



Java Web



Banco de dados relacionais	4
O que é banco de dados?	5
Sistema Gerenciador de Banco de Dados	6
Tabelas e chaves	8
Comandos SQL	9
União de Tabelas	13
JDBC - Java DataBase Connectivity	15
DAO - Data Access Object	17
Introdução a Java Enterprise Edition	18
O que é Java Enterprise Edition (JEE)?	19
Containers	21
APIs Java EE	23
Arquitetura de camadas JEE	25
Arquitetura MVC para Web	27
html, css e javaScript	29
Introdução ao HTML	30
O que são Tags e Elementos HTML?	31
Estrutura do HTML	32
Atributos	34
Formulários	36
Introdução ao CSS	38
Seletores	40
Introdução ao JavaScript	43
Sintaxe do JavaScript	45
Servlets	50
O que são servlets?	51
Estruturas de Aplicação Web	54
Deployment Descriptor: web.xml	56
Request e Response	58
Ciclo de Vida de um Servlet	61
Acessando um servlet	63

Trabalhando com servlets	65
Formulários	66
Navegação	68
Manipulando Cookies	71
Gerenciamento de Sessão	74
Interface Servlet Context	77
Filtros (Filter)	79
Listeners	82
Suporte a Annotations	84
JSP - JavaServer Pages	86
Ciclo de vida do JSP	87
Elemento JSP	89
Sintaxe e Semântica JSP	91
Objetos Implícitos	94
JavaBeans	97
Componentes Web Reutilizáveis	99
Tratamento de Erros	101
Expression Language JSTL	105
Expression Language	106
Sintaxe EL	108
Habilitando e Desabilitando EL e scriptlets	111
JSTL - JavaServer Pages Standard Tag Library	112
Biblioteca de Intercionalização	116
Custom Tags	119
Tag	120
Tag File	123
Diretivas (tag, attribute)	125
Simple Tag	127

Java Web

Banco de Dados Relacionais



Java Web

Bancos de Dados e Sistemas de Banco de Dados têm se tornado um componente essencial todos os dias da vida moderna. Aplicações de banco de dados tradicionais dizem respeito a sistemas bancários, sistemas de reservas aéreas, sistemas de controle de estoque.

Um banco de dados relacional organiza seus dados em **relações**. Cada relação pode ser vista como uma tabela, onde cada coluna corresponde a **atributos** da relação e as linhas correspondem às **tuplas** ou elementos da relação. Em uma nomenclatura mais próxima àquela de sistemas de arquivos, muitas vezes as tuplas são denominadas registros e os atributos, campos.

A visão de dados organizados em tabelas oferece um conceito simples e familiar para a estruturação dos dados, sendo um dos motivos do sucesso de sistemas relacionais de dados. Certamente, outros motivos para esse sucesso incluem o forte **embasamento matemático** por trás dos conceitos utilizados em bancos de dados relacionais e a uniformização na linguagem de manipulação de sistemas de bancos de dados relacionais através da linguagem **SQL**.

O que é banco de dados?



Java Web

O **banco de dados** é onde guardamos os dados que pertencem ao nosso sistema. A maioria dos bancos de dados comerciais hoje em dia são relacionais e derivam de uma estrutura diferente daquela **orientada a objetos**.

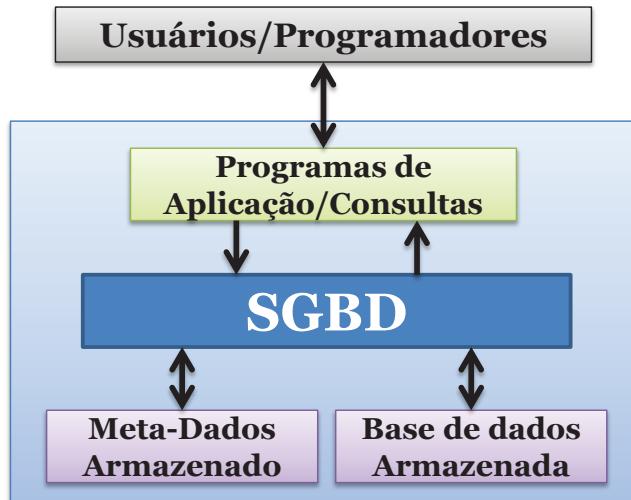
Assim como o nome já diz, um **banco de dados relacional** armazena dados como uma série de informações relacionadas. Grupos relacionados são expressos na forma de tabelas. Cada tabela contém colunas que definem as propriedades de cada grupo de dados armazenados.

A biblioteca padrão de persistência em banco de dados em Java é a JDBC mas já existem diversos projetos do tipo **ORM (Object Relational Mapping)** que solucionam muitos problemas que a estrutura da api do **JDBC (e ODBC)** gerou.

Alguns termos típicos:

- **Dados** - fatos que podem ser armazenados ex:nomes, telefones, endereços;
- **Base de dados** - coleção de dados inter-relacionados logicamente, ex: Agenda de telefones;
- **Sistema de Gerência de Bases de Dados (SGBD)** - coleção de programas que permite a criação e gerência de bases de dados.

Sistema Gerenciador de Banco de Dados



ORACLE
DATABASE



Java Web

Num computador o conjunto de dados é associado a um conjunto de programas para acessar estes dados, chamamos este sistema de Sistema Gerenciador de Banco de Dados (SGBD), note que o SGBD não é apenas o conjunto de dados, nem apenas o conjunto de programas, ele é os dois. O principal objetivo de um SGBD é proporcionar um ambiente tanto conveniente quanto eficiente para a recuperação e armazenamento das informações.

O Sistema de Banco de Dados veio evitar estes problemas. Os principais **objetivos de um Sistema de Banco de Dados** são:

- **Gerenciar grande quantidade de informação:** um Sistema de Banco de Dados pode armazenar simplesmente dados referentes a uma agenda de amigos, como também pode armazenar as informações relativas a uma usina nuclear. Em ambos os casos o Sistema de Banco de Dados tem que nos dar segurança e confiabilidade, independente se ele guardará 10 Megabytes ou 900 Gigabytes de informação.

- **Evitar redundância de dados e inconsistência:** redundância é manter a mesma informação em lugares diferentes, isto acontecia muito nos Sistemas de Arquivos, visto que novos programadores poderiam criar novos arquivos que conteriam um determinado dado que já está sendo armazenado em outro arquivo. Um dos problemas da redundância é que podemos atualizar um determinado dado de um arquivo e esta atualização não ser feita em todo o sistema, este problema é chamado de inconsistência. Um Sistema de Banco de Dados tenta evitar ao máximo estes erros, vendo ainda que a redundância causa desperdício de memória secundária e tempo computacional.

• **Facilitar o acesso:** um Sistema de Banco de Dados facilita ao máximo o acesso aos dados, visto que estes dados estarão no mesmo formato, o mesmo não acontecia no Sistema de Arquivos, onde os dados poderiam estar em diversos formatos e o acesso poderia até ser impossível. Outro ponto que um Sistema de Banco de Dados facilita é o acesso concorrente, onde podemos ter a mesma informação sendo compartilhada por diversos usuários.

• **Segurança aos Dados:** nem todos os usuários de banco de dados estão autorizados ao acesso a todos os dados. Imagine, se numa empresa todos funcionários tivessem acesso à folha de pagamento. O Sistema de Banco de Dados garante a segurança implementando senhas de acessos.

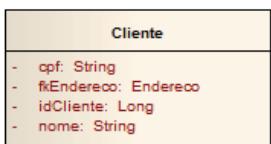
• **Garantir a Integridade:** é fazer com que os valores dos dados atribuídos e armazenados em um banco de dados devam satisfazer certas restrições para manutenção da consistência e coerência. Por exemplo, não podemos permitir a entrada de números onde é para entrar a sigla do Estado.

• **Facilitar Migração se necessário:** às vezes por motivos de velocidade ou de atualização precisamos mudar todo o Sistema Computacional, e os dados serão armazenados em um outro Banco de Dados. O ato de transferir as informações de um Banco de Dados para outro Banco de Dados é chamado de Migração, e facilitar esta Migração é um dos objetivos de um Sistema de Banco de Dados.

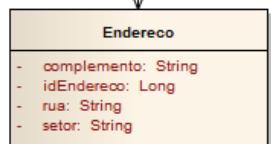
Tabelas e chaves

Modelo de Classe

[UML]



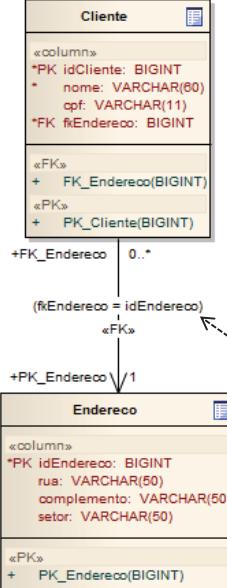
-fkEndereco



Java Beans

Modelo de Banco de Dados

[UML]



Representação das Tabelas

Tabela - Cliente

idCliente	nome	fkEndereco	cpf
001	JOAO AFONSO	101	02153565584
002	MARIA JOSEFINA	102	69535412584

Tabela – Endereco

idEndereco	rua	complemento	setor
101	Rua 200	Qd.20 Lt.15	Bueno
102	Rua T-7	Qd.15 Lt.18	Bueno

Chave Estrangeira



Java Web

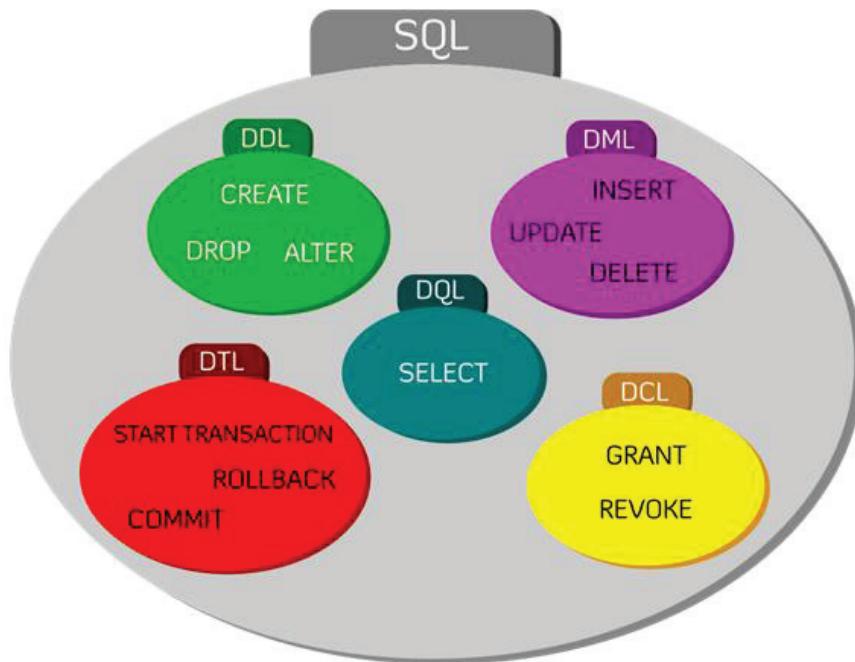
Um banco de dados relacional armazena dados em **tabelas**. Tabelas são organizadas em colunas, e cada coluna armazena um tipo de dados (inteiro, números reais, strings de caracteres, data, etc.). Os dados de uma simples “instância” de uma tabela são armazenados como uma **linha**. Por exemplo, a tabela *Cliente* teria colunas como *idCliente*, *primeiroNome* e *sobrenome*, e uma linha na tabela teria algo como **{001, “Arilo”, “Dias”}**.

No banco de dados relacional, é comum representar um contato (entidade) em uma tabela de contatos.

Tabelas tipicamente possuem **chaves**, uma ou mais colunas que unicamente identificam uma linha na tabela. No caso da tabela *Cliente* a chave seria a coluna *idCliente*. Para melhorar o tempo de acesso aos dados de uma tabela, são definidos **índices**. Um índice provê uma forma rápida para buscar dados em uma ou mais colunas em uma tabela, da mesma forma que o índice de um livro permite que nós encontremos uma informação específica rapidamente.

Chave estrangeira (foreign key) é o campo que estabelece o relacionamento entre duas tabelas. Assim, uma coluna corresponde à mesma coluna que é a chave primária de outra tabela. Dessa forma, deve-se especificar na tabela que contém a chave estrangeira quais são essas colunas e à qual tabela está relacionada, no nosso caso a chave estrangeira é **fkEndereco** presente na tabela *Cliente* que faz referência à tabela *Endereco*, isso é um relacionamento.

Comandos SQL



Java Web

SQL (*Structured Query Language*) ou Linguagem de Consulta Estruturada, uma linguagem padrão de gerenciamento de dados que interage com os principais bancos de dados baseados no modelo relacional.

Alguns dos principais sistemas que utilizam SQL são: MySQL, Oracle, Firebird, Microsoft Access, PostgreSQL (código aberto), HSQLDB (código aberto e escrito em Java).

SQL é uma linguagem essencialmente declarativa. Isso significa que o programador necessita apenas indicar qual o objetivo pretendido para que seja executado pelo SGBD.

A linguagem SQL é dividida em subconjuntos de acordo com as operações que queremos efetuar sobre um banco de dados, tais como: (Não seria melhor enumera-las aqui e depois elas continuam sendo explicadas?)

DML | Linguagem de Manipulação de Dados

DML é um subconjunto da linguagem SQL que é utilizado para realizar inclusões, consultas, alterações e exclusões de dados presentes em registros. Estas tarefas podem ser executadas em vários registros de diversas tabelas ao mesmo tempo. Os comandos que realizam respectivamente as funções acima referidas são Insert, Select, Update e Delete.

Função	Comandos SQL	Descrição do comando	Exemplo
Inclusões	INSERT	É usada para inserir um registro (formalmente uma tupla) a uma tabela existente.	Insert into Pessoa (id, nome, sexo) value;
Consultas	SELECT	O Select é o principal comando usado em SQL para realizar consultas a dados pertencentes a uma tabela.	Select * From Pessoa;
Alterações	UPDATE	Para mudar os valores de dados em uma ou mais linhas da tabela existente.	UPDATE Pessoa SET data_nascimento = '11/09/1985' WHERE id_pessoa = 7
Exclusões	DELETE	Permite remover linhas existentes de uma tabela.	DELETE FROM pessoa WHERE id_pessoa = 7

DDL | Linguagem de Definição de Dados

Uma DDL permite ao utilizador definir tabelas novas e elementos associados. A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL.

Os comandos básicos da DDL são poucos:

- **CREATE:** cria um objeto (uma Tabela) dentro da base de dados.
- **DROP:** apaga um objeto do banco de dados.

Alguns sistemas de banco de dados usam o comando ALTER, que permite ao usuário alterar um objeto, por exemplo, adicionando uma coluna a uma tabela existente.

Outros comandos *DDL* são: **CREATE TABLE, CREATE INDEX, CREATE VIEW, ALTER TABLE, ALTER INDEX, DROP INDEX, DROP VIEW.**

DCL | Linguagem de Controle de Dados

DCL controla os aspectos de autorização de dados e licenças de usuários para controlar quem tem acesso para ver ou manipular dados dentro do banco de dados.

Duas palavras-chaves da DCL:

- **GRANT:** Autoriza ao usuário executar ou setar operações.
- **REVOKE:** Remove ou restringe a capacidade de um usuário de executar operações.

DTL - Linguagem de Transação de Dados

- **BEGIN WORK** (ou **START TRANSACTION**, dependendo do dialeto SQL) pode

ser usado para marcar o começo de uma transação de banco de dados que pode ser completada ou não.

- **COMMIT** finaliza uma transação dentro de um sistema de gerenciamento de banco de dados.

- **ROLLBACK** faz com que as mudanças nos dados existentes desde o último COMMIT ou ROLLBACK sejam descartadas.

- **COMMIT** e **ROLLBACK** interagem com áreas de controle como transação e locação. Ambos terminam qualquer transação aberta e liberam qualquer cadeado ligado a dados. Na ausência de um BEGIN WORK ou uma declaração semelhante, a semântica de SQL é dependente da implementação.

DQL | Linguagem de Consulta de Dados

Embora tenha apenas um comando, a DQL é a parte da SQL mais utilizada. O comando SELECT permite ao usuário especificar uma consulta (“query”) como uma descrição do resultado desejado. Esse comando é composto de várias cláusulas e opções, possibilitando elaborar consultas das mais simples às mais elaboradas.

Cláusulas

As cláusulas são condições de modificação utilizadas para definir os dados que deseja selecionar ou modificar em uma consulta.

- **FROM** - Utilizada para especificar a tabela que se vai selecionar os registros.
- **WHERE** - Utilizada para especificar as condições que devem reunir os registros que serão selecionados.
- **GROUP BY** - Utilizada para separar os registros selecionados em grupos específicos.
- **HAVING** - Utilizada para expressar a condição que deve satisfazer cada grupo.
- **ORDER BY** - Utilizada para ordenar os registros selecionados com uma ordem específica.
- **DISTINCT** - Utilizada para selecionar dados sem repetição.

Operadores Lógicos

- **AND** - E lógico. Avalia as condições e devolve um valor verdadeiro caso ambos sejam corretos.
- **OR** - OU lógico. Avalia as condições e devolve um valor verdadeiro se algum for correto.
- **NOT** - Negação lógica. Devolve o valor contrário da expressão.

Operadores relacionais

O SQL possui operadores relacionais, que são usados para realizar comparações entre valores, em estruturas de controle.

Operador	Descrição
<	Menor
>	Maior
<=	Menor ou igual
>=	Maior ou igual
=	Igual
<>	Diferente

- **BETWEEN** - Utilizado para especificar um intervalo de valores.
- **LIKE** - Utilizado na comparação de um modelo e para especificar registros de um banco de dados. “like” + extensão % significa buscar todos resultados com o mesmo início da extensão.
- **IN** - Utilizado para verificar se o valor procurado está dentro de uma lista.
Ex.: valor IN (1,2,3,4).

Funções de Agregação

As funções de agregação, como os exemplos abaixo, são usadas dentro de uma cláusula SELECT em grupos de registros para devolver um único valor que se aplica a um grupo de registros.

- **AVG** - Utilizada para calcular a média dos valores de um campo determinado.
- **COUNT** - Utilizada para devolver o número de registros da seleção.
- **SUM** - Utilizada para devolver a soma de todos os valores de um campo determinado.
- **MAX** - Utilizada para devolver o valor mais alto de um campo especificado.
- **MIN** - Utilizada para devolver o valor mais baixo de um campo especificado.

União de Tabelas

União da tabela Cliente com Endereco							
idCliente	nome	fkEndereco	cpf	idEndereco	rua	complemento	setor
001	JOAO AFONSO	101	02153565584	101	Rua 200	Qd.20 Lt.15	Bueno
002	MARIA JOSEFINA	102	69535412584	102	Rua T-7	Qd.15 Lt.18	Bueno



Java Web

Uma das dificuldades dos iniciantes na linguagem SQL é a construção de consultas utilizando junções entre tabelas, um recurso fundamental para visualizar os dados de um banco relacional.

O resultado de uma junção pode ser utilizado para retornar os dados para um usuário ou aplicação, mas também pode ser referenciado em uma consulta como uma nova tabela, permitindo que novas junções e filtros sejam realizados sobre este sub-resultado. Além disso, pode se selecionar colunas específicas de cada tabela participante de uma junção ou mesmo não retornar colunas de uma tabela.

Na maioria dos casos, uma **INNER JOIN** rende os resultados mais relevantes para operações de união.

Há casos onde as entradas de uma tabela deveriam aparecer não importando a existência de dados de outra tabela, a utilização de **LEFT JOIN** ou **RIGHT JOIN** é mais apropriada.

Inner Join

A junção mais utilizada para explorar os relacionamentos em tabelas é o **INNER JOIN**. Através desta junção, são retornadas todas as linhas das tabelas A e B que correspondam ao critério estabelecido na cláusula **ON**.

Left Join

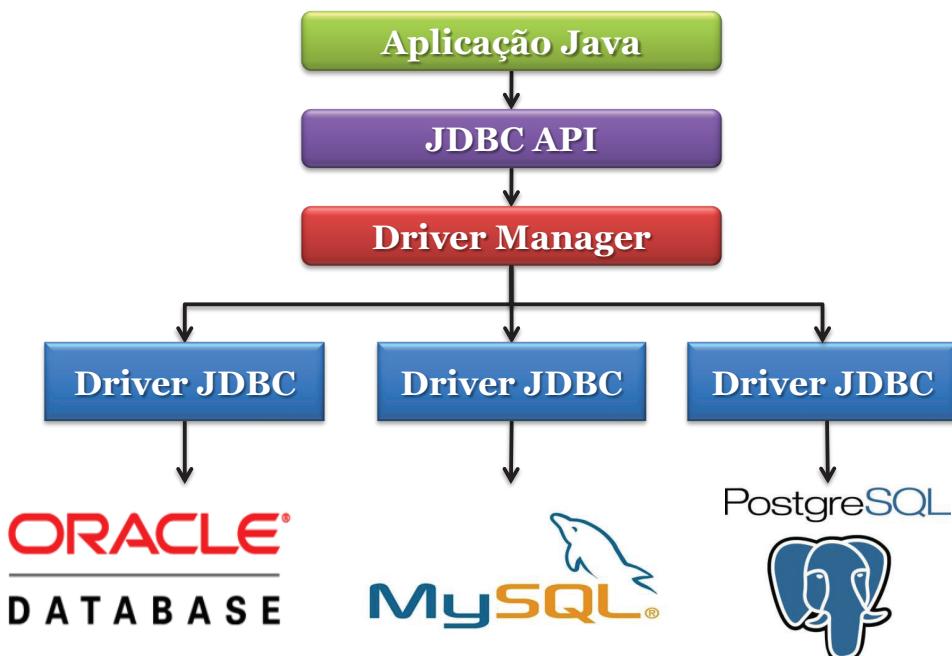
Um outro tipo de junção disponível na linguagem SQL é o *LEFT JOIN*. Ao juntar a tabela A com a tabela B utilizando o *LEFT JOIN*, todas as linhas de A serão retornadas, mesmo que não seja encontrada uma linha em B que atenda ao critério de alguma linha em A.

Se uma linha da tabela A não estiver associada a uma linha da tabela B e o resultado da consulta apresentar colunas da tabela B, estas colunas serão retornadas com o valor *NULL* para esta linha.

Right Join

O *RIGHT JOIN* é muito semelhante ao *LEFT JOIN*. A diferença fundamental é que serão retornadas todas as linhas da segunda tabela participante da junção.

JDBC - Java DataBase Connectivity



Java Web

Java Database Connectivity ou JDBC é um conjunto de classes e **interfaces (API)** escritas em Java que faz o envio de **instruções SQL** para qualquer banco de dados relacional. Por meio desta os desenvolvedores podem acessar bases de dados não importando quem seja seu fabricante; os desenvolvedores de um JDBC proveem a implementação para as interfaces definidas nesta API, fornecendo o mesmo grupo de funcionalidades ao desenvolvedor do sistema.

As seguintes classes estão na **API JDBC**:

- **java.sql.Connection** – Representa a conexão com o banco de dados. Encapsula os detalhes de como a comunicação com o servidor é realizada.
- **java.sql.DriverManager** – Gerencia os **drivers JDBC** utilizados pela aplicação. Em conjunto com o endereço e a autenticação, pode fornecer objetos de conexão.
- **java.sql.Statement** – Fornece meios ao desenvolvedor para que se possa executar comandos SQL.
- **java.sql.ResultSet** – Representa o resultado de um comando SQL. Estes objetos normalmente são retornados por métodos.

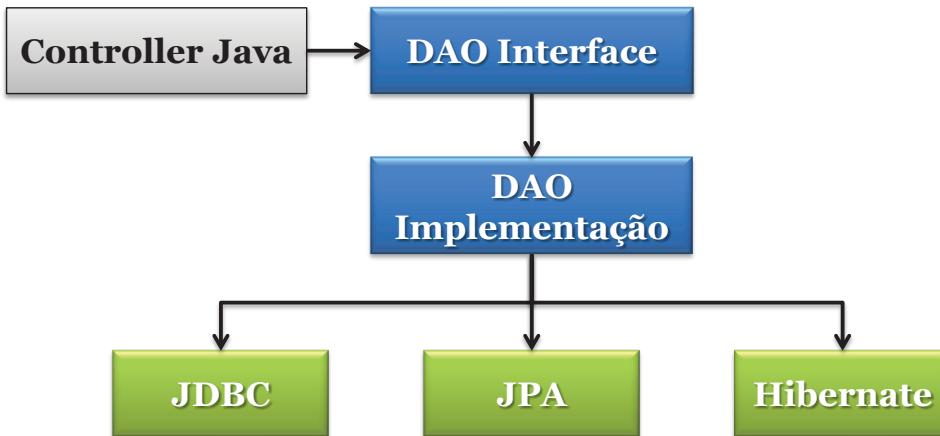
java.sql.DriverManager

Utilizando esta classe, o desenvolvedor pode retornar um objeto de conexão que pode ser usado para executar tarefas relativas ao banco de dados. Dois passos são necessários para tal:

- **Primeiro**, o **driver JDBC** deve estar registrado com **DriverManager**. Isto pode ser feito utilizando o método **Class.forName** que carrega a classe do driver para a memória.

- **Segundo**, utilizando o método **getConnection()**, mediante informação de uma **URL**, assim como a **senha** e o nome do **usuário** autenticado no banco de dados. A URL deve seguir a sintaxe requisitada pela implementação do banco de dados.

DAO – Data Access Object



Java Web

O padrão Data Access Object (DAO) é um padrão introduzido no ambiente JEE para simplificar e desacoplar a interação das aplicações Java com a API JDBC.

Este padrão permite criar as classes de dados independentemente da fonte de dados ser um **BD** relacional, um arquivo texto, um arquivo XML, etc. Para isso, ele encapsula os mecanismos de acesso a dados e cria uma interface de cliente genérica para fazer o acesso aos dados permitindo que os mecanismos de acesso a dados sejam alterados independentemente do código que utiliza os dados.

O objeto DAO é responsável por operar o mecanismo de persistência em nome da aplicação tipicamente executando os quatro tipos de operações - Criar, Recuperar, Alterar, Apagar - conhecidas pela sigla CRUD.

Existem diversas implementações do padrão **DAO** mas em geral podemos relacionar algumas características desejáveis em uma implementação do padrão **DAO**:

- Todo o acesso aos dados deve ser feita através das classes DAO de forma a se ter o encapsulamento;
- Cada instância da DAO é responsável por um objeto de domínio;
- O DAO deve ser responsável pelas operações CRUD no domínio;
- O DAO não deve ser responsável por transações, sessões ou conexões que devem ser tratados fora do DAO;

Java Web

Introdução a Java Enterprise Edition



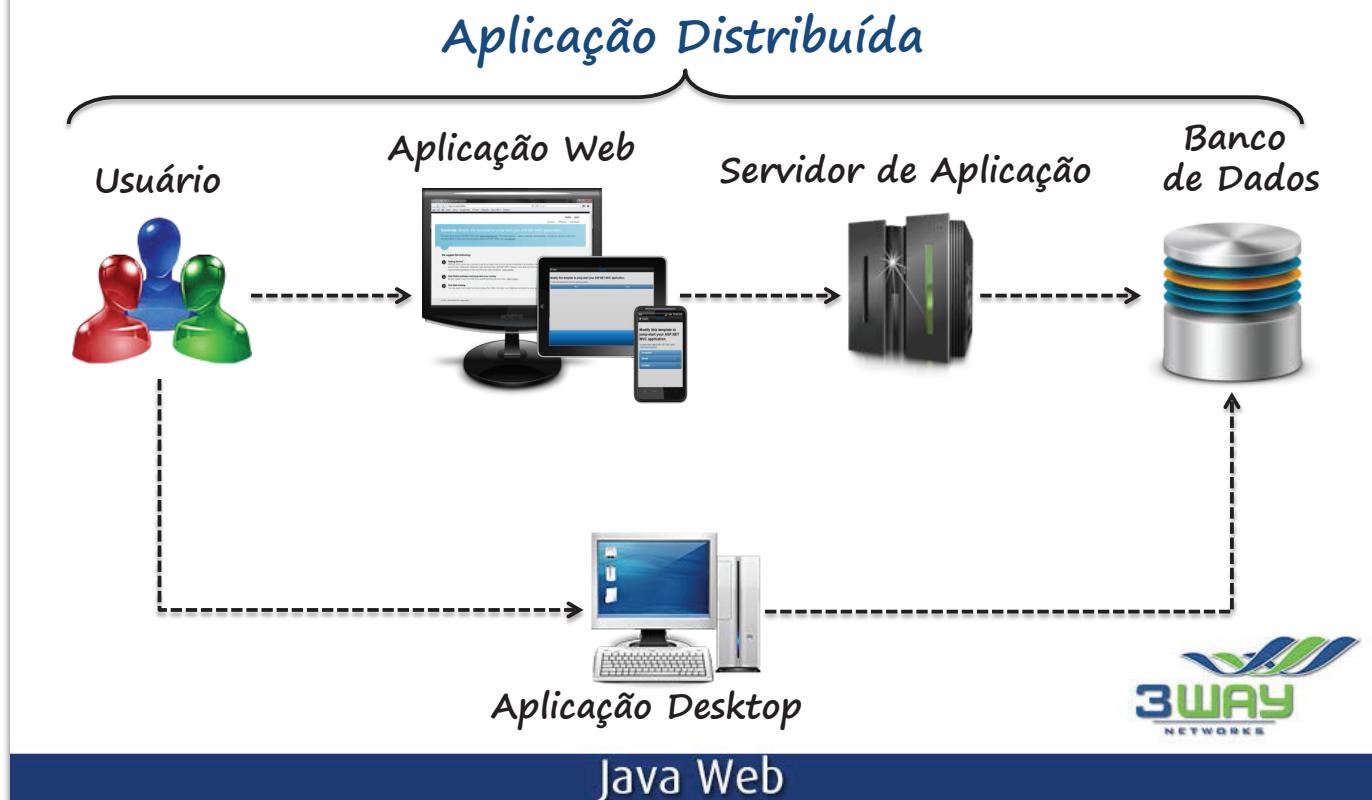
Java Web

Java EE é um padrão para desenvolvimento de sistemas web em Java, mas as diversas especificações, implementações e frameworks necessários para desenvolvimento de projetos profissionais, deixam as pessoas um pouco confusas no começo.

