



Java Web

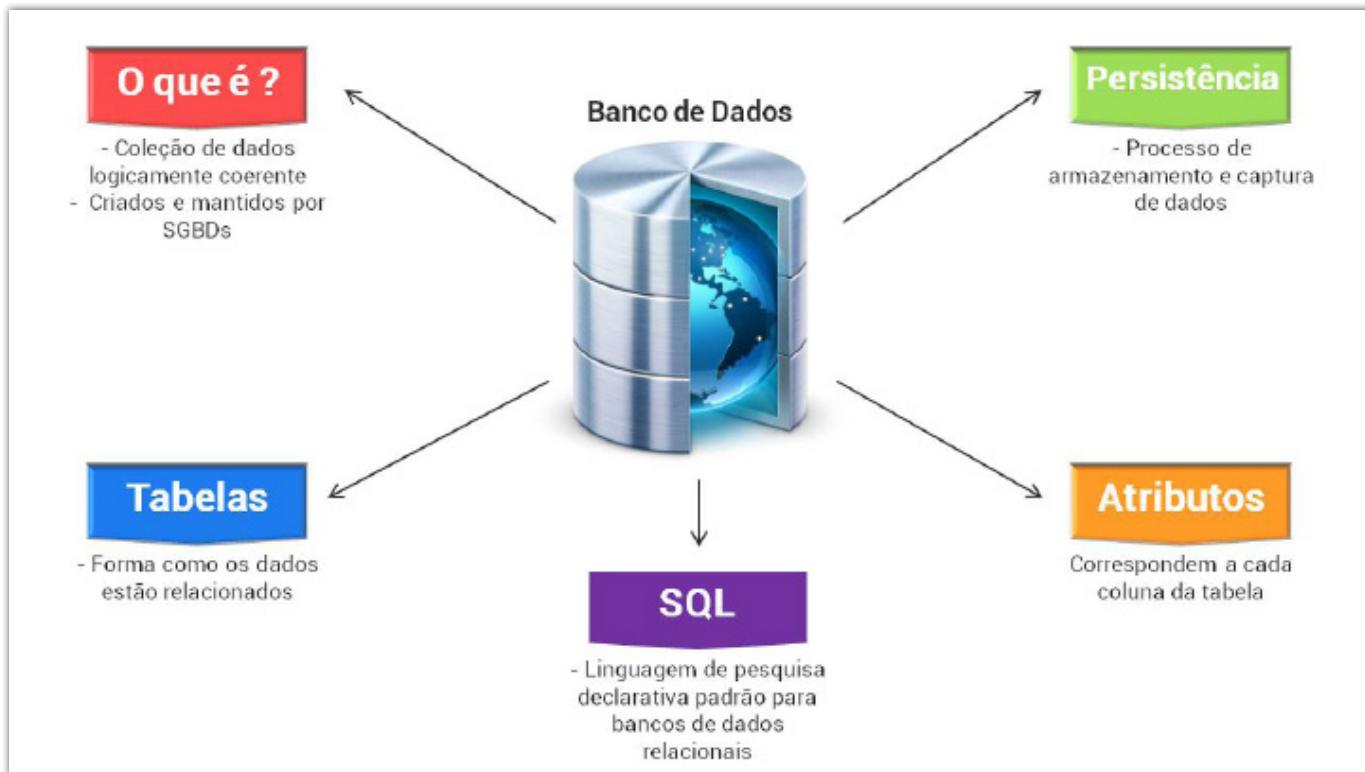


Bancos de Dados Relacionais	6
O que é banco de dados?	6
Sistema Gerenciador de Banco de Dados	7
Estrutura de uma Tabela.....	9
Tabelas e chaves	10
SQL	11
Comando Select.....	14
Comando Insert.....	15
Comando Update.....	16
Comando Delete.....	17
União de Tabelas	18
INNER JOIN.....	19
LEFT JOIN	20
RIGHT JOIN	22
Objeto Relacional.....	23
Mapeamento das classes e seus tributos.....	25
Mapeamento de Associação 1-1	26
Mapeamento de Associação 1 - Muitos.....	27
Mapeamento de Associação Muitos-Muitos.....	28
Mapeamento de Agregação	28
Mapeamento de Associação Reflexivas	29
Mapeamento de Associação n-varias	27
Mapeamento de Classes Associativas	31
Mapeamento de Herança	32
JDBC Java DataBase Connectivity.....	36
Prepared Statement	37
Padrão Fábrica - Conexão com BD.....	39
SQL no Código	40
DAO Data Acess Object	42
Padrão DAO	44
Padrão Aplicado.....	43

