# DIM0547 - DESENVOLVIMENTO DE SISTEMAS WEB II

09-12

# As regras do jogo

- **Não será aceito uso de IA** no desenvolvimento dos Trabalho ou Provas;
- **Trabalhos copiados** em todo ou em parte de outros estudantes ou da Internet receberão automaticamente nota zero;
- **Códigos que não compilam** serão avaliados, porém com penalidade;
- **Pontos extras** serão adicionados em códigos com uso de boas práticas de programação (**Se necessário para aprovação**);

# As regras do jogo

- **Trabalhos entregues fora do prazo não serão considerados;**

- **Atrasos resultam em meia falta;**

# As regras do jogo

***Faltas justificadas apenas mediante atestado***

***Qualquer problema, fale comigo, logo, não espero para o final***

# Avaliação

- Prova escrita;
- Trabalho Prático*;
- Trabalho Prático*;
- Reposição:
  - Prova escrita;
  - Trabalho Prático Correspondente + Algumas coisas;

* Apresentação é obrigatória

* Trabalhos em trio

# Aula 09

Thymeleaf - 01

# JSP (Java Server Pages)

JSP é o acrônimo para Java Server Pages, uma linguagem criada pela SUN gratuita, JSP é uma linguagem de script com especificação aberta que tem como objetivo primário a geração de conteúdo dinâmico para páginas da Internet.

Podemos ao invés de utilizar HTML para desenvolver páginas Web estáticas e sem funcionalidade, utilizar o JSP para criar dinamismo. É possível escrever HTML com códigos JSP embutidos.

# JSP (Java Server Pages)

JSP é o acrônimo para Java Server Pages, uma linguagem criada pela SUN gratuita, JSP é uma linguagem de script com especificação aberta que tem como objetivo primário a geração de conteúdo dinâmico para páginas da Internet.

Podemos ao invés de utilizar HTML para desenvolver páginas Web estáticas e sem funcionalidade, utilizar o JSP para criar dinamismo. É possível escrever HTML com códigos JSP embutidos.

# JSP (Java Server Pages)

Código Java misturado dentro do HTML (<% %>), o que deixava os arquivos difíceis de manter.

O HTML com JSP não era válido: designers não conseguiam abrir e editar diretamente no navegador.

A sintaxe era poluída e engessava o desenvolvimento.

# JSP (Java Server Pages)

Código Java misturado dentro do HTML (<% %>), o que deixava os arquivos difíceis de manter.

O HTML com JSP não era válido: designers não conseguiam abrir e editar diretamente no navegador.

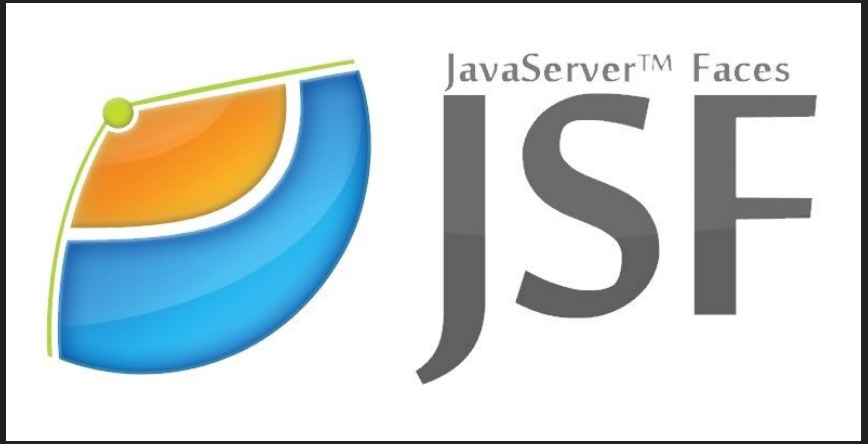
A sintaxe era poluída e engessava o desenvolvimento.

```
<!-- Saída de variável com Expression Language (EL) -->
<p>${mensagem}</p>

<!-- Uso de scriptlet (Java misturado no HTML) -->
<%
    String nome = "Maria";
    out.println("<p>Nome: " + nome + "</p>");
%>

<!-- Uso de loop com JSTL -->
<%
    java.util.List<String> nomes = java.util.Arrays.asList("Ana", "Bruno", "Carlos");
    request.setAttribute("nomes", nomes);
%>
<ul>
    <c:forEach var="n" items="${nomes}">
        <li>${n}</li>
    </c:forEach>
</ul>
```

# JavaServer Faces (JSF)

O JavaServer Faces (JSF) é um framework de interface de usuário criado pela Oracle no início dos anos 2000, como uma evolução em relação ao JSP.

A proposta do JSF era oferecer uma maneira mais estruturada e orientada a componentes para construir páginas web em Java. Em vez de escrever HTML misturado com código Java, como acontecia no JSP, o desenvolvedor utilizava tags especiais, geralmente com o prefixo <h:*>, que representavam componentes reutilizáveis, como formulários, campos de texto, botões e tabelas. Esses componentes eram conectados a Managed Beans, classes Java responsáveis por armazenar dados e controlar o fluxo da aplicação.