Uma aplicação Java Web gera páginas Web interativas, que contêm vários tipos de linguagem de marcação (HTML, XML, etc.) e conteúdo dinâmico. Normalmente é composto por componentes Web, como JavaServer Pages (JSP), servlets e JavaBeans.

O Java EE (Enterprise Edition) é uma plataforma amplamente usada que contém uma gama de tecnologias coordenadas que reduz significativamente o custo e a complexidade do desenvolvimento, implantação e gerenciamento de aplicações de várias camadas centradas no servidor. O Java EE é construído sobre a plataforma Java SE e oferece um conjunto de APIs (interfaces de programação de aplicações) para desenvolvimento e execução de aplicações portáteis, robustas, escaláveis, confiáveis e seguras no lado do servidor.

O que é Java Enterprise Edition (JEE)?



O que é uma Aplicação Java Web?

Uma aplicação Java Web gera páginas Web interativas, que contêm vários tipos de linguagem de marcação (HTML, XML, etc.) e conteúdo dinâmico. Normalmente é composto por componentes Web, como JavaServer Pages (JSP), servlets e JavaBeans para modificar e armazenar dados temporariamente, interagir com bancos de dados e Web services processando o conteúdo como resposta às solicitações do cliente.

Como a maioria das tarefas envolvidas no desenvolvimento de aplicações Web, pode ser repetitiva ou exigir um excedente de código padrão, os frameworks da Web podem ser aplicados para aliviar a sobrecarga associada às atividades comuns. Muitos frameworks, como JavaServer Faces, fornecem, por exemplo, bibliotecas para páginas de modelo e gerenciamento de sessão, e geralmente fomentam a reutilização do código.

O que é Java Enterprise Edition (JEE)?

O Java EE (Enterprise Edition) é uma plataforma amplamente usada que contém um conjunto de tecnologias coordenadas que reduz significativamente o custo e a complexidade do desenvolvimento, implantação e gerenciamento de aplicações de várias camadas centradas no servidor. O Java EE é construído sobre a plataforma Java SE e oferece um conjunto de APIs (interfaces de programação de aplicações) para desenvolvimento e execução de aplicações portáteis, robustas, escaláveis, confiáveis e seguras no lado do servidor.

Porque Java EE tem sido tão utilizado?

Além de ter uma vantagem numérica, pois não elimina parte do mercado que usa um sistema operacional específico, também aumenta a integrabilidade da aplicação, ou seja, permite que às empresas integrem os sistemas que estão rodando em diferentes plataformas como, por exemplo, o sistema Web.

Containers

GlassFish



Java Web

Containers são **servidores de objetos**, também chamados de **servidores de aplicação** que oferecem serviços e infraestrutura para a execução de componentes. O conceito de **Container** é independente da plataforma Java EE, utilizado em outras linguagens e plataformas.

Existem outros tipos de Contêiner utilizados em Java, considera-se a seguinte divisão de perfil de Containers Java e Java EE:

Tipo	Descrição / Exemplo
Client-side	<p>Responsável pelo ciclo de vida da aplicação, gerenciamento de eventos, bibliotecas, entre outros.</p> <p>Exemplos:</p> <p>Applet Contêiner - para painéis gráficos desenvolvidos com AWT/Swing controlados por browser;</p> <p>Application client Contêiner - aplicações standalone (AWT/Swing), podendo, opcionalmente, ser distribuídas por Java Web Start.</p>

Tipo	Descrição / Exemplo
Server-side	<p>Um Contêiner server-side, gerencia, além do ciclo de vida de componentes, recursos e meios de acesso. Configuramos no Contêiner os recursos que desejamos disponibilizar para que nossas aplicações os acessem através de APIs de serviços, como no caso de um pooling de Conexões a Banco de Dados.</p> <p>Exemplos:</p> <p>Web Contêiner - para objetos dirigidos por HTTP (Servlets e JSP);</p> <p>EJB Contêiner - para objetos de negócio server-side.</p>

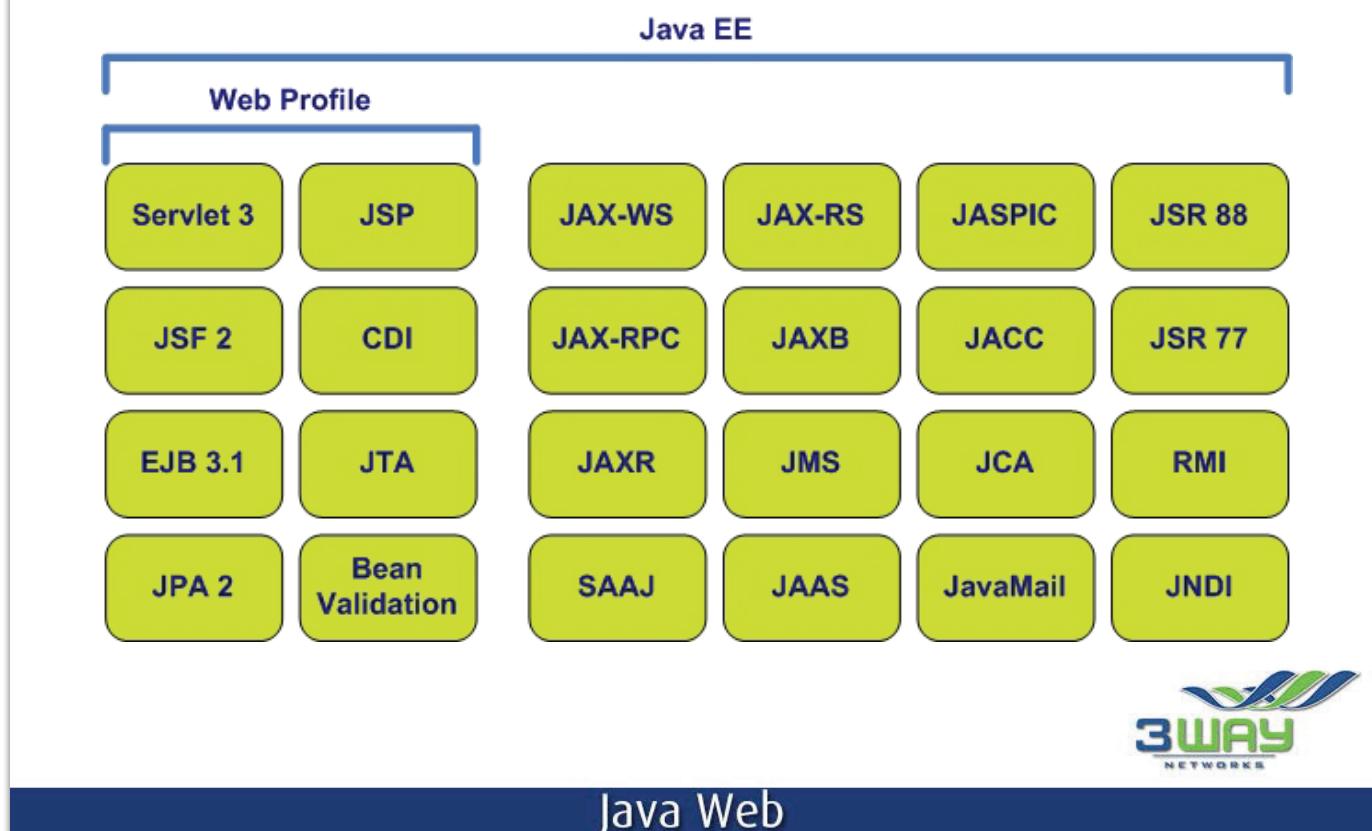
Além da definição das interfaces na API, **Java Enterprise Edition** também especifica como o Contêiner deve ser, quais recursos ele deve obrigatoriamente implementar, quais serviços ele deve oferecer. Permitindo assim que diversas empresas implementem seus próprios Containers.

Como já vimos a plataforma Java EE está fortemente baseada em Containers, como o foco do nosso curso é o desenvolvimento de aplicações Java para Web, nosso alvo são os **Web Containers**. Alguns dos serviços oferecidos pelos Web Containers são:

- Gerenciamento dos recursos utilizados pelos componentes, como **pool de conexões**;
- Gerenciamento do ciclo de vida dos componentes (**Servlets, JSPs e Custom Tags**);
- Gerenciamento de sessões de usuários;
- Controle de acesso.

As duas principais **APIs** suportadas por um **Contêiner Web** são: **Java Servlets e Java Server Pages**. A implementação de referência de **Container Web** é o **Tomcat**.

APIs Java EE



Java Web

O conjunto de **APIs Java EE** é definido, em sua maioria, por interfaces que podem ser empregadas pelos desenvolvedores das aplicações corporativas. Tais APIs possuem vínculos com o núcleo (kernel) do servidor que executa tarefas voltadas para o gerenciamento de recursos e infra-estrutura.

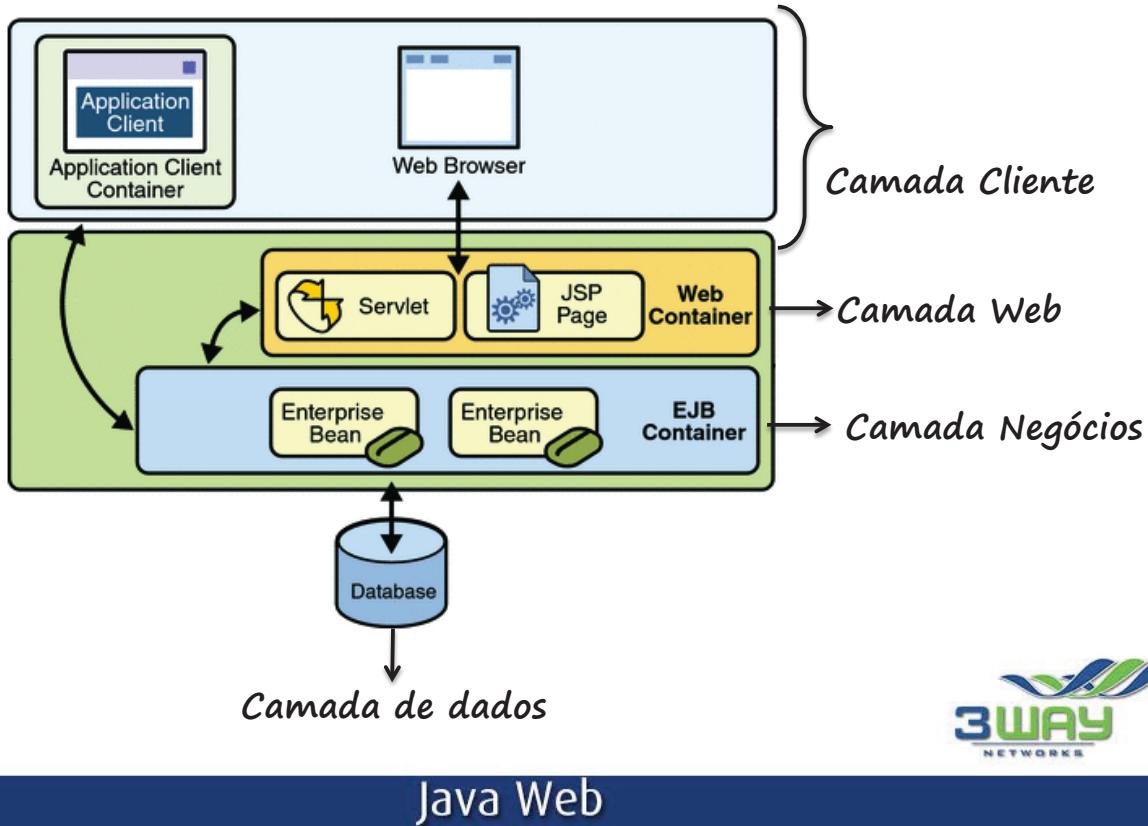
Podemos dizer que, ao desenvolvermos aplicações Java EE através do uso de APIs disponibilizadas no **Java Enterprise Edition (Reference Implementation)**, elas podem ser executadas nos servidores de aplicações que tenham implementado as especificações técnicas dos servidores JAVA EE.

Contamos com as seguintes APIs na plataforma Java EE:

API	Descrição
JDBC Extension	Extensão da API JDBC.
Enterprise JavaBeans (EJB)	Componentes gerenciados pelo EJB Contêiner , que oferece serviços de transação, multi-threading, persistência automática , entre outros para os componentes.
Java Servlets	Componentes frequentemente utilizados para integração entre as páginas Web e a camada de negócio.
JavaServer Pages (JSP)	API utilizada principalmente para construção de páginas dinâmicas .
Java Message Service (JMS)	API para tratamento de mensagens assíncronas .

API	Descrição
Java Transaction API (JTA)	API para controle manual de transações .
JavaMail	Utilizada para envio e recebimento de e-mails .
Java API for XML Processing (JAXP)	Processamento de XML .
Java Naming and Directory Interface (JNDI)	API que oferece acesso a Catálogo de Objetos .
Java Connector Architecture	API que padroniza os conectores para integração de aplicações .
Java API for XML Web Services (JAX-WS)	APIs para construção e utilização de Web Services .

Arquitetura de camadas JEE



Java Web

Java JEE tem 4 camadas básicas definidas no modelo de aplicação. São as seguintes:

- **Camada Cliente** - Parte do software que roda no computador do usuário;
- **Camada Web** - Parte do software que roda no servidor web, que por sua vez fica normalmente dentro do servidor de aplicações;
- **Camada de negócios** - Parte do software que roda no servidor de aplicações;
- **Camada de dados** - Banco de dados e sistemas externos;

Camada cliente

Estão na camada cliente softwares que rodam no browser, por exemplo, como páginas HTML, JavaScript, CSS, etc. Também rodam na camada cliente aplicações desktop, como Visual Basic ou Delphi, aplicativos Swing, SWT, AWT e afins. Todos esses tem em comum o fato de rodarem na máquina do usuário.

Nessa camada as aplicações não realizam funções complexas como acesso a banco de dados, cálculos, processamento de regras de negócio, conexão com sistemas externos, etc. Eles devem conter pura e simplesmente a interface com o usuário, tendo somente lógica de apresentação.

Camada web

Nessa camada estão os JSPs, servlets que normalmente rodam num servidor web como o Tomcat. Normalmente, esse tipo de aplicação é empacotado em um arquivo WAR (Web Archive), que pode ser instalado num servidor web.

Camada de negócios

Nessa camada Ficam os EJBs, classes de negócio, DAOs, classes que acessam sistemas externos e consomem web services e etc.

Normalmente a camada de negócios fica empacotada em um arquivo JAR contendo os EJBs e demais classes de negócio e acesso a dados da aplicação, que pode ser instalada em um container EJB.

Outra forma de empacotamento oferecida pela especificação J2EE são os arquivos EAR, ou Enterprise Application Archive. Esses pacotes simplesmente agrupam arquivos WAR, EJB-JAR e suas dependências em um único pacote, que guarda toda a parte de software que deve rodar em um servidor J2EE, ou seja, as camadas WEB e de negócios.

Camada de dados

Nessa camada ficam as tabelas de bancos de dados, índices e tudo mais que o servidor de banco suporta. Essa camada deve se limitar somente ao fornecimento de dados, sem qualquer lógica de negócio. O motivo disso é muito simples, não há como a especificação J2EE garantir escalabilidade executando regras de negócio rodando num banco Oracle, por exemplo. Já rodando num container J2EE, a especificação permite configurar a aplicação para rodar em um Cluster sem qualquer modificação (ou com mínimas modificações) no código fonte da aplicação.

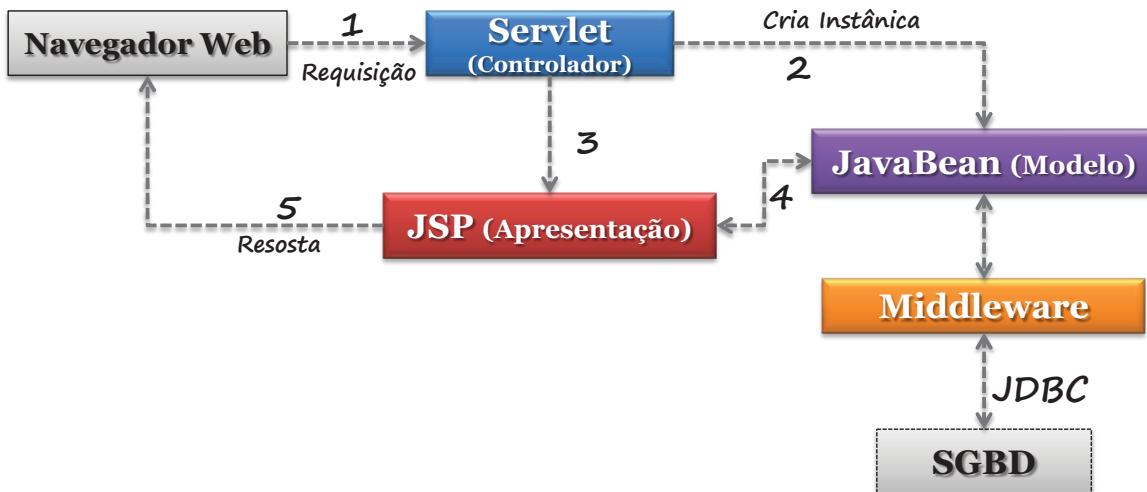
Para conseguir as vantagens oferecidas pela especificação J2EE, é necessário seguir a linha guia proposta pela mesma.

Acoplamento de camadas

Algumas aplicações processam todas as regras de negócio na aplicação web. Nesse caso, tanto a camada web quanto a camada de negócios ficam numa mesma aplicação web, sem necessidade de um servidor de aplicações. Isso não significa, contudo, que a aplicação não tenha uma camada de negócios. A definição de camada de negócios diz que esta é a parte do software que realiza as regras de negócio.

Em casos como esse, cuidado deve ser tomado para não misturar código web com código que contém regras de negócio. Por exemplo, processar regras de negócio em servlets torna as camadas web e de negócio acopladas, pois a mesma classe que processa lógica de apresentação contém as regras de negócio. Mesmo não usando EJBs, é uma boa prática criar classes de negócio (podem ser POJO mesmo) que contenham as regras de negócio e são chamadas a partir dos servlets ou das actions.

Arquitetura MVC para Web



Java Web

Para exemplificar a figura acima contém um diagrama de blocos que mostra a participação de **Servlets**, **JSP** e **JavaBeans** em uma arquitetura. A ideia é isolar cada aspecto do modelo MVC com a tecnologia mais adequada. A página JSP é ótima para fazer o papel da visão, uma vez que possui facilidades para a inserção de componentes visuais e para a apresentação de informação. No entanto, é um pouco estranho usar uma página JSP para receber e tratar uma requisição. Esta tarefa, que se enquadra no aspecto de controle do modelo MVC é mais adequada a um Servlet, uma vez que neste momento componentes de apresentação são indesejáveis. Finalmente, é desejável que a modelagem do negócio fique isolada dos aspectos de interação. A proposta é que a modelagem do negócio fique contida em classes de **JavaBeans**. Em aplicações mais sofisticadas a modelagem do negócio deve ser implementada por classes de **Enterprise JavaBeans (EJB)**, no entanto esta forma de implementação foge ao escopo deste material.

- **Servlets** - Atuam como controladores, recebendo as requisições dos usuários. Após a realização das análises necessária sobre a requisição, instancia o JavaBean e o armazena no escopo adequado (ou não caso o Bean já tenha sido criado no escopo) e encaminha a requisição para a página JSP.

- **JavaBeans** - Atuam como o modelo da solução, independente da requisição e da forma de apresentação. Comunicam-se com a camada intermediária que encapsula a lógica do problema.

- **JSP** - Atuam na camada de apresentação utilizando os JavaBeans para obtenção dos dados a serem exibidos, isolando-se assim de como os dados são obtidos. O objetivo é minimizar a quantidade de código colocado na página.

- **Camada Intermediária (Middleware)** - Incorporam a lógica de acesso aos dados. Permitem isolar os outros módulos de problemas como estratégias de acesso aos dados e desempenho. O uso de **EJB (Enterprise JavaBeans)** é recomendado para a implementação do **Middleware**, uma vez que os EJBs possuem capacidades para gerência de transações e persistência. Isto implica na adoção de um servidor de aplicação habilitado para EJB.

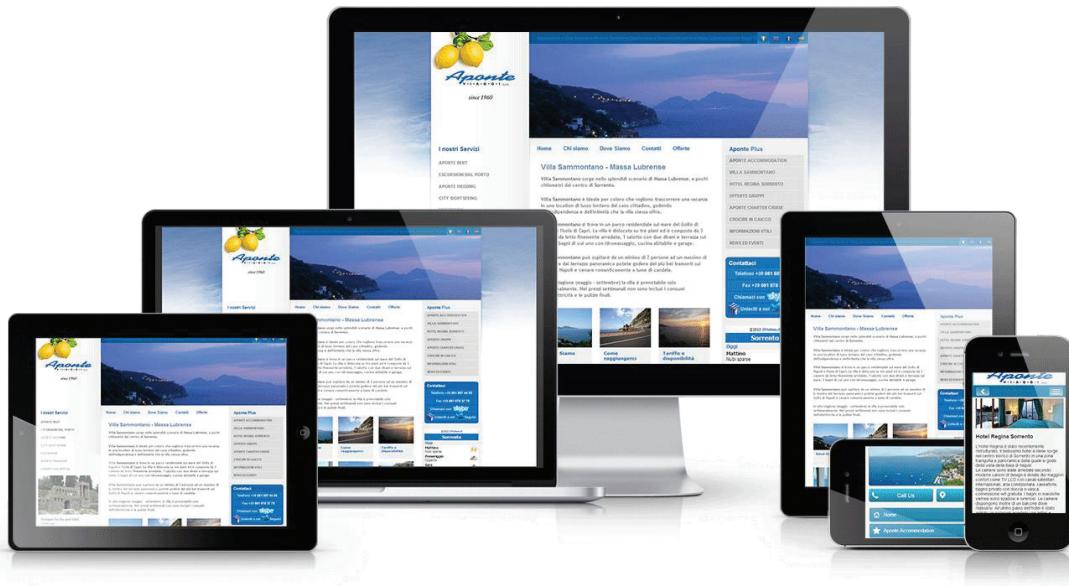
Vantagens da arquitetura MVC

1. Facilidade de manutenção: A distribuição lógica das funções entre os módulos do sistema isola o impacto das modificações.

2. Escalabilidade: Modificações necessárias para acompanhar o aumento da demanda de serviços (database pooling, clustering, etc) ficam concentradas na camada intermediária.

Java Web

html, css e javaScript



Java Web

HTML é uma linguagem de marcação que possibilita apresentar informações na Internet. Aquilo que você vê quando abre uma página na Internet é a interpretação que seu navegador faz do HTML.

CSS é uma linguagem para estilos que define o layout de documentos HTML. Por exemplo, CSS controla fontes, cores, margens, linhas, alturas, larguras, imagens de fundo, posicionamentos e muito mais.

HTML é usado para estruturar conteúdos. CSS é usado para formatar conteúdos estruturados.

Javascript é uma linguagem de programação escrita dentro do código HTML por meio de scripts e interpretada pelo browser (navegador) ao invés de compilada. Apesar de sua sintaxe se assemelhar muito com à do Java, Javascript não tem muito mais que sua origem semelhante à do Java.

Introdução ao HTML

H Hyper
T Text
M Markup
L Language



Linguagem de Marcação de Hipertexto



Java Web

O HTML sigla que significa ***HyperText Markup Language***, é uma linguagem de marcação de hipertexto, criado por Tim Benners-lee por volta de 1990, e passou por várias revisões e versões. Hoje o HTML se encontra na versão 5 onde ele sofreu várias aprimorações facilitando ainda mais a vida dos desenvolvedores front-end.

O HTML é mantido pela **W3C - World Wide Web Consortium**, que é um consórcio formado por empresas e instituições educacionais, fundada em 1994, onde seu objetivo é, estabelecer padrões para várias áreas e desenvolvimento web. Em 1997 o grupo lançou o HTML 4 e poucos meses depois foi publicado o XML 1. Em 1998 foi iniciado a reescrita do HTML em XML, o que originou o XHTML 1 que foi lançado em 1999, e em Maio de 2001 foi publicado o XHTML 1.1.

Como o HTML ficou bastante tempo estagnado sem nenhum tipo de melhoria ou novas versões, em 2004 foi fundada a **WHATWG - Web Hypertext Application Technology Working Group** - que era composta por membros do XForms, um grupo que trabalhava em paralelo com a W3C, mas voltada para formulários em HTML. O WHATWG é o grupo de trabalho tecnológico de aplicações de hipertexto para Web. Trata-se de um grupo livre, não oficial e de colaboração dos desenvolvedores de navegadores e de seus interessados. Mas em 2006 Tim Benners-lee reconheceu o grande feito pelo pessoal da WHATWG e anunciou que eles deveriam trabalhar juntos de forma unificada, foi a partir daí que começou os primeiros passos para o HTML 5.

O que são Tags e Elementos HTML?



Java Web

Tags são rótulos usados para informar ao navegador como deve ser apresentado o website. Toda tag deve ser escrita dentro de `<e>`. Existe dois tipos de tag: as de abertura de um elemento: `< nome da tag >`, e as de fechamento: `</ nome da tag >`.

Sempre que uma tag for aberta, ele deve ser fechada, isso mostra para o navegador onde você quer a aquela formatação ou elemento termine.

```

1  <!doctype html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="utf-8" />
5      <title>Tags e Elementos HTML</title>
6  </head>
7  <body>
8      <nomedatag>
9          Isto é um texto dentro de um elemento.
10     </nomedatag>
11
12 </body>
13 </html>

```

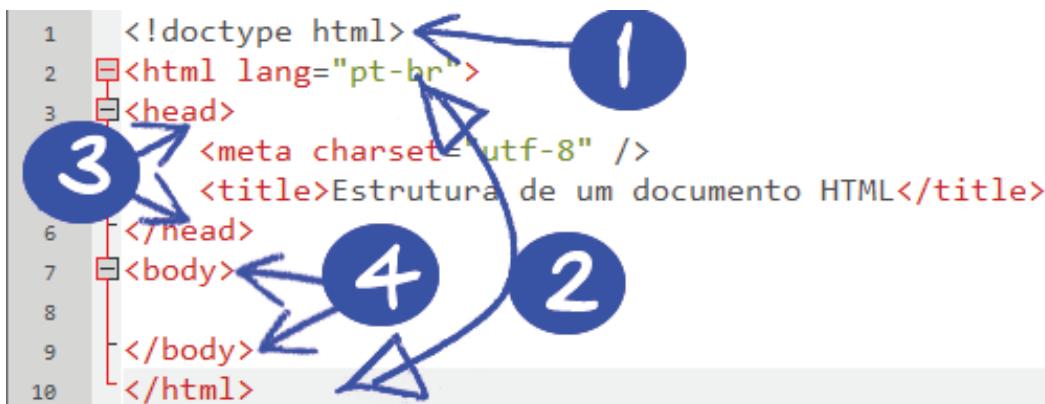
Diagrama explicativo sobre a estrutura HTML:

- TAG ABERTURA:** Aponta para a tag `<nomedatag>`.
- TAG DE FECHAMENTO:** Aponta para a tag `</nomedatag>`.

Elementos

Elementos são componentes de um arquivo HTML. Elementos em HTML são compostos por tags.

Estrutura do HTML



1 – doctype

2 - html

3 - head

4 - body



Java Web

Todo documento HTML deve conter as extensões html ou htm. Ele pode ser escrito por qualquer editor de texto, como o bloco de notas por exemplo. Como já mencionado o HTML, não é uma linguagem de programação, e sim um linguagem de marcação de hipertexto.

Todo documento HTML deve obrigatoriamente obedecer um padrão de estrutura como na imagem abaixo, isso se deve para que seja executado corretamente pelo navegador.

1) <!doctype html> : O doctype não é uma tag HTML, ele é uma instrução necessária para que o navegador saiba que tipo e versão de documento ele deverá executar. O doctype deve ser escrito na primeira linha do seu documento HTML e antes da tag html. Em versões anteriores, era necessário referenciar o DTD diretamente no código do Doctype. Com o HTML5, a referência por qual DTD utilizar é responsabilidade do Browser.

