03

Usando RestTemplate

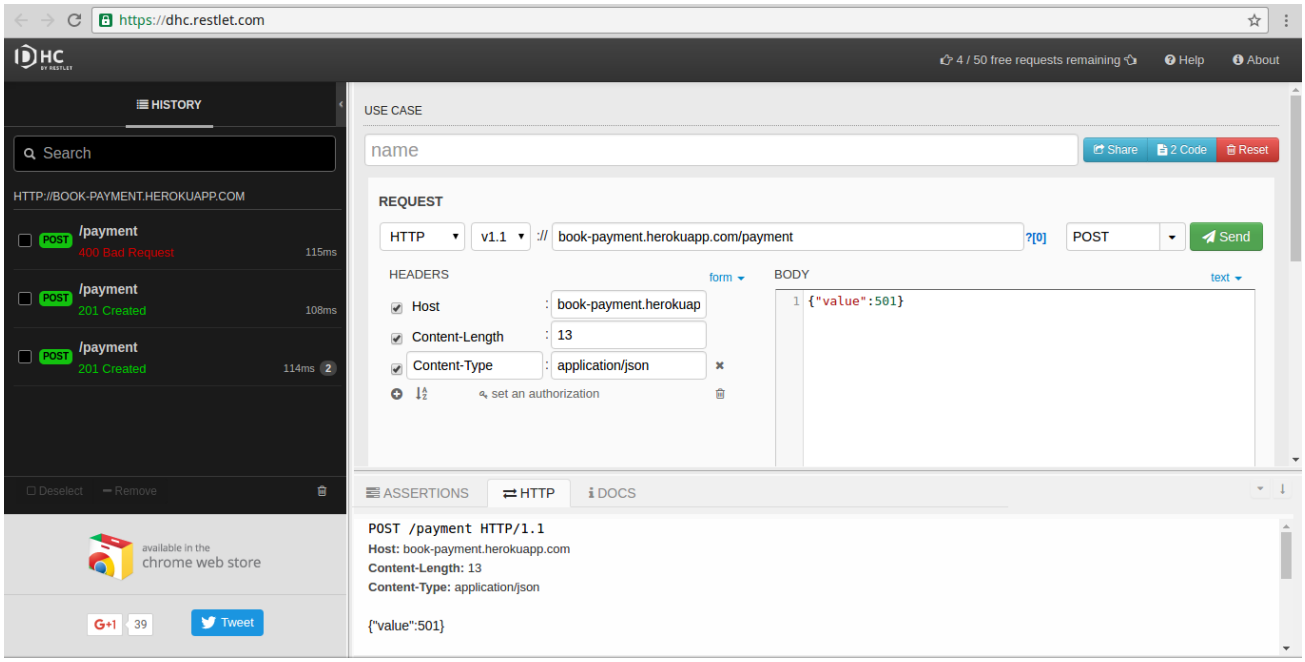
Transcrição

Ao encerrar uma compra, é muito comum que as aplicações usem uma outra aplicação terceirizada para o processamento do pagamento. Aplicações como Paypal e PagSeguro são exemplos de sistemas que geralmente se usam para processar o pagamento da compra. Em nosso caso, usaremos uma pequena aplicação hospedada em (http://book-payment.herokuapp.com/payment)http://book-payment.herokuapp.com/payment (http://book-payment.herokuapp.com/payment).

