



Visão Geral

Estática vs. Dinâmica

Freemarker

Java

Padrões

Verificação de Aprendizado

## VISÃO GERAL

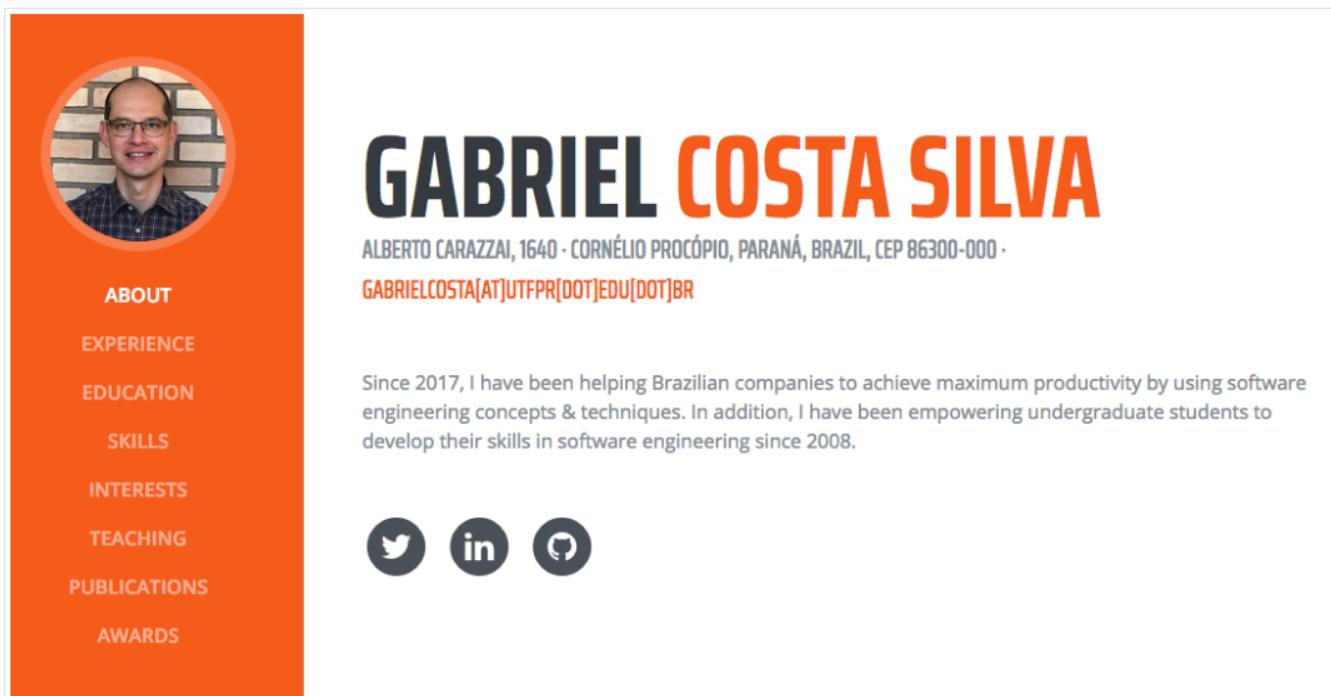
Nesta aula, nós transformamos a página estática em uma página dinâmica. Isso é necessário porque queremos que a tabela de cidades seja atualizada à medida que novas cidades são inseridas. Para isso, vamos precisar de mais uma tecnologia - o Freemarker. Em seguida, mudamos a página existente para uma nova pasta. Assim, o Spring Boot reconhece a página como uma página dinâmica. Também alteramos a extensão da página. O próximo passo é colocar o código dinâmico na página, usando a sintaxe do Freemarker. Também fazemos os ajustes necessários para implementar o padrão *MVC* no projeto.

Esses ajustes constituem os princípios fundamentais para o desenvolvimento de páginas dinâmicas. O vídeo mostra todo o processo enquanto as demais Seções explicam em detalhes o que está sendo feito e as razões para as mudanças.

Todo o projeto desenvolvido nesta aula está disponível no [Github](#), na branch `semana03-10-projeto-dinamico`.

## PÁGINA ESTÁTICA x DINÂMICA

A estrutura criada anteriormente serve bem se tivermos apenas páginas estáticas. O conteúdo de páginas estáticas não é atualizado a cada requisição do navegador. Veja o caso do meu Website pessoal (Figura abaixo).



