Spring Data Reactive Repositories with MongoDB

Introdução

Neste tutorial, veremos como configurar e implementar operações de banco de dados usando programação reativa por meio de repositórios reativos de dados Spring com MongoDB.

Veremos os usos básicos do repositório *ReactiveCrud , ReactiveMongoRepository ,* bem como *ReactiveMongoTemplate.*

Embora essas implementações usem programação reativa , esse não é o foco principal deste tutorial.

Ambiente

Para usar o Reactive MongoDB, precisamos adicionar a dependência ao nosso *pom.xml.*

Também adicionaremos um MongoDB incorporado para teste:

```
<dependencies>
    // ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
    </dependency>
    <dependency>
        <groupId>de.flapdoodle.embed</groupId>
        <artifactId>de.flapdoodle.embed.mongo</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Configuração

Para ativar o suporte reativo, precisamos usar o *@EnableReactiveMongoRepositories* junto com alguma configuração de infraestrutura:

```
@EnableReactiveMongoRepositories
public class MongoReactiveApplication
  extends AbstractReactiveMongoConfiguration {

    @Bean
    public MongoClient mongoClient() {
        return MongoClients.create();
    }

    @Override
    protected String getDatabaseName() {
        return "reactive";
    }
}
```

Observe que o descrito acima seria necessário se estivéssemos usando a instalação autônoma do MongoDB. Mas, como estamos usando Spring Boot com MongoDB incorporado em nosso exemplo, a configuração acima não é necessária.

Criando um *documento*

Para os exemplos abaixo, vamos criar uma classe *Account* e anotá-la com *@Document* para utilizá-la nas operações do banco de dados:

```
@Document
public class Account {

    @Id
```

```
    private String id;
    private String owner;
    private Double value;

    // getters and setters
}
```

Usando repositórios reativos

Já estamos familiarizados com o modelo de programação de repositórios, com os métodos CRUD já definidos e suporte para algumas outras coisas comuns também.

Já com o modelo Reativo, obtemos o mesmo conjunto de métodos e especificações, exceto que trataremos os resultados e parâmetros de forma reativa.

ReactiveCrudRepository

Podemos usar este repositório da mesma forma que o bloqueio *CrudRepository*:

```
@Repository
public interface AccountCrudRepository
  extends ReactiveCrudRepository<Account, String> {

    Flux<Account> findAllByValue(String value);
    Mono<Account> findFirstByOwner(Mono<String> owner);
}
```

Podemos passar diferentes tipos de argumentos como simples (*String*), embrulhado (*Optional* , *Stream*) ou reativo (*Mono* , *Flux*) como podemos ver no método *findFirstByOwner()*.

ReactiveMongoRepository

Há também a interface *ReactiveMongoRepository* , que herda de *ReactiveCrudRepository* e adiciona alguns novos métodos de consulta:

```
@Repository
public interface AccountReactiveRepository
  extends ReactiveMongoRepository<Account, String> { }
```

Usando o *ReactiveMongoRepository* , podemos consultar por exemplo:

```
Flux<Account> accountFlux = repository
  .findAll(Example.of(new Account(null, "owner", null)));
```

Como resultado, obteremos todas as *contas* iguais ao exemplo passado.

Com nossos repositórios criados, eles já possuem métodos definidos para realizar algumas operações de banco de dados que não precisamos implementar, como save e findById por exemplo:

```
Mono<Account> accountMono
  = repository.save(new Account(null, "owner", 12.3));
Mono<Account> accountMono2 = repository
  .findById("123456");
```

RxJava2CrudRepositoryName

Com *o RxJava2CrudRepository,* temos o mesmo comportamento do *ReactiveCrudRepository,* porém com os resultados e tipos de parâmetros do *RxJava*:

```
@Repository
public interface AccountRxJavaRepository
  extends RxJava2CrudRepository<Account, String> {