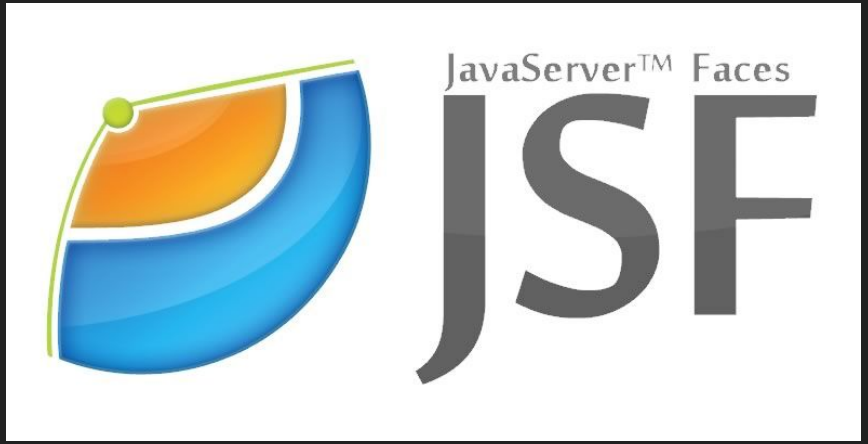


# JavaServer Faces (JSF)

O funcionamento do JSF seguia um ciclo de vida complexo. Quando o usuário enviava um formulário, o framework passava por diversas etapas internas, como restauração da visualização, aplicação dos valores recebidos, validação, atualização do modelo e, por fim, renderização da resposta.

Essa arquitetura permitia abstrair boa parte da lógica de tratamento das requisições, mas tornava o framework difícil de entender e, muitas vezes, complicado de debugar. Além disso, a navegação entre páginas e a configuração de beans eram feitas por meio de arquivos XML, o que aumentava a burocracia no desenvolvimento.

Apesar de sua proposta ambiciosa, o JSF apresentava problemas práticos. A rigidez dos componentes dificultava a personalização da interface, e o HTML gerado não era facilmente editável por designers, já que dependia de tags específicas que não funcionavam sem o servidor.



# JavaServer Faces (JSF)

O funcionamento do JSF seguia um ciclo de vida complexo. Quando o usuário enviava um formulário, o framework passava por diversas etapas internas, como restauração da visualização, aplicação dos valores recebidos, validação, atualização do modelo e, por fim, renderização da resposta.

Essa arquitetura permitia abstrair boa parte da lógica de tratamento das requisições, mas tornava o framework difícil de entender e, muitas vezes, complicado de debugar. Além disso, a navegação entre páginas e a configuração de beans eram feitas por meio de arquivos XML, o que aumentava a burocracia no desenvolvimento.

Apesar de sua proposta ambiciosa, o JSF apresentava problemas práticos. A rigidez dos componentes dificultava a personalização da interface, e o HTML gerado não era facilmente editável por designers, já que dependia de tags específicas que não funcionavam sem o servidor.

```
<h:head>
    <title>Exemplo JSF</title>
</h:head>
<h:body>
    <h:form>
        <h:outputLabel for="nome" value="Digite seu nome: " />
        <h:inputText id="nome" value="#{usuarioBean.nome}" />
        <h:commandButton value="Salvar" action="#{usuarioBean.salvar}" />
    </h:form>
</h:body>
</html>
```

# Thymeleaf

O Thymeleaf é um motor de templates (template engine) para aplicações Java.

Ele é usado principalmente junto com o Spring Framework / Spring Boot, mas pode ser integrado a qualquer aplicação Java.

A ideia central é permitir que o desenvolvedor misture HTML + dados dinâmicos vindos do backend.

# Thymeleaf

O Thymeleaf usa como base o HTML e incrementa o uso com “instruções dinâmicas” inseridas como atributos nas tags HTML.

Esses atributos têm o prefixo th:* (ex.: th:text, th:if, th:each) e por baixo dos panos, quem envia os dados é o Java, e quem renderiza no navegador continua sendo HTML puro.

# Thymeleaf

Ele surgiu como alternativa ao JSP (JavaServer Pages), que era a tecnologia mais usada antes.

# Thymeleaf

Nasceu inspirado em outras engines de template, como **Velocity** e **FreeMarker**, que também eram usadas em projetos Java.

Diferença:

Velocity/FreeMarker não mantêm os arquivos como HTML puro → eles usam sintaxes próprias.

O grande diferencial do Thymeleaf foi justamente o conceito de Natural Templates: o HTML continua sendo HTML válido, que pode ser aberto por qualquer navegador ou editor sem problemas.

# Thymeleaf

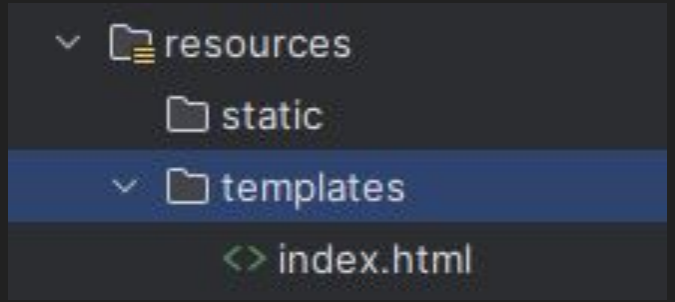
Para usa o Thymeleaf é necessário adicionar a seguinte dependência no pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

# Thymeleaf

**resources**: A pasta src/main/resources no Spring Boot é o local destinado a todos os recursos não Java da aplicação. Tudo que estiver aqui é carregado no classpath da aplicação.