2) html: esta tag indica que ali começa o HTML. O atributo LANG é necessário para que os user-agents saibam qual a linguagem principal do documento. Lembre-se que o atributo LANG não é restrito ao elemento HTML, ele pode ser utilizado em qualquer outro elemento para indicar o idioma do texto representado. Todas as outras tags e elementos devem estar dentro desta tag.

3) head: O head funciona como o cabeçalho do documento HTML. É nele que você insere arquivos de folha de estilo (CSS), javascript e metadados. Metadados são informações

sobre a página e o conteúdo ali publicado. O título da página ou do site, é inserido dentro desse elemento usando o elemento **<title>**, que é escrito com a tag **<title> ... </title>**.

4) body: Esta tag é responsável pelo corpo da página HTML, ou seja, tudo que estiver dentro dela, será exibido na tela do computador.

Esta é a estrutura base de um documento HTML. Essa estrutura deve ser sempre respeitada, isso evita de sua página não ser executada, ou apresentar erros no navegador.

Atributos

```
<p style="color:red;">  
Este é um parágrafo para a aula sobre atributos  
</p>
```

Este é um parágrafo para a aula sobre atributos



Java Web

Elementos em HTML têm **atributos**, que são valores adicionais que configuram os elementos ou ajustam seu comportamento de várias maneiras para cumprir os critérios que os usuários querem.

No HTML existem muitos atributos, mas iremos citar os mais usados. Todos os atributos devem seguir a seguinte sintaxe: **nome="valor"**, onde nome é o nome do atributo seguido do sinal de igual (=), e valor é o valor da propriedade dentro de aspas, duplas ou simples.

alt - Alternative text

O atributo alt, é usado em elementos de imagem. Ele exibe um texto no caso de ocorrer algum erro e a imagem não for exibida.

title - Título

Título é bastante usado em descrições de elemento. Sua função é exibir um texto em forma de caixa quando se passa o mouse em cima do elemento.

src - url de arquivos

Este atributo tem como função informar o endereço url de arquivo externos que serão inseridos no documento HTML, como imagens, arquivos de áudio e vídeo, scripts etc.

rel - Relações entre links

O atributo rel especifica a relação entre o objeto de destino para o objeto link.

language - Tipos de script

Este atributo é usado basicamente quando se irá inserir um bloco ou arquivo de script no HTML. Ele especifica o tipo de script que está sendo inserido.

lang - Idioma do conteúdo

Como já vimos, o atributo lang é usado para informar o idioma da página html. Mas este atributo não é exclusivo da tag <html>. Ele pode ser usado em qualquer tag ou bloco do seu documento HTML exercendo a mesma função, de informar o idioma do conteúdo do bloco, parágrafo etc.

id - Identificação de elementos únicos

Um dos atributos que mais se usa em documentos HTML. Este atributo identifica um elemento com um nome que você pode criar. É como você pegar um elemento do código e dar um nome a ele. Muito útil, pois com essa identificação você pode formatá-lo pelo CSS, criando regras para aquele específico elemento.

class - formatações mais amplas

O atributo class, é bastante usado para formatação em css para elementos que possuem propriedades em comum.

href - linkando

O href serve exclusivamente para linkar. Ele faz a linkagem entre páginas, arquivos, mídia, arquivo para download etc.

height - altura

O atributo height é usado quando precisamos especificar altura de elementos. Pode ser de parágrafos, blocos, imagens etc.

width - largura

Tem a mesma funcionalidade do height, mas ele é responsável pela largura.

Esses são os atributos mais usados, que praticamente você irá se cansar de escrever, mas o HTML possui muito mais atributos que está no guia de referência. Nossa objetivo aqui é deixar bem claro o que é um atributo e como usá-lo.

Formulários

```
<input />
```

```
<select>
    <option>...</option>
</select>
```

```
<textarea cols="20" rows="5">
</textarea>
```

- **text**
- **password**
- **hidden**
- **checkbox**
- **radio**
- **submit**
- **button**
- **image**
- **file**
- **reset**



Java Web

Formulários são um dos elementos mais usados na web. De pequenos formulários de contato a extensos cadastros em lojas virtuais.

Todo formulário é composto pelo elemento form `<form> </form>`. Todos os campos que estão dentro dele serão enviado para o servidor. Para que o formulário seja processado, os dados devem ser enviados a um script externo, escrito por alguma linguagem de programação como php, asp, ruby ou no nosso caso java. Para informar o script que irá manipular esses dados, usamos o atributo **action** no formulário. Exemplo:

```
<form action="url do script ou do servlet">
  ...
</form>
```

A tag input

Existem vários tipos de inputs, e cada um se comporta de maneira diferente, o que diferencia cada um desses inputs é o atributo type, que pode ser qualquer um dos abaixo:

- **text** - Cria uma caixa de texto de uma linha.
- **password** - Cria uma caixa de texto de uma linha escondendo os caracteres digitados.
- **checkbox** - Cria uma caixa que assume dois estados: checado e “deschecado”. Em conjunto com o atributo name é possível que se crie um grupo de checkboxes no qual uma ou mais checkboxes sejam “checados”.

• **radio** - Cria uma caixa que assume dois estados: checado e “deschecado”. Em conjunto com o atributo name é possível que se crie um grupo de rádios no qual apenas um rádio seja “checado”.

• **button** - Cria um botão e através do atributo value definimos o texto do botão.

• **submit** - Cria um botão para o envio do formulário e através do atributo value definimos o texto do botão.

• **file** - Cria um botão que, ao ser clicado, abre uma caixa de diálogo para a escolha de um arquivo no computador do usuário.

• **reset** - Cria um botão que descarta todas as alterações feitas dentro de um formulário. Através do atributo value definimos o texto do botão.

• **image** - Cria um botão para o envio do formulário e deve ser utilizado em conjunto com o atributo src para que uma imagem de fundo seja utilizada no botão.

• **hidden** - Cria um elemento que não fica visível para o usuário, porém pode conter um valor que será enviado pelo formulário.

Introdução ao CSS



Cascading Style Sheets Folhas de Estilo em Cascata



Java Web

CSS (Cascading Style Sheets) ou folhas de estilo em cascata é uma linguagem de estilo usada para apresentação e formatação de página web. Todo documento CSS deve ser na extensão “.css”. O principal objetivo do uso do CSS é separar a formatação da página do seu conteúdo.

Inserindo CSS ao HTML

Para estilizarmos nossa página HTML devemos vincular o CSS ao documento HTML, e isso podemos fazer de três maneiras.

Usando um arquivo externo

Você pode criar um arquivo “.css” e linká-lo a sua página HTML, para isso devemos inseri-lo ao elemento head.

Para isso usamos a tag link como alguns atributos. Veja o exemplo:

```
<link rel="stylesheet" type="text/css" href="url do arquivo css" media="all" />
```

Explicando os atributos.

- **rel:** significa o tipo relacionamento desse link que no caso é uma folha de estilo (stylesheet).
- **type:** informa o tipo de arquivo que estamos linkando ao nosso HTML.

- **href:** Usado também em links de página, ele mostra o endereço do arquivo.
- **media:** Esse atributo informa ao navegador para qual dispositivo deve ser usado esse arquivo CSS. Neste caso, que o valor é all, ele será usado em todos os dispositivos, telas grandes, celulares, impressão.

Blocos de CSS

Você pode inserir também em qualquer parte do documento HTML um bloco com propriedades de CSS em cima do bloco que . Essa maneira não é muito boa pois deixa seu código um pouco bagunçado.



Dica

Quando você precisar inserir bloco de código insira dentro do elemento head, assim você deixa seu código mais limpo e mais organizado.

Vamos inserir um bloco de CSS agora. Para inserir um bloco de código CSS devemos abrir o elemento de CSS com a tag **<style>**, e informar o atributo type com os valores de text/css. Exemplo:

```
<style type="text/css">
    ...
</style>
```

Dentro do elemento inserimos os seletores e propriedades CSS. Veja outro exemplo:

```
<style type="text/css">
p {
    color: red;
}
</style>
```

Como podemos observar a sintaxe continua a mesma.

Estilo dentro do Elemento

Outra forma de inserir CSS dentro de um elemento é inserir o estilo dentro da tag de abertura do elemento. Exemplo:

```
<div style="color:red;">
    ...
</div>
```

A sintaxe muda um pouco, mas a base é a mesma e as propriedades são inseridas dentro do atributo style.

Seletores

Seletores



Java Web

O seletor é o elemento html que você deseja alterar o formato, onde ele pode ser especificado pelo elemento e diferenciado por uma classe, id, tipo e de hierarquia. Vou mostrar agora alguns tipos de seletores que vocês irão usar frequentemente em seus projetos.

Propriedades são as configurações que irão estilizar o elemento HTML. Toda propriedade deve ter um valor e ser finalizado com um ponto e vírgula.

Seletor Universal

O seletor universal seleciona todos os elementos de um documento HTML. Exemplo:

```
* {
    margin: 0px;
    padding: 0px;
}
```

O exemplo acima é uma técnica muito usada como reset de propriedades padrão dos navegadores. Veremos isso melhor mais a frente.

Seletor de tipo

Os seletores de tipo alteram as propriedades de todos os elementos que são formados pela mesma tag. Exemplo:

```
p{
    color: red;
}
```

No exemplo acima, todos os parágrafos, independente de classe ou id irão ser de cor vermelha.

Seletores de classe

Neste caso todos os elementos que possuírem a mesma classe irão ter a mesma formatação, ou seja, todos os elementos que possuírem o atributo class="" com o mesmo valor. Exemplo:

```
div.minhaclasse {  
    color: blue;  
}
```

Seletores de id

Esses seletores fazem a mesma função dos seletores de classe, só que seu atributo é id="". Exemplo:

```
div#conteudo {  
    color: black;  
}
```



Dica importante sobre id e classe

- Como foi exemplificado acima, id e classe são atributos diferentes. As classes em CSS, são representadas por .nomedaclasse (.) e a id por #nomedaid (#).
- A maneira mais certa de usar id e classe é usando id para elementos específicos, como elemento pai, ou seja, elementos em que suas propriedades serão exclusivas a ele. E usar classe para elementos de propriedades mais amplas, ou que você precisa usá-las várias vezes em seu site.

Seletor de descendentes

Chamamos de seletor de descendentes a seleção de um ou mais elementos fazendo o uso de outros seletores separados por espaços. Um espaço indica que os elementos selecionados pelo seletor à sua direita são filhos diretos ou indiretos dos elementos selecionados pelo seletor à sua esquerda. Exemplo:

```
li a {  
    color: red;  
}
```

Seletor de filhos

Chamamos de seletor de filhos a seleção de um ou mais elementos fazendo o uso de outros seletores separados pelo caractere >. Um caractere > indica que os elementos selecionados pelo seletor à sua direita são filhos diretos dos elementos selecionados pelo seletor à sua esquerda. Exemplo:

```
div.conteudo > a.link {  
    color: red;  
}
```

Propriedades importantes do CSS

Como mostramos anteriormente, para alterar os estilos de um elemento HTML, usamos propriedades para tal tarefa. Vamos mostrar a vocês algumas das propriedades mais usadas para a criação de sites e páginas HTML estilizadas.

Cores e Medidas

Muito importante antes de você aprender as várias propriedades do CSS, é preciso saber como o CSS trata as cores e várias medidas, pois muitos valores dessas propriedades são em medidas e cores.

Cores

As cores em CSS são tratadas praticamente em duas formas. Essas são as mais usadas e são as que você irá usar sempre, que são cores em RGB ou Hexadecimal.

Você não precisa decorar e nem ser um expert em código de cores, pois os próprios programas que são usados para o desenho do layout como Photoshop, Gimp entre outros, te fornecem esses códigos, sejam em Hexadecimal ou RGB.

Exemplo em Hexadecimal:

```
p {  
    color: #000000;  
}
```

Exemplo em RGB;

```
p {  
    color: rgb(0,0,0);  
}
```

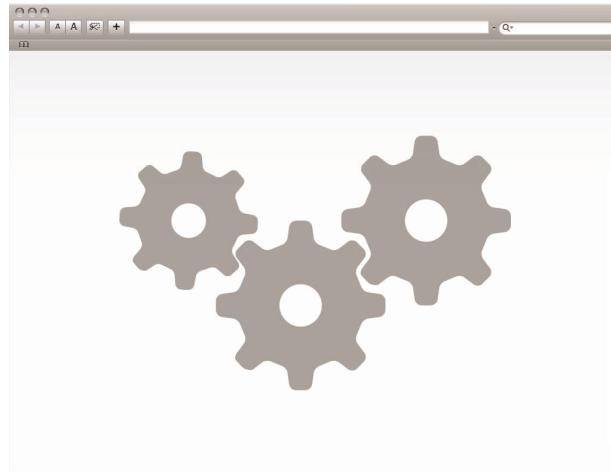
Medidas

As medidas em CSS são relativas ao tamanho da resolução da tela. Elas podem ter os valores em:

- **in** - polegada.
- **cm** - centímetro.
- **mm** - milímetro.
- **em** - tamanho relativo ao tamanho de fonte atual no documento. 1em é igual ao tamanho da fonte atual, 2em o dobro do tamanho da fonte atual e assim por diante.
- **ex** - essa unidade é igual à altura da letra “x” minúscula da fonte atual do documento.
- **pt** - ponto (1pt é o mesmo que 1/72 polegadas).
- **px** - pixels (um ponto na tela do computador).

Confira a uma descrição detalhada das propriedades CSS em nosso guia de referência.

Introdução ao javaScript



Client-side



Java Web

Para as criações de páginas web dinâmicas, usaremos JavaScript. O JavaScript é uma linguagem client-side, ou seja, ela é interpretada pelo navegador. Iremos aprender uma base de JavaScript e como usar ela em nosso site, deixando - o mais dinâmico e atraente.

Inserindo JavaScript a nossa página

Existe duas maneiras de você inserir JavaScript na sua página web, uma delas é inserindo um bloco de código usando a tag script, assim como mostramos no CSS anteriormente. Esse bloco pode ser inserido em qualquer parte do documento HTML, porém seu código ficará mais organizado se você o inserir dentro do elemento head. Exemplo:

```
<!doctype html>
<html lang="pt-br">
<head>
<meta charset="utf-8" />
<title>Aula de JavaScript</title>
<script>
...
</script>
</head>
...

```

Outra maneira, e a que eu recomendo, pois deixa a estrutura do seu site muito mais organizada e fácil de manusear é inserir um documento JavaScript externo, usando o atributo src na tag <script>. Exemplo:

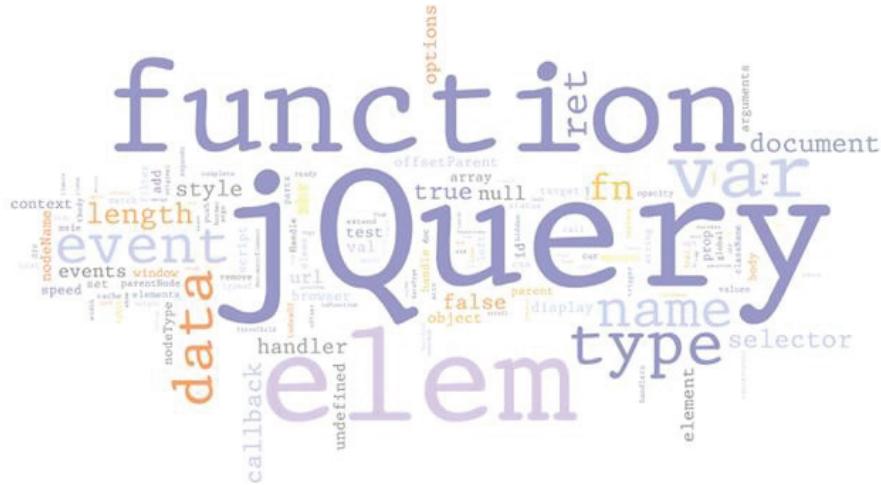
```
<!doctype html>
<html lang="pt-br">
<head>
<meta charset="utf-8" />
<title>Aula de JavaScript</title>
<script src="url do arquivo javascript"></script>
</head>
...

```

Simples, não? O seu documento de JavaScript deve conter a extensão .js. Exemplo:

meus-scripts.js
scripts.js.
etc.

Sintaxe do JavaScript



Java Web

Variáveis, são espaços localizados na memória que usamos para armazenar um valor ou uma expressão. Para declararmos uma variável em Java script, usamos a seguinte sintaxe.

```
var nomedavariavel = "valor da variável";
```

Exemplo:

```
var meunome = "Caio Vinicius";
```

As variáveis podem ser números, números com casas decimais, textos, e booleanos (verdadeiro ou falso).

Operadores

Como em toda linguagem de programação, os operadores servem para manipular os dados. Podemos fazer desde operações matemáticas a comparações de dados. O JavaScript nos oferece vários tipos de operadores, segue alguns tipos:

- Matemáticos ou aritméticos: (+, -, *, /)
 - Atribuição (=, +=, -=, *=, /=, %=)
 - Relacional (==, !=, <, <=, >, >=)
 - Lógico (&&, ||)

Operadores Matemáticos

Servem para realizar operações matemáticas. Exemplo:

- var dois = 1 + 1;
- var tres = 4 - 1;
- var dez = 5 * 2;
- var cinco = 10 / 2;

Operadores de Atribuição

Serve para atribuir algum valor. Existem seis tipos de operadores de atribuição:

- Simples =
- Incremental +=
- Decremental -=
- Multiplicativa *=
- Divisória /=
- Modular %=

Exemplo:

- var valor = 1; // valor = 1
- 2 valor += 2; // valor = 3
- 3 valor -= 1; // valor = 2
- 4 valor *= 6; // valor = 12
- 5 valor /= 3; // valor = 4
- 6 valor %= 3; // valor = 1

Basicamente os operadores de atribuição nos economizam em escrita de códigos, já que essas mesmas funções poderíamos escrever como operações matemáticas.

Operadores Relacionais

Como o próprio nome diz esses operadores fazem relações entre variáveis ou valores. Podemos fazer as seguintes relações de dados:

- Igualdade ==
- Diferença !=
- Menor <
- Menor ou igual <=
- Maior >
- Maior ou igual >=

Exemplo:

```
var valor = 2;  
var t = false ;
```

```
t = ( valor == 2); // t = true  
t = ( valor != 2); // t = false  
t = ( valor < 2); // t = false  
t = ( valor <= 2); // t = true  
t = ( valor > 1); // t = true  
t = ( valor >= 1); // t = true
```

Operadores Lógicos

Para verificarmos duas ou mais condições ao mesmo tempo, usamos operadores lógicos. O JavaScript nos oferece dois tipos de operadores lógicos, que são:

- “E” - &&
- “OU” - ||

```
var valor = 30;  
var teste = false ;  
teste = valor < 40 && valor > 20; // teste = true  
teste = valor < 40 && valor > 30; // teste = false  
teste = valor > 30 || valor > 20; // teste = true  
teste = valor > 30 || valor < 20; // teste = false  
teste = valor < 50 && valor == 30; // teste = true
```

Estude bastante operadores pois eles são a base para muitos scripts que você irão escrever em sua vida de desenvolvedor.

if e else

if e esle (se , senão), são comando de digamos verificação, onde se uma função for verdadeiro ele é executado ou senão não for outra função é executada. Imaginamos que um amigo seu lhe pede um dinheiro emprestado, e você passa pra ele a seguinte condição:

Se eu tiver o dinheiro no banco eu lhe empresto, mas se não tiver não tem como te emprestar.

É basicamente isso que os comandos if e else fazem se a resposta de if for verdadeira ele executa a primeira ação, senão for verdadeira ele executa a segunda ação. Exemplo:

```
var dinheiro = 100;  
var saldobanco = 110;
```

```
if(saldobanco <= dinheiro) {  
document . writeln ('Posso te emprestar o dinheiro ' );  
} else {  
document . writeln ('Não posso te emprestar o dinheiro.' );  
}
```

for e while

Os comandos for e while são bem úteis quando precisamos repetir alguns comandos. Imagine que precisemos exibir na tela a seguinte mensagem:

Olá, tenha um bom dia!

Imagine que tenhamos que exibir essa mensagem vinte vezes na tela. Para nos poupar do trabalho de digitar ou copiar e colar as vinte vezes, podemos usar o comando while. Exemplo:

```
// aqui mostramos o numero de vezes que a mensagem foi exibida
var numimpressoes = 0;

// aqui mostramos o numero de vezes que a mensagem será exibida.
while(numimpressoes < 10) {

document.writeln('Olá, tenha um bom dia!');

// aqui o operador ++ incrementa o código a quantidade de vezes que
foi informada.
numimpressoes++;
}
```

O comando for, faz o mesmo trabalho do while, a diferença é que ele deve receber três argumentos.

```
for ( var contador = 0; contador < 100; contador ++ ) {
document . writeln ('Rafael Cosentino ');
document . writeln ('<br/>');
}
```

Essa é base para trabalhar com JavaScript, e se você tiver alguma noção de alguma noção de alguma outra linguagem de programação, você vai ver que tudo isso que falamos é base muitas linguagens de programação.

Objetos e funções

Uma função JavaScript é uma sequência de instruções JavaScript que serão executadas quando você chamá-la através do seu nome.

Funções em JavaScript devem obedecer a seguinte sintaxe:

```
function nomedafuncao(parametro) {
    ...
}
```

Exemplo:

```
function mostraola() {  
    document.writeln('Olá Mundo');  
}
```

Para que a função seja executada basta chama-la em algum ponto da página HTML como em um link, por exemplo:

```
<a href="javascript:;" onclick="mostraola();">Mostrar Olá</a>
```

Java Web Servlets

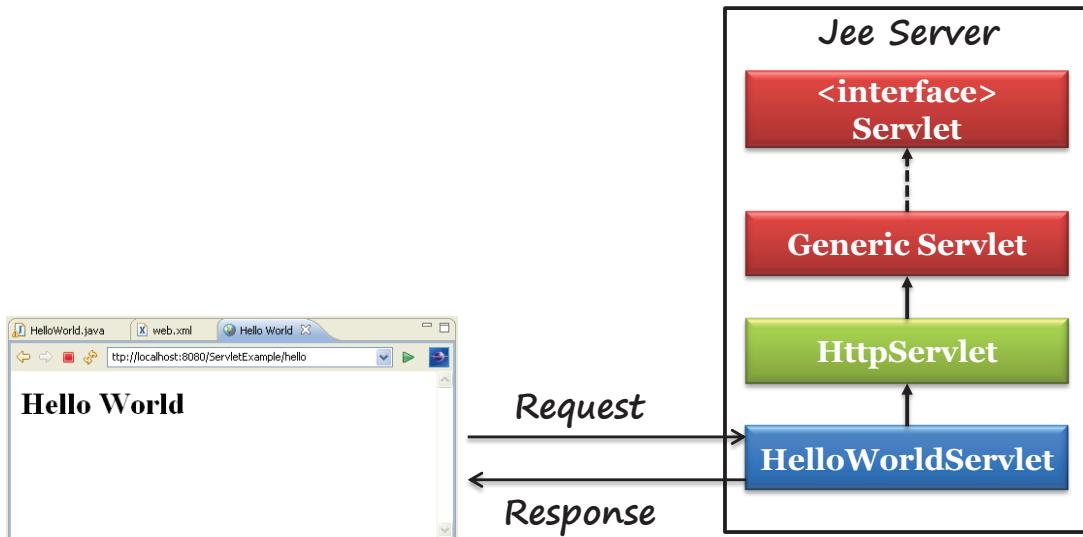


Java Web

Algum tempo atrás a Internet era composta principalmente de páginas estáticas com conteúdo institucional, hoje ela oferece uma infinidade de aplicações com conteúdo dinâmico e personalizado.

Os **Servlets** foram a primeira maneira de criar páginas dinâmicas com Java. Seu nome vem da ideia de um pequeno servidor (servidorzinho, em inglês). Sua principal função é receber chamadas HTTP, processá-las e devolver uma resposta ao cliente.

O que são servlets?



Java Web

Servlets são classes Java que são instanciadas e executadas em associação com servidores Web, atendendo requisições realizadas por meio do protocolo **HTTP**. Ao serem acionados, os objetos Servlets podem enviar a resposta na forma de uma página **HTML** ou qualquer outro conteúdo **MIME**. Na verdade os Servlets podem trabalhar com vários tipos de servidores e não só servidores Web. Uma vez que a API dos Servlets não assume nada a respeito do ambiente do servidor, sendo independentes de protocolos e plataformas.

Os Servlets são tipicamente usados no desenvolvimento de sites dinâmicos. Sites dinâmicos são sites onde algumas de suas páginas são construídas no momento do atendimento de uma requisição HTTP. Assim é possível criar páginas com conteúdo variável, de acordo com o perfil do usuário ou informações armazenadas em um banco de dados.

O protocolo HTTP é utilizado na navegação nas páginas da Internet: quando você abre uma janela de um browser, acessa uma página Web e navega em seus links, você está, na verdade, utilizando esse protocolo para visualizar, em sua máquina, o conteúdo que está armazenado em servidores remotos.

O HTTP é um protocolo **stateless (sem estado)**, que permite **comunicação cliente-servidor**.

Quando você digita o endereço de uma página em um Browser Web, estamos gerando uma requisição a um servidor, que irá, por sua vez, devolver para o browser o conteúdo da página HTML requisitada.

Existem diversos métodos HTTP que podem ser especificados em requisições, sendo os mais comuns o método **GET**, normalmente utilizado para obter o conteúdo de um arquivo

no servidor, e o método POST, utilizado para enviar dados de formulários HTML ao servidor. Além desses métodos, o protocolo HTTP versão admite os seguintes métodos:

- **HEAD:** permite que o cliente obtenha somente os headers da resposta
- **PUT:** transfere um arquivo do cliente para o servidor
- **DELETE:** remove um arquivo do servidor
- **OPTIONS:** obtém a lista dos métodos suportados pelo servidor
- **TRACE:** retorna o conteúdo da requisição enviada de volta para o cliente

HTTP Get

O método GET tem por objetivo **enviar uma requisição** por um recurso. As informações necessárias para a obtenção do recurso (como informações digitadas em formulários HTML) são **adicionadas à URL** e, por consequência, **não são permitidos caracteres inválidos** na formação de URLs, como espaços em branco e caracteres especiais.

A parte após a (?) é denominada de **query string**, ela consiste de vários pares nomes de **parâmetros e valor separados pelo (&)**, como em:

nome1=valor1&nome2=valor2&nome3=valor3&...&nomeN=valorN

HTTP Post

As **requisições POST** a princípio podem ter tamanho ilimitado. No entanto, elas não são idempotente, o que as tornam ideais para formulários onde os usuários precisam digitar informações confidenciais, como número de cartão de crédito. Desta forma o usuário é obrigado a digitar a informação toda vez que for enviar a requisição, não sendo possível registrar a requisição em um bookmark, por exemplo.

De forma geral use POST para:

- Enviar grandes quantidades de dados; por exemplo, grandes formulários.
- Fazer upload de arquivos
- Capturar nome de usuário e senha, assim você evita que estes dados fiquem visíveis na URL.

A classe HttpServlet

O comportamento dos servlets que iremos ver neste capítulo está definido na classe **HttpServlet** do pacote **javax.servlet**. Eles se aplicam às servlets que trabalham através do **protocolo Http**.

Para cada método HTTP há um método correspondente na classe HttpServlet, de modo geral eles tem a seguinte assinatura:

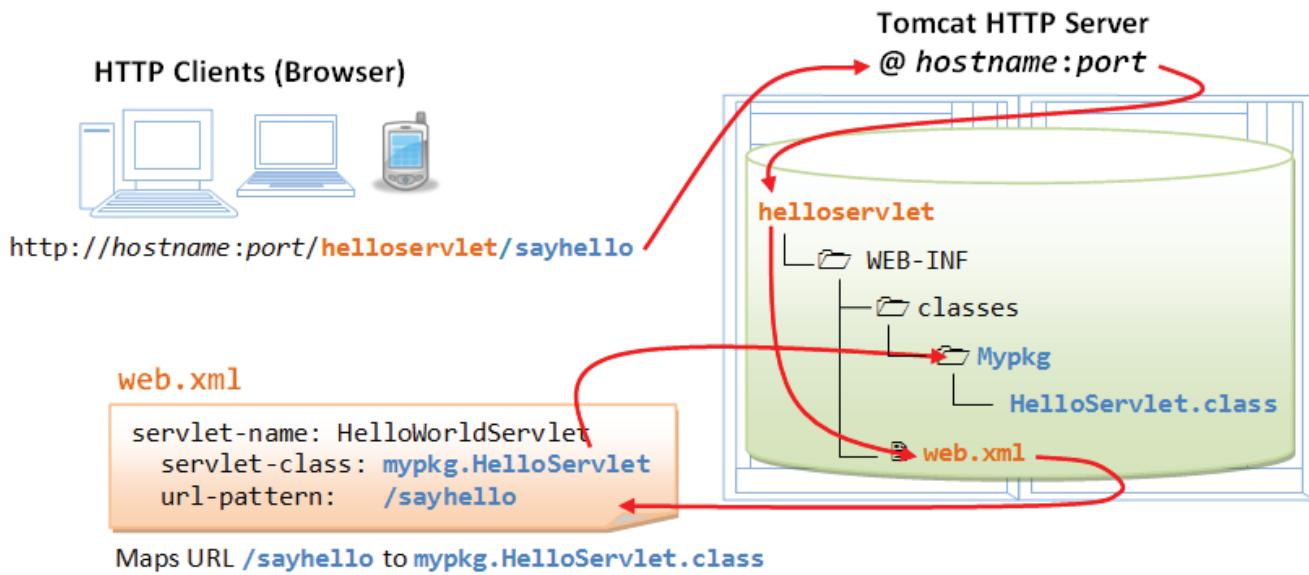
```
protected void doXXX(HttpServletRequest, HttpServletResponse)
throws ServletException, IOException;
```

onde **doXXX()** depende do **método HTTP**, como mostrado abaixo.

método HTTP	método HttpServlet
GET	doGet()
HEAD	doHead()
POST	doPost()
PUT	doPut()
DELETE	doDelete()
OPTIONS	doOptions()
TRACE	doTrace()

A classe **HttpServlet** fornece um implementação vazia para cada método **doXXX()**. Você deve sobrescrever o método **doXXX()** que for tratar em seu servlet para implementar a lógica de negócio.

