02

Enviando email

Transcrição

Uma prática muito comum nas lojas online é o envio de email para o usuário, notificando-o que sua compra foi realizada com sucesso. Note que nossa aplicação apesar de fazer o processamento dos itens no carrinho de compras e exibir uma mensagem de sucesso, ela não envia emails. É o que faremos nesta aula, evoluir um pouco mais nossa aplicação.

Enviando Emails

Nós processamos os itens do carrinho e redirecionamos o usuário para página de produtos com o método `finalizar` da classe `PagamentoController` . Faremos agora com que antes do redirecionamento, seja chamado um método `enviaEmailCompraProduto` que será privado e o criaremos logo a seguir.

```
@RequestMapping(value="/finalizar", method=RequestMethod.POST)
public Callable<ModelAndView> finalizar(RedirectAttributes model){
    return () -> {
    String uri = "http://book-payment.herokuapp.com/payment";
        try {
        String response = restTemplate.postForObject(uri, new DadosPagamento(carrinho.getTotal()
      System.out.println(response);

      // envia email para o usuário
      enviaEmailCompraProduto();

      model.addFlashAttribute("sucesso", response);

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

        } catch (HttpClientErrorException e) {
            e.printStackTrace();
      model.addFlashAttribute("falha", "Valor maior que o permitido");
      return new ModelAndView("redirect:/produtos");
        }
    };
}
```

Note que estamos querendo enviar um email para o usuário que fez a compra e não estamos capturando este usuário de lugar algum. O método `enviaEmailCompraProduto` precisa saber para quem o email será enviado.

O *Spring Security* consegue nos passar este usuário com a anotação `@AuthenticationPrincipal` . Desta forma, podemos adicionar um novo parâmetro (`Usuario`) no método `finalizar` anotado com esta anotação, e após isso, passar este usuário para o método `enviaEmailCompraProduto` .

A assinatura do método então muda para:

```
@RequestMapping(value="/finalizar", method=RequestMethod.POST)
public Callable<ModelAndView> finalizar(@AuthenticationPrincipal Usuario usuario, RedirectAttril
```

E a chamada do método que envia o email agora passa a receber o usuário obtido através do *Spring Security*

```
//envia email para o usuário
enviaEmailCompraProduto(usuario);
```

Para a criação de emails, o próprio Java facilita esta tarefa por meio da classe `SimpleMailMessage` da qual, usando de uma instância, podemos fazer uso dos métodos `setSubject` , `setTo` , `setText` e `setFrom` que nos permite configurar o assunto, o destinatário, a mensagem e o remetente do email respectivamente. Veja como ficará nosso exemplo:

```
private void enviaEmailCompraProduto(Usuario usuario) {
    SimpleMailMessage email = new SimpleMailMessage();
    email.setSubject("Compra finalizada com sucesso");
    email.setTo(usuario.getEmail());
    email.setText("Compra aprovada com sucesso no valor de " + carrinho.getTotal());
    email.setFrom("compras@casadocodigo.com.br");
}
```

Perceba que estamos apenas enviando para o usuário uma mensagem de que sua compra foi realizada com sucesso e junto com esta estamos enviando o valor total da compra.

Com o email pronto, nos falta apenas envia-lo! Para isto, usaremos um "enviador de emails" do *Spring*: o **EmailSender**. Assim, criaremos um novo atributo do tipo `EmailSender` , anotaremos este com `@Autowired` e o chamaremos de `sender` . Ao final do método `enviaEmailCompraProduto` usaremos este objeto para enviar o email com o método `send` .

```
@RequestMapping("/pagamento")
@Controller
public class PagamentoController {

  @Autowired
  private CarrinhoCompras carrinho;

  @Autowired
  private RestTemplate restTemplate;

  @Autowired
  private MailSender sender;

  @RequestMapping(value="/finalizar", method=RequestMethod.POST)
  public Callable<ModelAndView> finalizar(@AuthenticationPrincipal Usuario usuario,
      RedirectAttributes model) {
    return () -> {
      String uri = "http://book-payment.herokuapp.com/payment";

      try {
        String response = restTemplate.postForObject(uri, new DadosPagamento(carrinho.getTotal(
        System.out.println(response);

        enviaEmailCompraProduto(usuario);

        model.addFlashAttribute("sucesso", response);
        return new ModelAndView("redirect:/produtos");
```

```
        } catch (HttpClientErrorException e) {
          e.printStackTrace();
          model.addFlashAttribute("falha", "Valor maior que o permitido");
          return new ModelAndView("redirect:/produtos");
        }
      };
    }

    private void enviaEmailCompraProduto(Usuario usuario) {
      SimpleMailMessage email = new SimpleMailMessage();

      email.setSubject("Compra finalizada com sucesso");
      email.setTo(usuario.getEmail());
      email.setText("Compra aprovada com sucesso no valor de " + carrinho.getTotal());
      email.setFrom("compras@casadocodigo.com.br");

      sender.send(email);
    }

}
```