Dentro dela temos várias subpastas especiais que o Spring reconhece automaticamente.



# Thymeleaf

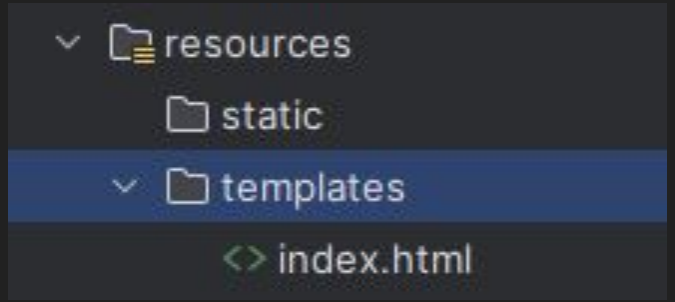
**static**: Essa pasta é usada para recursos estáticos.

Arquivos que não precisam ser processados no servidor antes de chegar ao navegador.

Exemplos típicos: CSS, JavaScript, imagens, fontes, PDFs.

O Spring Boot serve automaticamente qualquer arquivo colocado aqui.

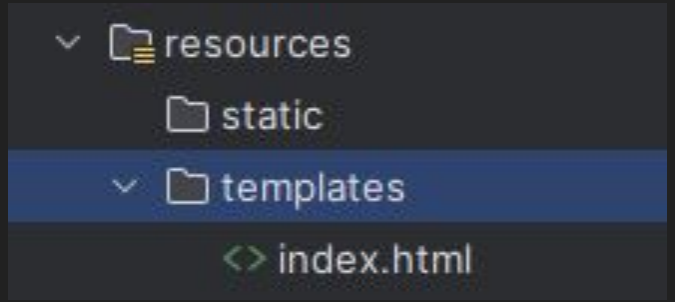
Se você colocar um arquivo src/main/resources/static/css/style.css, ele estará disponível no navegador em http://localhost:8080/css/style.css



# Thymeleaf

**templates**: Aqui ficam os templates do Thymeleaf.

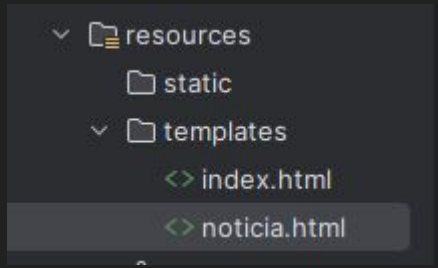
Os templates do Thymeleaf são arquivos HTML que contêm marcações especiais usadas para tornar a página dinâmica.



# Thymeleaf

Ao anotar a classe com @Controller e retornar um String, o Spring vai procurar a página com nome correspondente “noticia” e a engine do thymeleaf vai processar a página e devolver para o cliente.

```java
@Controller  no usages
@RequestMapping("/noticia")
public class NoticiaController {


    @GetMapping("/{id}/{titulo}")  no usages
    String noticia(@PathVariable String titulo, @PathVariable long id) {

        return "noticia";

    }
```



# Thymeleaf

Metadados da página

O <meta> é uma tag HTML usada para fornecer informações sobre a página. Essas informações não são exibidas diretamente ao usuário, mas são importantes para navegadores, mecanismos de busca (SEO) e para definir como o conteúdo deve ser interpretado.

```
<head>
    <meta charset="UTF-8">
    <title>Noticia</title>
</head>
```

# Thymeleaf

Metadados da página

O atributo charset dentro da tag <meta> define a codificação de caracteres da página. No seu exemplo, UTF-8 significa Unicode Transformation Format, 8 bits, que é a codificação mais comum atualmente.

```
<head>
    <meta charset="UTF-8">
    <title>Noticia</title>
</head>
```

# Thymeleaf

Por que o charset="UTF-8" é importante?

**Compatibilidade de caracteres**: garante que acentos, cedilhas, símbolos especiais e emojis sejam exibidos corretamente.

**Evita problemas de exibição**: sem a codificação correta, caracteres como á, ç ou é podem aparecer como símbolos estranhos (Ã¡, Ã§).

**Padrão moderno**: UTF-8 suporta praticamente todos os idiomas do mundo, diferente de codificações antigas como ISO-8859-1.

```
<head>
    <meta charset="UTF-8">
    <title>Noticia</title>
</head>
```

# Thymeleaf

Exibindo dados

O Model é apenas um objeto que o Spring fornece para você adicionar atributos que serão usados pelo template (como Thymeleaf).

