04

Requisições assíncronas

Transcrição

O comportamento padrão da nossa aplicação atualmente tem um problema bem comum em diversas aplicações do gênero. Perceba que ao finalizar uma compra a aplicação envia os dados de pagamento para um outro sistema e fica aguardando uma resposta. Enquanto aguarda uma reposta a aplicação simplesmente para. Isto porque ela executa em uma única *thread*.

Atualmente como estamos fazendo poucas operações ao finalizar uma compra, isso pode não aparentar um problema, mas em um sistema real será, pois ao finalizar uma compra, geralmente a operação envolve envio de emails, confirmação de pagamento por terceiros e registro da compra. Juntando essa quantidade de atividades a um possível grande número de usuários acessando o sistema, teremos problemas com um sistema lento que pode trazer diversos outros problemas como queda nas vendas por exemplo.

Pensando nisso, podemos fazer uma pequena otimização na funcionalidade de finalização da compra. Faremos com que ao o usuário finalizar uma compra a requisição seja feita de forma assíncrona e que somente este usuário aguarde a resposta do processamento. Desta forma os demais usuários continuam usando a aplicação sem nenhum problema.

Para implementarmos essa modificação, precisamos apenas modificar a assinatura do método `finalizar` na classe `PagamentoController` para que retorne um objeto `Callable` do tipo `ModelAndView` . Veja como fazemos isso:

```
@RequestMapping(value="/finalizar", method=RequestMethod.POST)
public Callable<ModelAndView> finalizar(RedirectAttributes model){
    return () -> {
        try {
            String uri = "http://book-payment.herokuapp.com/payment";
            String response = restTemplate.postForObject(uri, new DadosPagamento(carrinho.getTot
            model.addFlashAttribute("message", response);
            System.out.println(response);
            return new ModelAndView("redirect:/produtos");
        } catch (HttpClientErrorException e) {
            e.printStackTrace();
            model.addFlashAttribute("message", "Valor maior que o permitido");
            return new ModelAndView("redirect:/produtos");
        }
    };
}
```

Observação: Observe que estamos usando novamente recursos do Java 8. Esta forma de usar *lambda* nos permite criar um objeto do mesmo tipo esperado pelo retorno do método, evitando que criemos uma classe anônima. Neste caso é perfeitamente aplicável o recurso, por que na interface `Callable` só há um método, de nome `call` .

Conheça mais sobre Java 8 fazendo o curso disponível aqui no Alura. Clique aqui para ver o curso (https://cursos.alura.com.br/course/java8-lambdas)

A otimização feita não poderá ser analisada em ambiente local como estamos desenvolvendo, mas sim acompanhando o dia a dia da aplicação, com muitos usuários e requisições transitando pela aplicação. Mas é possível fazer alguns testes

através do JMetter, uma ferramenta para testes de performance. Leia mais sobre o JMetter na documentação do mesmo em: (http://jmeter.apache.org/)http://jmeter.apache.org/ (http://jmeter.apache.org/).