Estruturas de Aplicação Web



Java Web

Uma **aplicação Web** é um conjunto de **Servlets, JSPs, Classes Java, bibliotecas, imagens, páginas HTML e outros elementos**, que podem ser empacotados juntos e que provê as funcionalidades da aplicação.

Essa definição está contida, na verdade, na própria especificação de Servlets de Java, não sendo específica, portanto, à utilização do um único contêiner Web. Isso significa que as aplicações desenvolvidas por você podem ser instaladas em qualquer servidor que implemente a especificação de Servlets (como o **Apache Tomcat, JBoss, Glassfish** entre outros).

Uma aplicação web precisa ter um local (pasta/diretório) onde são armazenados os recursos disponibilizados pela aplicação para acesso por parte do usuário. Tenha como recursos páginas web (HTML), imagens, arquivos CSS (Cascade Style Sheet – Folhas de Estilo em Cascata), Javascripts e outros recursos a mais. Note que estes recursos o usuário pode acessar diretamente através do browser. Existem outros recursos, como dados armazenados em uma base de dados, que o usuário não pode (e nem deve) acessar diretamente.

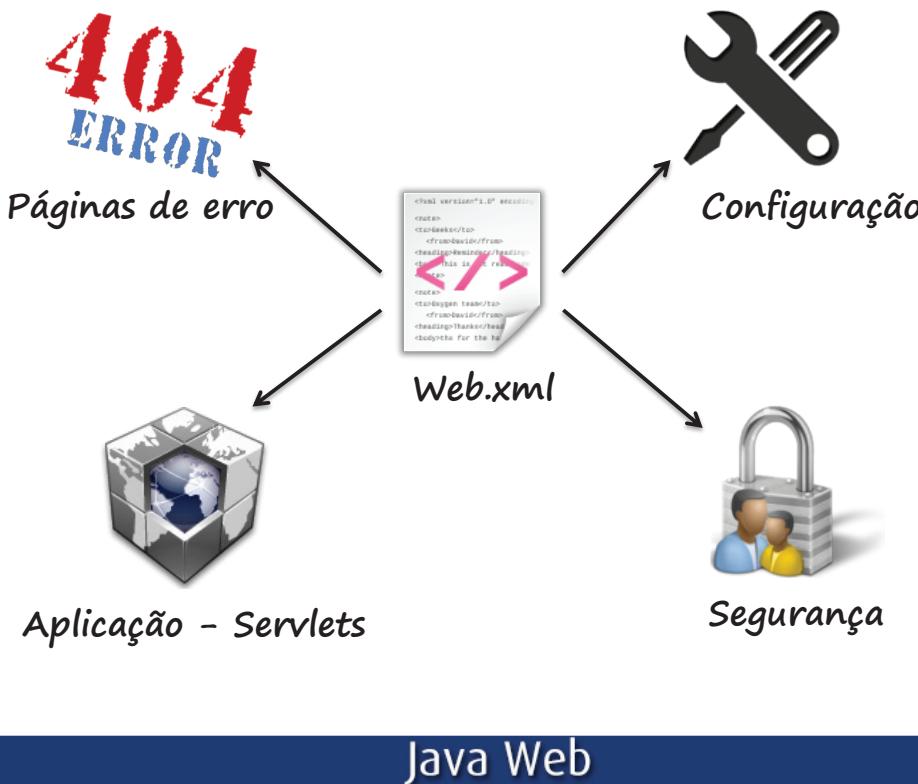
Normalmente, uma aplicação faz uso de “**outras aplicações**”, também chamadas como bibliotecas que desempenham uma função específica, ou ainda, mais tecnicamente falando, frameworks. Sua aplicação pode precisar acessar dados de forma mais ágil e produtiva, imprimir relatórios ou gerar gráficos, dentre muitas outras finalidades, desta forma, não é produtivo que sua aplicação crie tudo isso do zero, sendo que existem soluções disponíveis, e, na maioria das vezes, gratuitas. Assim sendo, é preciso que a aplicação armazene estes recursos em algum “local” (pasta/diretório) de sua aplicação.

Uma vez apresentado alguns dos diversos recursos possíveis para uma aplicação, olhe a figura abaixo, apresenta a estrutura básica necessária para uma aplicação web.



Na **Figura**, a pasta **WebContent** representa o **root** de sua aplicação web. É nesta pasta que todos os recursos acessíveis de forma direta pelo seu usuário devem estar. Nela, é possível criar uma estrutura de pastas, distribuindo os arquivos de acordo a seu tipo, ou responsabilidade. A pasta **WEB-INF** é uma pasta que o usuário não acessa de forma direta pelo browser, o acesso é feito pelo container. A pasta **lib** possui recursos (bibliotecas) utilizados por sua aplicação. É comum ouvir desenvolvedores que dizem que se mais de uma aplicação utiliza uma mesma biblioteca, esta deve ser colocada na pasta de bibliotecas comuns do container. Não faça isso, pois podem acontecer sérios problemas relacionados ao **class loader**. A pasta **classes** é a pasta que terá todas as classes utilizadas por seu projeto, já compiladas e na estrutura de pacotes das mesmas.

Deployment Descriptor: web.xml



Java Web

O **deployment descriptor** é utilizado para informar ao contêiner como executar seus servlets, ele permite mapear uma URL pública, conhecida pelo seu cliente, para próprio nome interno do seu servlet. O ponto mais importante, por hora, é saber que você pode modificar o comportamento de sua aplicação usando o **web.xml**, sem qualquer alteração em seu código fonte.

O que define o **Deployment Descriptor (DD)**, o elemento **<web-app>** é o elemento **root (raiz)** desse XML, ou seja, deve haver somente um elemento **<web-app>**, e abaixo dele devem ficar todos os outros elementos do XML.

Os principais elementos abaixo do elemento root são os seguintes:

- **<display-name>** O elemento deve conter um nome da aplicação a ser apresentado por ferramentas GUI de gerenciamento/ desenvolvimento de Aplicações Web. Esse elemento é opcional, porém caso você decida utilizá-lo, é importante que haja somente um desses elementos por DD.

- **<context-param>** O elemento serve para que se possam definir **parâmetros de inicialização** do contexto da aplicação; esses parâmetros estarão disponíveis para todos os Servlets e páginas JSP da aplicação. Cada elemento presente deve conter o nome de um parâmetro e o seu valor correspondente. O desenvolvedor pode também optar por não utilizar nenhum desses elementos em seu XML.

• **<welcome-file-list>** Os elementos **<welcome-file-list>** e **<error-page>** contém, respectivamente, a lista ordenada de páginas a serem utilizadas como **index** e as páginas a serem apresentadas em casos de **erros HTTP** ou exceções não tratadas pela aplicação. Esses dois elementos são opcionais, sendo que somente o primeiro admite uma instância por DD.

• **<servlet>** O elemento servlet serve para definir, os Servlets da aplicação, com seus respectivos parâmetros. Cada elemento servlet, por sua vez, é composto dos seguintes elementos:

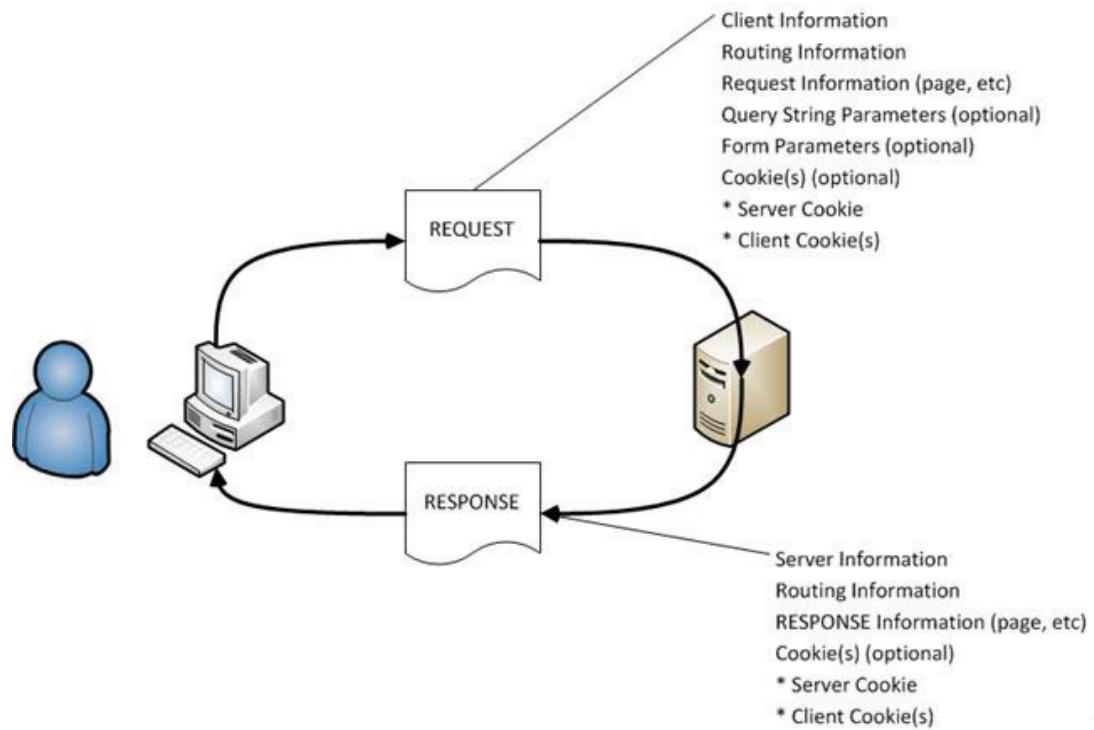
- **<servlet-name>** deve conter o nome do Servlet.
- **<servlet-class>** deve conter o nome da classe completamente qualificado (inclui a informação sobre o package, se existir).

• **<init-param>** deve conter um parâmetro de inicialização do Servlet; pode haver nenhum, somente um, ou mais de um elemento deste tipo para cada Servlet.

• **<load-on-startup>**: deve conter um inteiro positivo indicando a ordem de carga deste Servlet em relação aos outros Servlets da aplicação, sendo que inteiros menores são carregados primeiro; se este elemento não existir, ou seu valor não for um inteiro positivo, fica a cargo do Servlet Container decidir quando o Servlet será carregado (possivelmente, no instante em que chegar a primeira requisição a esse Servlet).

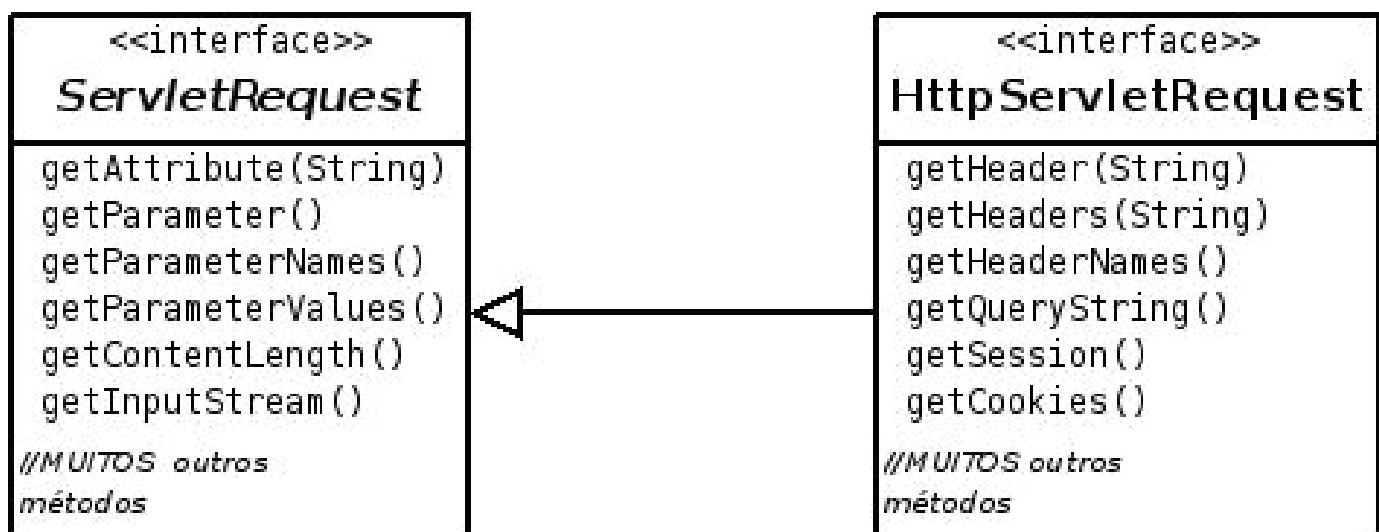
• **<servlet-mapping>** O elemento contém um nome de Servlet, conforme definido em **servlet-name**, e um **<url-pattern>**, padrão de URL do Servlet no servidor.

Request e Response



Java Web

A Interface HttpServletRequest



Hierarquia Interface HttpServletRequest

O objeto **HttpServletRequest** passado para o **Servlet** contém várias informações importantes relacionadas com a requisição, como por exemplo o protocolo utilizado, o endereço remoto, informações contidas no cabeçalho e muitas outras.

Veja no exemplo mostra uma página **HTML** que permite ao usuário enviar dois parâmetros ao servidor.

```
<form action="CaminhoDoServlet" method="POST">
    Tecnologia:
    <input type="text" name="lstconsulta" value="java"> <br> <br>
    Estado:
    <select name="estado" size="5" multiple>
        <option value="GO">GOIÂNIA</option>
        <option value="DF">BRASÍLIA</option>
    </select> <br>
    <br>
    <input type="submit" value="Busca Emprego">
</form>
```

O **<form>** possui um campo texto, uma caixa de listagem, e um botão de submissão. O atributo **action** especifica o servlet mapeado para o nome **CaminhoDoServlet** que irá manipular a requisição. Observe que o atributo **method** do **<form>** é **POST**, então os parâmetros serão enviados ao servidor usando uma requisição **HTTP POST**.

Uma vez que a requisição tenha sido enviada ao servidor, o servlet que está mapeado no Deployment Descriptor é invocado. Abaixo é mostrado o método **doPost()** do servlet.

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    // retorna valor do parametro enviado na requisicao
    String consultaString = req.getParameter("lstconsulta");
    System.out.println(consultaString);
    // retorna valores do parametro enviado na requisicao
    String[] listaEstado = req.getParameterValues("estado");

    for (String estado : listaEstado)
        System.out.println(estado);
}
```

No código acima, nós sabemos o nome dos parâmetros (**lstconsulta** e **estado**) enviados com a requisição, então usamos o método **getParameter()** e **getParameterValues()** para recuperar os valores dos parâmetros. Quando não se sabe os nomes dos parâmetros você pode usar o método **getParameterNames()** e recuperar o nome de todos os parâmetros submetidos na requisição.

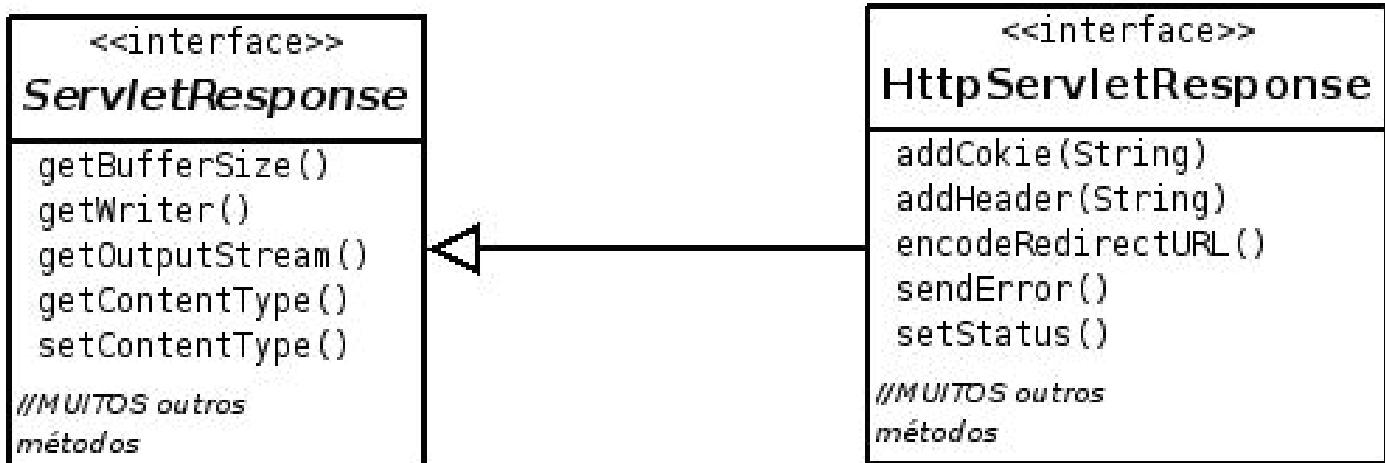
A Interface **HttpServletResponse**

Ele recebe os dados que o servlet necessita enviar para o cliente e os formata dentro de uma mensagem como especificado pelo protocolo HTTP.

A interface **ServletResponse** fornece métodos relevantes para qualquer protocolo, enquanto **HttpServletResponse** estende **ServletResponse** e adiciona métodos específicos.

cos para o protocolo **HTTP**.

ServletResponse declara vários métodos genéricos, incluindo **getWriter()**, **getOutputStream()**, **setContentType()** entre outros.



Hierarquia interface HttpServletResponse

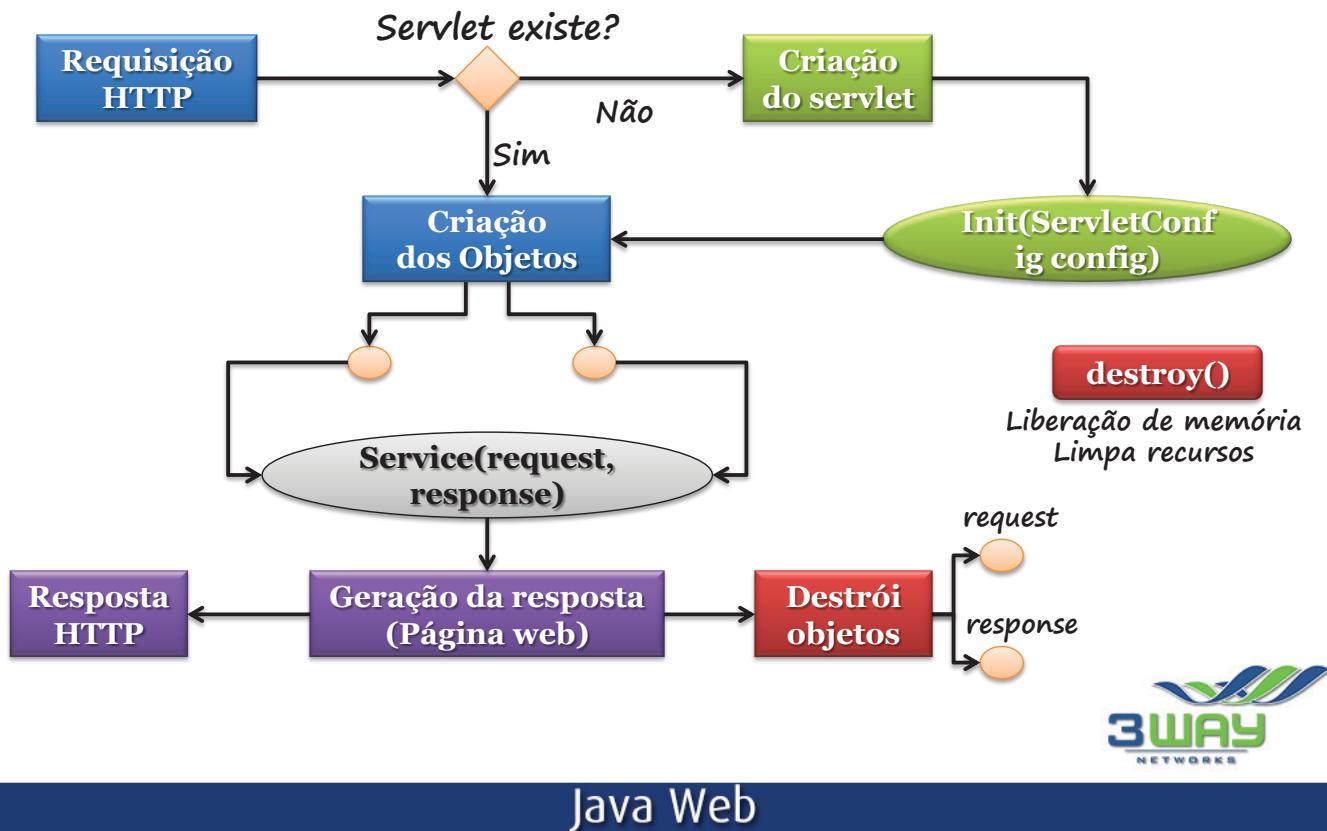
Olhe para o método **getWriter()** a chamada desse método irá retornar uma referência a um objeto da classe **java.io.PrintWriter**, que pode ser usada para enviar dados em formato de caracteres para um cliente. Os métodos **print()** e **println()** dessa classe, por exemplo, podem ser utilizados para adicionar Strings ao stream de saída do **Servlet**; como a saída é mantida em um buffer por questões de performance, você pode também utilizar o método **flush()** para forçar a liberação desse buffer de saída, fazendo que o conteúdo da resposta definido por você seja imediatamente enviado para o cliente.

Assim podemos usar os métodos **print()** e **println()** dessa classe para adicionar Strings ao **stream de saída** do servlet para gerar páginas HTML dinamicamente ou enviar qualquer outro recurso aceito pelo browser cliente.

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    PrintWriter pw = response.getWriter();
    pw.println("<html>");
    pw.println("<head>");
    pw.println("<body>");
    pw.println("<h3> Posso adicionar html dinamicamente</h3>");
    pw.println("</body>");
    pw.println("</html>");
    pw.flush();
}
  
```

Ciclo de Vida de um Servlet



A implementação da interface **Servlet** (`javax.servlet.Servlet`) é usada para todos os servlets, porque é essa interface que encapsula os métodos do ciclo de vida do servlet. Nesse ciclo, existem os métodos **init()**, **service()** e **destroy()**, que estão apresentados abaixo.

A classe Servlet é carregada pelo contêiner durante seu processo de inicialização. Durante este processo, o contêiner lê um conjunto de arquivos de configuração, denominados **deployment descriptors** (descritores de distribuição da aplicação). Cada aplicação tem seu próprio arquivo descritor de distribuição, `web.xml`, que inclui uma entrada especificando nome do servlet e o nome da classe do servlet, para cada um dos servlets em uso pelo contêiner. Um instância da classe do servlet é criada pelo contêiner usando o método `Class.forName(classname).newInstance()`. Isto exige que o servlet tenha um construtor **default**, normalmente você deixará isto a cargo do compilador. Após isto o servlet estará **carregado**.

O contêiner invoca o método **init(ServletConfig)**. Este método inicializa o servlet e deve ser chamado antes que o servlet possa responder a qualquer requisição. O objeto **ServletConfig** contém todos os parâmetros de inicialização que nos especificamos no descritor de distribuição. O procedimento de inicialização do servlet é chamado de **lazy loading** (carga preguiçosa), deste modo o tempo de inicialização do contêiner é reduzido. Porém se o servlet realizar muitas tarefas durante a sua inicialização o tempo de resposta para a primeira requisição será muito alto. Para melhorar este inconveniente você pode

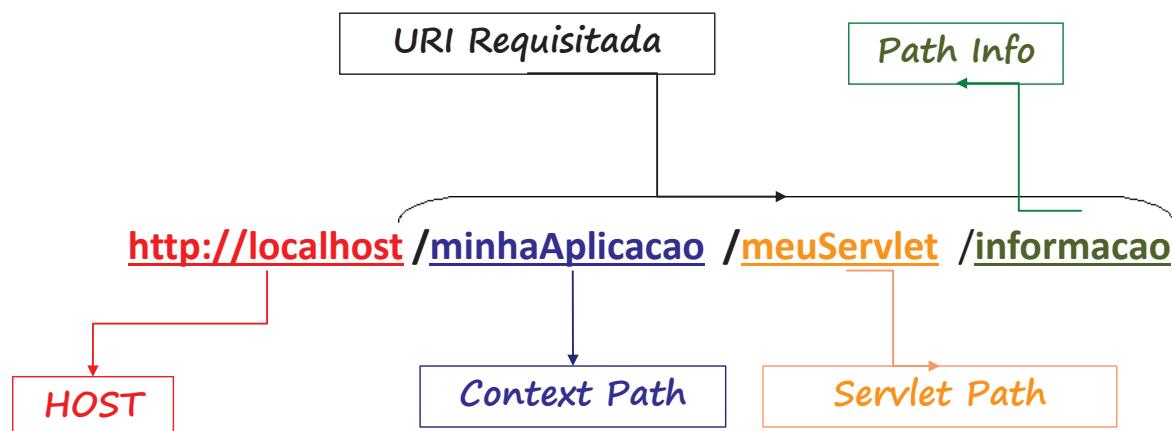
usar a tag **<load-on-startup>** no descriptor de distribuição, exigindo que o contêiner initialize o servlet no mesmo momento da sua própria inicialização.

Após a inicialização, o servlet pode atender as requisições dos clientes. Cada requisição é tratada, separadamente, por seu próprio thread. O contêiner chama o método **Servlet.service(ServletRequest, ServletResponse)** do servlet para toda requisição. O método **service()** determina o tipo de requisição que está sendo feito e a despacha para um método apropriado a fim de manipular a requisição. O desenvolvedor do servlet deve disponibilizar a implementação deste método. Se uma requisição for para o método que não esteja implementado, o método da super-classe será chamado, certamente resultando em um erro que será retornado ao requisitor.

Finalmente, se o contêiner decidir que não necessita mais da instância da servlet, ele chama o método **destroy()** que inativa o serviço da servlet. O método **destroy()**, assim como o método **init()**, é chamado somente uma única vez durante o ciclo de vida de um Servlet.

Acessando um servlet

Acessando recursos



Java Web

Um servlet tem um **nome do caminho para o arquivo**, obviamente, como **classes/pacote/UmServlet.class** (um caminho para um arquivo de classe real). O desenvolvedor da classe do servlet escolhe o nome da classe (e o nome do pacote que define parte da estrutura de diretórios), e o local no servidor define o nome completo do caminho. Mas qualquer pessoa que distribua o servlet também pode atribuir a ele um nome de **distribuição especial**. Um nome de distribuição é simplesmente um nome interno secreto, que não precisa ser igual ao nome da classe ou do arquivo. Ele pode ser igual ao nome da classe (**pacote.UmServlet**) ou o **caminho relativo** para o arquivo da classe (**classes/pacote/UmServlet.class**), mas também pode ser algo completamente diferente, como **ListarServlet**, se definido assim no arquivo web.xml usando **< servlet-mapping >**.

Assim, o servlet tem um **nome público de URL** – nome que o cliente conhece. Ou seja, o nome codificado no HTML de modo que, quando o usuário clicar em um link, que se supõe que vá a aquele servlet, este nome público URL é enviado ao servidor na solicitação HTTP.

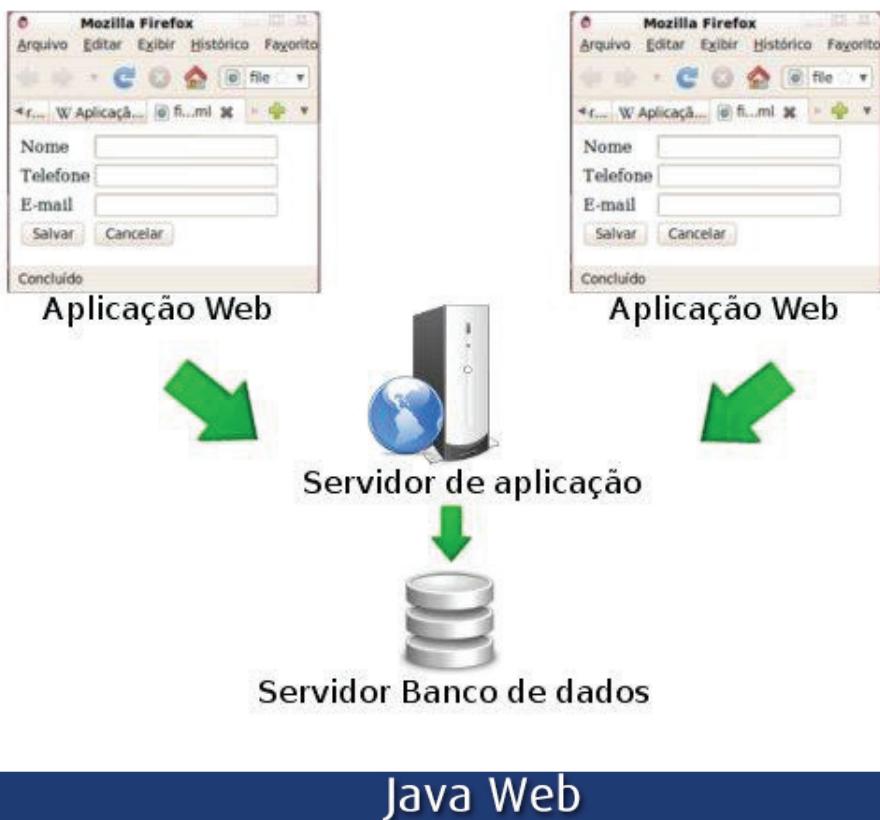
Rotear a requisição para um servlet é um processo feito pelo servlet contêiner em duas etapas. Primeiro o contêiner identifica a aplicação web à qual a requisição pertence, e então ele encontra um servlet apropriado para manipular a requisição.