Não importa a ordem em que você declara ele nos parâmetros; o Spring vai injetá-lo corretamente.

```
@GetMapping("/{id}/{titulo}")  no usages
String noticia(@PathVariable String titulo, @PathVariable long id, Model model) {

    Noticia noticia = new Noticia();
    noticia.id = id;
    noticia.titulo = titulo;

    model.addAttribute( attributeName: "noticia", noticia);
    return "noticia";

}
```

# Thymeleaf

Exibindo dados

O Model é apenas um objeto que o Spring fornece para você adicionar atributos que serão usados pelo template (como Thymeleaf).

Não importa a ordem em que você declara ele nos parâmetros; o Spring vai injetá-lo corretamente.

```
@GetMapping("/{id}/{titulo}")  no usages
String noticia(@PathVariable String titulo, @PathVariable long id, Model model) {

    Noticia noticia = new Noticia();
    noticia.id = id;
    noticia.titulo = titulo;

    model.addAttribute( attributeName: "noticia", noticia);
    return "noticia";

}
```

# Thymeleaf

Exibindo dados

O Model é apenas um objeto que o Spring fornece para você adicionar atributos que serão usados pelo template (como Thymeleaf).

Não importa a ordem em que você declara ele nos parâmetros; o Spring vai injetá-lo corretamente.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="ISO-8859-5">
    <title>Notícia</title>
</head>
<body>
    <span th:text="${noticia.titulo}"></span>

</body>
</html>
```

# Dialetos

O `th:text` substitui o conteúdo do elemento pelo valor de uma expressão do modelo (Model). É equivalente a “injetar” texto no HTML, mantendo o HTML válido.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="ISO-8859-5">
    <title>Notícia</title>
</head>
<body>
    <span th:text="${noticia.titulo}"></span>

</body>
</html>
```

# Dialetos

O `th:utext` substitui o conteúdo do elemento pelo valor de uma expressão do modelo (Model).

É equivalente a “injetar” texto no HTML, mantendo o HTML válido.

Diferente do `th:text` ele renderiza as tag HTML do texto.

```
<body>
    <span th:utext="${noticia.titulo}"></span>

</body>
```

http://localhost:9090/minhaapi/noticias/123?titulo=texto

# Dialetos

O `th:utext` substitui o conteúdo do elemento pelo valor de uma expressão do modelo (Model).

É equivalente a “injetar” texto no HTML, mantendo o HTML válido.

Diferente do `th:text` ele renderiza as tag HTML do texto.

```
<body>
    <span th:utext="${noticia.titulo}"></span>

</body>
```

http://localhost:9090/minhaapi/noticias/123?titulo=<script>alert('XSS')</script>

Exemplo que XSS

# Dialetos

O `th:if` controla se um elemento deve ou não ser renderizado, baseado em uma condição booleana.

```
<li th:if="${ativo}">...</li> <!-- TRE -->
<li th:if="${nome}">...</li> <!-- TRE -->
<li th:if="${quantidade}">...</li> <!-- FALSE -->
<li th:if="${lista}">...</li> <!-- TRUE -->
```

```
model.addAttribute( attributeName: "ativo", attributeValue: true);
model.addAttribute( attributeName: "nome", attributeValue: "");
model.addAttribute( attributeName: "quantidade", attributeValue: 0);
model.addAttribute( attributeName: "lista", Arrays.asList());
```

# Dialetos

O `th:if` controla se um elemento deve ou não ser renderizado, baseado em uma condição booleana.

```
<li th:if="${ativo}">...</li> <!-- TRE -->
<li th:if="${nome}">...</li> <!-- TRE -->
<li th:if="${quantidade}">...</li> <!-- FALSE -->
<li th:if="${lista}">...</li> <!-- TRUE -->
```

```
model.addAttribute( attributeName: "ativo", attributeValue: true);
model.addAttribute( attributeName: "nome", attributeValue: "");
model.addAttribute( attributeName: "quantidade", attributeValue: 0);
model.addAttribute( attributeName: "lista", Arrays.asList());
```

# Dialetos

O `th:unless` controla se um elemento deve ou não ser renderizado, baseado em uma condição booleana.

É equivalente ao um `th:if` com negação

```
<li th:if="${nome}">Nome não está vazio</li>
<li th:unless="${nome}">Nome está vazio</li>
```

# Dialetos

O Thymeleaf vem com várias expressões utilitárias (Utility Objects), que já estão disponíveis por padrão no contexto. Elas ajudam a manipular listas, strings, números, datas etc., direto no HTML.

```
<p th:text="${#strings.toUpperCase('ola mundo')}"></p>
<p th:text="${#strings.contains(nome, 'Silva')}"></p>
<p th:text="${#strings.length(nome)}"></p>
```

```
<p th:text="${#numbers.formatDecimal(1234.567, 1, 2)}"></p>
<p th:text="${#numbers.formatInteger(42,3)}"></p>
```

# Dialetos

Trabalhar com datas (java.util.Date, LocalDate, LocalDateTime).

```
<p th:text="${#dates.format(hoje, 'dd/MM/yyyy')}"></p>
<p th:text="${#dates.dayOfWeekName(hoje)}"></p>
```

# Dialetos

Semelhante ao #dates, mas voltado a java.util.Calendar.

```
<p th:text="${#calendars.format(cal, 'dd-MM-yyyy HH:mm')}"></p>
```

