Basically Available, Soft state, Eventual consistency) principles, which prioritise availability over strict consistency.
- **Data Models**: NoSQL databases use diverse data models like document, key-value, column-family, and graph, depending on the use case.

Spring Data MongoDB: CRUD Operations on NoSQL Documents

MongoDB is a popular document-based NoSQL database that stores data in JSON-like structures called BSON (Binary JSON) documents. Spring Data MongoDB provides seamless integration for performing CRUD operations on MongoDB collections.

MongoDB Entities

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Document(collection = "doctors")
public class Doctor extends Employee {
    private String speciality;
    @JsonIgnore
    @DBRef
    private Set<Department> departments;
    @JsonIgnore
    @DBRef
    private Set<Patient> patients;
}
```

CRUD Operations

Create and Update Operations:

```
@Service
@RequiredArgsConstructor
public class EmployeeServiceImpl implements EmployeeService {

    private final DoctorRepository doctorRepository;

    @Transactional
    @Override
    public Doctor createDoctor(DoctorDTO doctorDTO) {
        Doctor doctor = Doctor.builder()
                .firstName(doctorDTO.getFirstName())
                .surname(doctorDTO.getSurname())
                .address(doctorDTO.getAddress())
                .telephoneNumber(doctorDTO.getTelephoneNumber())
                .speciality(doctorDTO.getSpeciality())
                .build();
        return doctorRepository.save(doctor);
    }

    @Transactional
    @Override
    public Doctor updateDoctor(String doctorId, DoctorDTO doctorDTO) {
        return doctorRepository.findById(doctorId).map(doctor -> {
            doctor.setFirstName(doctorDTO.getFirstName());
            doctor.setSurname(doctorDTO.getSurname());
            doctor.setAddress(doctorDTO.getAddress());
            doctor.setTelephoneNumber(doctorDTO.getTelephoneNumber());
            doctor.setSpeciality(doctorDTO.getSpeciality());
            return doctorRepository.save(doctor);
        }).orElseThrow(() -> new IllegalArgumentException("Doctor not found"));
    }
}
```

Read Operation:

```
@Override
public Doctor getDoctorById(String doctorId) {
    return doctorRepository.findById(doctorId)
            .orElseThrow(() -> new IllegalArgumentException("Doctor not found"));
}

@Override
public List<Doctor> getAllDoctors() {
    return doctorRepository.findAll();
}
```

Querying NoSQL Databases: Performing Advanced Queries

MongoDB Query Example:

Spring Data MongoDB allows you to perform complex queries using the `@Query` annotation.

```
public interface DoctorRepository extends MongoRepository<Doctor, String> {
    @Query(value = "{ 'speciality' : ?0 }", fields = "{'id': 1,'firstName': 1 ,'surname': 1,
'address': 1, 'telephoneNumber' : 1, 'speciality': 1}")
    List<DoctorDTO> findingBySpeciality(String speciality);
}
```

This query retrieves all doctors with a specific specialty, projecting only the relevant fields in the results.

Redis Integration with Spring Data

Redis is an in-memory key-value store known for its high performance and support for various data structures (e.g., strings, lists, sets, hashes). Spring Data Redis provides a convenient way to interact with Redis, including CRUD operations and caching.

Example Code: Redis Entity

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@RedisHash("Patient")
public class Patient implements Serializable {
    @Id
    private String id;
    private String surname;
    private String firstName;
    private String address;
    private String telephoneNumber;
    private String diagnosis;
    private String doctorId;
    private String hospitalisationId;
}
```

Redis Configuration

```
@Configuration
@EnableRedisRepositories
public class RedisConfig {
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);
        template.setKeySerializer(new StringRedisSerializer());
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(new JdkSerializationRedisSerializer());
        template.setValueSerializer(new JdkSerializationRedisSerializer());
        template.afterPropertiesSet();
        return template;
    }
}
```

CRUD Operations in Redis

Create Operation:

```
@Service
@RequiredArgsConstructor
public class PatientServiceImpl implements PatientService {

    private final PatientRepository patientRepository;

    @Transactional
    @Override
    public Patient createPatient(Patient patient) {
        return patientRepository.save(patient);
    }
}
```

Read Operation:

```
@Override
public Patient getPatient(String id) {
    return patientRepository.findById(id).orElseThrow(() -> new
IllegalArgumentException("Patient not found"));
}

@Override
public Iterable<Patient> getAllPatients() {
    return patientRepository.findAll();
}
```

Caching with Spring Data

This enhances performance by reducing the load on the primary data store and allowing fast access to frequently accessed data.

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager("doctorsBySpeciality");
        cacheManager.setCaffeine(caffeineCacheBuilder());
        return cacheManager;
    }

    Caffeine<Object, Object> caffeineCacheBuilder() {
        return Caffeine.newBuilder()
                .initialCapacity(100)
                .maximumSize(500)
                .expireAfterWrite(1, TimeUnit.HOURS)
                .recordStats();
    }
}
```

With this configuration, the cache will store the results of the `findDoctorsBySpeciality` method, which can be retrieved from the cache instead of querying the database directly.

Conclusion

Spring Data provides comprehensive support for integrating NoSQL databases like MongoDB and Redis. By leveraging the flexibility of NoSQL databases, applications can scale more effectively and handle unstructured data more efficiently. Spring Data MongoDB and Redis enable seamless CRUD operations, advanced querying, and caching to optimise application performance and data handling.