Ambos os passos requerem que o contêiner quebre a URI em três partes: caminho contexto (**context path**), caminho do servlet (**Servlet Path**) e informação de caminho (**path info**).

- **Context Path** – o servlet contêiner tenta comparar a requisição com o maior pedaço possível da URI, começando como os nomes das aplicações web disponíveis. Este pedaço é chamado de **context path**. Por exemplo, se a URL é **/banco/ContaServlet/pessoafisica**, então **/banco** é o **context path** (assumindo que a aplicação **banco** exista no servlet contêiner). Se não houver um nome de aplicação compatível o **context path** é vazio, neste caso ele será associado com a aplicação web padrão (ou raiz **/**, no Tomcat é o **webapps/ROOT**).
- **Servlet Path** – após extrair o context path, o contêiner tenta comparar a maior parte possível da URL restante com algum mapeamento de servlet (**<servlet-mapping> do web.xml**) definido para a aplicação web, que esteja especificado com context path. Esta parte é chamada de **Servlet path**. Por exemplo, se a URL é **/banco/ContaServlet/pessoafisica**, então **/ContaServlet** é o **Servlet path** (assumindo que **ContaServlet** é o nome de um servlet definido para aplicação). Se não for possível encontrar um padrão compatível é retornado uma página de erro.
- **Path Info** – qualquer coisa que sobre URI após a determinação do servlet path é chamado de path info. Por exemplo, se a URI é **/banco/ContaServlet/pessoafisica**, então **/pessoafisica** é o **path info**.

Java Web

Trabalhando com Servlets



Servlets são classes Java, desenvolvidas de acordo com uma estrutura bem definida, e que, quando instaladas junto a um Servidor que implemente um Servlet Container (um servidor que permita a execução de Servlets, muitas vezes chamado de Servidor de Aplicações Java), podem tratar requisições recebidas de clientes.

Servlet trabalha com servidores de aplicação web (Servlet Container) baseados na plataforma Java que são capazes de executá-los, por exemplo: Apache Tom Cat, Jboss, GlassFish entre outros.

Para se criar um Servlet deve ser criada uma classe que estende as classe HttpServlet, depois sobrescrever os métodos necessários.

Formulários

Agenda de contatos

Cadastro de contatos

Cadastrar novo contato	Relatório dos cadastrados
Nome	Funções
iMasters	
Leandro Vieira Pinho	
Total de contatos	

Nome:
 Observações:
 DDD: Telefone: Celular:
 E-mail:
 Blog / site:
 MSN:
 gTalk:
 Skype:



Java Web

Ser capaz de lidar com as informações contidas em formulários HTML é fundamental para qualquer tecnologia de desenvolvimento de aplicações para Web. É por meio de formulários que os usuários fornecem dados, preenchem pedidos de compra e (ainda mais importante) digitam o número do cartão de crédito. As informações digitadas no formulário chegam até o Servlet por meio de um objeto do tipo **ServletRequest** e são recuperadas por meio do métodos **getParameter()**, **getParameterNames()** , **getParameterValues()** deste objeto. Todo item de formulário HTML possui um nome e esse nome é passado como argumento para o método **getParameter()** ou **getParameterValues()** que retorna na forma de um String o valor do item de formulário.

Alguns tipos de entradas de dados, como um grupo de checkbox ou lista de seleção, podem ter mais de um valor. Isso quer dizer que um único parâmetro terá diversos valores, dependendo de quantos itens o usuário selecionou. Um formulário em que o usuário possa selecionar diversos itens é algo como:

```
<form action="recruta.do" method="POST">
  Tecnologia:
  <input type="text" name="MeuServletMapeado" value="java"> <br>
  Estado:
  <select name="estado" size="5" multiple>
    <option value="AC">ACRE</option>
    <option value="DF">BRASILIA</option>
```

```
</select>
<br>
Certificação:
<input type="checkbox" name="certificados" value="scjp6">JAVA 6.0
<input type="checkbox" name="certificados" value="scejb3">EJB 3.0
<br>
<input type="submit" value="Busca Emprego">
</form>
```

Note que um formulário é criado por meio do tag **<form>**. Como parâmetros opcionais deste tag temos método da requisição (**method**), é a URL para onde será submetida à requisição (**action**). No caso do exemplo, o método adotado é o **POST** e a requisição será submetida ao próprio servlet que trata o formulário.

No seu código, você usará o método **getParameterValues()** que retorna um array:

```
String um = request.getParameter("lstconsulta");
String[] varios = request.getParameterValues("certificacoes");
```

Navegação

Reencaminhamento



Redirecionamento



Java Web

Até agora nós temos estudado os servlets do ponto de vista e uso de um único servlet. Mas em aplicações reais utilizar um único servlet para tratar todas as possíveis tarefas não é nada prático. Comumente você irá dividir suas regras de negócio em múltiplas tarefas. Por exemplo, considere um processo simplificado de uma loja virtual de livros e suas regras de negócio. Um usuário deveria poder:

- Selecionar livros para compra
- Remover ou adicionar mais livros
- Visualizar livros num carrinho virtual
- Fazer pagamento dos livros que deseja comprar
- Informar endereço de entrega

Você normalmente irá quebrar o processo de negócio em processos menores, tendo um servlet mais específico para tratar cada tarefa em foco. No processo descrito poderíamos ter um servlet **CarrinhoServlet**, que mantém dados das opções de livros já selecionado, que invoca um servlet **FinalizaCompraServlet**, após ação do usuário para encerrar compra; o servlet **FinalizaCompraServlet**, verifica se o usuário já está cadastrado em sua base de dados de clientes, senão estiver ele invoca o servlet **CadastroClientServlet**, que ao encerrar cadastro encaminha requisição novamente para **FinalizaCompraServlet**, que confirma o usuário e finaliza a compra emitindo fatura pedido.

Para implementar as funcionalidades requeridas neste exemplo, os servlets terão que coordenar seus processos e também o compartilhamento de informações. Por exemplo, o

servlet **FinalizaCompraServlet** deve conhecer os livros que foram selecionados pelo uso de **CarrinhoServlet**.

A API Servlet fornece uma maneira elegante para compartilhar dados e coordenar os processos de servlets.

Redirecionamento

Existem algumas situações onde pode ser desejável transferir uma requisição para outra URL. Isto é feito com frequência em sistemas que combinam o uso de Servlets juntamente com JSP. No entanto, a transferência pode ser para qualquer recurso. Assim, podemos transferir uma requisição de um Servlet para uma página JSP, HTML ou um Servlet. Da mesma forma uma página JSP pode transferir uma requisição para uma página JSP, HTML ou um Servlet.

Existem dois tipos de transferência de requisição: o redirecionamento e o reencaixamento. O redirecionamento é obtido usando o método **sendRedirect()** de uma instância **HttpServletResponse**, passando como argumento a URL de destino. Abaixo é mostrado o código de um redirecionando para uma página HTML.

```
response.sendRedirect("caminhoDoServletOuPaginaHtml");
```

Você deve manter em mente um conjunto de pontos importantes sobre o método **sendRedirect()**. Você não pode invocar este método se uma resposta já tiver sido emitida – ou seja, se o cabeçalho de resposta já tiver sido enviado ao browser. Se você fizer a chamada sobre esta condição o método irá disparar um exceção do tipo **java.lang.IllegalStateException**. Por exemplo, na listagem abaixo o código irá gerar uma **IllegalStateException**.

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
    PrintWriter pw = response.getWriter();
    pw.println("<html><body>Dipara exception</body></html>");
    pw.flush(); //envia resposta ao browser
    //tenta redirecionar
    response.sendRedirect("http://www.apache.org");
}
```

Neste código está forçando o envio imediato do header de resposta e a geração de texto para o browser pelo chamado do método **pw.flush()**. A resposta é tida como committed neste ponto. Outro ponto importante é que a invocação de **sendRedirect()** não é transparente para o browser, em outras palavras o servlet envia uma mensagem dizendo ao browser que deve buscar um recurso em algum outro lugar; na prática o servidor envia uma mensagem **HTTP 302** de volta para o cliente informando que o recurso foi transferido para outra URL e o cliente envia uma nova requisição para a URL informada.

Reencaminhamento

Diferentemente de **sendRedirect()**, no reencaminhamento, a requisição é enca-minhada diretamente para a nova URL mantendo todos os objetos associados e evitando uma nova ida ao cliente. O uso de reencaminhamento é mais eficiente do que o uso de redirec-ionamento. O reencaminhamento é obtido usando o método **forward()** de uma instânci-a do objeto **RequestDispatcher**, passando como argumento os objetos **HttpServletRequest** e **HttpServletResponse** para o recurso de destino.

O exemplo abaixo mostra o código de um Servlet reencaminhando a requisição para outro servlet.

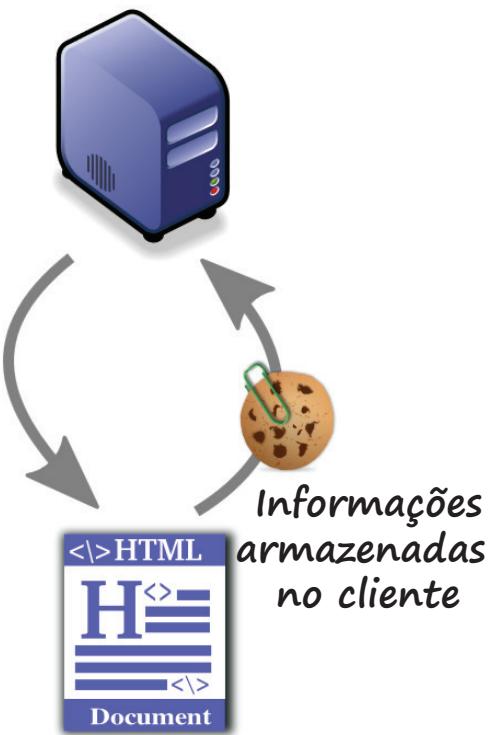
```
public class EncaminhamentoServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException, IOException {  
  
        ServletContext sc = getServletContext();  
        RequestDispatcher rd = sc.getRequestDispatcher("outroServlet-  
Mapeado");  
        rd.forward(request, response);  
    }  
}
```

No exemplo o servlet **EncaminhamentoServlet** pode processar parcialmente a re-quisição recebida e então encaminhá-la para outro servlet que irá gerar a resposta final.

Redirect versus Forward

Há uma diferença importante entre usarmos **RequestDispatcher.forward()** e **HttpServletResponse.sendRedirect()**, é que **forward()** é completamente manipulada pelo web contêiner enquanto **sendRedirect()** envia uma mensagem de redirecionamento ao browser. Em resumo, **forward()** é transparente ao browser e **sendRedirect()** não.

Manipulando Cookies



Java Web

Muitos sites existentes na Internet, as configurações principais, como idioma e modo de exibição, são mantidas no navegador, mesmo depois de um tempo sem uso. Por exemplo, caso escolha o idioma Russo na pesquisa do Google, ele será usado até você mudar de ideia. Para que isso seja possível, o navegador utiliza arquivos de textos chamados Cookies, os quais possuem como principal função armazenar as preferências dos usuários sobre um determinado site na Internet. Cada Cookie em seu PC armazena dado para um endereço web específico.

O que é um Cookie?

Cookies são pacotes de dados, gerados pelo servidor, e que são enviados junto com a resposta de uma requisição, ficando armazenados na máquina cliente acessíveis ao browser do usuário. Posteriormente, a cada requisição enviada, o browser anexa também as informações desses Cookies, permitindo que o servidor recupere os valores definidos anteriormente.

Enquanto o cookie estiver salvo em seu PC, toda vez que você digitar o endereço do site, o seu navegador irá enviar este arquivo para o site que você está conectado. Desta maneira, as suas configurações serão aplicadas de maneira automática.

Manipulação de Cookies

Os cookies fazem parte dos recursos que podemos adotar para guardar dados de identificação de clientes utilizando protocolo HTTP, uma vez que este protocolo, como já dissemos, é sem estado ou não orientado à conexão. A necessidade da identificação do cliente de

onde partiu a requisição e o monitoramento de sua interação com o site (denominada de sessão) é usado tipicamente em sistemas web para:

- Controlar, em um carrinho de compras, a associação dos itens selecionados para compra com o usuário que deseja adquiri-los. Na maioria das vezes a seleção dos itens e compra é feita por meio da navegação de várias páginas do site e a todo instante é necessário distinguir os usuários que estão realizando as requisições.
- Acompanhar as interações do usuário com o site para observar seu comportamento e, a partir dessas informações, realizar adaptações no site para atrair um maior número de usuários ou realizar campanhas de marketing.
- Saber se o usuário está acessando o site para fornecer uma visualização e um conjunto de funcionalidades adequadas às suas preferências de acordo com o seu perfil.

Infelizmente, existem situações em que cookies não irão funcionar: quando o browser do usuário estiver configurado para não aceitar cookies. Usuários preocupados com sua privacidade normalmente bloqueiam o armazenamento de cookies por seus browsers. O problema é privacidade, não segurança:

- Se você fornecer informações pessoais, servidores podem associar estas informações com ações anteriores;
- Servidores podem compartilhar informações sobre cookies;
- Sites mal projetados armazenam informações confidenciais como número de cartão de crédito diretamente nos cookies ;
- Bugs em JavaScript permitem sites hostis roubarem cookies (navegadores antigos).

Assim concluímos, se cookies não é vital para sua aplicação não use. Evite criar servlets que falhem totalmente quando cookies estão desabilitados e nunca coloque dados confidenciais em cookies.

A Classe `javax.servlet.http.Cookie`

Manipulamos um cookie através de instâncias da classe **`javax.servlet.http.Cookie`**. Essa classe fornece apenas um construtor que recebe dois argumentos do tipo String, que representam o nome e o valor do cookie.

A classe **Cookie** apresenta os seguintes métodos:

Método	Descrição
<code>String getName()</code>	retorna o nome do cookie.
<code>String getValue()</code>	retorna o valor armazenado no cookie.
<code>String getDomain()</code>	retorna o servidor ou domínio do qual o cookie pode ser acessado.
<code>String getPath()</code>	retorna o caminho de URL do qual o cookie pode ser acessado.
<code>boolean getSecure()</code>	indica se o cookie acompanha solicitações HTTP ou HTTPS.
<code>void setValue(String newValue)</code>	atribui um novo valor para o cookie.

Método	Descrição
void setDomain(String pattern)	define o servidor ou domínio do qual o cookie pode ser acessado.
void setPath(String url)	define o caminho de URL do qual o cookie pode ser acessado.
void setMaxAge(int expiry)	define o tempo restante (em segundos) antes que o cookie expire.
void setSecure(Boolean flag)	retorna o valor de um único cabeçalho de solicitação como um número inteiro.

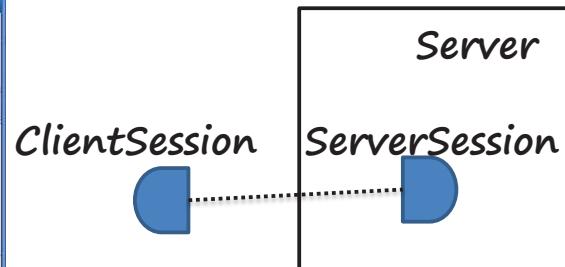
Para enviar Cookies para o Cliente siga o roteiro:

- Crie um objeto do tipo **javax.servlet.http.Cookie**.
- Chame o construtor do cookie com um nome do cookie e um valor, ambos strings.
Cookie c = new Cookie("userID", "a1234");
- Atribua a idade máxima para tempo de vida cookie, informe ao navegador para armazenar cookies em disco ao invés de apenas na memória, use:
c.setMaxAge(60*60*24*7); // Uma semana
- Insira o cookie na resposta http, use **response.addCookie(c)**; Se você esquecer este passo, o cookie não será enviado para o browser!

Par ler Cookies do Cliente faça:

- Chame **request.getCookies()**. Isto retornará um array de objetos do tipo **Cookie**.
- Percorra o array, chamando **getName()** para cada entrada até você encontrar o cookie de seu interesse. Use o valor retornado com **getValue()** para usar o dado armazenado em sua aplicação.

Gerenciamento de Sessão



Java Web

Imaginando uma aplicação simples como um carrinho virtual de compras de uma livraria virtual, por exemplo, como poderemos manter o histórico dos livros já selecionados pelo nosso cliente? Se a seleção de cada livro gera uma ou mais requisições, no momento do fechamento da compra, como fazemos para saber quais foram todos os livros selecionados?

Possíveis maneiras de resolver esse problema, de maneira a contornar essa limitação do protocolo HTTP, são:

- **Cookies** – neste caso, na primeira requisição respondida pelo servlet contêiner é adicionado um identificador de sessão (**jsessionid**) em um **cookie**. A cada nova requisição do cliente o cookie é recuperado e associado a possíveis contêiners **HttpSession** que tenham sido criados para este cliente. Esta estratégia é adotada como procedimento padrão pelo contêiner, sendo seu funcionamento transparente ao desenvolvedor. Porém, como vimos anteriormente o usuário pode desabilitar o suporte a cookies em seu browser, essa ação deliberada do usuário impede o funcionamento desta técnica.

- **Reescrita de URL** – essa estratégia é utilizada quando cookies não podem ser usados. O contêiner adiciona o identificador de sessão (**jsessionid**) no fim de cada **URL**. Assim o servidor pode associar o identificador de sessão com dados armazenados para sessão. Por exemplo: **http://host/caminho/arquivo;jsessionid=123**

Informação da sessão é **jsessionid=123**. Diferentemente de cookies esta estratégia não é transparente para o desenvolvedor; a interface **HttpServletResponse** fornece dois métodos que facilitam o manuseio de **URL reescritas**: **encodeURL()** e **encodeRedirectURL()**.

Obtendo uma sessão

Para controlar a sessão é necessário obter um objeto **HttpSession** por meio do método **getSession()** do objeto **HttpServletRequest**. Para associar um objeto ou informação à sessão usa-se o método **setAttribute()** do objeto **HttpSession**, passando para o método uma **chave (String)** e um **valor (Object)**. Note que o método aceita qualquer objeto e, portanto, qualquer objeto pode ser associado à sessão. Os objetos associados a uma sessão são recuperados com o uso método **getAttribute()** do objeto **HttpSession**, que recebe como argumento o nome associado ao objeto. Para se obter uma enumeração dos nomes associados à sessão usa-se o método **getAttributeNames()** do objeto **HttpSession**. Abaixo é mostrado como podemos utilizar **HttpSession** para armazenar dados de sessão.

```
//recupera objeto HttpSession
HttpSession session = request.getSession(true);
//extrai dados da sessão
out.println("Identificador: " + session.getId() + "<br>");
out.println("Data:"+(new Date(session.getCreationTime())));
out.println("<br>");
out.println("Último acesso: " + (new Date(session.getLastAccessedTime())));
```

Atributos da sessão

Os atributos de uma sessão são mantidos em um objeto **HttpSession**. Pode-se armazenar valores em uma sessão por meio do método **setAttribute()** e recuperá-los por meio do método **getAttribute()**.

```
//armazena dado na sessão
session.setAttribute("nomeDoAtributo" , "valorDoAtributo");
//recupera todos atributos armazenados na sessão
Enumeration valueNames = session.getAttributeNames();
```

Se um usuário não realizar nenhuma ação durante um certo período de tempo o servidor irá assumir que o usuário está inativo e irá invalidar a sessão. O **web.xml** pode ser utilizado para configurar o tempo máximo, em minutos, de uma sessão:

```
</web-app>
...
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
...
</web-app>
```

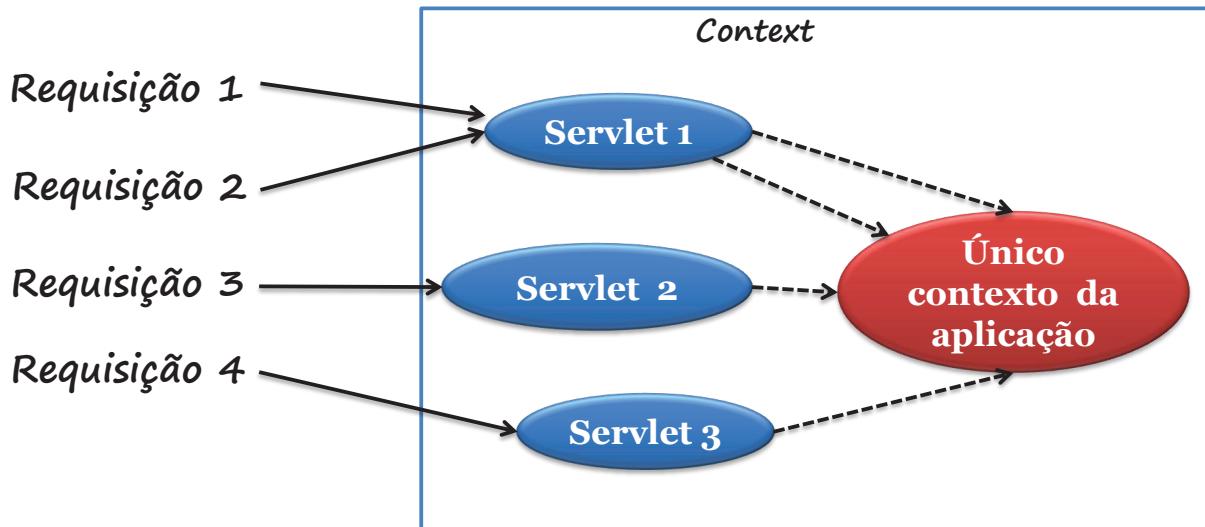
O tempo de duração da sessão, também, pode ser alterado por meio do método **setMaxInactiveInterval(int segundos)**, ele afetará somente a sessão em que for invocado. Observe que o método trabalha como tempo em segundos enquanto a tag **<session-timeout>** é configurada em minutos.

Invalidando uma sessão

Você viu na seção anterior que uma sessão é finalizada automaticamente quando o usuário permanece inativo por um período de tempo específico. Em alguns casos você poderá desejar encerrar uma sessão programaticamente. Por exemplo, num programa de carrinho virtual de compra é desejável terminar a sessão após o processo de pagamento ter completado, assim, ao enviar uma nova requisição o usuário terá uma nova sessão com um carrinho de compras sem nenhum item. HttpSession fornece o método invalidate() para este propósito, veja seu uso abaixo:

```
public class LogoutServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException, IOException {  
        // supondo que alguém tenha invocado este  
        // método a partir de um Hyperlink  
        request.getSession().invalidate();  
        // encaminha para pagina principal do usuário  
    }  
}
```

Interface Servlet Context



Java Web

Você pode imaginar a interface **ServletContext** como uma janela por onde você vê o ambiente do servlet. Um servlet usa essa interface para obter informação sobre a aplicação, tais como: **parâmetros de inicialização** para a aplicação ou **versão do servlet contêiner**. Ele pode ser usado pelos servlets para **compartilhar dados** uns com os outros, assim como fizemos com **ServletRequest** e **HttpSession**, a diferença entre eles está no escopo destes dados.

Parâmetros de inicialização

O contexto é de uma aplicação web é inicializado no momento em que a aplicação está sendo carregada. Assim como tem parâmetros de inicialização para um servlet, também os temos para o contexto. Estes parâmetros são definidos no **deployment descriptor** da sua aplicação web.

Adicionando parâmetros de inicialização no Deployment Descriptor (web.xml). Veja este exemplo:

```

<web-app>
  <context-param>
    <param-name>dburl</param-name>
    <param-name>jdbc:basedadosurl</param-name>
  </context-param>
</web-app>
  
```

O servlet de uma aplicação web pode recuperar os parâmetros definidos acima, utilizando os métodos **getInitParameter()** e **getInitParameterNames()** da interface **ServletContext**.

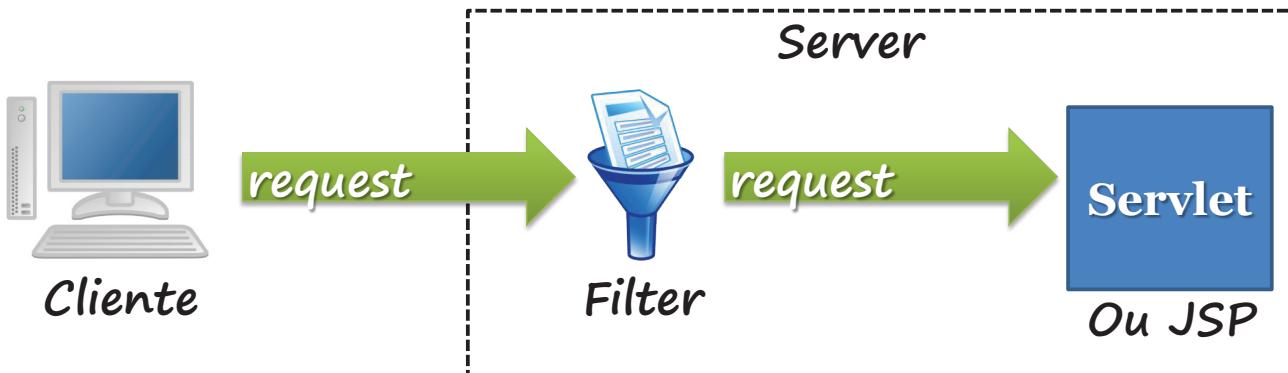
Os parâmetros de inicialização do contexto são usados para especificar informações para toda a aplicação, tais como: **conexão com banco de dados, e-mail do desenvolvedor, etc.** Claro, antes usar estes métodos você precisará de uma referência para um objeto **ServletContext**. O trecho de código demonstra como fazer isto dentro do método **init()**.