CRUD	45
HTML, CSS e JavaScript	47
Tags Semântica	48
CSS Container e background imagem	49
Elementos de Cabeçalho	50
Elementos de Rodapé	51
Barra Lateral	52
Java Script e Janelas de Alerta.....	54
Formulário Boostrap	55
Servlets E JSP.....	56
Servlets	56
Estruturas de Aplicação Web	58
Criação de uma Servlet	59
Implementação de uma Classe Servlet.....	60
Mapeamento na web.xml	61
Implementação	63
Recebendo Parâmetros.....	65
Dados de Formulário	67
Expression Language	68
Página JSP	72
Redirecionar (redirect).....	74
Encaminhar (forward).....	76
Cookies	74
Paginas de Erro	81
JSTL JavaServer Pages Standart Tag Library	83
JSTL Core Tags	74
Servlet com <c:forEach>.....	87
Internacionalização de Páginas	88
JSTL Tags de Formatação	89
Internacionalização de Páginas	90
JSF	92
TagLibs do JSF	93

MVC.....	95
Fluxo com Tudo no Servlet	96
Fluxo com Geração do HTML fora do Servlet	97
Fluxo com Tarefas bem Distribuídas.....	99
Arquitetura MVC	100
Ciclo de Vida JSF.....	101
Àrvore de Componentes.....	102
Aplicar Valores Requisição.....	103
Atualizar Valores de Modelo.....	105
Managed beans	107
Escopos de Navegação	109
RequestScoped.....	110
SessionScoped.....	111
ViewScoped.....	112
ApplicationScoped.....	113
NoneScoped.....	114
Componentes UI.....	115
Exemplo de Utilização de Algumas Tags	116
Tabela de Dados	119
Command Link e Command Button	120
Ajax com HTML e JavaScript	121
Ajax com JSF	122
Conversor Padrão	123
Conversor Customizado	126
Validador Padrão	127
Validador Customizado	129
Mensagens	130
Internacionalização	131
Mapeamento dos Arquivos Properties no faces-config.xml	132
Navegação	132
Navegação bean	134
Faces-config.xml	135
Navegação Explícita	136

Navegação Implícita	137
Action Handlers	138
Event Listeners	139
Navegação	133
Navegação bean	135
Faces-config.xml	136
Navegação Explícita	137
Navegação Implícita	138
Action Handlers	139
Event Listeners	140
Facelets	141
Templates	142
Página de Template Criado Layout.xhtml.....	143
Executando a Página1.xhtml.....	145