# Dialetos

Manipulação de listas e arrays.

```
<p th:text="${#lists.size(lista)}"></p>
<p th:text="${#lists.isEmpty(lista)}"></p>
<p th:text="${#lists.contains(lista, 'ABC')}"></p>
<p th:text="${#lists.sort(lista)}"></p>
```

# Dialetos

O `th:href` substitui o atributo href de links (<a>)

```
<a th:href="@{'/noticias/'+${noticia.id}}">/noticias</a>
<a th:href="@{/}">/index</a>
```

# Dialetos

O `th:src` substitui o atributo src de imagens, scripts ou iframes.

```
<img th:src="@{/imgs/spring-boot.png}" />
<link rel="stylesheet" th:href="@{/css/style.css}">
```

# Dialetos

`th:action` define o caminho da submissão do formulário, equivalente a action no HTML.

```
<form th:action="@{/usuarios/salvar}" th:object="${usuario}" method="post">

    <label>Nome:</label>
    <input type="text" th:field="*{nome}" />
    <br/><br/>
```

# Dialetos

`th:object` define o objeto do Model que será usado para binding automático com th:field

```
<form th:action="@{/usuarios/salvar}" th:object="${usuario}" method="post">

    <label>Nome:</label>
    <input type="text" th:field="*{nome}" />
    <br/><br/>
```

# Dialetos

`th:field="*{prop}"` conecta cada input a uma propriedade do objeto definido no th:object.

```
<form th:action="@{/usuarios/salvar}" th:object="${usuario}" method="post">

    <label>Nome:</label>
    <input type="text" th:field="*{nome}" />
    <br/><br/>
```

# Dialetos

O `th:value` define o valor de elementos de formulário, como <input>, <textarea> ou <select>, de forma dinâmica.

```
<label>Sexo:</label>
<input type="radio" th:field="*{sexo}" value="M"/> Masculino
<input type="radio" th:field="*{sexo}" value="F"/> Feminino
<br/><br/>
```

# Dialetos

O `th:each` é usado para iterar sobre listas, arrays ou coleções, permitindo criar elementos repetidos dinamicamente.

```
<ul>
    <li th:each="p : ${produtos}" th:text="${p.nome}"></li>
</ul>
```

# Dialetos

Você pode usar a sintaxe T(PACOTE.Classe)
para acessar enums ou constantes estáticas.

```
<select>
    <option th:value="${T(br.ufrb.imd.web2.atividade.Situacao).FAZENDO}"
            th:text="${T(br.ufrb.imd.web2.atividade.Situacao).FAZENDO.descricao}"></option>
    <option th:value="${T(br.ufrb.imd.web2.atividade.Situacao).FEITO}"
            th:text="${T(br.ufrb.imd.web2.atividade.Situacao).FEITO.descricao}"></option>
</select>
```

# Exercício

Desenvolva sistema de cadastro de notícias onde deve ser possível listar as notícias em uma página, com as opções de cadastrar e excluir.

Caso alguma regra de negócio seja quebrada, trate usando um controller advice que exibe uma página correspondente para o erro lançado.

# Aula 10

Thymeleaf - 02

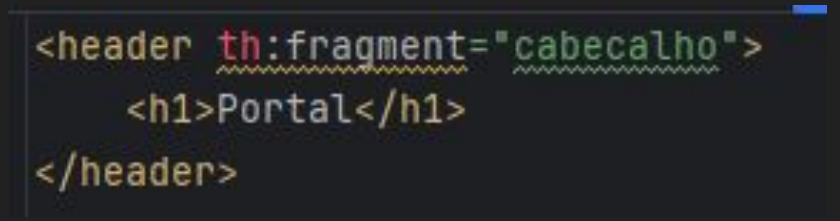
# templates

No Thymeleaf você pode criar templates reutilizáveis (cabeçalho, rodapé, menu, etc.) e depois incluir esses pedaços em outras páginas. Isso é feito principalmente com os fragmentos (`th:fragment` e `th:insert/th:replace`).

# templates

th:fragment

Serve para declarar uma parte do HTML como reutilizável.



# templates

`th:insert="arquivo :: fragmento"` → insere dentro da tag onde você chamou.

```
<div th:insert="fragments/header :: cabecalho"></div>
```

`th:replace="arquivo :: fragmento"` → substitui a tag inteira pelo fragmento.

```
<div th:replace="fragments/header :: cabecalho"></div>
```

# templates

Você também pode criar fragmentos dinâmicos e passar parâmetros

```
<header th:fragment="cabecalho(titulo)">
    <h1 th:text="${titulo}">Título Padrão</h1>
</header>
```

```
<div th:replace="fragments/header :: cabecalho('Página Inicial')"></div>
```

# Criando o Próprio Dialeto

Você pode criar dialetos personalizados e criar propriedade do tipo:

noticia:titulo-simples=”${noticia.titulo}”

```java
public class NoticiaDialect extends AbstractProcessorDialect {

    private static final String DIALECT_NAME = "Noticia Dialect";
    private static final String PREFIX = "noticia";

    public NoticiaDialect() {
        super(DIALECT_NAME, PREFIX, processorPrecedence:1000);
    }

    @Override
    public Set<IProcessor> getProcessors(String dialectPrefix) {

        Set<IProcessor> processors = new HashSet();
        processors.add(new TituloSimpleProcessor(dialectPrefix));
        return processors;

    }

}
```