Tudo configurado! Vamos ver se realmente funciona. O servidor deve inicializar normalmente, mas ao tentar acessar alguma página um erro no console do *Eclipse* é exibido.

```
No qualifying bean of type [org.springframework.mail.MailSender]
```

O erro notifica que o *Spring* não conseguiu encontrar um objeto *MailSender*. Já passamos por erros parecidos, lembra? Geralmente, este tipo de erro acontece por que não configuramos o que estavamos querendo usar. Vamos configurar o *MailSender*!

Na classe `AppWebConfiguration`, criaremos um novo método anotado com `@Bean` que retorna um objeto do tipo `MailSender`. A classe que implementa esta interface é a `JavaMailSenderImpl` e será através do objeto desta classe que iremos configurar todo o acesso ao servidor de emails.

```
@Bean
public MailSender mailSender(){
    JavaMailSenderImpl mailSender = new JavaMailSenderImpl();

    mailSender.setHost("smtp.gmail.com");
    mailSender.setUsername("alura.springmvc@gmail.com");
    mailSender.setPassword("alura2015");
    mailSender.setPort(587);

    Properties mailProperties = new Properties();
    mailProperties.put("mail.smtp.auth", true);
    mailProperties.put("mail.smtp.starttls.enable", true);

    mailSender.setJavaMailProperties(mailProperties);
    return mailSender;
}
```

O uso do objeto `mailProperties` é feito para que configurações adicionais sobre como a comunicação com o servidor **SMTP** irá acontecer.

Nestas configurações, estamos definindo o endereço do servidor, o `host`. O usuário e senha usados para a autenticação e a porta de comunicação. Além disso estamos habilitando a autenticação *SMTP* e também usando o tipo de conexão segura por meio de **TLS**.

Considerações Importantes: Apesar de disponibilizarmos estes dados para testes rápidos não é recomendado o uso em aplicações em produção. Para questões de testes, use seus próprios dados para verificar que a aplicação se comporta como esperado.

Usar o servidor do **GMail** para envio de emails através de aplicações deste tipo em produção também não é recomendado. Lembre-se de trocar as configurações para utilizar o servidor de emails da sua empresa quando publicar a aplicação online.

É provavel que estas configurações não funcionem quando forem executadas, pois o servidor do **GMail** pode bloquear a conexão por questões de segurança. Experimente! Caso não funcione, tente trocar as informações para usar seus próprios dados e se ainda não funcionar, procure rever as configurações de segurança da sua conta no **GMail** a fim de possibilitar o envio de email por aplicações externas.

Por último, precisaremos adicionar como dependência do nosso projeto a biblioteca do **Java Mail**, pois sem esta, o envio de email simplesmente não irá funcionar. No `pom.xml` adicionaremos o seguinte:

```
<dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
    <version>1.4.7</version>
</dependency>
```

Observação: Ao tentar executar a operação de compra, na finalização da mesma, seremos levados ao formulário de login. É o normal que se espera nestas aplicações, porém, por questões de teste, podemos liberar o acesso ao caminho `/pagamento/` na classe `SecurityConfiguration`.

```
.antMatchers("/pagamento/**").permitAll()
```

Lembre-se de trocar o email de destinatário para seu email na classe `PagamentosController` caso faça esta configuração. Caso não troque, poderá ter problemas. No mínimo não terá certeza de que o email está sendo enviado corretamente.

Agora ao executar o teste de operação de uma compra. Ao finalizar o carrinho, o email deve ser enviado sem problemas.