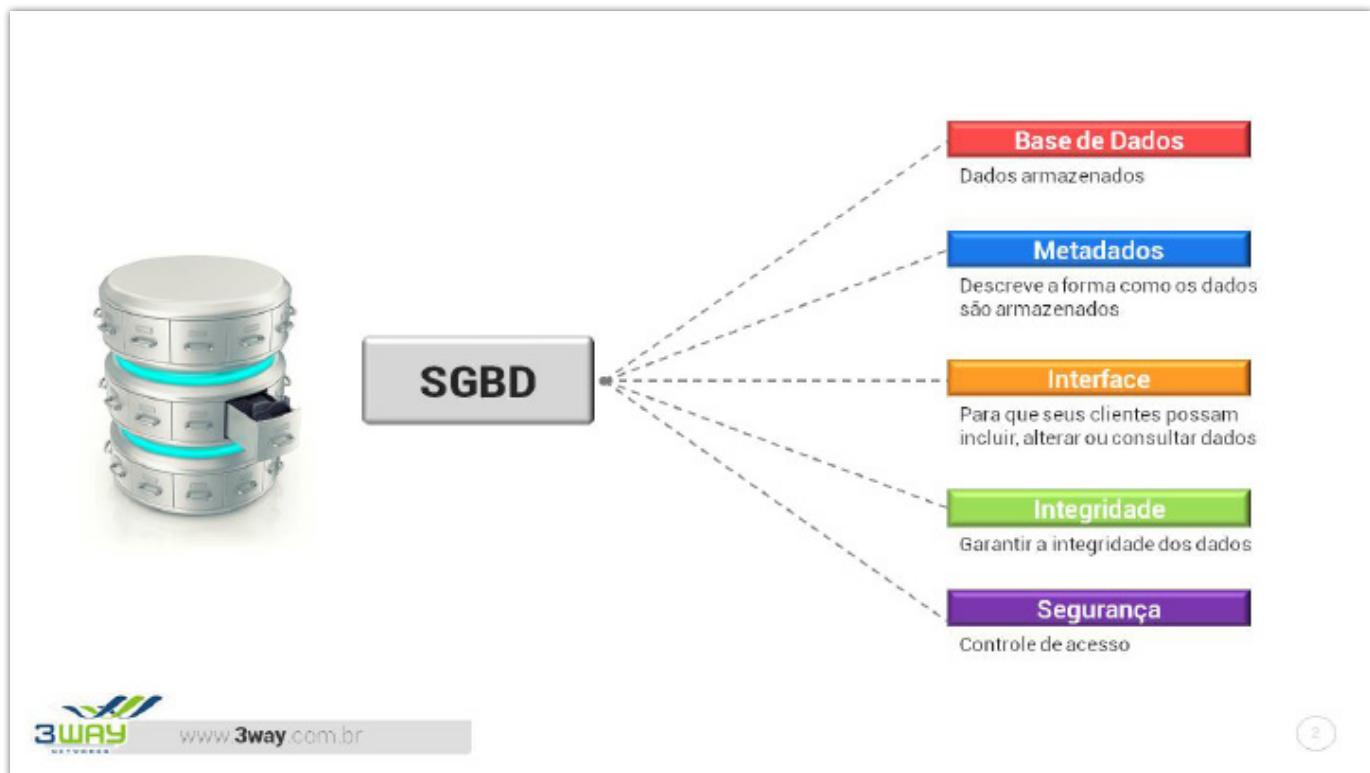
Banco de dados é onde guardamos os dados que pertencem ao nosso sistema. A maioria dos bancos de dados comerciais hoje em dia são relacionais

Banco de dados relacional armazenam dados em tabelas. Cada tabela contém linhas e 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, essa API define as interfaces de manipulação dos dados no SGDB. Além do JDBC existem diversos projetos do tipo ORM (Object Relational Mapping) que facilitam a criação de programas que acessam dados relacionais em programas desenvolvidos com o paradigma de programação orientado a objetos.

Podemos citar alguns termos típicos quando o assunto é banco de dados:

- **Dados:** fatos que podem ser armazenados. Exemplo: nomes, telefones, endereços;
- **Base de Dados:** coleção de dados inter-relacionados logicamente. Exemplo: Agenda de telefones;
- **Persistência:** processo de armazenamento e captura de dados;
- **Tabelas:** Forma como os dados estão relacionados;
- **Atributos:** correspondem a cada coluna da tabela;
- **SQL:** linguagem de pesquisa declarativa padrão para banco dados;
- **Sistema de Gerência de Base de Dados (SGBD):** coleção de programas que permite a criação e gerência de base de dados.



Sistema Gerenciador de Banco de Dados (SGBD) é o conjunto de programas responsáveis por armazenar e recuperar dados de uma base de dados. O principal objetivo de um SGBD é retirar da aplicação cliente a responsabilidade de gerenciar o acesso, a manipulação e organização dos dados.

Os principais objetivos de um Sistema de Banco de Dados são:

- Gerenciar grande quantidade de informações:** 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.

- 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 acontece

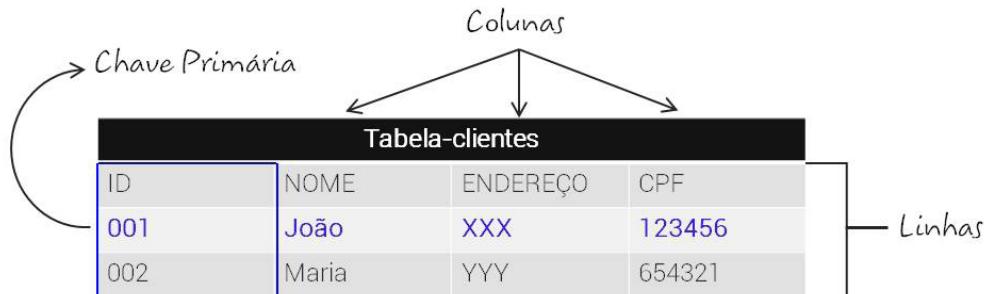
cia no Sistema de Arquivos, onde os dados poderiam estar em diversos formatos e o acesso poderia até ser impossível. Outro ponto que um Sistema de Banco de Dados facilita é o acesso concorrente, onde podemos ter a mesma informação sendo compartilhada por diversos usuários.