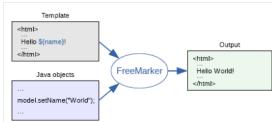
É uma página HTML simples e alguma formatação em CSS. Apesar do site ser atualizado uma ou duas vez por ano, a atualização de conteúdo é feita diretamente no código. É diferente de um site de comércio eletrônico, que dá boas-vindas personalizada usando seu nome, como a Figura a seguir.



A Figura acima ilustra um caso de página dinâmica. Uma página, ou site, dinâmico é aquele que pode ser atualizado automaticamente a cada requisição feita pelo navegador. No projeto do CRUD de cidades, a lista de cidades existentes é atualizada cada vez que a página é recarregada pelo navegador. Portanto, precisamos de uma página dinâmica.

## TORNANDO UMA PÁGINA DINÂMICA COM FREEMARKER

O Spring Boot, por meio do [Spring MVC](#), fornece suporte para criação de páginas dinâmicas. Isso é feito com o auxílio de uma [template engine](#). A template engine substitui os valores fixos por valores variáveis na página Web durante o processamento da requisição (Figura abaixo). O Spring Boot coopera com esse processo fornecendo o que a template engine precisa para fazer seu trabalho.



O Spring Boot se integra muito bem com várias template engines. Nesse curso, nós vamos usar o [Freemarker](#). O Freemarker é um projeto estável da Apache Software Foundation. Ele possui as características necessárias para criar uma página dinâmica que atende uma gama de projetos.

The screenshot shows the Apache Freemarker homepage. At the top, there's a navigation bar with links like Home, Manual, Java API, Contribute, Report a Bug, and Download. Below the header, there's a section titled "WHAT IS APACHE FREEMARKER™?" with a sub-section "What is Apache Freemarker™?". It features a diagram showing a "Template" box with code like "<html><body>Hello \${name}</body></html>" and a "Java objects" box with code like "... model.setName("World"); ...". An arrow labeled "FreeMarker" points from the template to the output, which shows the result: "<html><body>Hello World!</body></html>". The page also includes sections for Documentation, Tooling, and Community, along with a brief history of the tool.

A primeira coisa para adicionar o Freemarker ao nosso projeto é adicionar uma dependência do Freemarker no `pom.xml` do nosso projeto. As linhas 30 a 34 na Figura a seguir mostram a nova dependência inserida no `pom.xml`.

```

19   <dependencies>
20     <dependency>
21       <groupId>org.springframework.boot</groupId>
22       <artifactId>spring-boot-starter-web</artifactId>
23     </dependency>
24
25     <dependency>
26       <groupId>org.springframework.boot</groupId>
27       <artifactId>spring-boot-starter-test</artifactId>
28       <scope>test</scope>
29     </dependency>
30
31     <dependency>
32       <groupId>org.springframework.boot</groupId>
33       <artifactId>spring-boot-starter-freemarker</artifactId>
34     </dependency>
35   </dependencies>
  
```

O Spring Boot é um framework *opinionated*. Isso significa que ele usa configurações padrão. Como o Spring Boot tem suporte nativo ao Freemarker, ele já pré-configura tudo que é necessário para que o Freemarker funcione no projeto. Agora, só precisamos colocar as coisas no lugar certo e inserir o código.

Para que o Spring Boot reconheça nossa página dinâmica com o Freemarker, precisamos mover a página `crud.html` (atualmente estática) para a pasta `/resources/templates/`. Essa é a pasta padrão para páginas dinâmicas no Spring Boot. Outra alteração é a mudança da extensão da página. O Spring Boot reconhece por padrão páginas dinâmicas com o Freemarker desde que a página use a extensão `.ftl`. A Figura a seguir mostra como ficou a estrutura de arquivos após essas modificações.



O próximo passo é alterar o código estático da página usando a sintaxe do Freemarker. Vamos alterar a tabela para que as linhas da tabela sejam criadas dinamicamente de acordo com uma lista de cidades. Observe a diretiva `list` do Freemarker nas linhas 41 e 52, na Figura abaixo.

```

32   <table class="table table-striped table-hover mt-5">
33     <thead class="thead-dark">
34       <tr>
35         <th>Nome</th>
36         <th>Estado</th>
37         <th>Ações</th>
38       </tr>
39     </thead>
40     <tbody>
41       <#list listaCidades as cidade >
42         <tr>
43           <td>Londrina</td>
44           <td>Paraná</td>
45           <td>
46             <div class="d-flex d-justify-content-center">
47               <a class="btn btn-warning mr-3">ALTERAR</a>
48               <a class="btn btn-danger">EXCLUIR</a>
49             </div>
50           </td>
51         </tr>
52       </#list>
53     </tbody>
54   </table>
  