```
public void init() throws ServletException {  
    //use SevletConfig para obter ServletContext  
    ServletContext contexto= getServletConfig().getServletContext();  
  
    //ou use  
    //GenericServlet.getServletContext  
    //ServletContext contexto= getServletContext();  
  
    String dburl = contexto.getInitParameter("dburl");  
    //use dburl para criar coneçôs com banco de dados  
  
}
```

Utilizando atributos do ServletContext

Você pode adicionar informações a objetos de contexto, objetos de sessão e objetos da requisição. Felizmente cada uma dessas classes contém os mesmos quatro métodos para realizar esta operação de atribuição: **getAttribute(String)**, **getAttributeNames(String)**, **setAttribute(String, Object)**, **removeAttribute(String)**.

Filtros (Filter)



Java Web

Filtros são componentes que se interpõem entre uma requisição do cliente e um determinado recurso. Qualquer tentativa de recuperar o recurso deve passar pelo filtro. Um recurso pode ser qualquer conteúdo estático ou dinâmico (HTML, JSP, GIF, etc.).

Filtros interceptarão as requisições do cliente ao servidor antes de chegar ao **Servlet ou JSP** de destino para processamento. Quando a requisição passa pelo filtro, esse filtro pode processar os dados contidos na requisição e também decidir sobre o próximo passo da requisição – que pode ser encaminhada para um outro filtro, acessar o recurso diretamente ou impedir que o usuário acesse o recurso desejado.

Filtros podem realizar muitos tipos diferentes de funções, como exemplo:

- **Autenticação** – bloqueio de requisições baseados na identidade do usuário.
- **Auditoria e registro** – monitorando usuários de uma aplicação web.
- **Conversão de imagem** – redimensionamento de mapas entre outros.
- **Compressão de dados** – empacotamento de downloads
- **Localização** – configurando as requisições e respostas para uma localização (uso de formatos específicos de uma determinada região como hora, moeda, idioma, etc) em particular.
- **Transformações XSL/T para conteúdo XML** – convertendo as respostas de um ou mais tipos de clientes.

Criando um filtro

Uma classe Filtro deve implementar a interface **javax.servlet.Filter** que define os seguintes métodos:

- **void init(FilterConfig config) throws ServletException** – esse método é invocado pelo contêiner durante a primeira vez que o filtro é carregado na memória. Códigos de inicialização devem ser colocado, incluindo o uso de parâmetros de inicialização contidos no arquivo **web.xml**.

- **void destroy** – esse método é invocado pelo contêiner quando o filtro é descarregado da memória. Isso ocorre normalmente quando a aplicação WEB é encerrada. O código de liberação de recursos utilizados pelo filtro deve ser inserido aqui – o encerramento de uma conexão ao banco de dados, por exemplo.

- **void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException** – esse método contém toda a funcionalidade do filtro, ou seja, as regras que devem ser verificadas, antes que uma requisição possa continuar seu caminho até um recurso. Esse método é chamado pelo contêiner quando o servidor decide que o filtro deve interceptar uma determinada requisição ou resposta ao usuário. Os parâmetros passados para esse método são instâncias das classes **ServletRequest** e **ServletResponse** do pacote **javax.servlet.http** e da classe **FilterChain** do pacote **javax.servlet**. Se o filtro estiver sendo usado por uma aplicação web (o caso mais comum), o desenvolvedor pode realizar o **casting** entre esses objetos para instâncias das classes **HttpServletRequest** e **HttpServletResponse** do pacote **javax.servlet.http**, respectivamente. Isso permite a recuperação de informações específicas do protocolo HTTP.

Configuração do Deployment Descriptor

A configuração de filtros é muito parecida com a configuração de Servlets. Existe uma sessão necessária para configurar cada filtro utilizado na aplicação bem como sessões para configurar o mapeamento entre os filtros e os padrões de URLs aos quais as cadeias de filtros serão aplicadas. Um exemplo de configuração de filtro é apresentado abaixo:

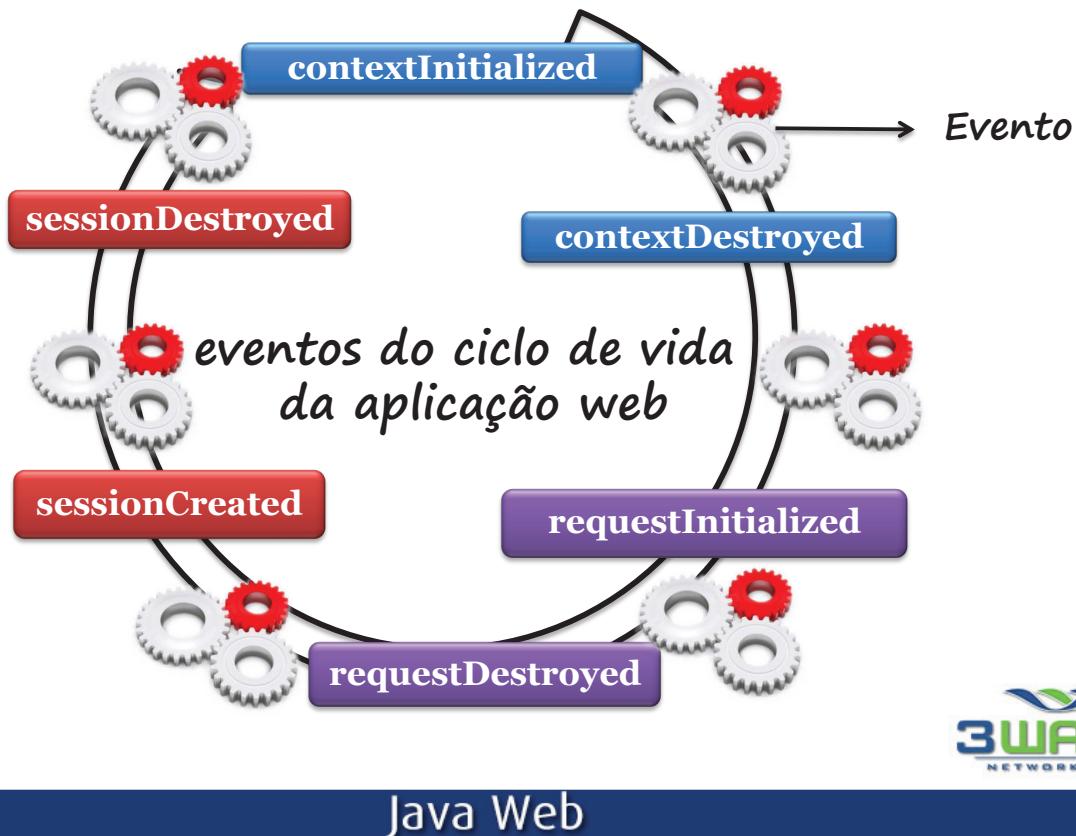
```
<filter>
<filter-name>RegistroFilter</filter-name>
<filter-class>CaminhoDoFilter</filter-class>
</filter>
<filter-mapping>
<filter-name>RegistroFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Uma vez que a interpretação do arquivo **web.xml** sofre influência da ordem em que os elementos são declarados, é recomendada a declaração dos filtros após os elementos **<context-param>** e antes de qualquer definição de servlets. Não se esqueça de colocar os elementos **<filter-mapping>** após a definição dos filtros.

```
</context-param>
<filter>
    <filter-name>RegistroFilter</filter-name>
    <filter-class>cap8. CaminhoDoFilter </filter-class>
</filter>
<filter-mapping>
    <filter-name>RegistroFilter</filter-name>
    <url-pattern>/*</url-pattern>
<dispatch>REQUEST</dispatch>
    <dispatch>INCLUDE</dispatch>
</filter-mapping>
```

Elementos **<dispatch>** possuem um dos seguintes valores: **REQUEST, INCLUDE, FORWARD** e **ERROR**. Isso indica quando um filtro deve ser aplicado apenas a requisições do cliente, apenas em includes, apenas em requisições, apenas em caso de erro ou na combinação desses quatro valores.

Listeners



Java Web

Com os listeners e eventos de ciclo de vida do aplicativo é possível notificar os listeners interessados ao alterar os contextos e sessões do servlet. Por exemplo, é possível notificar os usuários quando atributos forem alterados e se sessões ou contextos de servlet forem criados ou destruídos.

Os listeners de ciclo de vida fornecem ao desenvolvedor de aplicativos maior controle sobre as interações com objetos ServletContext e HttpSession. Os atendentes de contexto de servlet gerenciam recursos no nível de um aplicativo. Os listeners de sessão gerenciam recursos associados a uma série de pedidos de um único cliente. Os listeners estão disponíveis para os eventos de ciclo de vida e para os eventos de modificação de atributos.

Um ponto interessante é que pode haver múltiplos **Listeners** para cada tipo de evento e o desenvolvedor tem a flexibilidade para definir a ordem de eventos serão chamados. O container gerencia o ciclo de vida dos **Listeners**. É de responsabilidade do servidor instanciar cada uma das classes registradas como Listener antes da execução do primeiro pedido para a aplicação. Além disso, cada uma das classes deve ser referenciado até a última solicitação.

Classes para alterações de contexto e sessão do servlet

Os métodos a seguir são definidos como parte da interface javax.servlet.ServletContextListener:

- **void contextInitialized(ServletContextEvent)**

Notificação de que o aplicativo da Web está pronto para processar pedidos. Coloque

o código nesse método para ver se o contexto criado é para seu aplicativo da Web e se for, aloque uma conexão de banco de dados e armazene a conexão no contexto do servlet.

- **void contextDestroyed(ServletContextEvent)**

Notificação de que o contexto do servlet está prestes a ser encerrado. Coloque o código nesse método para ver se o contexto criado é para seu aplicativo da Web e se for, feche a conexão do banco de dados armazenada no contexto do servlet.

Os métodos a seguir são definidos como parte da interface **javax.servlet.ServletRequestListener**:

- **public void requestInitialized(ServletRequestEvent re)**

- Notificação de que o pedido está prestes a entrar no escopo
- Um pedido é definido como entrando no escopo quando está prestes a entrar no primeiro filtro da cadeia de filtros que processa o pedido.

- **public void requestDestroyed(ServletRequestEvent re)**

- Notificação de que o pedido está prestes a sair do escopo
- Um pedido é definido como saindo do escopo quando ele sai do último filtro de sua cadeia de filtros.

As interfaces listener a seguir são definidas como parte do pacote javax.servlet:

- ServletContextListener
- ServletContextAttributeListener

Uma interface de filtro a seguir é definida como parte do pacote javax.servlet:

- Interface FilterChain - métodos: doFilter()

As classes de eventos a seguir são definidas como parte do pacote javax.servlet:

- ServletContextEvent
- ServletContextAttributeEvent

As interfaces a seguir são definidas como parte do pacote javax.servlet.http:

- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionActivationListener

A classe de eventos a seguir é definida como parte do pacote javax.servlet.http:

- HttpSessionEvent

Listeners precisão ser registrados no **deployment descriptor**, **web.xml**, da aplicação. São definidos usando tags **<listener>** e serão executados na ordem que você especificar. A tag **<listener>** tem um elemento filho, **<listener-class>**, que define a classe que implementa a interface.

```
<listener>
<listener-class>
    com.listeners.MeuListener
</listener-class>
</listener>
```

Suporte a Annotations

@WebServlet

@WebListener

@WEBFILTER

@WebInitParam

@MultipartConfig



Java Web

A especificação do Servlet 3.0 foi considerada um divisor de águas para o programador Java, sua especificação, tem como principais características:

- Facilidade de desenvolvimento;
- Capacidade de conexão e extensibilidade;
- Suporte assíncrono;
- Melhorias de segurança e APIs existentes.

Como foi visto anteriormente, criar Servlets, Filtros e Listeners mapeando no web.xml é um processo muito trabalhoso. Um dos grandes problemas é que temos que configurar cada um no web.xml e se quisermos acessar esse servlet de maneiras diferentes, temos que criar vários mapeamentos para o mesmo servlet, o que pode com o tempo se tornar um problema devido a difícil manutenção.

@WebServlet

Para definir um Servlet, basta incluir esta anotação. Obrigatoriamente você deverá incluir pelo menos uma urlPattern que é utilizado para indicar o padrão de URL que deverá ser mapeado para o servlet, ele permite também a definição de mais de um pattern, o que representa, na prática, que você pode delegar a um servlet o atendimento de requisições que usam diversos padrões de URL. A vantagem desta abordagem é que você centraliza suas operações em apenas um objeto e, dependendo da requisição feita, pode tratá-la de forma adequada. Não esqueça de que as classes anotadas com o @WebServlet também deverão estender javax.servlet.http.HttpServlet.

A annotation @WebServlet também possui outros atributos, como name (para definir um nome para o servlet), description (para definir uma descrição), asyncSupported (para suporte a servlets assíncronos, que serão abordados mais adiante) e initParams.

@WebFilter

Você pode facilmente criar um filtro anotando a classe com javax.servlet.annotation.WebFilter. Da mesma forma que o @WebServlet devemos incluir a URL para mapeamento do filtro, podendo ser mais de uma URL no urlPatterns.

@WebInitParam

A anotação javax.servlet.annotation.WebInitParam é utilizada para especificar parâmetros iniciais para servlets e filters ou seja, pode ser utilizada tanto para @WebFilter como @WebServlet.

@WebListener

Podemos capturar vários eventos web com esta anotação. As classes anotadas com o @WebListener devem implementar pelo menos algumas das seguintes interfaces:

- HttpSessionAttributeListener
- HttpSessionListener
- ServletContextAttributeListener
- ServletContextListener
- ServletRequestAttributeListener
- ServletRequestListener

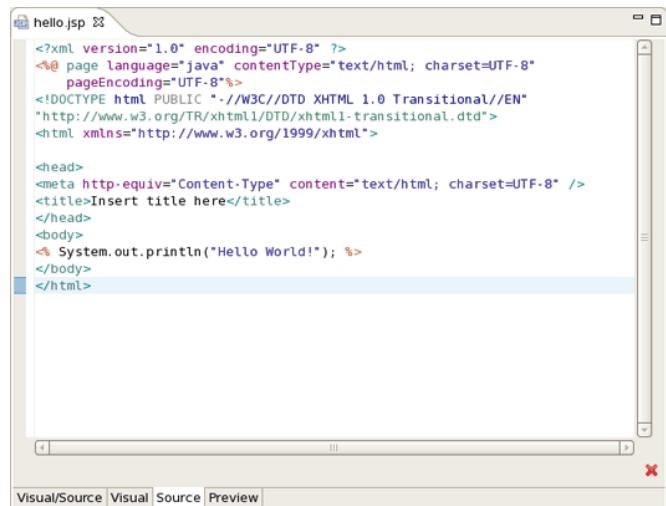
As possibilidades são várias, com destaque para as notificações quando uma sessão é criada ou invalidada através da implementação do HttpSessionListener, quando algum objeto é adicionado ou excluído da sessão através do HttpSessionAttributeListener ou até mesmo adicionar filtros de forma dinâmica com a implementação do ServletContextListener.

@MultipartConfig

Esta anotação indica que o Servlet vai esperar uma requisição do tipo multipart/form-data. Com ela podemos setar o caminho absoluto de um diretório para upload, tamanho máximo do arquivo para upload, tamanho máximo da requisição em bytes e o tamanho do arquivo em bytes após o qual será temporariamente armazenado no disco.

Java Web

JSP – JavaServer Pages

```

<?xml version="1.0" encoding="UTF-8" ?>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<% System.out.println("Hello World!"); %>
</body>
</html>

```

Visual/Source Visual Source Preview



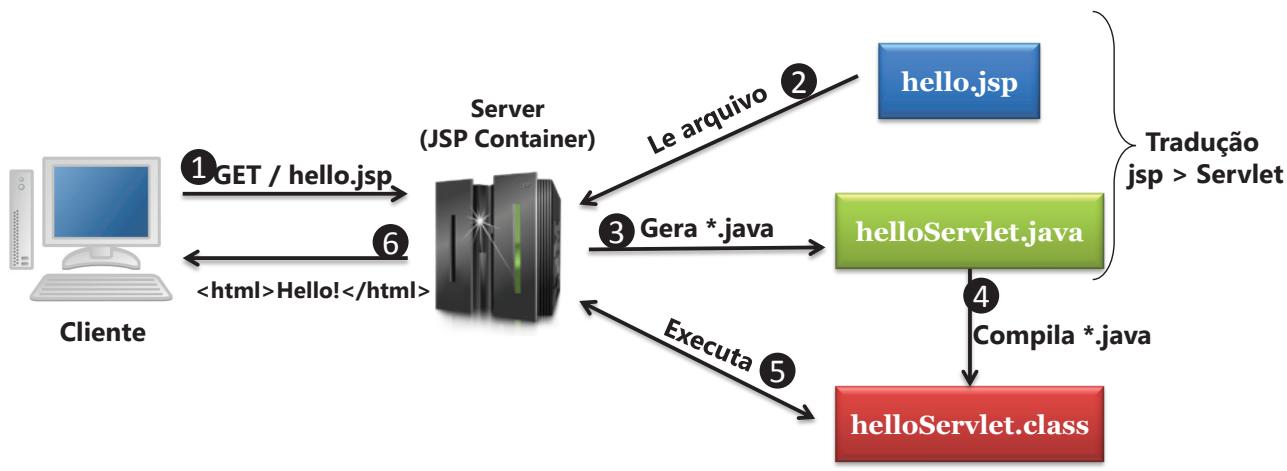
Java Web

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. Como o HTML é uma linguagem estática, o JSP será o responsável por criar dinamismo.

Uma pagina JSP possui extensão .jsp e consiste em uma página com codificação HTML e com codificação Java, inserida entre as tag's , denominada scriptlets e funcionando da seguinte forma: o servidor recebe uma requisição para uma página JSP, interpreta esta página gerando a codificação HTML e retorna ao cliente o resultado de sua solicitação. É necessário apenas desenvolver as páginas JSP e disponibilizá-las no Servlet Container (Tomcat, por exemplo). O trabalho restante será realizado pelo servidor que faz a compilação em tempo de uso transformando o jpg em bytecode.

Assim, pode-se definir o JSP como uma tecnologia que provê uma maneira simples e prática de desenvolver aplicações dinâmicas baseadas em web, sendo independente de Plataforma de Sistema Operacional.

Ciclo de vida do JSP



Java Web

Servlet é uma boa ideia, mas dá para imaginar montar uma página complexa usando **println()**? Muitas vezes o desenvolvimento de aplicações web é uma tarefa complexa que envolve vários profissionais. A tarefa de projeto do layout da página fica a cargo do Web Designer, incluindo a diagramação dos textos e imagens, aplicação de cores, tratamento das imagens, definição da estrutura da informação apresentada na aplicação e dos links para navegação pela mesma. Já o Desenvolvedor Web é responsável pela criação das aplicações que vão executar em um site.

O trabalho destes dois profissionais é somado na criação de um único produto, mas durante o desenvolvimento a interferência mútua deve ser a mínima possível. Ou seja, um profissional não deve precisar alterar o que é feito pelo outro profissional para cumprir sua tarefa. A tecnologia Servlet não nos permite atingir esse ideal. Por exemplo, suponha que um Web Designer terminou o desenvolvimento de uma página e a entregou para o Desenvolvedor Web codificar em um Servlet. Se após a codificação o Web Designer desejar realizar uma alteração na página será necessário que ele altere o código do Servlet (do qual ele nada entende) ou entregar uma nova página para o Desenvolvedor Web para que ele a codifique totalmente mais uma vez. Quaisquer umas dessas alternativas são indesejáveis e foi devido a esse problema a Sun desenvolveu uma tecnologia baseada em Servlets chamada de JSP.

A idéia de se usar **scripts** de linguagens de programação em páginas HTML que são processados no lado servidor para gerar conteúdo dinâmico não é restrita à linguagem Java.

Java Server Pages são páginas HTML que incluem código Java e outras tags especiais. Desta forma as partes estáticas da página não precisam ser geradas por **println()**. Elas são

fixadas na própria página. A parte dinâmica é gerada pelo código JSP. Assim a parte estática da página pode ser projetada por um Web Designer que nada sabe de Java.

A primeira vez que uma página JSP é carregada pelo container JSP o código Java é compilado gerando um Servlet que é executado, gerando uma página HTML que é enviada para o navegador. As chamadas subsequentes são enviadas diretamente ao Servlet gerado na primeira requisição, não ocorrendo mais às etapas de geração e compilação do Servlet.

O exemplo mostra um esquema das etapas de execução de uma página JSP na primeira vez que é requisitada. Na etapa (1) a requisição é enviada para um container Servlet/JSP. Na etapa (2) o container verifica que não existe nenhuma instância de Servlet correspondente à página JSP. Na etapa (3) a página JSP é traduzida para código fonte de uma classe Servlet que será usada na resposta à requisição. Na etapa (4) o código fonte do Servlet é compilado. Finalmente, na etapa (5) é invocado o método **service()** da instância Servlet para gerar a resposta à requisição.

Elementos JSP

Tag JSP	Descrição	Sintaxe da tag
Diretiva	Especifica instruções de tradução para o motor JSP	<%@ Diretivas%>
Declaração	Declara e define métodos e variáveis	<%! Declarações Java%>
Scriptlet	Permite ao desenvolvedor escrever código Java de forma livre na página JSP	<% Código Java %>
Expressão	Usado para imprimir conteúdo de variáveis na saída HTML gerada por uma página JSP	<%= uma expressão%>
Ação	Fornece instruções em tempo de requisição para o motor JSP	<jsp:nome_da_ação />
Comentário	Usado para comentar partes do código JSP e para documentação	<%-- qualquer texto --%>



Java Web

Elementos utilizados em JSPs

Como qualquer outra linguagem a **linguagem de scripting**, JSP tem uma gramática bem definida e inclui elementos sintáticos para realizar várias tarefas, tais como: declaração de variáveis e métodos, expressões de impressão e invocação de outra página JSP. Os elementos sintáticos, também chamados de tags JSP, estão agrupados em seis categorias, veja **Tabela 10.1** para uma rápida visão deste elementos.

Diretivas JSP

Existem dois tipos principais de diretivas. **Diretiva page** que permite situações como importação de classes, customização de super classes servlet entre outras. **Diretiva include** que permite que seja inserido o conteúdo de um arquivo no servlet no momento em que o arquivo JSP é traduzido para servlet.

Diretiva Page

```
<%@page atributo1=valor1 atributo2=valor2 atributo3=valor3 ... %>
```

Abaixo relacionamos os atributos mais utilizados nas diretivas Page:

Atributo	Utilização	Descrição
contentType	<code>contentType="text/html"</code>	Indica qual o tipo de conteúdo que a página JSP estará gerando e enviando para o browser ex: text/html ou text/plain (não interpretado pelo browser).
Import	<code>import="java.util.*"</code>	Permite que seja especificado qual o pacote a ser importado. O atributo import é o único que pode aparecer várias vezes.
isThreadSafe	<code>isThreadSafe = "true false"</code>	O valor de true (default) indica o processamento normal do servlet quando múltiplas requisições podem ser acessadas simultaneamente na mesma instância de servlet. O valor false indica que o servlet deve implementar SingleThreadModel , como requisição para cada requisição sinalizada ou com requisições simultâneas sendo uma em cada instância.
session	<code>session = "true false"</code>	O valor de true (default) indica que a variável predefinida session (do tipo HttpSession) deve estar ligada a sessão existente, caso não exista uma sessão, uma nova sessão deve ser criada para ligá-la. O valor false indica que sessões não devem ser usadas.
buffer	<code>buffer = "sizekb none"</code>	Especifica o tamanho do buffer para o JspWriter out . O buffer padrão é definido pelo servidor.
autoFlush	<code>autoFlush = "true false"</code>	O valor de true (default) indica se o buffer deve ser esvaziado quando estiver cheio. O valor false, indica que uma exceção deve ser mostrada quando ocorrer overflows .
errorPage	<code>errorPage = "url"</code>	Especifica que a página JSP deve processar algum Throwables , mas não carregá-lo na página corrente.
isErrorPage	<code>isErrorPage = "true false"</code>	Define se uma página pode atuar como uma página de erro para uma outra página JSP. O default é false.

Diretiva include

A diretiva include permite a inclusão estática de arquivos na hora em que a página JSP é traduzida no servlet. A inclusão é estática porque não há processamento do arquivo incluído, somente ocorre a cópia do texto do arquivo referenciado no local onde se encontra umas das definições:

```
<%@ include file="URL" %>
```

O atributo **file** é mandatório e deve conter uma URL relativa, não contém informação de protocolo, host ou porta. A **URL** deve ser **relativa o documento JSP** corrente (não usa a / no início do caminho), ou relativa ao **document root** (começa com a / no caminho).

Sintaxe e Semântica JSP

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Sintaxe e Semântica JSP</title>
    </head>
    <body>
        <%
            class Calculadora {
                <%-- Método para somar dois valores --%>
                public int somar(int x, int y) {
                    return x + y;
                }
            };
            Calculadora ca = new Calculadora();
        %>
        <font face="verdana, arial" size=5>
            Soma Escolhida: <%=ca.somar(5, 5)%>
        </font>
    </body>
</html>

```

.....> Diretiva Page

.....> Comentário

.....> Expressão

Declarções



Java Web

Declarções

As declarções são usadas para **definir variáveis** e métodos específicos para uma página JSP. Os métodos e variáveis declaradas podem então ser referenciados por outros elementos de criação de **scriptlets** na mesma página. Pode assumir a seguinte sintaxe - **<%! Declaração %>**, veja no slide acima:

O escopo de uma declaração é geralmente o arquivo JSP, mas se for incluído outros arquivos com a diretiva **include**, o escopo se expande para o escopo do arquivo incluído, isto é, passa a fazer parte de uma única página JSP.

Expressões

Pode conter alguma expressão válida da linguagem de script usada nessa página, no caso a linguagem é Java, mas sem ponto-e-vírgula. Exemplo abaixo:

```

<%= new Date() %>
<%= aux1 + aux2 + aux3 %>

```

Essa avaliação é feita em tempo de execução, quando a página é solicitada, permitindo fácil e rápido acesso a informação que foi requisitada. Por exemplo, uma exibição de data e hora em que a página é acessada.

Para construir uma expressão em JSP você pode colocar entre as tags qualquer expressão definida na **Especificação da Linguagem Java**. Ao contrário dos **scriptlets** (que veremos a seguir), uma expressão não aceita ponto e vírgula e define somente uma expressão da Linguagem.

Scriptlets

Scriptlets permite inserir trechos de código em Java na página JSP. Uma vez que scriptlets podem conter qualquer código Java, eles são comumente usados para agregar lógica computacional dentro da página JSP. O exemplo abaixo demonstra o uso.

```
<%  
    out.println("Isso é um Scriptlet");  
    out.println("Pode adicionar qualquer código JAVA");  
%>
```

Quando você escreve um script, você pode usar os **objetos implícitos** (veremos a seguir) do JSP ,das classes importadas, variáveis ou métodos (declarados entre as **tags <%! e %>**) ou objetos nomeados através da tag **<jsp:useBean>**.

Ações

Ações permitem que você execute tarefas sofisticadas como **instanciação de objetos, comunicação com** recursos do lado do servidor como **Servlets e JSP**, sem que para isto seja necessária codificação Java. Entretanto o mesmo pode ser obtido usando código Java dentro dos **scriptlets** como visto anteriormente. Todas as tags de ação são construídas com a **sintaxe XML**.

<jsp:include>

A tag **<jsp:include>** pode ser utilizada para redirecionar o pedido para qualquer recurso estático ou dinâmico a partir de uma página JSP principal. Os recursos podem ser desde um servlet, um arquivo HTML e também outra JSP.

```
<jsp:include page ="url" flush="true" />
```

A tag possui dois atributos **page** e **flush**. O atributo **page** informa a URL para onde irá ocorrer a chamada. O atributo **flush** serve para **informar a página JSP** principal que o **seu buffer** de conteúdo de saída **será descarregado** antes de se passar o controle para a página referenciada na tag, permitindo que a página incluída possa adicionar seu conteúdo de saída corretamente. Por esse motivo ele deve sempre ser utilizado com o valor true.

<jsp:forward>

A tag **<jsp:forward>** serve para **redirecionar pedidos** para qualquer **JSP, Servlet** ou **página HTML**, dentro do mesmo contexto o qual a página que disparou a ação se encontra. A página que realiza o desvio tem seu processamento finalizado exatamente no ponto onde o redirecionamento acontece embora todo o processamento até este ponto ainda aconteça. Se houver dados no buffer de saída colocados por esta página, estes serão apagados antes do desvio de controle.

```
<jsp:forward page="url"/>
```

Comentários

Existem dois tipos principais de comentários que podem ser usados em uma página JSP. Comentário de Conteúdo: esses comentários são transmitidos de volta para o navegador como parte da resposta de JSP e são visíveis na visualização do código da página.

Aqueles familiarizados com HTML percebem que é a mesma sintaxe de comentário para essa linguagem de marcação. Tais comentários não produzem qualquer output visível, mas podem ser visualizados pelo usuário final através do item **view source** do navegador.

Comentários JSP não são enviados para o cliente e são visíveis apenas nos arquivos fontes JSP originais. O corpo do comentário é ignorado pelo container JSP. Os comentários JSP podem assumir duas sintaxes:

```
<%-- comentário jsp --%>
```

E / OU

```
<% /* comentário scriptlet */ %>
```

Esse segundo comentário é introduzido dentro da página através de um scriptlets, usando a sintaxe de comentário nativa da linguagem de criação de scripts, no caso Java.

Objetos Implícitos

Objeto	Classe ou Interface	Descrição
page	java.lang.Object	Instância de servlet da página.
config	javax.servlet.ServletConfig	Dados de configuração de servlet.
request	javax.servlet.http.HttpServletRequest	Dados de solicitação, incluindo parâmetros.
response	javax.servlet.http.HttpServletResponse	Dados de resposta.
out	javax.servlet.jsp.JspWriter	Fluxo de saída para conteúdo da página.
session	javax.servlet.http.HttpSession	Dados de sessão específicos do usuário.
application	javax.servlet.ServletContext	Dados compartilhados por todas as páginas da aplicação.
pageContext	javax.servlet.jsp.PageContext	Dados de contexto para execução da página.
exception	javax.lang.Throwable	Erros não capturados ou exceção.



Java Web

Assim como todo objeto em Java, cada objeto implícito é uma instância de uma classe ou interface e segue uma API correspondente. Abaixo segue um resumo dos objetos implícitos disponíveis em JSP, suas respectivas classes/interfaces e uma pequena descrição do objeto.

Page

O objeto **page** representa a própria página JSP ou, mais especificamente, uma instância da classe de servlet gerada na tradução da página JSP. O objeto **page** é do tipo **java.lang.Object**.

Esta variável raramente é utilizada. De fato, uma vez que esta variável é do tipo **Object**, ela não pode ser usada diretamente por invocações de servlets.

```
<%= page.getServletInfo()           %> ----- Erro
<%= ((Servlet)page).getServletInfo() %> ----- Ok, typecast
<%= this.getServletInfo()           %> ----- OK
```

A primeira expressão dá erro de compilação indicando que **getServletInfo()** não é um método de **java.lang.Object**.

Na segunda há um **cast** da referência **page** para **Servlet**. Uma vez que **page** refere-se ao servlet gerado, que por sua vez implementa a interface **Servlet**. Perceba que neste caso a variável **page** pode sofrer **casting** para **JspPage** ou **HttpJspPage**, porque as duas interfaces são derivadas de **Servlet**.

A terceira expressão nós usamos a referência **this** que é uma instância da servlet gerada.

config

O objeto **config** armazena dados de configuração do servlet, em forma de parâmetros de inicialização para o servlet de uma página JSP quando compilada. Pelo fato das páginas JSP raramente serem escritas para interagir com parâmetros de inicialização, este objeto implícito raramente é usado na prática.

request e response

O objeto **request** e **response** representa a solicitação e resposta, respectivamente. O objeto **request** implementa a interface **javax.servlet.http.HttpServletRequest** enquanto **response** implementa a interface **javax.servlet.http.HttpServletResponse**. Esses objetos são passado ao método `_jspService()` no momento em que um cliente faz uma requisição à página JSP. Este objetos funcionam da mesma forma que utilizados em servlets, ou seja, analisando a requisição e enviando a resposta:

```
<body>
    Seu IP :<%=request.getRemoteAddr()%><br>
    Seu Host :<%=request.getRemoteHost()%><br>
</body>
```

Objeto out

Este objeto implícito representa o fluxo de saída para a página, cujo conteúdo será enviado para o navegador com o corpo de sua resposta. O objeto **out** é uma instância da classe **javax.servlet.jsp.JspWriter**. Esse objeto implementa todos os métodos **print()** e **println()** definidos por **java.io.Writer**. Por exemplo, o objeto **out** pode ser usado dentro de um script para adicionar conteúdo à página gerada. Veja o exemplo abaixo.

