Spring Data Redis

Spring Data Redis ist ein fantastisches Werkzeug zur Integration von Redis—a high-performance, in-memory data store—in Ihre Spring-Anwendung. Es erleichtert die Arbeit mit Redis, indem es eine vertraute repository-ähnliche Schnittstelle und Abstraktionen über niedrigstufige Redis-Operationen bietet. Lassen Sie uns Schritt für Schritt durch die Einrichtung und Verwendung gehen.

1. Projekt einrichten

Zunächst müssen Sie die Spring Data Redis-Abhängigkeit in Ihr Projekt einbinden. Wenn Sie Maven verwenden, fügen Sie dies zu Ihrer `pom.xml` hinzu:

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

Für Gradle verwenden Sie:

```
implementation 'org.springframework.boot:spring-boot-starter-data-redis'
```

Dies schließt den Redis-Client (Jedis oder Lettuce—Lettuce ist der Standard) ein. Wenn Sie Jedis bevorzugen, können Sie es explizit hinzufügen und Lettuce ausschließen:

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

2. Redis konfigurieren

Konfigurieren Sie Ihre Redis-Verbindung in `application.properties` oder `application.yml`. Für eine lokale Redis-Instanz, die auf dem Standardport (6379) läuft:

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password= # Optional, wenn Ihr Redis-Server ein Passwort hat
spring.redis.database=0 # Standard-Datenbankindex
```

Wenn Sie einen entfernten Redis-Server oder einen Dienst wie AWS ElastiCache verwenden, passen Sie den Host und die Anmeldeinformationen entsprechend an.

3. Grundlegende Verwendung mit RedisTemplate

Spring Data Redis stellt `RedisTemplate` für niedrigstufige Operationen bereit. Sie können es in Ihren Service oder Ihre Komponente injizieren:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Service;

@Service
public class RedisService {
    private final RedisTemplate<String, String> redisTemplate;

    @Autowired
    public RedisService(RedisTemplate<String, String> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    public void saveData(String key, String value) {
        redisTemplate.opsForValue().set(key, value);
    }

    public String getData(String key) {
        return redisTemplate.opsForValue().get(key);
    }
}
```

- `RedisTemplate` ist generisch: `<String, String>` bedeutet, dass Schlüssel und Werte Zeichenketten sind. Sie können andere Typen (z. B. `<String, Object>`) verwenden.

- `opsForValue()` ist für einfache Schlüssel-Wert-Operationen. Andere Methoden umfassen `opsForList()`, `opsForSet()`, `opsForHash()` usw. für verschiedene Redis-Datenstrukturen.

4. Verwendung mit Objekten

Um Java-Objekte zu speichern und abzurufen, konfigurieren Sie `RedisTemplate` mit Serialisierern. Spring Boot konfiguriert es automatisch, aber Sie können es bei Bedarf anpassen:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
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.setValueSerializer(new Jackson2JsonRedisSerializer<>(Object.class));
        template.afterPropertiesSet();
        return template;
    }
}
```

Jetzt können Sie Objekte speichern und abrufen:

```
public class Person {
    private String firstName;
    private String lastName;

    // Standardkonstruktor (für Deserialisierung)
    public Person() {}

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

```
    // Getter und Setter
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
}

@Service
public class PersonRedisService {
    private final RedisTemplate<String, Object> redisTemplate;

    @Autowired
    public PersonRedisService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    public void savePerson(String key, Person person) {
        redisTemplate.opsForValue().set(key, person);
    }

    public Person getPerson(String key) {
        return (Person) redisTemplate.opsForValue().get(key);
    }
}
```

5. Repository-Ansatz

Für eine höhere Abstraktion verwenden Sie Spring Data Redis-Repositories. Definieren Sie eine Entität und ein Repository:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;

@RedisHash("Person") // Zu einem Redis-Hash mit Präfix "Person" zuordnen
public class Person {
    @Id
    private String id; // Redis-Schlüssel wird "Person:<id>"
    private String firstName;
    private String lastName;
```

```
    // Konstruktoren, Getter, Setter (wie oben)
}

import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, String> {
}
```

Verwenden Sie es wie folgt:

```
@Service
public class PersonService {
    private final PersonRepository repository;

    @Autowired
    public PersonService(PersonRepository repository) {
        this.repository = repository;
    }

    public void savePerson() {
        Person person = new Person("John", "Doe");
        repository.save(person);
        System.out.println("Gespeicherte Person mit ID: " + person.getId());
    }

    public void findPerson(String id) {
        Person person = repository.findById(id).orElse(null);
        if (person != null) {
            System.out.println(person.getFirstName() + " " + person.getLastName());
        }
    }
}
```

- `@RedisHash` ordnet die Entität einem Redis-Hash zu.
- Das Repository bietet CRUD-Operationen aus der Box.

6. Ihre Anwendung ausführen

Stellen Sie sicher, dass Redis lokal läuft (z. B. über Docker: `docker run -d -p 6379:6379 redis`) oder verbinden Sie sich mit einer entfernten Instanz. Starten Sie dann Ihre Spring Boot-App:

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

7. Erweiterte Funktionen

- **TTL (Time-to-Live):** Legen Sie eine Ablaufzeit für Schlüssel mit `redisTemplate.expire(key, 10, TimeUnit.MINUTES)` fest.
- **Hashes:** Verwenden Sie `opsForHash()`, um direkt mit Redis-Hashes zu arbeiten.
- **Pub/Sub:** Verwenden Sie `RedisMessageListenerContainer` für die Nachrichtenübermittlung.

Beispiel-Workflow

1. Speichern Sie ein `Person`-Objekt mit `repository.save()` oder `redisTemplate.opsForValue().set()`.
2. Rufen Sie es mit `repository.findById()` oder `redisTemplate.opsForValue().get()` ab.
3. Überprüfen Sie Ihre Redis-Instanz (z. B. mit `redis-cli`), um die Daten zu sehen.

Das sind die Grundlagen! Wenn Sie mehr über Caching, Pub/Sub oder etwas Spezielles erfahren möchten, lassen Sie es mich wissen, und ich werde tiefer eintauchen. Wie passt das zu dem, was Sie vorhaben?