# Criando o Próprio Dialeto

Você pode criar dialetos personalizados e criar propriedade do tipo:

noticia:titulo-simples="${noticia.titulo}"

```java
public class TituloSimpleProcessor extends AbstractAttributeTagProcessor {

    private static final String ATTR_NAME = "titulo-simples";
    private static final int PRECEDENCE = 1000;

    public TituloSimpleProcessor(final String dialectPrefix) {
        super(
            TemplateMode.HTML,          // This processor will apply only to HTML mode
            dialectPrefix,              // Prefix to be applied to name for matching
            elementName:null,           // No tag name: match any tag name
            prefixElementName:false,    // No prefix to be applied to tag name
            ATTR_NAME,                  // Name of the attribute that will be matched
            prefixAttributeName:true,   // Apply dialect prefix to attribute name
            PRECEDENCE,                 // Precedence (inside dialect's precedence)
            removeAttribute:true);      // Remove the matched attribute afterwards
    }


    protected void doProcess(
            ITemplateContext context,
            IProcessableElementTag tag,
            AttributeName attributeName,
            String attributeValue,
            IElementTagStructureHandler structureHandler) {

        IStandardExpression expression = StandardExpressions.getExpressionParser(context.getConfiguration())
                .parseExpression(context, attributeValue);

        Object rawValue = expression.execute(context);
        String titulo = rawValue == null ? "" : rawValue.toString();

        // Transforma em "titulo-noticia-simples"
        String simples = titulo.replace(target:" ", replacement:"-").toLowerCase();

        structureHandler.setBody(simples, processable:false);
    }
```

# Criando o Próprio Dialeto

Você pode criar dialetos personalizados e criar propriedade do tipo:

noticia:titulo-simples=”${noticia.titulo}”

```java
@Configuration
public class ThymeleafConfig {

    @Bean
    public SpringTemplateEngine templateEngine(SpringResourceTemplateResolver resolver) {
        SpringTemplateEngine engine = new SpringTemplateEngine();
        engine.setTemplateResolver(resolver);

        // Adiciona o seu dialect
        engine.addDialect(new NoticiaDialect());

        return engine;
    }

}
```

# Exercício

Faça uma aplicação onde é necessário criar um cabeçalho e rodapé.

Adicionar o cabeçalho e rodapé na tela de listagem de notícias.

Além disso, crie um dialeto para tratar alguma informação da notícia, por exemplo, exibir o título com todas as letras maiúsculas

# Aula 11

Atividade prática

# Exercício

Implemente um sistema de cadastro de notícias que possua:

API REST para gerenciar as notícias (CRUD completo).

- Deve respeitar os níveis do REST (mín: 2)
- Preparar para um possível versionamento

Páginas Web (Thymeleaf) que exiba os dados para o usuário final.

- Exibir uma tabelas o título da notícia;
- Ao clicar no título abrir uma página com o conteúdo da notícia.
- Adicionar um botão ‘Nova’ para adicionar um nova notícia.
- Estilizar a página com css ou algum framework/biblioteca.

Utilize exceções para tratar os erros de negócio juntamente com controller Advice para exibir a página ou mensagem correta na API

# Aula 12

Inversão de Controle (IoC), Injeção de Dependência (DI) e Services

# Inversão de Controle (IoC)

Uma classe não deve controlar diretamente como seus fluxo ou suas dependências são criadas ou gerenciadas; esse controle é “invertido” e passado para um mecanismo externo.

Ou seja:

- Antes (sem IoC): a própria classe cria e controla tudo que precisa.
- Com IoC: a classe apenas declara o que precisa, e alguém de fora entrega isso para ela.

# Inversão de Controle (IoC)

Exemplo sem IoC

A classe carro é responsável por definir que motor o carro vai usar.

Dessa maneira o código fica muito acoplado e não tem como, no momento da criação, trocar o motor de uma carro para outro tipo de motor.

```
class Motor {

}

class MotorFiat extends Motor {

}


class MotorWV extends Motor {

}

class Carro {
    private Motor motor = new Motor();
}
```

# Inversão de Controle (IoC)

Exemplo com IoC

Agora o Carro não controla a criação do motor.

O controle foi invertido: alguém de fora (um container ou outro código) cria o Motor e fornece ao Carro.

```
class Motor {

}

class MotorFiat extends Motor {

}


class MotorWV extends Motor {

}

class Carro {

    private Motor motor;

    public Carro(Motor motor) {
        this.motor = motor;
    }

}



Carro c = new Carro(new Motor());
Carro c = new Carro(new MotorFiat());
Carro c = new Carro(new MotorVW());
```

# Inversão de Controle (IoC)

IoC é um princípio onde o controle de criação e gerenciamento de dependências ou fluxo sai da classe e vai para fora dela, tornando o sistema mais flexível, desacoplado e fácil de manter.

```
class Motor {

}

class MotorFiat extends Motor {

}


class MotorWV extends Motor {

}

class Carro {

    private Motor motor;

    public Carro(Motor motor) {
        this.motor = motor;
    }

}



Carro c = new Carro(new Motor());
Carro c = new Carro(new MotorFiat());
Carro c = new Carro(new MotorVW());
```

# Injeção de Dependência (DI)

Em vez de uma classe criar suas dependências, essas dependências são fornecidas (injetadas) por um código externo, geralmente um container ou outra parte da aplicação.