```

Uma diretiva é uma instrução do Freemarker que faz algo no código. Nesse caso, a diretiva `list` é usada para iterar sobre uma lista, chamada `listaCidades`. Essa diretiva funciona de forma similar ao `forEach` no Java. Cada cidade na lista será mapeada para uma variável chamada `cidade`. Depois, vamos usar essa variável para extrair o nome e o estado da cidade, nas linhas 43 e 44. Mas, por enquanto, vamos manter o código como está.

## ALTERANDO O CÓDIGO JAVA

Precisamos de uma classe para representar uma cidade. Para isso, vamos criar uma classe `Cidade` dentro do pacote `visao` (Figura a seguir).



A classe `Cidade` tem os atributos que representam uma cidade. No nosso caso, `nome` e `estado`. Nesse momento, tudo que precisamos é de um construtor que recolha valores iniciais e `get`s que retornem os valores dos atributos. Observem que essa classe é imutável (`final`). Isso porque não esperamos ficar alterando os dados de uma cidade. Veja como ficou a classe na Figura a seguir.

```
1 package br.edu.utfpr.cp.espjava.crudcidades.visao;
2
3 public final class Cidade {
4     private final String nome;
5     private final String estado;
6
7     public Cidade(final String nome, final String estado) {
8         this.nome = nome;
9         this.estado = estado;
10    }
11
12    public String getEstado() {
13        return estado;
14    }
15
16    public String getName() {
17        return nome;
18    }
19 }
```

Agora que já temos uma representação da cidade – a classe `Cidade`, podemos alterar a classe `CidadeController` para retornar uma lista de cidades. Vamos começar criando uma lista de cidades.

Para isso, vamos usar um `Set`. O `Set` é uma das interfaces do *Collections framework*, parte da biblioteca padrão do Java. A diferença entre o `Set` e outras interfaces do framework é que ele não permite elementos repetidos. Afinal, você não vai querer ter duas cidades com o mesmo nome no mesmo estado, não é mesmo?

Desde o Java 8, as interfaces do *Collections framework* tem o método estático `of`, usado para criar uma lista com valores pré-definidos. Criamos três novas cidades, estipulando um nome e estado para cada uma, conforme a Figura abaixo.

```
15 public String listar(Model memoria) {
16
17     var cidades = Set.of(
18         new Cidade("Cornélio Procópio", "PR"),
19         new Cidade("Assis", "SP"),
20         new Cidade("Itajaí", "SC")
21     );
22
23     memoria.addAttribute("listaCidades", cidades);
24 }
```

Agora, precisamos tomar essa lista de cidades (`cidades`) disponível para que o FreeMarker seja capaz de acessá-la e colocar os valores na página. Isso é feito usando um objeto do tipo `org.springframework.ui.Model`.

Para isso, criamos um parâmetro para o método `CidadeController.listar()` do tipo `org.springframework.ui.Model` (linha 15). Esse parâmetro representa uma espécie de memória compartilhada durante as requisições. Toda vez que uma solicitação vier ou for enviada para o navegador, ela traz essa memória junto.

Essa memória pode armazenar valores que você precisa enviar ou receber do navegador. No nosso caso, precisamos enviar a lista de cidades. A classe `org.springframework.ui.Model` tem o método `addAttribute()`, que permite inserir um par de valores. Nesse caso, queremos criar uma variável com o nome `listaCidades`. Essa é a mesma variável usada na página dinâmica pelo FreeMarker para listar as cidades (Veja a Figura na Seção anterior). Como valor, queremos enviar o `Set cidades`, que tem atualmente três cidades (linha 23).