- **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 acesso.

- **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 em um campo onde a entrada deve ser a sigla do estado.

- **Recuperação de banco de dados:** existem alguns mecanismos capazes de permitir a recuperação de um banco de dados de alguma inconsistência causada por falhas internas (erros de consistência, como recuperação de um estado anterior à uma transação que deu erro) e externas (queda de energia, catástrofe ambiental).

Estrutura de uma Tabela



3

As tabelas em um banco de dados são organizadas em colunas, cada coluna armazena um tipo de dado (inteiro, números reais, strings de caracteres, datas, etc...). Em geral todos os bancos de dados dão suporte aos tipos de dados SQL ANSI.

ID	NOME
CHARACTER	Caractere de tamanho fixo - CHAR
CHARACTER VARYING	Caractere de tamanho variável - VARCHAR
CHARACTER LARGE OBJECT	Caractere longo - CLOB (4GB)
BINARY LARGE OBJECT	Binário para objetos longos - BLOB (4GB)
NUMERIC	Numérico extato
DECIMAL	Numérico extato
SMALLINT	Numérico extato
INTEGER	Numérico extato
BIGINT	Numérico extato
FLOAT	Numérico aproximado
REAL	Numérico aproximado
DOUBLE PRECISION	Numérico aproximado
BOOLEAN	Booleano
DATE	Data com dia, mês e ano,
TIME	Horário com hora, minuto e segundos
TIMESTAMP	Momento com ano, mês, dia, hora, minuto e segundo

Chave Estrangeira (Foreign Key)

Estabelece relacionamento entre duas tabelas
Corresponde à chave primária de outra tabela

Tabela_A	
pk_id	<coluna>
Tabela_B	
id	fk_idA

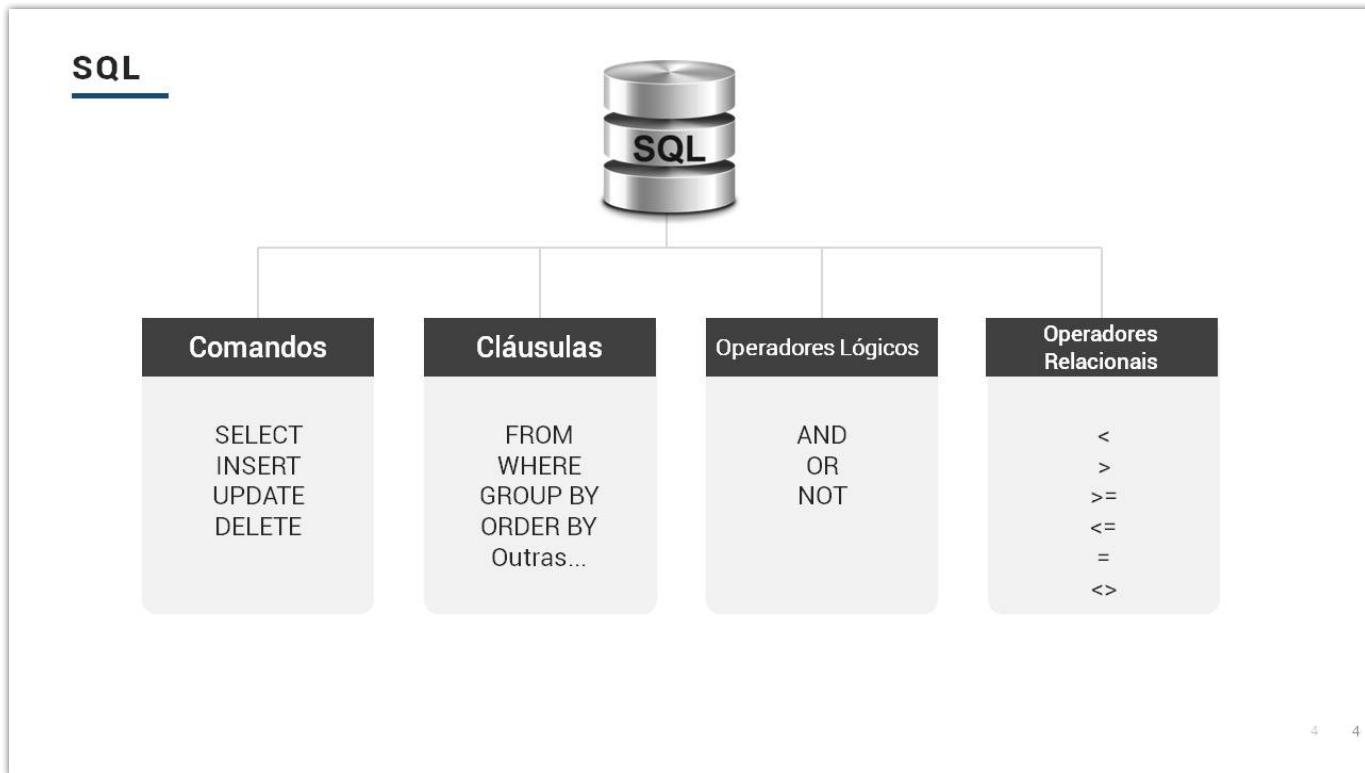
Chave Primária (Primary Key)

