CETEJ35 - Java Web - JAVA_XXX (2024_01)

API Reativa

✓ **Feito:** Ver ✓ **Feito:** Gastar pelo menos 20 minutos na atividade **A fazer:** Passar pela atividade até o fim

Fecha: segunda-feira, 2 dez. 2024, 00:00

Nesta aula vamos construir uma nova aplicação que funciona como uma API, recebendo dados usando a arquitetura REST. Para fazer isso, vamos usar uma nova tecnologia do Spring - o Spring WebFlux.

PERSISTÊNCIA

Nossa API do gerenciador de tarefas usa o MongoDB para salvar as tarefas de forma persistente. O MongoDB é um banco de dados não relacional (NoSQL), que armazena dados como *documentos*. Dessa forma, a estrutura dos dados pode variar a cada *documento*. Para o MongoDB, um documento é como um arquivo em formato `JSON`.

As *tarefas* no nosso gerenciador de tarefas tem uma estrutura bem definida. Poderíamos usar um banco de dados relacional sem qualquer problema. Porém, estamos usando o Spring WebFlux para desenvolver uma API não bloqueante. No momento em que essa aula está sendo preparada, o Spring Data JPA ainda não oferece suporte para o WebFlux. Porém, você ainda pode usar bancos relacionais com WebFlux adotando o Spring Data R2DBC.

Escolher o tipo mais adequado de banco de dados pode ser bem complicado. Se quiser entender melhor as diferenças entre SQL e as diversas variantes de NoSQL, esses links podem ajudar:

- **Understand data store models**
- **From SQL to NoSQL**
- **NoSQL Databases: a Survey and Decision Guidance**

Felizmente, criar a parte de persistência com o Spring WebFlux não é diferente do que já fizemos com o Spring MVC. Isso porque o Spring Data oferece uma interface para repositórios reativos (`org.springframework.data.repository.reactive.ReactiveCrudRepository`). Tanto a interface para repositórios *reativos* quanto para *não reativos* é baseada na mesma interface (`org.springframework.data.repository.Repository`). Dessa forma, a implementação é exatamente a mesma. A figura abaixo mostra o código da interface `TodoRepository`. Essa interface é responsável pelas operações de gerenciamento dos dados de *tarefas* (representada pelo Java record `Todo`).

```java
package com.example.demo;

import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.springframework.stereotype.Repository;

import reactor.core.publisher.Flux;

@Repository
public interface TodoRepository
    extends ReactiveMongoRepository<Todo, String> {

    Flux<Todo> findByFeito(boolean feito);

}
```

A linha 08 identifica essa interface como responsável por gerenciar os dados da aplicação. A linha 10 herda comportamentos da interface `org.springframework.data.mongodb.repository.ReactiveMongoRepository`. Os comportamentos herdados correspondem às operações básicas de um CRUD, como ler e salvar. A interface `org.springframework.data.mongodb.repository.ReactiveMongoRepository` é uma especialização da interface `org.springframework.data.repository.reactive.ReactiveCrudRepository` para o MongoDB.

> Por que `CidadeRepository` não tem a anotação `org.springframework.stereotype.Repository` e `TodoRepository` tem? Essa anotação **é indicada para identificar** uma interface como responsável por gerenciar dados persistentes. Na prática, ela é indiferente para a execução da aplicação.

Note que a interface é do tipo genérico - assim como no exemplo com JPA. Dessa forma, precisamos identificar a entidade persistente (`Todo`) e o tipo do identificador único (`String`). Enquanto bancos SQL costumam usar identificadores com tipos numéricos (ex: `int`, `long`), bancos NoSQL podem usar diferentes tipos de dados.

Ao herdar os comportamentos da interface `org.springframework.data.mongodb.repository.ReactiveMongoRepository`, nossa interface já recebe automaticamente métodos para criar, alterar, excluir e consultar os dados, entre outros. Contudo, nesse projeto criamos um método adicional usando palavras-chave de consulta (linha 12). Isso nos permite gerar uma lista de *tarefas* de acordo com seu status - feita ou não.

Observe que o retorno do método `findByFeito()` é do tipo `reactor.core.publisher.Flux`, e não um `java.util.List`. Isso porque `reactor.core.publisher.Flux` é o tipo que controla uma sequência de dados variando de zero a muitos.

Pode parecer um pouco confuso no início, mas é importante lembrar que o WebFlux está baseado no padrão *publisher-subscriber*. Dessa forma, o retorno dos dados não é imediato, como acontece em um método não reativo. O objeto é populado à medida que os dados são emitidos pelo *publisher* - nesse caso, o banco de dados. Um `java.util.List` não é apropriado para esse tipo de operação não bloqueante. Por outro lado, o `reactor.core.publisher.Flux` é projetado para isso.

> Posso usar um retorno do tipo `reactor.core.publisher.Flux` em um código não reactivo? Não. O `reactor.core.publisher.Flux` não é apenas um container para dados, como `java.util.List`. Tem muito mais coisa acontecendo para tornar o código reativo.

Assim como foi feito no projeto anterior, o código que implementa a interface `TodoRepository` é criado automaticamente pelo Spring Data quando o projeto é compilado. Dessa forma, não precisamos nos preocupar com mais nada - nem mesmo com a configuração do MongoDB. **Apenas garanta que o MongoDB estará disponível na porta padrão no seu computador quando executar o projeto.**

O código desenvolvido nesta Seção está disponível no **Github**, na branch `semana06-31-persistencia`.

Retroceder Avançar

◄ Verificação de aprendizado - Integração

Seguir para...

Atividade II WebConf ►