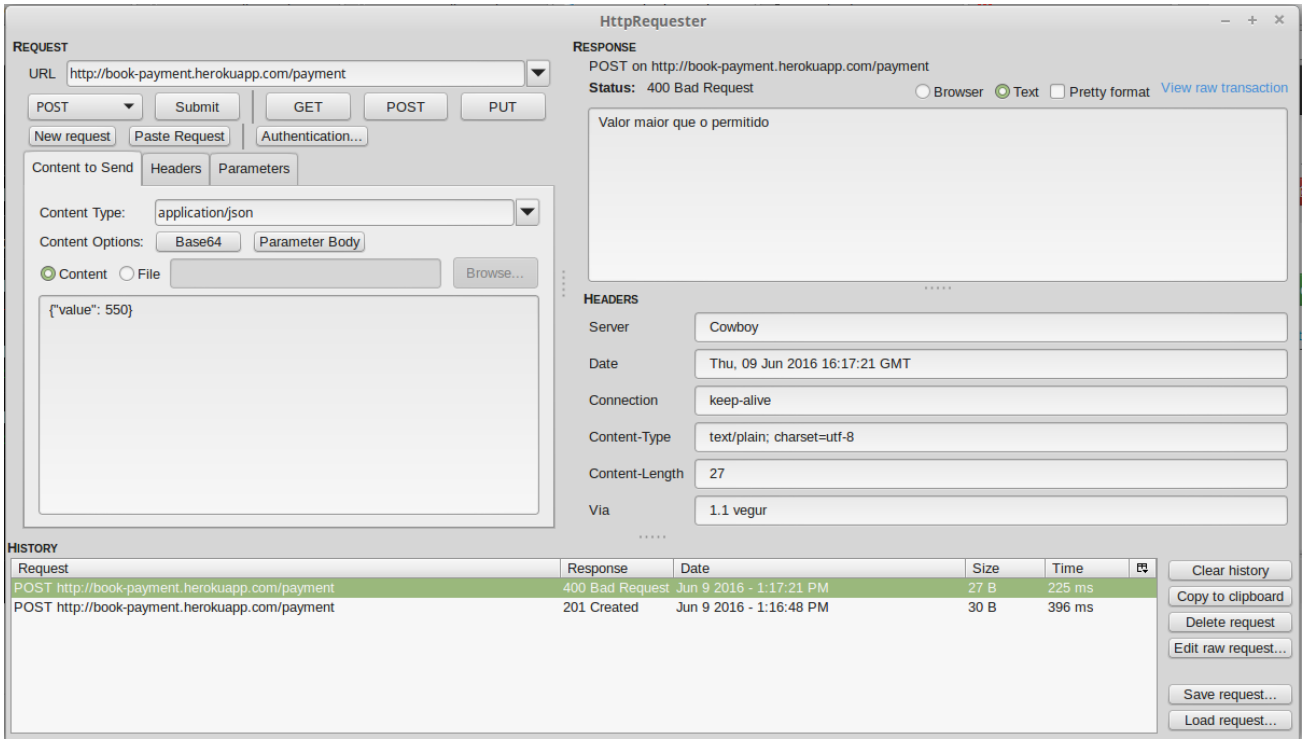
A aplicação de pagamento apresentada anteriormente serve para podermos verificar se os dados estão sendo enviados corretamente, ela não processa realmente o pagamento, mas simula a situação. Ela espera receber uma requisição do tipo `POST` enviando um `JSON` com o seguinte formato: `{"value": 500}` onde no lugar do valor `500` enviaremos o valor da compra.

Observação: Esta aplicação limita o valor do pagamento a 500. Valores acima deste retornam uma mensagem de que o valor não é permitido.

Para alguns testes pontuais, usaremos uma outra ferramenta para verificar algumas coisas, como por exemplo o tempo e a resposta em que a requisição ocorre e é respondida. No Google Chrome, você pode instalar uma extensão chamada DHC.



E no Mozilla Firefox você pode usar o HttpRequester:



Observação: Note que as imagens mostram uma captura de tela das extensões com alguns testes realizados. Faça alguns também para ter certeza que está tudo funcionando corretamente.

O problema agora é conseguir fazer com que a nossa aplicação consiga enviar uma requisição para outra aplicação. Não podemos redirecionar o navegador neste caso. O que precisamos fazer é chamado de integração, os sistemas se comunicam entre si de forma independente do navegador.