    Observable<Account> findAllByValue(Double value);
    Single<Account> findFirstByOwner(Single<String> owner);
}
```

Testando nossas operações básicas

Para testar nossos métodos de repositório, usaremos o assinante de teste:

```
@Test
public void givenValue_whenFindAllByValue_thenFindAccount() {
    repository.save(new Account(null, "Bill", 12.3)).block();
    Flux<Account> accountFlux = repository.findAllByValue(12.3);

    StepVerifier
      .create(accountFlux)
      .assertNext(account -> {
          assertEquals("Bill", account.getOwner());
          assertEquals(Double.valueOf(12.3) , account.getValue());
          assertNotNull(account.getId());
      })
      .expectComplete()
      .verify();
}

@Test
public void givenOwner_whenFindFirstByOwner_thenFindAccount() {
    repository.save(new Account(null, "Bill", 12.3)).block();
    Mono<Account> accountMono = repository
      .findFirstByOwner(Mono.just("Bill"));

    StepVerifier
      .create(accountMono)
      .assertNext(account -> {
          assertEquals("Bill", account.getOwner());
          assertEquals(Double.valueOf(12.3) , account.getValue());
          assertNotNull(account.getId());
      })
      .expectComplete()
      .verify();
}

@Test
public void givenAccount_whenSave_thenSaveAccount() {
    Mono<Account> accountMono = repository.save(new Account(null, "Bill", 12.3));

    StepVerifier
      .create(accountMono)
      .assertNext(account -> assertNotNull(account.getId()))
      .expectComplete()
      .verify();
}

/* O código fornecido é um teste de unidade escrito em Java usando o framework de teste JUnit e o projeto Reactor para programação reativa.

A anotação @Test indica que o método é um teste unitário.

O método givenValue_whenFindAllByValue_thenFindAccount é o caso de teste em si.

A linha repository.save(new Account(null, "Bill", 12.3)).block(); salva uma nova conta com nome "Bill" e valor 12.3 no repositório. O métod

A linha Flux<Account> accountFlux = repository.findAllByValue(12.3); cria um fluxo reativo de contas encontradas no repositório com o valor

StepVerifier.create(accountFlux) cria um verificador de etapas para o fluxo reativo accountFlux. O verificador permite definir as expectati

.assertNext(account -> { ... }) define uma ação que será executada quando uma conta for recebida no fluxo. Neste caso, asserções são feitas

.expectComplete() espera que o fluxo esteja completo, ou seja, que todas as contas tenham sido recebidas.
```

```
.verify() inicia a verificação. Se todas as asserções passarem e o fluxo estiver completo, o teste será considerado bem-sucedido.

No geral, esse teste verifica se o método findAllByValue do repositório retorna corretamente as contas com o valor especificado e se essas
```

ReactiveMongoTemplate

Além da abordagem de repositórios, temos o ***ReactiveMongoTemplate***.

Antes de tudo, precisamos registrar *o* ***ReactiveMongoTemplate*** como um bean:

```
@Configuration
public class ReactiveMongoConfig {

    @Autowired
    MongoClient mongoClient;

    @Bean
    public ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoClient, "test");
    }
}
```

E então, podemos injetar este bean em nosso serviço para realizar as operações do banco de dados:

```
@Service
public class AccountTemplateOperations {

    @Autowired
    ReactiveMongoTemplate template;

    public Mono<Account> findById(String id) {
        return template.findById(id, Account.class);
    }

    public Flux<Account> findAll() {
        return template.findAll(Account.class);
    }
    public Mono<Account> save(Mono<Account> account) {
        return template.save(account);
    }
}
```

O ***ReactiveMongoTemplate*** também possui vários métodos que não estão relacionados ao domínio que temos, você pode conferir na documentação.

Conclusão

Neste breve tutorial, abordamos o uso de repositórios e modelos usando programação reativa com o **MongoDB** com o framework **Spring Data Reactive Repositories**.

O código-fonte completo dos exemplos está disponível no GitHub.