Identifica de forma única uma linha da tabela

3

Tabelas devem possuir um ou mais campos chave, que são uma ou mais colunas que unicamente identificam uma linha na tabela. 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. No caso da tabela A, a chave primária é a coluna *pk_id*. Para mantermos uma relação da tabela A com B, acrescentamos a coluna *fk_idA* em B, que contém a mesma informação da coluna *pk_id* em A, assim, *fk_idA* é a chave estrangeira de A em B. Desse modo é possível identificar todas as linhas da tabela B que tem relacionamento com a tabela A.



4 4

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 SqlServer, PostgreSQL (código aberto), HSQLDB (código aberto e escrito em java).

DDL – Linguagem de Definição de Dados

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

Os comando básicos da DDL são poucos:

CREATE: cria um objeto (exemplo uma tabela) dentro da base de dados.

CREATE TABLE Produto(

```
  id      INT PRIMARY KEY,  
  codigo  VARCHAR(10) NOT NULL UNIQUE,  
  descricao VARCHAR(255) NOT NULL,  
  preco    NUMBER(10,2) NOT NULL
```

);

DROP: apaga um objeto do banco de dados.

DROP TABLE Produto;

ALTER: que permite ao usuário alterar um objeto.

ALTER TABLE Produto ADD validade DATE;

DCL – Linguagem de Controle de dados

Controla os aspectos de autorização de dados e a utilização de licenças por usuários. Os principais comandos são GRANT e REVOKE.

- **GRANT** – Autoriza ao usuário executar ou setar operações.

GRANT SELECT, UPDATE, INSERT ON Produto TO aluno;
GRANT ALL PRIVILEGES ON Produto TO aluno;

- **REVOKE** – Remove ou restringe a capacidade de um usuário de executar operações.

REVOKE UPDATE ON Produto FROM aluno;
REVOKE ALL PRIVILEGES ON Produto FROM aluno;

DTL – Linguagem de Transação de Dados

Utilizado pelos desenvolvedores em transações. Existem dois comandos principais: COMMIT e ROLLBACK.

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.

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 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 SQL é dependente da implementação.

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, Update, Delete, Select.

Operadores Lógicos

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

- **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 controles.

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).

Comandos - SELECT



Resultado do SELECT

ID	NOME	ENDEREÇO	CPF
001	João	XXX	123456

6

Select é uma declaração SQL que retorna um conjunto de resultados de registros de uma ou mais tabelas. Ela recupera zero ou mais linhas de uma ou mais tabelas-base, tabelas temporárias ou visões em um banco de dados. Na maioria das aplicações, SELECT é o comando de DML mais utilizado. A estrutura mais básica do comando SELECT é:

Forma geral:

```
SELECT "nome_coluna" FROM "nome_tabela"
```

Retorna todas as linhas da tabela Cliente (o asterisco indica que você deseja selecionar todos os campos).

```
SELECT * FROM Cliente
```

Retorna o nome e o cpf de todos os clientes cujo o primeiro nome começa com “Jo”.

```
SELECT Nome, Cpf FROM Cliente WHERE Nome LIKE 'Jo%'
```

Retorna o nome e o cpf de todos os clientes cujo o primeiro nome começa com “Jo”, ordenados por Cpf.

```
SELECT Nome, Cpf FROM Cliente WHERE Nome LIKE 'Jo%'  
ORDER BY Cpf
```

Comandos - INSERT



Insert é uma instrução utilizada para inserir novos registros em uma tabela existente. Toda operação de inserção de um novo registro será feita através desta instrução. A estrutura mais básica do comando INSERT é:

Forma geral:

```
INSERT INTO "nome_tabela" ("coluna 1", "coluna 2", ...)
VALUES ("valor 1", "valor 2", ...)
```

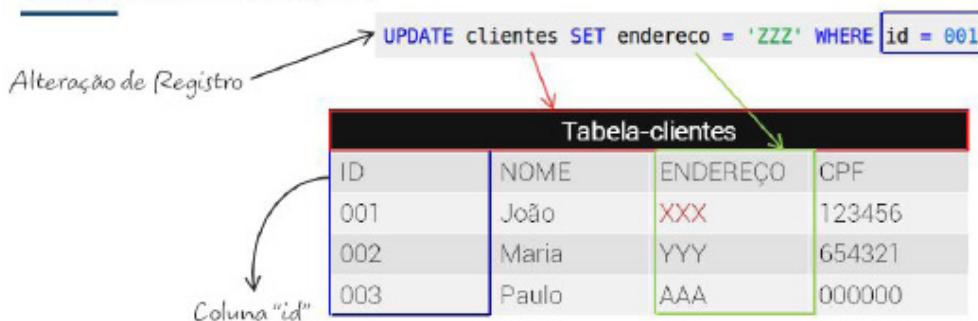
Insere um novo cliente na tabela Cliente com os dados especificados na ordem de criação das colunas.

```
INSERT INTO Clientes VALUES (003, 'Paulo', 'AAA', '000000')
```

Insere um novo cliente na tabela Cliente com os dados especificados conforme ordem das colunas.

```
INSERT INTO Clientes (nome, endereco, cpf, id)
VALUES ('Paulo', 'Rua da Prata', '12345678901', 003)
```

Comandos - UPDATE



Resultado do UPDATE

Tabela-clientes			
ID	NOME	ENDEREÇO	CPF
001	João	ZZZ	123456
002	Maria	YYY	654321
003	Paulo	AAA	000000

Update é uma instrução utilizada para realizar atualizações de dados existentes em uma ou várias tabelas. As atualizações de dados são utilizadas juntamente com a cláusula WHERE, permitindo delimitar quais registros serão atualizados. Se esta condição não for especificada, todas as linhas da tabela serão modificadas.

A tabela deve ser atualizável e não pode aparecer em nenhuma cláusula FROM de qualquer subconsulta presente na cláusula WHERE. A estrutura mais básica do comando UPDATE é:

Forma geral:

```
UPDATE "nome_tabela" SET "coluna 1" = [novo valor]
WHERE "condição"
```

Atualiza a coluna endereço da tabela Cliente onde o id do cliente é igual a 001

```
UPDATE Clientes SET endereco = 'ZZZ' WHERE id = 001
```

Comandos - DELETE



Resultado do DELETE

Tabela-clientes			
ID	NOME	ENDEREÇO	CPF
001	João	XXX	123456
003	Paulo	AAA	000000



www.3way.com.br



O comando **DELETE** nunca deve ser utilizado sem a cláusula WHERE, caso contrário, irá deletar todos os registros da tabela. A estrutura mais básica do comando DELETE é:

Forma geral:

DELETE FROM “nome_tabela” WHERE “condição”

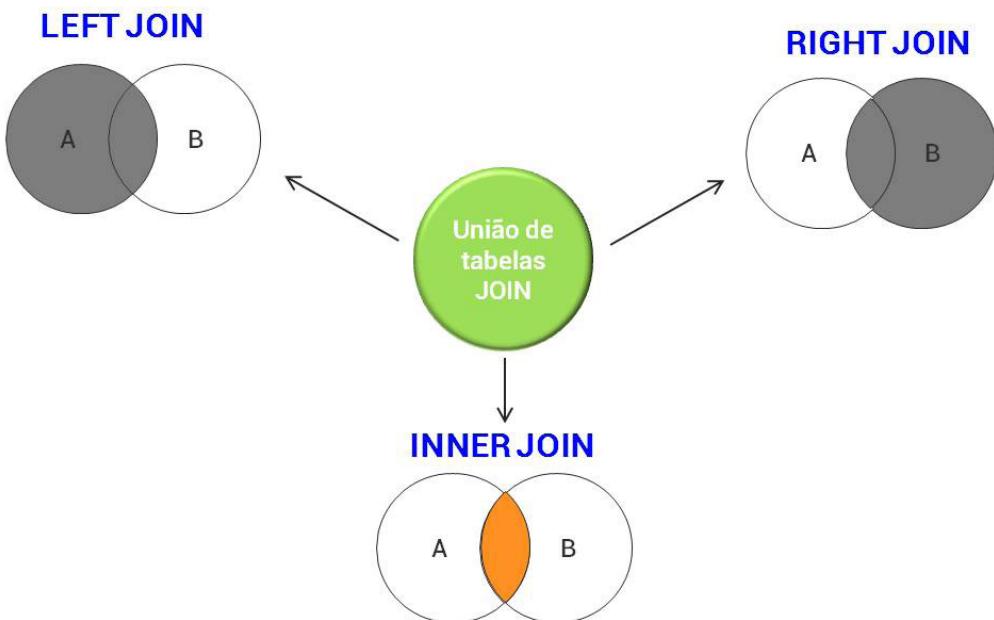
Remove todos os registros da tabela Clientes