Para a solução deste tipo de problema, o **Spring** nos disponibiliza uma classe com a qual podemos fazer as requisições que precisamos. Esta classe é a `RestTemplate`. Ela tem um método chamado `postForObject` que requer três parâmetros. O primeiro deles é a url na qual queremos enviar a requisição, o segundo é o objeto que representa a mensagem (o corpo) da requisição e por último uma classe na qual esperamos receber uma resposta do tipo.

Vamos então fazer uso do `RestTemplate` em nossa classe `PagamentoController`. Crie um atributo deste tipo e o assine com a anotação `@Autowired`.

```
@Controller
@RequestMapping("/pagamento")
@Scope(value=WebApplicationContext.SCOPE_REQUEST)
public class PagamentoController {

    @Autowired
    CarrinhoCompras carrinho;

    @Autowired
    RestTemplate restTemplate;
    [...]
}
```

No método `finalizar` faremos o seguinte procedimento: Criar uma `String` que guardará a url para onde a requisição será enviada. Chamar o método `postForObject` do objeto `restTemplate` usando a `String` criada, um objeto da classe `DadosPagamento` (que ainda não temos mas vamos criar daqui a pouco) usando o valor total do carrinho e por último, dizer que esperamos uma resposta do tipo `String`, passando `String.class` como último parâmetro.

```
String uri = "http://book-payment.herokuapp.com/payment";
restTemplate.postForObject(uri, new DadosPagamento(carrinho.getTotal()), String.class);
```

Já que se é esperada uma resposta da requisição enviada, vamos atribuir o resultado deste método a um objeto `String` que chamaremos de `response` e usa-lo para exibir a mensagem recebida tanto no console, quanto na *view*. Veja o método finalizar como fica ao final:

```
@RequestMapping(value="/finalizar", method=RequestMethod.POST)
public ModelAndView finalizar(RedirectAttributes model){

    String uri = "http://book-payment.herokuapp.com/payment";
    String response = restTemplate.postForObject(uri, new DadosPagamento(carrinho.getTotal()), S

    model.addFlashAttribute("sucesso", response);
    System.out.println(response);

    return new ModelAndView("redirect:/produtos");
}
```

Agora precisamos criar a classe `DadosPagamento` que deve receber o valor da compra e guardar em um atributo do tipo `BigDecimal`. Veja como fica simples esta classe:

```
public class DadosPagamento {
    private BigDecimal value;

    public DadosPagamento(BigDecimal value) {
        this.value = value;
    }

    public DadosPagamento() {
    }

    public BigDecimal getValue() {
        return value;
    }
}
```

Lembre-se de criar esta classe dentro do pacote de modelos. Observe que o nome do atributo é o mesmo que o sistema de pagamentos espera receber. Isto é necessário pois o **Spring** irá transformar o objeto desta classe em um objeto `JSON`. Por padrão ele irá criar a chave com o nome do atributo da classe e o valor será o mesmo do definido no atributo.

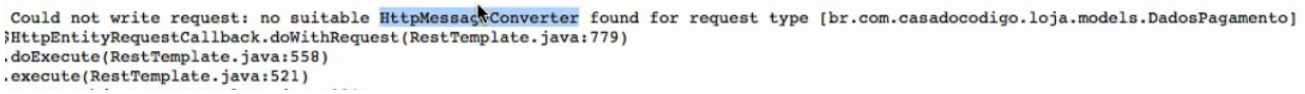
Para podermos testar, nos falta apenas criar uma configuração básica para que o **Spring** consiga criar o objeto `RestTemplate` corretamente. Para isso criaremos um novo método na classe `WebAppConfiguration` anotado com `@Bean` e que apenas retorna um objeto do tipo `RestTamplate`.

