02

O padrão Model-View-Controller

Código com muita responsabilidade

Quando um código faz muita coisa além de seu propósito, ou seja possui muita responsabilidade, ele se torna cada vez mais dificil de manter e testar. Se tornam partes problemáticas do nosso sistema.

Alguns exemplos muito conhecidos com a linguagem java são aqueles `Servlets` que além de realizar toda a lógica, ainda geravam o html que era devolvido como resposta ao usuário. Ou então os `JSP` s, que além de cuidar do HTML tinham pedaços de código java e até mesmo código SQL. Por exemplo, no lugar de usar a classe `ProdutoController` com o método lista que criamos, poderíamos ter feito tudo isso no arquivo `lista.jsp` , algo como:

```
<%@page import="br.com.caelum.vraptor.model.Produto"%>
<%@page import="java.util.List"%>
<%@page import="br.com.caelum.vraptor.util.JPAUtil"%>
<%@page import="javax.persistence.EntityManager"%>
<%@page import="br.com.caelum.vraptor.dao.ProdutoDao"%>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Lista de produtos</title>
    </head>
<body>
    <table>
        <thead>
            <tr>
                <th>Nome</th>
                <th>Preço</th>
                <th>Quantidade</th>
            </tr>
        </thead>
        <tbody>
            <%
                EntityManager em = JPAUtil.criaEntityManager();
                ProdutoDao dao = new ProdutoDao(em);
                List<Produto> produtoList = dao.lista();

                for(Produto produto : produtoList) {
            %>
            <tr>
                <td><%= produto.getNome() %></td>
                <td><%= produto.getValor() %></td>
                <td><%= produto.getQuantidade() %></td>
            </tr>
            <% } %>
        </tbody>
    </table>
</body>
</html>
```

Estranho? Isso porque é só um pedacinho de código. Se fosse alguma função mais complexa esse código estaria ainda mais carregado. Repare que nesse mesmo arquivo temos `imports` das classes java, código java, código html, e cada nova necessidade que aparecer em nossa listagem vai ficar nele também. Imagine se tivermos varios `for` s e `if` s, como ficaria bagunçado o fechamento das indentações dessas lógicas sempre com o `<% } %>` misturado no html.

Para não sobrecarregar alguma das partes de nosso sistema, uma solução comum é separar suas responsabilidades.

Como organizar meu código web?

Ao desenvolver essa primeira funcionalidade da nossa aplicação, a listagem de produtos, já foi possivel notar que nosso código tem uma certa distribuição. Vamos entender agora como ele foi e será organizado ao decorrer do curso, assim como as motivações para utilizar esse padrão de organização.

Quando um usuário executa alguma ação em nosso sistema, por exemplo abrir listagem de produtos, alguma regra de negócio de nosso código é executada e por fim o usuário tem como resposta as informações exibidas em sua tela.

Repare que para isso acontecer passamos por 3 fases, ou **camadas** como costumam ser chamadas. Uma delas é a nossa regra de negócio (listar todos os produtos, neste caso). Outra camada é a de resposta ao usuário, responsável por dispor as informações ao final da ação. Por fim, precisamos de alguém para intermediar essas duas camadas, saber qual a lógica deverá ser executada de acordo com a ação realizada pelo usuário.

O padrão arquitetural MVC

Essas 3 camadas são a base de um conhecido padrão arquitetural, o MVC (`Model-View-Controller`). Ele propoe uma forma educada de lidarmos com a distribuição do nosso código, que é:

Model : Onde temos nossas regras de negócio, são classes que podem representar nossas entidades (como a classe `Produto`) ou que te ajudam a trabalhar com os dados da nossa aplicação (como a `ProdutoDao`).

View : Onde temos nossas interfaces de uso do sistema, em geral nossas páginas web. A `lista.jsp` é um exemplo de item da camada de `View` .

Controller : Onde temos os intermediários que conectam as outras duas outras camadas. Essa camada recebe informações da `View` e transforma essas informações em chamadas das nossas regras de negócio, da camada `Model` . Como é o caso do nosso `ProdutoController` .

Outros frameworks que utilizam o padrão MVC

Existem diversos frameworks que assim como o VRaptor utilizam o padrão arquitetural `Model-View-Controller` . Alguns dos principais exemplos são o Spring MVC (http://www.alura.com.br/cursos-online-java/spring-mvc), que também é escrito em java, e em outras linguagens temos o Ruby on Rails (http://www.alura.com.br/cursos-online-ruby/ruby-on-rails) e Asp.NET MVC (http://www.alura.com.br/cursos-online-net/aspnetmvc). Independente da linguagem ou framework que você utilizar, conhecer MVC é um passo fundamental para o desenvolvimento web.