O último passo é ajustar o retorno do método para que ele refita a mudança na página (linha 23). Veja abaixo o código completo.

```
1 package br.edu.utfpr.cp.espjava.crudcidades.visao;
2
3 import java.util.Set;
4
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.Model;
7 import org.springframework.web.bind.annotation.GetMapping;
8
9 @Controller
10 public class CidadeController {
11
12     @GetMapping("/")
13     public String listar(Model memoria) {
14
15         var cidades = Set.of(
16             new Cidade("Cornélio Procópio", "PR"),
17             new Cidade("Assis", "SP"),
18             new Cidade("Itajaí", "SC")
19         );
20
21         memoria.addAttribute("listaCidades", cidades);
22
23         return "/crud";
24     }
25 }
```

Se você executar a aplicação nesse ponto, vai perceber que a cidade Londrina/PR é listada três vezes. A diretiva está funcionando, mas nós não atualizamos os valores com o nome da cidade e seu estado na página `crud.ftl`. Por isso, vamos voltar na página `/resources/templates/crud.ftl` e atualizar a tabela nas linhas 43 e 44 conforme a Figura a seguir.

```
8     <tbody>
9         <#list listaCidades as cidade >
10            <tr>
11                <td>${cidade.nome}</td>
12                <td>${cidade.estado}</td>
13                <td>
14                    <div class="d-flex d-justify-content-center">
15                        <a class="btn btn-warning mr-3">ALTERAR</a>
16                        <a class="btn btn-danger">EXCLUIR</a>
17                    </div>
18                </td>
19            </tr>
20        </#list>
21    </tbody>
```

Essa linha é usada para acessar a variável `cidade`, que representa um objeto do tipo `Cidade`. A classe `Cidade`, que criamos anteriormente, tem dois atributos: `nome` e `estado`. Esses atributos são acessados por meio dos `get`s, que retornam o valor atual para cada cidade. Todo código dentro da diretiva `list` é repetido para cada elemento dentro de `listaCidades`. Ao executar a aplicação, você deve ver os valores da lista, conforme a Figura abaixo.

Nome	Estado	Ações
Itajaí	SC	<a href="#">ALTERAR</a> <a href="#">EXCLUIR</a>
Cornélio Procópio	PR	<a href="#">ALTERAR</a> <a href="#">EXCLUIR</a>
Assis	SP	<a href="#">ALTERAR</a> <a href="#">EXCLUIR</a>

Agora essa página dinâmica retorna valores presentes em uma lista. Mas não é só isso que você espera, certo? Queremos inserir valores dinamicamente na lista. Também queremos ser capazes de alterar e excluir esses valores. Precisamos de um CRUD completo. Vamos fazer isso na próxima aula, além de conhecer dos padrões muito utilizados no desenvolvimento Web: MVC e Front-controller.

A próxima aula encerra esse projeto básico e, em seguida, começamos um novo projeto com várias funcionalidades novas.

## PADRÕES

O desenvolvimento de software depende de um número grande de padrões. Esses padrões são soluções bem testadas para problemas recorrentes. Os tipos de problemas podem variar desde o projeto até a implantação. Talvez, vocês já tenham ouvido falar de [padrões de projeto](#). Esses padrões são soluções propostas para problemas envolvendo o projeto de aplicações.



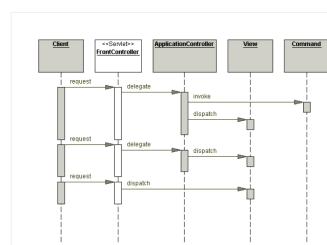
O desenvolvimento Web com Java (e na maioria das outras linguagens também) depende de dois padrões antigos e muito populares: [Front Controller](#) e MVC.

O [Front Controller](#) faz parte do catálogo J2EE. J2EE era como o plataforma Java para desenvolvimento empresarial costumava ser chamada há alguns anos. Hoje, ela é chamada de [Jakarta EE](#), e é mantida pela comunidade Eclipse. Você pode acessar o catálogo online, ou [comprar o livro](#) se preferir.

The screenshot shows the "Core J2EE Patterns" catalog page with the "Front Controller" pattern highlighted. The pattern description states: "This pattern defines a central point of entry for all requests into a system. It delegates the request to the appropriate object that handles it. This pattern is used in many contexts, such as the presentation layer of enterprise systems. These patterns are the most basic for real-world problems. The action of Core J2EE Patterns has been learned in a wide variety of patterns. (Not just the Front Controller pattern, but also the Command, Interpreter, Iterator, and Strategy patterns). The action of Core J2EE Patterns has learned in a wide variety of patterns. (Not just the Front Controller pattern, but also the Command, Interpreter, Iterator, and Strategy patterns). See also: Front Controller, Command, Interpreter, Iterator, Strategy, and State patterns." Other resources like Core J2EE Patterns Catalog and Core J2EE Patterns, 2nd Edition are also listed.