DELETE FROM Clientes

Remove todos os registros da tabela Clientes para as linhas com id igual a 002

DELETE FROM Clientes WHERE id = 002

União de tabelas JOIN



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.

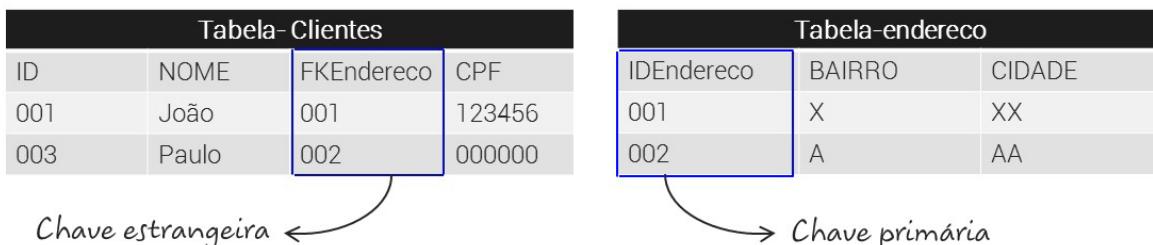
O resultado de uma junção pode ser utilizado para retonar os dados para um usuário ou aplicação, mas também pode ser referenciado em uma consulta como uma nova tabela, permitindo que novas junções e filtros sejam realizados sobre este sub-resultado. Além disso, pode se selecionar colunas específicas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

União de tabelas – INNER JOIN

```
SELECT * FROM clientes INNER JOIN enderecos ON clientes.fkEndereco = enderecos.idEndereco
```



Resultado do INNER JOIN

ID	NOME	FKEndereço	CPF	ID endereco	BAIRRO	CIDADE
001	João	001	123456	001	X	XX
003	Paulo	002	000000	002	A	AA

13

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 correspondem ao critério estabelecido na cláusula ON.

A estrutura de uma query usando INNER JOIN é:

```
SELECT “nome das colunas” FROM “tabela1”
LEFT JOIN “tabela2” ON tabela1.nome_da_coluna = tabela2.nome_da_coluna;
```

Considere as tabelas Funcionario e Info_Funcionario:

ID	NOME	SOBRENOME	IDADE
1	pedro	carvalho	25

ID_PESSOA	CARGO	SALARIO
1	Desenvolvedor	2500

```
SELECT ID, NOME, SALARIO FROM Funcionario
INNER JOIN Info_Funcionario
ON Funcionario.ID = Info_Funcionario.ID_PESSOA
```

ID_PESSOA	NOME	SALARIO
1	PEDRO	2500

União de tabelas – LEFT JOIN

```
SELECT * FROM clientes LEFT JOIN enderecos ON clientes.fkEndereco = enderecos.idEndereco
```

Tabela-Clientes			
ID	NOME	FKEndereço	CPF
001	João	001	123456
003	Paulo	002	000000
004	Maria	004	456789

Chave estrangeira ↙

Tabela-endereco		
IDEndereço	BAIRRO	CIDADE
001	X	XX
002	A	AA
003	Y	YY

↘ Chave primária

Resultado do LEFT JOIN

ID	NOME	FKEndereço	CPF	ID endereco	BAIRRO	CIDADE
001	João	001	123456	001	X	XX
003	Paulo	002	000000	002	A	AA
004	Maria	004	456789			

15

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.