Toda vez que você usa DI, você já está praticando IoC.

Não dá pra separar completamente, porque DI é uma forma de IoC.

```
class Carro {

    private Motor motor;

    public Carro(Motor motor) {
        this.motor = motor;
    }

}


Carro c = new Carro(new Motor());
Carro c = new Carro(new MotorFiat());
Carro c = new Carro(new MotorVW());
```

# Como Funciona o Spring

Quando você roda uma aplicação Spring Boot (SpringApplication.run(...)), o framework:

Sobe o IoC Container → é como uma “caixa” onde os objetos (beans) vão viver.

1. Escaneia o código → procura classes anotadas com @Component, @Service, @Repository, @Controller, etc.
2. Cria instâncias (beans) dessas classes.
3. Detecta dependências → vê quais classes precisam de outras (via construtor, @Autowired, etc.).
4. Faz a Injeção de Dependência (DI) → entrega automaticamente os objetos certos uns para os outros.
5. Gerencia o ciclo de vida dos beans (quando são criados, mantidos e destruídos).

# @Component

É o componente mais genérico do Spring. Diz apenas: “essa classe é um bean”.

Assim o Spring faz todo o processo de IoC e DI, mas perde conceitos semânticos e algumas funcionalidades extras

Útil quando a classe não se encaixa em “service”, “controller” ou “repository”.

# @Autowired

O @Autowired é uma anotação do Spring usada para injeção de dependência automática. Ele informa ao IoC Container que uma dependência deve ser resolvida e injetada automaticamente, sem que você precise criar a instância manualmente.

Funcionamento:

- O Spring procura um bean compatível no container (com base no tipo da variável).
- Se encontrar exatamente um, ele injeta.
- Se encontrar mais de um, pode ocorrer conflito, e aí você pode usar @Qualifier para especificar qual bean usar.
- Se não encontrar nenhum, lança exceção (a não ser que você marque como required = false).

# @Autowired

O @Autowired é uma anotação do Spring usada para injeção de dependência automática. Ele informa ao IoC Container que uma dependência deve ser resolvida e injetada automaticamente, sem que você precise criar a instância manualmente.

Funcionamento:

- O Spring procura um bean compatível no container (com base no tipo da variável).
- Se encontrar exatamente um, ele injeta.
- Se encontrar mais de um, pode ocorrer conflito, e aí você pode usar @Qualifier para especificar qual bean usar.
- Se não encontrar nenhum, lança exceção (a não ser que você marque como required = false).

```
@RestController  no usages
@RequestMapping("/noticias")
public class NoticiaController {

    @Autowired  1 usage
    NoticiaService noticiaService;
```

```
@Component  2 usages
public class NoticiaService {
```

# @Autowired

Mesmo para interações mais complexas como interfaces e suas implementações o @Autowired consegue mapear e reconhecer as dependências e injetar corretamente.

Mas há limitações…

```
public interface Pagamento {  4 usages

    abstract  public void pagar();  1
}
```

```
@Service  no usages
public class PagamentoPix implements Pagamento {


    @Override  no usages
    public void pagar() {
        System.out.println("Pagando por Pix");
    }
}
```

# @Autowired e suas limitações

Se há mais de uma implementação, o @Autowired não vai saber qual usar para resolver a dependência.

Nessa situação ele lança uma exceção.

```java
public interface Pagamento {  4 usages

    abstract  public void pagar();  1
}
```

```java
@Service  no usages
public class PagamentoPix implements Pagamento {


    @Override  no usages
    public void pagar() {
        System.out.println("Pagando por Pix");
    }
}
```

```java
@Service  no usages
public class PagamentoCartaoCredito implements Pagamento {

    @Override  1 usage
    public void pagar() {
        System.out.println("Pagando por Cartão de Crédito");
    }
}
```

# @Autowired e suas limitações

Se há mais de uma implementação, o @Autowired não vai saber qual usar para resolver a dependência.

Nessa situação ele lança uma exceção.

```
Description:

Field pagamento in br.ufrb.imd.web2.atividade.api.controller.NoticiaController required a single bean, but 2 were found:
    - pagamentoCartaoCredito: defined in file [D:\Downloads\atividade (1)\atividade\target\classes\br\ufrb\imd\web2\atividade\service\PagamentoCartaoCredito.class]
    - pagamentoPix: defined in file [D:\Downloads\atividade (1)\atividade\target\classes\br\ufrb\imd\web2\atividade\service\PagamentoPix.class]

This may be due to missing parameter name information

Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed
```

# @Primary

@Primary é uma anotação do Spring que você coloca em um bean para indicar que, quando houver mais de uma implementação compatível, esse bean deve ser escolhido por padrão.