```
<% out.print("Este é um exemplo da utilização do objeto out"); %>
```

Esse objeto é muito utilizado para gerar conteúdo dentro do corpo de um script, sem ter que fechá-lo temporariamente para inserir conteúdo de página estático.

Contudo, deve-se evitar usar os métodos **print()** ou **println()** para inserir cadeias de caracteres muito grandes, é aconselhável fechar o script e inserir o conteúdo estático.

session

Este objeto representa a sessão atual de um usuário individual. Todas as solicitações feitas por um usuário são consideradas parte de uma sessão. Desde que novas solicitações por aqueles usuários continuem a ser recebidas pelo servidor, a sessão persiste. Se, no entanto, um certo período de tempo passar sem que qualquer nova solicitação do usuário seja recebida, a sessão expira. O objeto **session** armazena informações a respeito da sessão.

O objeto **session** é uma instância de **javax.servlet.http.HttpSession**, funciona da mesma maneira que vimos em servlets. Uma sessão só estará ativa se existir a diretiva:

```
<%@ page session="true" %>
```

application

Este objeto representa a aplicação à qual a página JSP pertence. Ele é uma instância da interface **javax.servlet.ServletContext**. Os containers JSP tipicamente tratam do primeiro nome de diretório em uma URL como o nome de uma aplicação.

pageContext

O objeto **pageContext** fornece várias facilidades como gerenciamento de sessões, atributos, páginas de erro, inclusões e encaminhamento de requisições de fluxo de resposta. O objeto **pageContext** é uma instância da classe **javax.servlet.jsp.PageContext**.

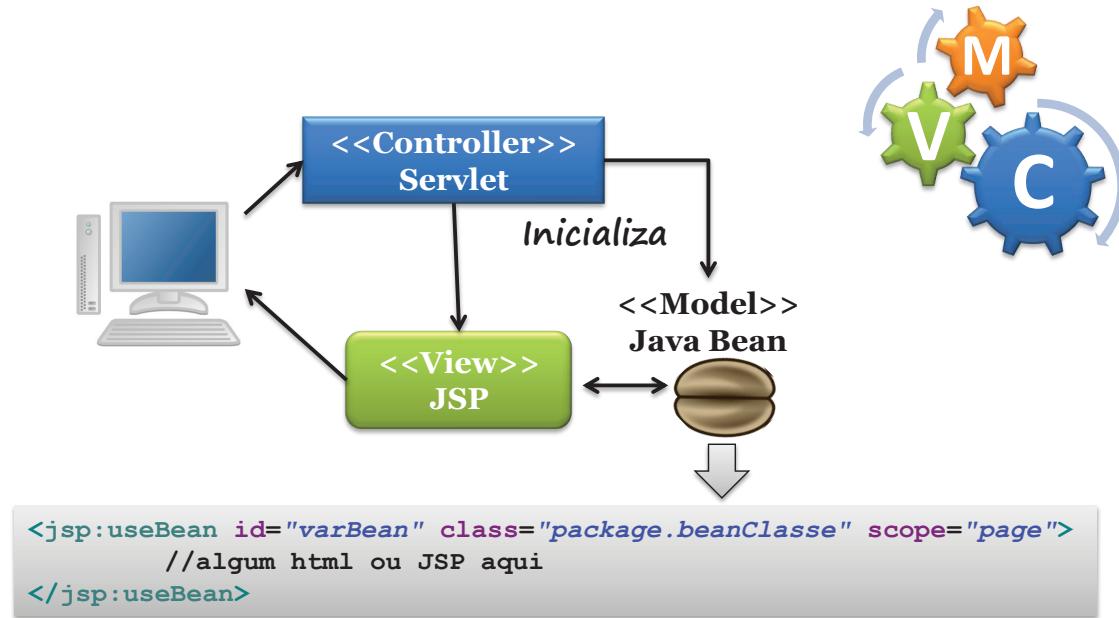
Exception

O objeto **exception** não está automaticamente disponível em todas as páginas JSP. Este objeto está disponível apenas nas páginas que tenham sido designadas como páginas de erro, usando o atributo **isErrorPage** configurado como **true** na diretiva **page**.

```
<%@ page isErrorPage="true" %>
```

O objeto **exception** é uma instância da classe **java.lang.Throwable** correspondente à exceção não tratada ou repassada que fez com que o controle fosse transferido para a página de erro.

JavaBeans



Java Web

JavaBeans são classes Java reutilizáveis que seguem algumas regras bem definidas para nomeação de seus métodos e variáveis. Embora a definição exata de JavaBeans fuja ao escopo desse material, para efeitos de uso do uso dessas classes em páginas JSP, é necessário que se siga algumas regras básicas no seu desenvolvimento:

- 1) Podem existir um ou mais métodos públicos para a definição de valores de propriedades do Bean; esses métodos são chamados de métodos **setter**.
- 2) Podem existir um ou mais métodos públicos para a obtenção de valores de propriedades do Bean; esses métodos são chamados de métodos **getter**.

Temos, a seguir, uma exemplo de classe JavaBean:

```

public class Alguem {
    private String nome;
    private int idade;

    // construtor padrão
    public Alguem() { }
    // criar métodos getters e setters
}
  
```

<jsp:useBean>

A tag **<jsp:useBean>** te permite acessar métodos de um JavaBean em sua página JSP.

```
<jsp:useBean id="varBean" class="package.beanClasse" scope="page">
//algum html ou JSP aqui
</jsp:useBean>
```

O atributo **id** especifica qual o nome da instância do bean dentro da aplicação. O atributo **scope** possui um dos valores de escopo já vistos anteriormente: **page**, **request**, **session** ou **application**. E por último o atributo **class** que define o nome da classe a qual o bean será instanciado.

Scope determina quanto tempo o objeto irá existir como variável de programa. Para que um bean possa ser instanciado não poderá existir objetos anteriores com mesmo identificador em **id** e **scope**, caso contrário, se já existir uma instância para o objeto esta será usada.

<jsp:setProperty>

A tag **<jsp:setProperty>** é utilizada para configurar valores das propriedades dos componentes JavaBeans em uma página JSP. Existem quatro formatos para a utilização desta tag.

Sintaxe	Descrição
<jsp:setProperty name="nomeBean" property="nomePropriedade" />	Usando este formato a propriedade especificada será setada com o valor que será passado como parâmetro no pedido HTTP(request).
<jsp:setProperty name="nomeBean" property="*" />	Usando esta forma cada parâmetro enviado no pedido HTTP request será avaliado a fim de se encontrar um método correspondente combinando com seu nome, caso isto aconteça o método set é invocado com o valor do parâmetro recebido no pedido HTTP.
<jsp:setProperty name="nomeBean" property="nomePropriedade" param="nomeParametro" />	Usando esta forma a propriedade do bean será configurada com o valor do parâmetro do pedido especificado na tag, invocando-se o método set apropriado com o valor do parâmetro como argumento.
<jsp:setProperty property="nomePropriedade" value="String <%= ... %>" />	Usando esta forma o valor especificado será atribuído a propriedade do bean em questão. Este valor poderá ser uma cadeia de caracteres ou até mesmo o resultado de uma expressão JSP (<%= ... %>).

<jsp:getProperty>

A tag **<jsp:getProperty>** é utilizada para recuperar valores de uma propriedade de um componente JavaBean, converte o valor recuperado em uma cadeia de caracteres (String) e insere no conteúdo de saída. Os dois atributos necessários para esta tag são o nome do Bean (propriedade **name**) já referenciado anteriormente na tag **<jsp:useBean>**, e o que indica a propriedade (**property**) que se deseja recuperar o valor. O atributo **property** deve possuir um método **getXXX()** correspondente no Bean.

```
<jsp:getProperty name="nomeBean" property="nomePropriedade" />
```

Componentes Web Reutilizáveis



Java Web

No mundo JSP reusar componentes web significa essencialmente que você vai incluir o conteúdo ou o resultado do processamento de outro componente web na página JSP. Isto pode ser feito de duas maneiras: estática ou dinâmica. Inclusão estática envolve incluir o conteúdo do componente web no arquivo JSP no momento em que o arquivo JSP está sendo traduzido, enquanto na inclusão dinâmica a saída gerada por outro componente é incluída juntamente como a saída da página JSP que recebe a requisição.

Fragmentos JSP

Um fragmento de página é a parte de uma página, como o cabeçalho, o rodapé ou a barra de navegação, que pode ser reutilizada em outras páginas. Por exemplo, você pode colocar um elemento comum, como um gráfico ou um campo Pesquisa, em um fragmento de página e incluí-lo como um cabeçalho em todas as páginas do aplicativo. Você pode também incluir o nome da sua companhia e informações sobre direitos autorais em um fragmento de página e usar tal fragmento como o rodapé do seu aplicativo. Assim como a página principal, o fragmento de página é uma página JSP com seu próprio Bean de página associado. Entretanto, a extensão de arquivo de um fragmento de página normalmente será algo como **jspf** em vez de **jsp** (isto é só uma convenção – mas também evita o processamento dos fragmentos).

Diretiva include

Conforme visto anteriormente, a diretiva **<@ include file="URL">** nos permite fazer inclusão estática de fragmento de código. Uma vez que o código da página JSP incluída torna-

se uma parte da página JSP que inclui, cada página incluída pode acessar as variáveis e métodos definidos na outra página. Elas também compartilham todos os objetos implícitos.

Quando uma diretiva include inclui um arquivo as seguintes regras devem ser observadas:

- Nenhum processamento pode ser feito em tempo de tradução, assim o valor do atributo **file** não pode ser uma expressão. O seguinte uso de include é inválido:

```
<% String umaURL = "copyright.html"; %>
<%@ include file="<%=" umaURL %>" %>
```

- Como parâmetros de requisição são propriedades da requisição e não têm qualquer significado em tempo de tradução, o atributo **file** não pode passar quaisquer parâmetros à página incluída. Desta forma o valor do atributo file no seguinte exemplo é inválido:

```
<%@ include file="outro.jsp?abc=pqr" %>
```

- A página incluída pode ou não ser compilada independentemente. Em geral, é melhor evitar estas dependências e usar uma variável implícita **pageContext** para compartilhar objetos entre as páginas incluídas estaticamente pelo uso dos métodos **pageContext.setAttribute()** e **pageContext.getAttribute()**.

Ação <jsp:include>

O elemento **jsp:include** é processado quando uma página JSP é executada. A ação **include** permite que você inclua um recurso estático ou um recurso dinâmico em um arquivo JSP. Os resultados entre a inclusão de recurso estático ou dinâmico são ligeiramente diferentes. Se o recurso é estático seu conteúdo é inserido na invocação do arquivo JSP. Se o recurso é dinâmico a requisição é enviada para o recurso incluído, a página incluída é executada e então o resultado é incluído na resposta da página JSP incluente.

Ação <jsp:param>

Você pode passar parâmetros para componentes incluídos dinamicamente usando a tag **<jsp:param/>**. O seguinte exemplo ilustra o uso da tag **<jsp:param>** para passar dois parâmetros para página incluída:

```
<jsp:include page="pagina.jsp">
<jsp:param name="nome1" value="valor1" />
</jsp:include>
```

Pode haver qualquer número de elementos **<jsp:param>** aninhados com **<jsp:include>** ou **<jsp:forward>**. O valor do atributo **value** também pode ser especificados usando expressões **em tempo de execução** da seguinte maneira:

```
<jsp:include page="pagina.jsp">
<jsp:param name="nome1" value="<%=" nomePessoa %>" />
</jsp:include>
```

Ação <jsp:foward>

A ação **<jsp:forward>** realiza um encaminhamento da requisição para outro recurso web da mesma forma que no Servlet, podemos utilizar **RequestDispatcher.forward(HttpServletRequest, HttpServletResponse)**.

Tratamento de Erros

OPS!

A página que você procura
não pôde ser encontrada.

Possíveis motivos:

- O conteúdo não está mais no ar;
- A página mudou de lugar;
- Você digitou o endereço errado.

HTTP Status 500 - Generate Error

type Status report

message Generate Error

description The server encountered an internal error (Generate Error) that prevented it from fulfilling this request.

Apache Tomcat/6.0.16



Java Web

Permitir que um usuário visualize a pilha de exceção quando ao algo dá errado ou um mensagem de erro padrão do tipo **404 Not Found**, não será um bom cartão de visitas para sua página.

Você poderá elaborar um página personalizada para manipular os erros, e então usar a **diretiva page** para configurá-la. Veja o exemplo:

```
<%@ page isErrorPage="true" %>
<html>
```

Veja bem ! ... Um erro ocorreu, por gentileza tente outra vez.

**
**

</html>

Agora qualquer página JSP poderá usar **erro.jsp** para informar (ou omitir) um erro ao usuário, basta que as páginas que possam lançar exceções informem qual será sua página de erro colocando a diretiva:

```
<%@ page errorPage="erro.jsp" %>
```

Como nada é perfeito teremos um grande inconveniente ao utilizarmos essa abordagem, se você tiver muitas páginas que precisem deste tratamento de erro você terá modificar cada uma para que contenham a **diretiva errorPage**. Tudo bem, você diria, posso fazer isso! Mas se precisar de um tratamento diferente para exceções de tipos diferentes? Então sua abordagem deverá ser mais abrangente, você poderá usar a tag **<error-page>** do **Deployment Descriptor**.

É possível declarar páginas de erro no **web.xml** para uma aplicação web inteira, e você pode até mesmo configurar páginas de erro para diferentes tipos de exceções, ou diferentes tipos de código de erros HTTP.

Declarando uma página de erro geral

Esta declaração se aplica a tudo na sua aplicação web, não apenas para páginas JSPs. As tags internas **<exception-type>** ou **<error-code>** são usados para indicar o tipo da exceção ou código de erro HTTP que será interceptado pelo contêiner. Veja exemplo abaixo.

```
<error-page>
<exception-type>java.lang.Throwable</exception-type>
<location>/erro.jsp</location>
</error-page>
```

Declarando página de erro para uma exceção mais específica

Esta declaração configura uma página de erro que só será invocada quando houver uma **SQLException**. Se você tiver, ao mesmo tempo, uma declaração geral como o exemplo abaixo e ocorrer uma declaração diferente de **SQLException**, ele continuará caindo em **erro.jsp**.

```
<error-page>
<exception-type>java.sql.SQLException</exception-type>
<location>/erroSQL.jsp</location>
</error-page>
```

Página de erro HTTP

Veja como configurar uma página de erro que só é chamada quando o código HTTP de status for **404** (arquivo não encontrado).

```
<error-page>
<error-code>404</error-code>
<location>/erroHTTP.jsp</location>
</error-page>
```

Tratando erros em Servlets

Se é bom observador deve ter notado que um servlet, através dos métodos **service()** ou **doXXX()**, só podem lançar exceções do tipo **ServletException** e **IOException**, ou suas subclasses destas. Então como poderíamos informar para uma página ou outro servlet a ocorrência de exceções como **SQLException** ou outra exceção checável? A resposta é, não poderíamos.

Criando nossa Exception

Se você realmente precisar de uma exceção você deverá criá-la como uma subclasse de **ServletException**. Veja o exemplo a seguir.

```
import javax.servlet.ServletException;
public class MinhaServletException extends ServletException {
    public MinhaServletException(String message) {
        super(message);
    }
}
```

O exemplo abaixo apresenta um servlet que lança a exceção **MinhaServletException**:

```
public class LancaExceptionServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        throw new MinhaServletException("Minha Exceção");
    }
}
```

Servlet que dispara Exception

Mas ainda temos um problema como poderemos repassar informações de exceção que não são subclasses de **ServletException** e **IOException**? A estratégia a ser utilizada neste caso é utilizar a classe **ServletException** como uma classe empacotadora. Você captura a exceção num bloco **try...catch** e “embrulha” a exceção capturada com o construtor **ServletException(java.lang.Throwable rootCause)** e relança-la com **throw**. Veja exemplo abaixo:

```
public class OutraException extends Exception {
    public OutraException(String msg){
        super(msg);
    }
}

public class LancaExceptionServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
        throw new ServletException(
            new OutraException("Um Exception não ServletException"));
    }
}
```

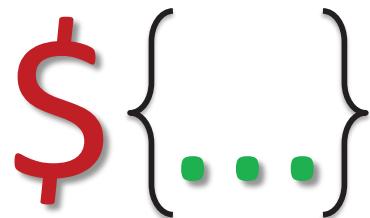
Mapeamento no Deployment Descriptor

Da mesma forma que fizemos com JSPs, também podemos configurar nossa exceção gerada pelo servlet no **web.xml**, de modo declarativo. Veja abaixo.

```
<error-page>
    <exception-type>OutraException</exception-type>
    <location>/erro.jsp</location>
</error-page>
```

```
<%@ page isErrorPage="true"%>
<h1>Erro Encontrado</h1>
O seguinte erro foi encontrado:
<br>
<b><%=exception.toString() %></b>
<br> <% exception.printStackTrace(); %>
```

Java Web Expression Language | JSTL



```
<c:if test="true or false">  
    ...  
</c:if>
```

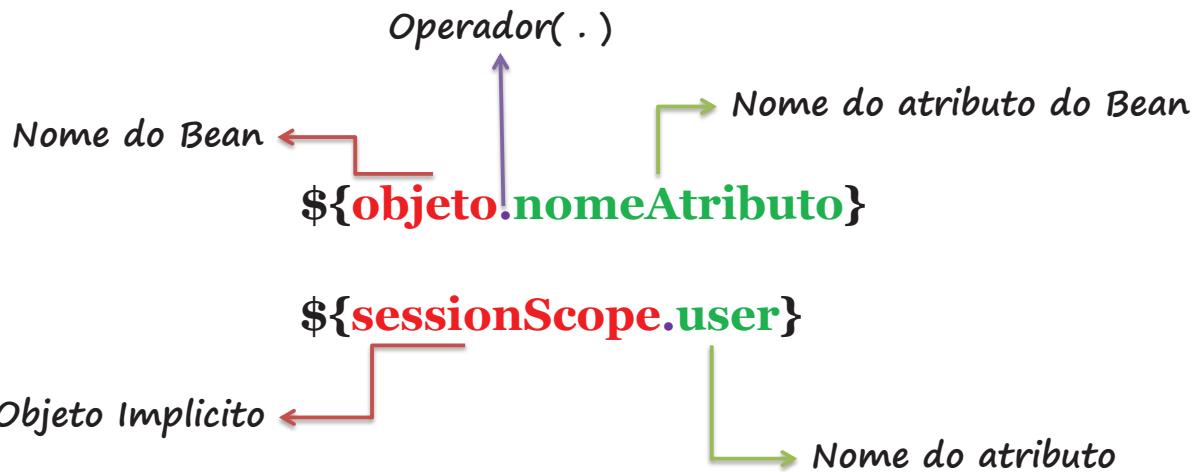


Java Web

Expression Language (Linguagem de Expressão) foi introduzido na versão 1.0 da JSTL (JSP Standard Tag Library), antes da JSTL scriplets eram usados para manipular dados nas aplicações. JSTL introduziu o conceito de Linguagem de Expressão – EL, no qual simplifica o desenvolvimento de uma página provendo recursos através de tags. As tags existentes provê recursos de tarefas estruturados como iterações e condições, processamento de arquivos XML, internacionalização e acesso a dados usando SQL.

O EL simplifica bastante operações e exibição de dados, onde com Scriplets exigiria mais código e manutenção mais difícil.

Expression Language



Java Web

O que é EL?

Expression Language (EL) é uma linguagem de script que permite o acesso a atributos/propriedades e propriedades aninhadas java a partir de páginas JSP (a partir da versão 2.0) sem a necessidade de inserir código java nas mesmas.

A utilização de EL torna fácil o trabalho de Web Designers que tem pouco ou nenhum conhecimento em Java, com ela é possível simplificar consideravelmente a camada de apresentação.

A sintaxe de EL é `${expressão}`, onde expressão é o componente/atributo a ser resgatado do Servlet/JavaBean.

Objetos implícitos

Em EL, os **objetos implícitos** que estão disponíveis na JSP surgem através de expressões com recurso a:

Nome	Descrição
param	Um Map contendo os parâmetros (String) da requisição.
paramValues	Um Map contendo os parâmetros (String[]) da requisição
header	Um Map contendo os cabeçalhos (String) da requisição.

Nome	Descrição
headerValues	Um Map contendo os cabeçalhos (String[]) da requisição
cookie	Um Map contendo os campos de um Cookie como um objeto simples
initParam	Um Map contendo os parâmetros de inicialização do contexto
pageScope	Um Map contendo os atributos do escopo da página (page)
requestScope	Um Map contendo os atributos do escopo da requisição (request)
sessionScope	Um Map contendo os atributos do escopo da sessão (session)
applicationScope	Um Map contendo os atributos do escopo do contexto (application)

Veja os seguintes exemplos:

```
<%-- imprime variaveis implicitas --%>
${param.foo} => ${param.foo}
${paramValues.tecno[0]} => ${paramValues.tecno[0]}
${header["host"]} => ${header["host"]}
${header["user-agent"]} => ${header["user-agent"]}
```

Sintaxe EL

Expressão EL	Resultado
<code> \${1 > (4/2)}</code>	false
<code> \${4.0 >= 3}</code>	true
<code> \${100.0 == 100}</code>	true
<code> \${((10*10) ne 100)}</code>	false
<code> \${'a' < 'b'}</code>	true
<code> \${'hip' gt 'hit'}</code>	false
<code> \${4 > 3}</code>	true
<code> \${1.2E4 + 1.4}</code>	12001.4
<code> \${3 div 4}</code>	0.75
<code> \${10 mod 4}</code>	2
<code> \${!empty param.Add}</code>	True se o parâmetro Add for null ou uma String vazia



Java Web

A melhor maneira de compreendermos as expressões EL é compará-las com expressões em scripts JSP. Por exemplo, se você quiser imprimir o valor de uma variável **temperatura** na sua página você o faria utilizando expressão JSP como se segue:

Temperatura externa é de <%= temperatura%> graus.

ou, você pode usar EL

Temperatura externa é de \${temperatura} graus.

Literais

A EL define os seguintes literais:

- **Boolean:** true e false
- **Integer:** como em Java
- **String:** com aspas simples ou duplas; aspas duplas (") é escapada com \" e aspas simples (') é escapada com \' e \\' é escapada com \\
- **Null:** null

Se você precisar utilizar uma expressão literal que inclua os caracteres \${}, reservados pela sintaxe EL, você precisa escapar estes caracteres como segue:

- Criando um expressão composta como em: \${"\${exprA}"}
- Ou o caractere de escape \ pode ser usado para evitar a avaliação de uma expressão: \\$\${exprA}

Navegação em variáveis

Para usar variáveis com EL você pode usar operadores combinados. Usamos operadores para acessar coleções ou propriedades. Os operadores de acesso a propriedades permitem acesso aos membros de objetos, enquanto os operadores de coleções retornam elementos de **Map**, **List** ou **Array**.

1) Operador ponto (.)

Este operador possui algumas restrições de uso. O literal à esquerda do **ponto (.)**, deve ser um **Map** ou um JavaBean. O segundo literal, à direita de **ponto (.)**, deve ser uma propriedade do bean ou uma chave do **Map** e deve obedecer aos padrão de nomenclatura de variáveis em Java (não iniciar com número, etc...). Veja exemplo:

```
 ${pessoa.nome} //atributo do bean pessoa invoca getName()
 ${map.chave} //chave do map
 ${pessoa.1} // não funciona pois não obedece o padrão de nomenclatura Java
```

2) operador []

Este operador é mais poderoso e flexível. O literal à esquerda de **[]** também pode ser um **List** ou um array de qualquer tipo. A variável à esquerda pode ser um número ou ainda qualquer valor que não respeite as regras de nomenclatura do Java. Veja exemplos do operador **[]**:

```
 ${pessoa["nome"]} mesmo que ${pessoa.nome}
 ${map["chave"]} mesmo que ${map.chave}
 ${map["br.com.servlet"]} // agora funciona
 ${minhaLista["1"]} // agora funciona, porém...
```

Navagação em Arrays e Listas

Quando a variável for um array ou lista, tudo que se coloca dentro do **[]** é convertido para um inteiro, ou seja, a chave ou índice deve ser um literal numérico, exemplo:

```
 ${minhaLista["0"]}
 ${minhaLista['0']}
 ${minhaLista[0]}
```

Lista de Operadores

Operador	Descrição
.	Acessar a propriedade de um bean
[]	Acessar um array ou um elemento do array
? :	Teste condicional: condição ? ifTrue : ifFalse.
+	Adição.
-	Subtração.
*	Multiplicação.
/ ou div	Divisão.
% ou mod	M Modulo (resto).
== ou eq	Teste de igualdade.
!= ou ne	Diferença.
< ou lt	Menor que.
> ou gt	Maior que.
<= ou le	Menor ou igual que.
>= ou ge	Maior ou igual que.
&& ou and	E lógico.
ou or	OU lógico.
empty	Teste para String, array e Collection se valor null ou vazio.

Habilitando e Desabilitando EL e scriptlets

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>false</el-ignored>
  </jsp-property-group>
</jsp-config>
```

|||

```
<%@ page isELIgnored="true" %>
```



Java Web

O padrão `${ }` não é símbolo reservado na especificação anterior a JSP 2.0, portanto pode haver aplicações em que este símbolo possa utilizados com outro intuito. A fim de prevenir a avaliação destes padrões dentro de uma página você tem três opções:

1) Escapar o caracteres `${ }` :

`\${expr}`

2) Configurar sua aplicação (`web.xml`) com o grupo de propriedade JSP:

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>false</el-ignored>
  </jsp-property-group>
</jsp-config>
```

3) Configurar a página com diretiva `page`:

`<%@ page isELIgnored="true" %>`

JSTL - JavaServer Pages Standard Tag Library

Pacote	Sugestão de prefixo	Descrição
JSTL Core	c	Tags relacionadas à lógica e controle
JSTL fmt	fmt	Tags para formatação e internacionalização de dados
JSTL sql	sql	Tags para relaizar operações em banco de dados
JSTL xml	xml	Tags para manipulação de arquivos XML
JSTL functions	fn	Conjunto de funções para processamento de objetos Strings e coleções



Java Web

JSTL significa JSP Standard Tag Library e consiste essencialmente, num conjunto de tags, que oferecem controle sobre o processamento das páginas sem aumento de complexidade.

Permitem substituir os **scriptlets** e assim estimular a separação entre apresentação e lógica, resultando num investimento significativo no sentido de conseguir seguir o modelo MVC.

Instalação da JSTL

Para instalar a JSTL você precisa:

- Fazer o download da última versão do site <https://jstl.java.net/download.html>.
- Copiar os JARs das bibliotecas desejadas para o diretório **WEB-INF/lib/** da sua aplicação Web.

Como usar JSTL em uma página JSP

O prefixo definido para utilização da tag pode ter qualquer valor mas no caso da **taglib core da jstl** a convenção é o padrão da letra **c**. Já a URI (que não deve ser decorada) é mostrada a seguir e não implica em uma requisição pelo protocolo http, mas sim um nome a ser utilizado numa busca entre os arquivos descritores de tags (TDL). A URI está para a tag, nos arquivos tlds, assim como **<servlet-name>** está para um servlet, no arquivo web.xml.

Existem cinco bibliotecas de tags JSTL (versão 1.1), para incluir uma taglib em sua aplicação proceda como:

Core library: tags para condicionais, iterações, urls, ...:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" />
```

Exemplo:

```
<c:if test="..." ... >...</c:if>
```

XML library: tags para processamento XML:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" />
```

Exemplo:

```
<x:parse>...</x:parse>
```

Internationalization library I18N:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" />
```

Exemplo:

```
<fmt:message key="..." />
```

SQL library: manipular banco de dados

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" />
```

Exemplo:

```
<sql:update>...</sql:update>
```

Functions: Manipulação de string e coleções

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" />
```

Exemplo:

```
<sql:update>...</sql:update>
```

Biblioteca Core

Tag	Descrição
c:catch	- bloco do tipo try/catch
c:choose	- bloco do tipo switch
c:forEach	- um for para iterar sobre coleções
c:forTokens	- for em tokens (ex: "a,b,c" separados por vírgula)
c:if	- if
c:import	- import
c:otherwise	- default do switch
c:out	- saída
c:param	- parâmetro
c:redirect	- redirecionamento
c:remove	- remoção de variável
c:set	- criação de variável
c:url	- veja adiante
c:when	- teste para o switch



Java Web

Suporte a variáveis - c:set e c:remove

O conjunto de tags que permite configurar valores de variáveis EL ou propriedades de uma variável EL em qualquer scope JSP (**page, request, session ou application**). Se a variável não existir ela é criada.