O objetivo do [Front Controller](#) é centralizar o tratamento de requisições na camada de apresentação. Isso faz muito sentido quando se pensa em aplicações Web. Afinal, não queremos que usuário tenha um conjunto de endereços e fique digitando os endereços para acessar cada uma das páginas da nossa aplicação. O que queremos é que, a partir de uma página, o usuário tenha acesso às demais páginas da aplicação.

O catálogo J2EE explica que o [Front Controller](#) recebe as requisições da aplicação e delega essas requisições para outras partes da aplicação, dependendo de regras de roteamento. Essa centralização facilita muito o controle de acesso à aplicação, por exemplo.



O [Spring MVC](#), projeto do Spring que é usado no Spring Boot para desenvolvimento de aplicações Web, é baseado primordialmente no [Front Controller](#).

### 1.1. DispatcherServlet

WebFlux

[Spring MVC](#), as many other web frameworks, is designed around the front controller pattern, where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components. This model is flexible and supports diverse workflows.

Gracias ao framework, você não precisa se preocupar em implementar o padrão ou gerenciar as requisições. O framework faz todo o trabalho pesado pra você e você só precisa se preocupar em definir as regras de roteamento. É exatamente isso que foi feito quando usamos a anotação `@GetMapping("/")` para definir o acesso à página `crud.ftlh`.

O outro padrão frequentemente usado em aplicações Web é o [MVC](#). MVC é a sigla em Inglês para *Model-View-Controller*. É importante ressaltar que o projeto do Spring que usa o [MVC](#) é chamado de [Spring Web MVC](#), isso serve para mostrar que o [MVC](#) é o padrão central para o desenvolvimento de aplicações Web com o Spring.

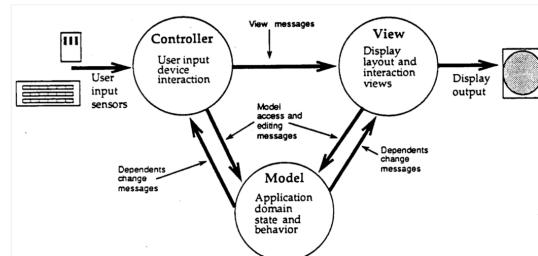
Eu tenho certeza que você já ouviu falar do [MVC](#). Um dos grandes problemas que temos em computação é que cada indivíduo implementa as coisas da forma que bem entende. Assim, uma procura rápida no Google e você vai encontrar 100 formas diferentes de implementar o [MVC](#), além de definições infinitas.

Aqui, vou usar o artigo [original](#) que definiu o [MVC](#) para explicar como ele é usado dentro do Spring Boot. Logo no começo, o artigo explica que o objetivo é criar aplicações gráficas, se referindo à interface gráfica com o usuário (GUI). Dessa forma, o padrão se aplica apenas à GUI. Eu já vi muita coisa nessa vida, inclusive gente juntando banco de dados e falando que isso é [MVC](#). Não de acordo com o artigo original!

## A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80

Glenn E. Krasner  
Stephen T. Pope

O padrão consiste de três papéis, que são exercidos na camada de apresentação (GUI). O Controller é responsável por receber a solicitação vindas do Front Controller, bem como é responsável por acionar o View. O View é responsável por receber e enviar dados de volta para o usuário da aplicação – normalmente, essa é a página Web. Por fim, o Model é responsável por encapsular dados e permitir que eles sejam transportados do Controller para o View e vice-versa. Veja o modelo no artigo original:



No Spring Boot, uma classe Java é usada para representar o Controller e outra, o Model. O View é representado por uma página Web, no nosso caso, uma página Web com Freemarker para permitir o dinamismo. No código criado até agora, a página `crud.ftlh` é o View, a classe `CidadeController` é o Controller, e a classe `Cidade` é o Model. Observe que o Spring Boot usa a anotação `@Controller` para identificar a classe como uma classe do tipo Controller, enquanto a classe `Cidade` não precisa de anotações especiais. Observe também que a classe `CidadeController` disponibiliza uma lista da classe `Cidade` para o `crud.ftlh` por meio do objeto `org.springframework.ui.Model`. Não confunda o objeto `org.springframework.ui.Model` com o `Model` do MVC. Esse objeto `org.springframework.ui.Model` funciona como uma memória compartilhada entre o Controller e o View, e é usado pelo Spring Boot para permitir que os dados transitem entre os dois.