Assim, por padrão, PagamentoPix vai ser a Injeção padrão do sistema.

```java
@Service  no usages
@Primary
public class PagamentoPix implements Pagamento {


    @Override  1 usage
    public void pagar() {
        System.out.println("Pagando por Pix");
    }
}
```

# @Qualifier

@Qualifier é uma anotação do Spring que resolve ambiguidades na injeção de dependência quando existem vários beans do mesmo tipo.

```java
@RestController  no usages
@RequestMapping("/noticias")
public class NoticiaController {

    @Autowired  1 usage
    NoticiaService noticiaService;

    @Autowired  1 usage
    @Qualifier("pagamentoPix")
    Pagamento pagamento;
```

# @Qualifier

Você pode definir o nome do Quilifier passando a informação no service.

```
@Service("servicePix")  no usages
public class PagamentoPix implements Pagamento {
```

Se nada for passado, o Spring considera como nome o nome da classe com a primeira letra minúscula

```
@RestController  no usages
@RequestMapping("/noticias")
public class NoticiaController {

    @Autowired  1 usage
    NoticiaService noticiaService;

    @Autowired  1 usage
    @Qualifier("servicePix")
    Pagamento pagamento;
```

# @Autowired em métodos e construtores

Quando você anota um método com @Autowired, o Spring chama esse método automaticamente e injeta os beans necessários como parâmetros.

É a recomendação do Spring inicializar via construtores, afinal, é pelos construtores que os objetos são criados.

```
public NoticiaController(Map<String, Pagamento> pagamentos) {  no u
    this.pagamentos = pagamentos;
}

@GetMapping(path = "/{id}")  no usages
public Noticia noticia(@PathVariable long id, @RequestParam(requir

    Noticia noticia = new Noticia();
    noticia.id = id;
    noticia.titulo = titulo;

    noticiaService.logNoticia(noticia);
    pagamentos.get("pagamentoCartaoCredito").pagar();

    return noticia;

}
```

# Mas como carregar dinamicamente

Tanto o @Primary como o Qualifier são fixos no código, então para carregar todas as dependências da minha classes de é via construtor.

O Spring varre todas as classes que implementam Pagamento e cria beans gerenciados para cada uma (PixService, CartaoCreditoService etc.).

Cada bean já passou pelo processo de injeção de dependências do Spring

```
public NoticiaController(Map<String, Pagamento> pagamentos) {  no usages
    this.pagamentos = pagamentos;
}

@GetMapping(path = "/{id}")  no usages
public Noticia noticia(@PathVariable long id, @RequestParam(required = fal

    Noticia noticia = new Noticia();
    noticia.id = id;
    noticia.titulo = titulo;

    noticiaService.logNoticia(noticia);
    pagamentos.get("pagamentoCartaoCredito").pagar();

    return noticia;

}
```

# @Order

@Order é uma anotação usada para definir a prioridade ou a ordem de execução de beans ou componentes em coleções quando o Spring precisa lidar com múltiplos beans do mesmo tipo.

```java
@RestController  no usages
@RequestMapping("/noticias")
public class NoticiaController {

    @Autowired  1 usage
    NoticiaService noticiaService;

    private final List<Pagamento> pagamentos;  2 usages

    public NoticiaController(List<Pagamento> pagamentos) {
        this.pagamentos = pagamentos;
    }
@Service  no usages
@Order(1)
public class PagamentoCartaoCredito implements Pagamento {

    @Service("servicePix")  no usages
    @Order(2)
    public class PagamentoPix implements Pagamento {
```

# @Order

A lista de pagamentos vai ter no primeiro índice o service de PagamentoCartaoCredito e no segundo índice PagamentoPix.

```
@RestController  no usages
@RequestMapping("/noticias")
public class NoticiaController {

    @Autowired  1 usage
    NoticiaService noticiaService;

    private final List<Pagamento> pagamentos;  2 usages

    public NoticiaController(List<Pagamento> pagamentos) {
        this.pagamentos = pagamentos;
    }
@Service  no usages
@Order(1)
public class PagamentoCartaoCredito implements Pagamento {

    @Service("servicePix")  no usages
    @Order(2)
    public class PagamentoPix implements Pagamento {
```

# Exercício - Sistema de Pagamentos com Spring Boot

Você deve implementar um sistema simples de processamento de pedidos, onde cada pedido pode ser pago por diferentes formas de pagamento.

Crie uma interface Pagamento com o método, com o método processar pagamento.

Crie a classe PedidoService, que deverá receber uma instância de Pagamento via injeção no construtor.

O deve ser informado o valor e o meio do pagamento.

Quando o pagamento estiver processado, retornar a informação que foi processado.

# Exercício - Sistema de Pagamentos com Spring Boot

Para cada meio de pagamento, adicionar provedores.

Um provedor pode estar disponível ou indisponível. Se um provedor estiver indisponível, tenta com o próximo.

Ao processar, exibir a informação dizendo que o processamento foi realizado pelo provedor X

Se todos estiverem indisponíveis, retornar a mensagem que não foi possível processar o pagamento.