A variável EL ou propriedade pode receber valor através do atributo **value**:

```
<c:set var="num" scope="session" value="${4*5}" />
```

Para configurar propriedades de um **JavaBean** ou um **java.util.Map**, mas só isso, você não pode usá-lo para adicionar elementos em uma lista ou em arrays. Você deve usar o **atributo target** ao invés de **var**, quando estiver configurando um **JavaBean** ou um **Map** veja como:

```
<c:set target="${carrinho}" property="items" value="${lista}" />
```

A tag **<c:remove>** é usada para remover uma variável de seu escopo. O **atributo var** tem que ser um literal String não pode ser uma expressão. Veja um simples exemplo:

```
<c:remove var="num" scope="session" />
```

Controle de Fluxo

- **Condisional** `<c:if>`, `<c:choose>`

Você pode usar a tag `<c:if>` para construir expressões condicionais simples. Por exemplo:

```
<c:if test="#${condicional}">
```

Se a condicional for verdadeiro executa o trecho de código dentro da tag

```
</c:if>
```

A tag `<c:if>` tem uma condição e um “bloco” de código (qualquer coisa aceitável em JSP). Caso a condição da tag seja satisfeita o “bloco” de código é executado.

A tag `<c:choose>` e suas parceiras `<c:when>` e `<c:otherwise>`, funcionam como uma estrutura `switch` do java.

- **Iteradores** `<c:forEach>` e `<c:forTokens>`

A tag `<c:forEach>` é capaz de iterar por uma coleção. O exemplo a seguir mostra o uso de **expression language** de uma maneira mais limpa que o **script JSP**.

Tag `<c:forEach>`:

```
<c:forEach var="contato" items="#${lstContatos.lista}">
    <li>${contato.nome}, ${contato.email}: ${contato.endereco}</li>
</c:forEach>
```

Também é possível criar um contador do tipo int no corpo da tag `<c:forEach>`. Para isso basta definir o atributo chamado **varStatus** para a variável desejada e utilizar a propriedade **count** dessa variável.

```
<c:forEach var="contato" items="#${dao.lista}" varStatus="id">
    <li>${id.count} é ${contato.nome}</li>
</c:forEach>
```

Na tag `<c:forTokens>` o atributo items é uma String constituída por tokens separados por algum delimitador. Se você imaginar uma String como um coleção de substrings é possível notar a semelhança com `<c:forEach>`.

```
<c:set var="nomes" value="A:B:C|D" scope="page" />
<c:forTokens items="#${pageScope.nomes}"
    delims=":|"
    var="nomeAtual"
    varStatus="status" >
    Membro família #<c:out value="#${status.count}" /> is
        <c:out value="#${nomeAtual}" /> <br>
</c:forTokens>
```

Biblioteca de Internacionalização



messages_pt_BR.properties

Label.ok=Ok
Label.save=Salvar

messages_en_US.properties

Label.ok=Ok
Label.save=Save



Java Web

Internacionalizar aplicações é cada vez mais uma tarefa corriqueira de todo desenvolvedor web. A maioria dos frameworks web tem a sua maneira particular de prover esse mecanismo.

Locale

A tag `<fmt:setLocale>` é utilizada para fazer com que o sistema passe a ser exibido em uma língua diferente da que está previamente definida pelo browser cliente.

`<fmt:setLocale value="${param.lingua}" scope="session"/>`

Você também poderia utilizar a classe `javax.servlet.jsp.jstl.core.Config` em um **Servlet**. Esta classe permite controlar as configurações da **JSTL** programaticamente, deixando transparente o controle da localização para suas páginas JSP.

O seguinte trecho de código mostra como a tag `<fmt:setLocale>` é utilizada para especificar explicitamente a configuração de localização na sessão do usuário:

`<fmt:setLocale value="pt_BR" scope="session"/>`

Após a execução deste fragmento JSP as preferências de idioma especificadas pelo usuário para seu browser serão ignoradas.

Mensagens

Textos de localização de idiomas são utilizadas na JSTL com uso da tag `<fmt:message>`. Esta tag permitirá a você retornar mensagens e textos de um arquivo de recurso (**resource bundle**) e mostrá-lo em sua página.

Um arquivo **resource bundle** é um arquivo contendo todas as mensagens (e rótulos) a serem utilizados pelo sistema. O exemplo mostra um arquivo **resource bundle**, **messages.properties**:

```
site.titulo = Sistema com i18n
saudacao = Bem vindo ao sistema
campo.nome = Nome:
campo.email = Email:
campo.rua = Rua:
campo.cidade = Cidade:
botao.enviar = Enviar
botao.cancelar = Cancelar
erro.campo.obrigatorio = Por favor, preencha o campo
```

Para usar este arquivo contendo as mensagens do seu sistema você referenciá-lo em sua página para que a tag `<fmt:message>` saiba onde encontrar os recursos. Isto pode ser feito de duas formas: usando a tag `<fmt:setBundle>` ou `<fmt:bundle>`:

```
<fmt:bundle basename="br.com.empresa.pacote.messages">
```

Ou adicionando a seguinte configuração ao seu **web.xml**:

```
<web-app ...>
  <context-param>
    <param-name>
      javax.servlet.jsp.jstl.fmt.localizationContext
    </param-name>
    <param-value>messages</param-value>
  </context-param>
</web-app>
```

A segunda opção é mais flexível e também mais desejável, já que não precisamos alterar as páginas JSP caso venhamos mudar localização do resource. Este arquivo de mensagens deve estar no *classpath* da sua aplicação web e deve possuir a extensão **.properties**. Existem várias formas de se fazer isso. A mais simples, é criar o arquivo **messages.properties** no diretório onde estão os fontes (*.java) da sua aplicação.

Caso queira deixá-lo dentro de algum pacote, a configuração no **web.xml** deverá conter o nome completo do arquivo:

```
<param-value>br.com.empresa.pacote.messages</param-value>
```

Para usar essas mensagens nas suas páginas JSP você basta usar a tag **<fmt:message>** da JSTL,

```
<fmt:message key="chave"/>
```

A tag **<fmt:message>** sempre procura o arquivo de mensagens mais adequado para o **Locale** associado ao usuário. Os navegadores enviam no cabeçalho das requisições com informações sobre os idiomas configurados pelo usuário em seu browser. Experimente mudar essas configurações no seu navegador e veja que o sistema passa a ser exibido em idiomas diferentes.

Se o Locale associado usuário for **pt_BR** (português do Brasil), a tag **fmt:message** irá tentar buscar as mensagens nos seguintes arquivos (contendo o texto para o locale desejado), **em ordem**:

```
messages_pt_BR.properties  
messages_pt.properties  
messages.properties
```

O primeiro a ser encontrado será usado. Portanto, a boa prática é ter o arquivo **messages.properties** com a língua padrão do sistema e um arquivo específico para cada língua adicional.

Formatação de Números e Datas

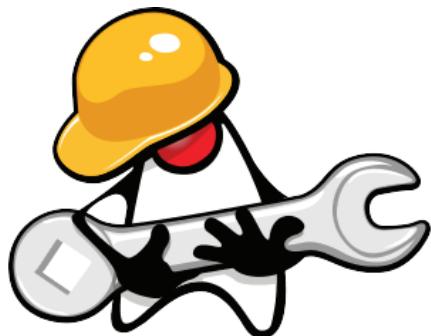
A biblioteca **fmt** inclui tags para manipular Data e Números: **<fmt:formatDate>**, **<fmt:parseDate>**, **<fmt:formatNumber>**. Como o próprio nome sugere, **<fmt:formatDate>** faz formatação de datas e mostra datas e horas (**saída dados**), enquanto **<fmt:parseDate>** é usada para fazer análise de valores de datas e horas (**entrada de dados**). Veja exemplo:

```
<c:set var="brDateString">4/1/03 19:03</c:set>  
<fmt:parseDate value="${brDateString}" parseLocale="pt_BR"  
    type="both" dateStyle="short" timeStyle="short" var="brDate"/><br>  
<ul>  
    <li> Analise <c:out value="${brDateString}"/> dado localização  
        Pt Brasil  
        resultando na data <c:out value="${brDate}"/>.</li>  
</ul>
```

A tag **<fmt:formatNumber>** é usada para mostrar dados numéricos, incluindo valores monetários e percentuais, de acordo com uma localização específica.

```
<fmt:TimeZone value="America/Sao_Paulo ">  
    <fmt:formatNumber value="0.02" type="currency"/>  
</fmt:TimeZone>
```

Java Web Custom Tags



<x:y></x:y>



Java Web

Tendo visto o uso e a facilidade de se programar páginas utilizando **JSTL** você provavelmente desejará utilizar a mesma estratégia para um conjunto de funcionalidades, específicas de sua implementação, que não tenham sido contempladas na **JSTL**.

Escrever novos manipuladores de tags é uma tarefa relativamente simples de ser feito com **JSP/Servlets**. Existem três maneiras de se criar seus próprios manipuladores de tags. Dessas três, duas foram introduzidas com **JSP 2.0** para tornar a tarefa de desenvolver tags mais simples (**Simple Tags e Tag Files**).

Tag



```
<%@ taglib uri="WEB-INF/custom.tld" prefix="prefixoQualquer" %>
```

```
<prefixoQualquer:minhaTag></prefixoQualquer:minhaTag>
```



Java Web

Construir tags JSP é algo tão simples quanto desenvolver uma classe Java normal ou um Servlet. Para usar JSP Custom Tags, você precisa definir separadamente três componentes:

- A classe tag handler (manipulador da tag) que define o comportamento da tag;
- O tag library descriptor (descritor da biblioteca de tag) que define um arquivo XML contendo elementos que são mapeados para a implementação da tag;
- Um arquivo JSP que usa a tag library.

A classe Tag Handler

Quando definimos uma nova tag, sua primeira tarefa é definir uma classe Java que diz ao sistema o que fazer quando encontrar a declaração da tag. Esta classe deve implementar a interface **javax.servlet.jsp.tagext.Tag**. Você normalmente conseguirá isto, fazendo herança da classe **TagSupport** ou **BodyTagSupport**. Logo abaixo é mostrado um exemplo de uma tag simples que insere o texto **Exemplo Custom Tag** em uma página JSP.

```
public class TagExemplo extends TagSupport {
    public int doStartTag() throws JspTagException {
        try {
            pageContext.getOut().write("Exemplo Custom tag.<br/>");
        } catch (IOException ex) {
```

```

        System.out.println("Erro em TagExemplo: " + ex);
    }
    return EVAL_PAGE;
}
public int doEndTag() throws JspTagException {
    return EVAL_BODY_INCLUDE;
}
}

```

Arquivo Descritor de Tag Library

Uma vez definido o **tag handler**, sua próxima tarefa é identificar a classe para o servidor e associando-a com uma tag XML, apropriada, do descritor de tags. Logo abaixo é mostrado um arquivo (em formato XML com extensão .tld), ele contém algumas informações fixas como: nome da sua biblioteca, uma descrição resumida e um série de tags de descrição. A parte não negritada da listagem é comum, e virtual igual, para todos os arquivos descritores de tags, podendo ser copiado para novos arquivos descritores de forma literal.

Note que o elemento **tag** define o nome principal da **tag** e identifica a classe que manipula a **tag**. O nome completamente qualificado da classe deve ser usado. A classe deve ser colocada em qualquer lugar que seu servidor aceite como local de instalação de **beans** ou qualquer outra classe de suporte.

```

<?xml version="1.0" encoding="UTF-8" ?>
<taglib>

<tlibversion>1.0</tlibversion>
<jspversion>2.0</jspversion>
<shortname>exemplos</shortname>

<tag>
    <name>minhaTag</name>
    <tagclass>tags.TagExemplo</tagclass>
    <bodycontent>JSP</bodycontent>
</tag>

</taglib>

```

O Arquivo JSP

Depois de ter implementado o manipulador da **tag** e registrado este manipulador no descriptor, você agora está pronto para utilizar sua tag customizada no em uma página JSP. Para utilizar a **tag** numa página JSP, precisamos adicionar a diretiva:

```
<%@ taglib uri="WEB-INF/custom.tld" prefix="prefixoQualquer" %>
```

A diretiva requer um atributo **uri** que pode ser absoluto ou relativo à URL. Por enquanto, nós iremos utilizar uma URL relativa, bem simples, que corresponde a uma descriptor de arquivo **TLD** colocado no mesmo diretório da página JSP que faz uso da **tag**.

O atributo **prefix**, também obrigatório, especifica um prefixo que será usado como nome da **tag** definida no descriptor de tags. Por exemplo, se o arquivo **TLD** defini uma **tag** nomeado como **tag1** e o atributo **prefix** tem como valor o literal **teste**, então o nome corrente da tag será **teste:tag1**. Esta tag será usada, dependendo se ela está definida como uma tag que aceita um corpo, de duas das seguintes maneiras:

```
<prefixoQualquer:minhaTag></prefixoQualquer:minhaTag>
```

ou apenas

```
<prefixoQualquer:minhaTag/>
```

Tag File

testaTagsCriadas.jsp

/WEB-INF



```

<%@ taglib prefix="ex" tagdir="/WEB-INF/tags" %>
<html>
<body>
  Os primeiros seis números da sequencia de Fibonacci são:
  <ex:exemplo/>
</body>
</html>
  
```

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>

<c:forTokens items="0;1;1;2;3;5" delims=";" var="fibNum">
  <c:out value="${fibNum}" />
</c:forTokens>
  
```



Java Web

Uma Tag File nada mais é do que um pequeno código de arquivo JSP. A sua função é justamente criar pequenos fragmentos de códigos para um determinado fim. Ela ajuda na organização e design da sua aplicação, retirando uma fatia de códigos comuns e repetitivos das suas páginas JSP. Suporta EL, JSTL e qualquer tag libs.

Com **Tag Files** você pode invocar conteúdo reutilizável usando um tag customizada, em vez dos genéricos **<jsp:include>** ou **<c:import>**. Numa comparação mais grosseira podemos ver as **Tag Files** como **includes** mais poderosos.

Tão simples quanto parece, um tag file é um arquivo contendo código JSP que tem como extensão **.tag** ou **.tagx**. De fato, o único elemento JSP que não pode ser usado num **tag file** são as diretivas **page**.

Pode parecer trivial, mas este novo recurso é importante. A chave é a simplicidade. Você não precisa, necessariamente, de arquivos descritores de tag. A nova especificação (JSP 2.0), tornou simples a integração de **tag file** com um JSP. Você só precisa seguir dois passo: adicionar a diretiva **taglib** ao JSP com um atributo **prefix** e o atributo **tagdir** apontando para **/WEB-INF/tags**.

Coloque a tag contendo o prefixo e o nome do **tag file** (sem a extensão) em qualquer lugar em que o JSP necessite usar a tag.

Este processo levanta uma questão importante. Se uma tag clássica necessita de **TLDs** para localizar suas classes, como estas **tag files** são localizadas? Para responder isto, nós precisamos saber como o contêiner acessa e processa as **tag files**.

Tag Files empacotados

Para ficar mais fácil e rápido, no exemplo acima, nos colocamos a tag file no diretório **WEB-INF/tags**, com isso não foi preciso gerar um **jar** para **tag**. Isto foi necessário porque o contêiner verifica automaticamente pelos **tag files** dentro deste diretório.

Mas se você quiser colocar suas tags dentro de um **JAR** a situação muda. Neste caso você precisará criar um arquivo **TLD**. Este descritor de tag é similar aos **TLDs** das tags clássicas, mas ao invés de associar tags a uma classe de manipulador de tag, ele irá associar um nome a um **tag file** e seu caminho.

Para tornar isto possível, o **TLD do tag file** usa um elemento **<tag-file>** no lugar de **<tag>**. Os únicos subelementos necessários são **<name>**, que especifica o nome do arquivo sem a extensão e **<path>**, que especifica o caminho para o arquivo de **tag**. Entretanto, **<path>** deve começar com **/META-INF/tags**. Aqui temos um exemplo de **TLD** para um **tag file** empacotado.

```
<taglib>
  <uri>http://minha.empresacom/tagfile/exemplo</uri>
  <tag-file>
    <name>exemplo</name>
    <path>/META-INF/tags/exemplo.tag</path>
  </tag-file>
</taglib>
```

Este arquivo deve ser colocado dentro do diretório **META-INF** e o **tag file(s)** deverão estar em **META-INF/tags** ou em um subdiretório. Um exemplo da estrutura de diretório segue abaixo:

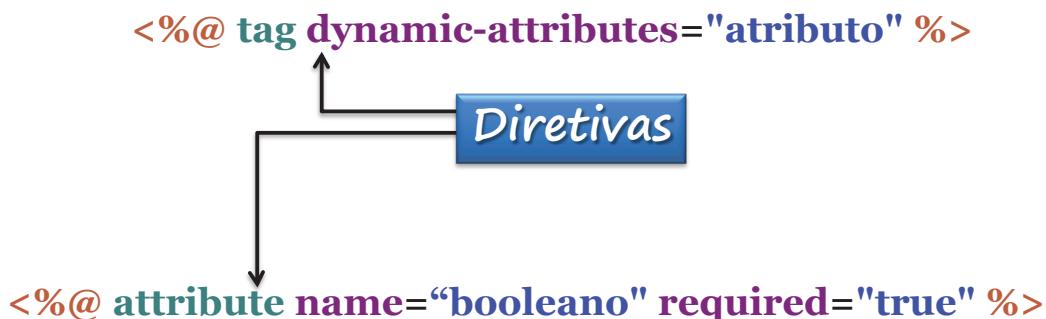
```
META-INF/
  exemplo.tld
  tags/
    exemplo.tag
```

Uma vez que o tag file não se encontra dentro ou abaixo de **WEB-INF/tags**, você não poderá usar o atributo **tagdir** na diretiva **taglib**. A invés disto, você precisa especificar a URI definida no seu TLD (**http://minha.empresacom/tagfile/exemplo**) usando o atributo **uri**. Para nosso exemplo, a diretiva **taglib** deverá ser usada com segue:

```
<%@ taglib prefix="ex" uri="http://minha.empresacom/tagfile/exemplo" %>
```

Outra diferença importante entre TLDs de tag file e TLDs de tags clássicas, são relativos aos elementos **<attribute>** e **<body-content>**. As tags clássicas podem conter esta informação dentro de seus TLDs, tag files não. Os tag files usam um conjunto de diretivas especiais. Elas dizem ao contêiner como processar o tag file, é importante entender como elas funcionam.

Diretivas (tag, attribute)



Java Web

JSP contém três tipos diferentes de diretivas: page, taglib e include. Tag files possui variable, cria e inicializa uma variável usada no processamento da tag. A tag, diz ao web container como processar o tag file. A attribute, descreve os atributos que podem ser usados na tag. Vejamos as duas últimas.

Tag

A nova diretiva, **tag**, funciona como a diretiva page em um JSP. Ela fornece ao web container um conjunto de configuração a ser usado por todo o arquivo de tag.

Um atributo que não paralelo no JSP é **dynamic-attributes**. Ele age como o sublemento **<dynamic-attributes>** do TLD, mas ao invés de informar atributos para um método em Java, o web container atualiza as variáveis locais especificadas pela diretiva.

Por exemplo, o seguinte **tag file** usa a diretiva **tag** para enviar dados ao atributo dinâmico. Estes dados são mostrados com tag de ação **forEach da JSTL**.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ tag dynamic-attributes="atributo" %>
<c:forEach items="${atributo}" var="localVar">
  ${localVar.value}<br>
</c:forEach>
dinatributo.tag
```

O código JSP, no exemplo acima mostra como usar a tag file (dinatributo.tag) e como configurar valores para os atributos da tag. Quando o JSP for invocado ele mostrará uma lista de valores.

```
<%@ taglib prefix="dyn" tagdir="/WEB-INF/tags" %>
<html><body>
    <dyn:dinatributo atr1="primeiro" atr2="segundo" atr3="terceiro"/>
    <dyn:dinatributo x="a" y="b" z="d"/>
</body></html>
atributoDinamico.jsp
```

Attribute

Atributos dinâmicos são bastante flexíveis, mas se você souber quais atributos sua tag irá precisar você poderá informá-las ao web contêiner usando atributos estáticos. As tags tradicionais tem o subelemento **<attribute>** no TLD para tal propósito. Mas para configurar um conjunto de atributos na tag file nos precisamos da diretiva **attribute**.

O atributo **name** fornece uma identificação **required** e informa ao contêiner se atributo é obrigatório e **rtpexprvalue** informa ao contêiner que o valor do atributo pode ser obtido em tempo de execução.

Vejamos um exemplo simples para demonstrar o uso destas diretivas em um tag file.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ attribute name="booleano" required="true" %>

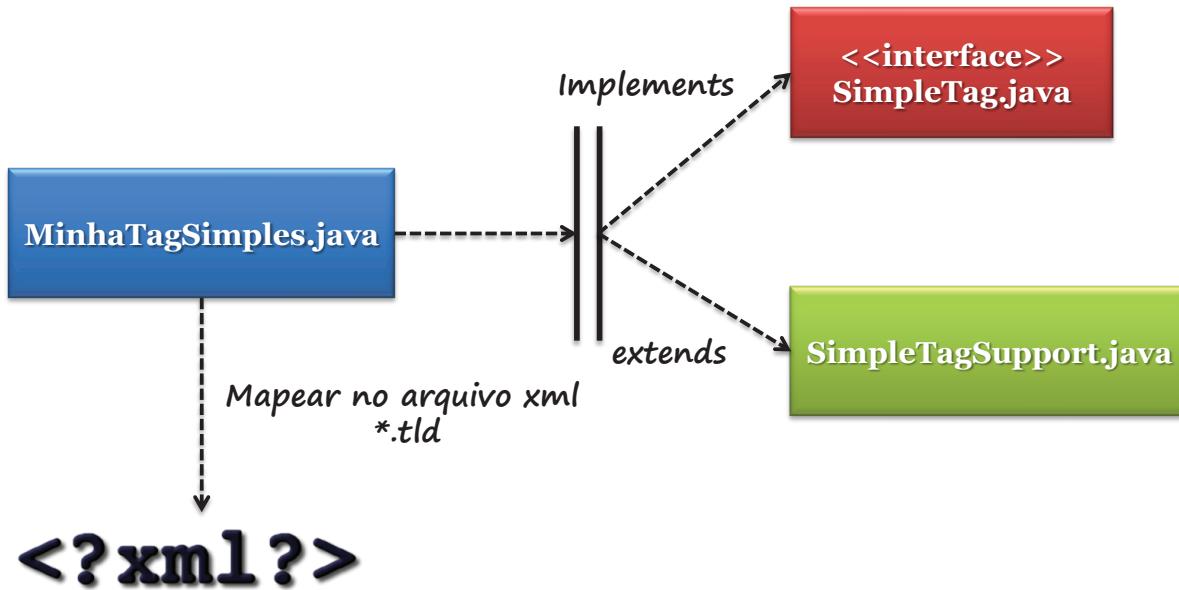
<c:if test="\$\{booleano == 'sim'\}">
    Entrou na condição
</c:if>
estatributo.tag
```

Então, o seguinte código JSP usa a tag file e configura o atributo:

```
<%@ taglib prefix="atr" tagdir="/WEB-INF/tags" %>
<html>
<body>
    <atr:estAtributo booleano="sim" />
</body>
</html>
estatributo.jsp
```

A diretiva **attribute** também permite que você insira code JSP dentro de atributos estáticos simplesmente configurando o atributo **fragment** para **true**. Entretanto, para processar esse fragmento, você precisará de recursos extras, fora do escopo das diretivas de **tag file**, ou seja, você vai precisar de novas ações padrões.

Simple Tag



Java Web

O JSP 2.0 introduziu uma série de novos mecanismos com o intuito de facilitar a vida do desenvolvedor. Com **Simple Tags**, podem ser criados todos tipos de tags com mais facilidade: desde tags com corpo vazio, passando por tags que, dependendo de uma determinada condição, incluem ou não seu corpo, e até tags que processam o corpo repetidas vezes.

Toda a lógica de processamento de um **Simple Tag** fica em um único método, **doTag()**, em uma classe que implementa a interface `javax.servlet.jsp.tagext.SimpleTag`.

Quando um JSP invoca uma tag, uma nova instância da classe handler é criada, dois ou mais métodos são chamados para o handler e quando o método **doTag()** é finalizado, o objeto handle desaparece, em outras palavras, estes objetos handler não são reutilizados pelo web contêiner.

Uma Tag handler **Simple** deve implementar a interface **SimpleTag** ou estender **SimpleTagSupport** e substituir apenas o método de que você precisa, **doTag()**. A **SimpleTagSupport** implementa os métodos de **SimpleTag**, mas o **doTag()** não faz nada, você precisa substituí-lo em seu tag handler.

Métodos da interface SimpleTag:

Nome	Descrição
setJspContext()	Torna o JspContext disponível para o processamento na tag
setParent()	Chamado pelo web contêiner para tornar a tag pai disponível
setJspBody()	Permite o processamento do corpo da tag
doTag()	Chamado pelo web contêiner no começar da operação da SimpleTag
getParent()	Invocado por classes Java para obter seu JspTag pai

Criando SimpleTags

O primeiro passo é escrever uma classe que estenda **SimpleTagSupport**:

Depois de criar a classe é necessário reimplementar o método **doTag()** oriundo da superclasse **SimpleTagSupport**, que por sua vez herdou esse método da interface **SimpleTag**.

```
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class SimpleTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().print("Simple Tag");
    }
}
SimpleTag1.java
```

Note que o método **doTag()** declara uma **IOException**, então não é preciso tratar a exceção com um bloco **try/catch**.

Mapeamento de tlds em páginas Jsp.

Crie um **TLD** para a tag.

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
         version="2.0">
    <description>Minha biblioteca SimpleTag</description>
    <tlib-version>1.0</tlib-version>
    <short-name>BibliotecaSimpleTag</short-name>
    <uri>simpleTags</uri>
    <tag>
        <description>Minha Tag</description>
        <name>minhaTag</name>
        <tag-class>SimpleTag1</tag-class>
        <body-content>empty</body-content>
    </tag>
</taglib>
simpleTag.tld
```

Coloque a TLD dentro de **WEB-INF**.

```
<%@ taglib prefix="tag" uri="simpleTags" %>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html">
    <title>Testando Simple Tags</title>
</head>
<body>
    <tag:minhaTag/>
</body>
</html>
SimpleTag1.jsp
```

Corpo da Tag

Se a tag precisar de um corpo, o **<body-content>** do **TLD** precisa refletir isso e é necessário adicionar um comando especial no método **doTag()**.

```
<%@ taglib prefix="tag" uri="simpleTags" %>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; ">
    <title> Testando Simple Tags com corpo</title>
</head>
<body>
    <tag:minhaTag> Isto é um corpo </tag:minhaTag>
</body>
</html>
Tag com corpo
```

Para que o JSP acima funcione, é necessário modificar o método **doTag()** da classe **SimpleTag1** conforme abaixo:

```
public void doTag() throws JspException, IOException {
    getJspBody().invoke(null);
    getJspContext().getOut().print("<br>Simple Tag Handler");
}
doTag() com suporte a um corpo
```

Você deverá informar essa nova opção em seu arquivo **TLD**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd">
```

```

version="2.0">

<description>Minha biblioteca SimpleTag</description>
<tlib-version>1.0</tlib-version>
<short-name>BibliotecaSimpleTag</short-name>
<uri>simpleTags</uri>
<tag>
    <description>Minha Tag</description>
        <name>minhaTag</name>
    <tag-class>SimpleTag1</tag-class>
    <body-content>scriptless</body-content>
</tag>
</taglib>
Reconfigurando simpleTags.tld

```

Tópicos Avançados: definindo atributos.

Imagine que você tenha uma tag com um corpo que usa uma expressão EL como atributo. Agora imagine que o atributo não exista no momento em que você invoca a tag! Em outras palavras, o corpo da tag depende que a tag handler defina o atributo.

```
<tag:minhaTag> A mensagem é: ${mensagem} </tag:minhaTag>
```

No ponto em que a tag é invocada, “mensagem” não é um atributo dentro do escopo! Se você tirasse esta expressão da tag, ela retornaria nula. Para definir o atributo mensagem usamos o método **setAttribute**:

```

public void doTag() throws JspException, IOException {
    getJspContext().setAttribute("mensagem", "Uma mensagem");
    getJspBody().invoke(null);
}

```

É possível configurar um atributo referenciando uma coleção, no exemplo abaixo a expressão EL no corpo da tag representa um único valor em uma coleção e o objetivo é fazer a tag gerar uma linha para cada elemento da coleção. Simplesmente o método **doTag()** só precisa fazer o trabalho em um loop, invocando o corpo em cada iteração do loop.

```

<table>
<tag:minhaTag>
    <tr><td>${mensagem}</td></tr>
</tag:minhaTag>
</table>

```

O método **doTag()** da tag handler:

```
String[] mensagem = {"oi!", "tudo bem?", "Bom dia!", "Boa noite!"};

public void doTag() throws JspException, IOException {

    for(String msg : mensagem){

        getJspContext().setAttribute("mensagem", msg);

        getJspBody().invoke(null);
    }
}
```

Cada loop da tag handler reinicia o valor do atributo “mensagem” e chama **getJspContext()** novamente.