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class, ProdutoDAO.class, FileSaver.class, Carı
public class AppWebConfiguration extends WebMvcConfigurerAdapter {
    [...]
```

```
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }

}
```

Agora podemos testar nossa aplicação. Inicie o servidor e efetue o processo de compra, vá até a listagem de produtos, adicione alguns ao carrinho e clique em `Finalizar Compra`. E veja o que acontece:



Obtemos um erro 500 ao finalizar a compra. Por que? Veja o que diz a mensagem de erro:

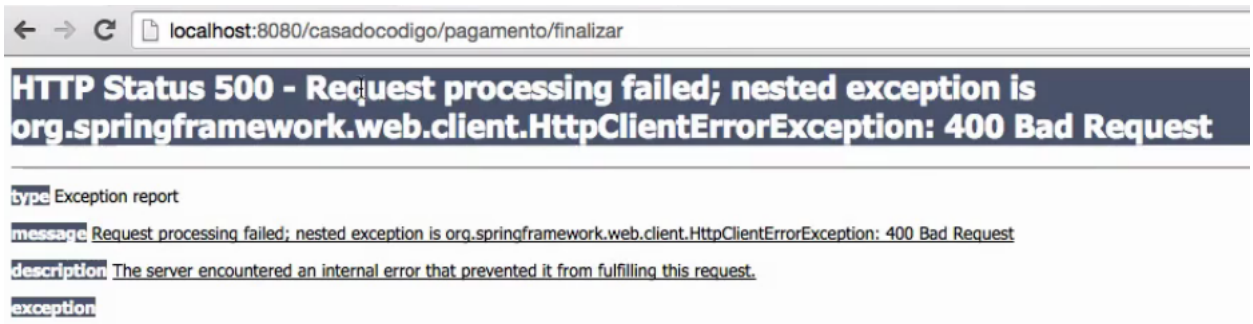
```
Could not write request: no suitable HttpMessageConverter found for request type [br.com.casado
```

O **Spring** não conseguiu transformar o objeto da classe `DadosPagamento` em `JSON`. Veja que ele pede por um `HttpMessageConverter`. Perceba que mesmo o **Spring** se encarregando de fazer a conversão do objeto de `DadosPagamento` para `JSON` não dissemos em lugar nenhum como ele deve fazer isto. Existe uma biblioteca que já faz essa tarefa, esta biblioteca se chama *jackson*. Vamos utilizá-la.

Abra seu `pom.xml` e adicione as seguintes dependencias:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.5.1</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.5.1</version>
</dependency>
```

Salve seu `pom.xml` e atualize o projeto. Tente iniciar e refazer o processo de compra. Tudo deve funcionar normalmente. Mas veja o que acontece quando testamos com um valor acima de 500.



Recebemos um erro do tipo `Bad Request`. Lembre-se que o sistema de pagamentos que estamos simulando retorna um erro de número 400 do `HTTP` quando o valor excede o valor de 500. Para que essa mensagem de erro seja mostrada para

o usuário, vamos fazer uso de um bloco `try/catch` e caso aconteça o erro, mostraremos uma mensagem amigável. Veja como ficará o método `finaliza` da classe `PagamentoController` :

```
@RequestMapping(value="/finalizar", method=RequestMethod.POST)
public ModelAndView finalizar(RedirectAttributes model){

    try {
        String uri = "http://book-payment.herokuapp.com/payment";
        String response = restTemplate.postForObject(uri, new DadosPagamento(carrinho.getTotal(
        model.addFlashAttribute("sucesso", response);
        System.out.println(response);
        return new ModelAndView("redirect:/produtos");
    } catch (HttpClientErrorException e) {
        e.printStackTrace();
        model.addFlashAttribute("falha", "Valor maior que o permitido");
        return new ModelAndView("redirect:/produtos");
    }
}
```

Com esta adição em nosso código, paramos de exibir a mensagem de erro do próprio Spring e passamos a exibir uma mensagem mais amigável ao usuário.Lembre-se de exibir a mensagem de falha através de `${falha}` no arquivo `lista.jsp` . Onde se encontra a listagem dos produtos.