A estrutura de uma query usando LEFT JOIN é:

```
SELECT "nome das colunas" FROM "tabela1"
LEFT JOIN "tabela2" ON tabela1.nome_da_coluna = tabela2.nome_da_coluna;
```

Considere as tabelas Funcionario e Info_Funcionario:

ID	NOME	SOBRENOME	IDADE
1	Pedro	Carvalho	25
2	Maria	Pereira	32
3	Carlos	José	28
4	Lucas	Oliveira	21

ID_PESSOA	CARGO	SALARIO
1	Desenvolvedor Junior	2300.00
2	Analista de Testes	2600.00

```
SELECT ID, NOME, SALARIO
FROM Funcionario
LEFT JOIN Info_Funcionario
ON Funcionario.ID = Info_Funcionario.ID_PESSOA
```

ID	NOME	SOBRENOME	IDADE
1	Pedro	Carvalho	2300.00
2	Maria	Pereira	2600.00
3	Carlos	José	NULL
4	Lucas	Oliveira	NULL

Note que como não há registros relacionados aos funcionários de ID 3 e 4 na tabela Info_Funcionário, a coluna SALARIO vem com o valor NULL.

União de tabelas – RIGHT JOIN

```
SELECT * FROM clientes RIGHT JOIN enderecos ON clientes.fkEndereco = enderecos.idEndereco
```

Tabela-Clientes			
ID	NOME	FKEndereco	CPF
001	João	001	123456
003	Paulo	002	000000
004	Maria	004	456789

Tabela-endereco		
IDEndereço	BAIRRO	CIDADE
001	X	XX
002	A	AA
003	Y	YY

Chave estrangeira ←

→ Chave primária

Resultado do RIGHT JOIN

ID	NOME	FKEndereco	CPF	ID endereco	BAIRRO	CIDADE
001	João	001	123456	001	X	XX
003	Paulo	002	000000	002	A	AA
				003	Y	YY

14

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.
A estrutura de uma query usando RIGHT JOIN é:

```
SELECT "nome das colunas" FROM "tabela1"
RIGHT JOIN "tabela2" ON tabela1.nome_da_coluna = tabela2.nome_da_coluna;
```

Considere as tabelas Funcionario e Responsavel_Projeto

ID	NOME	SOBRENOME	IDADE
1	Pedro	Carvalho	25
2	Maria	Pereira	32
3	Carlos	José	28
4	Lucas	Oliveira	21

ID_PROJETO	ID_FUNCIONARIO	PROJETO	COMPLEXIDADE
1	1	Projeto 1	Baixa
2	1	Projeto 2	Alta
3	2	Projeto 3	Alta
4	NULL	Projeto 4	Média

```
SELECT NOME, PROJETO, COMPLEXIDADE
FROM Funcionario
RIGHT JOIN Projetos Funcionario.ID = Responsavel_Projeto.ID_FUNCIONARIO
```

NOME	PROJETO	COMPLEXIDADE
Pedro	Projeto 1	Baixa
Maria	Projeto 2	Alta
Carlos	Projeto 3	Alta
Lucas	Projeto 4	Média

Objeto Relacional



16

O mapeamento objeto relacional é uma técnica de desenvolvimento utilizada para transformar as informações de um banco de dados que estão no modelo relacional para classes, dentro do paradigma de programação orientado a objetos. Assim, tabelas do banco de dados relacional são representados através de classes e os registros de cada tabela são como instâncias das classes correspondentes.

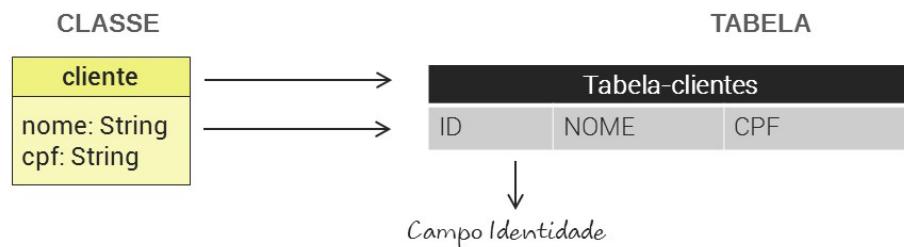
O desenvolvedor deve configurar a relação entre as tabelas de onde se originam os dados e o objeto que os disponibiliza.

Sua tarefa enquanto desenvolvedor é encontrar uma solução para os seguintes questionamentos:

- Como mapear as classes e atributos ?
- Como mapear as Relações ?
- Como mapear a Agregação ?
- Como mapear as Associações Reflexivas ?
- Como mapear as Associações n-árias ?
- Como mapear as Classes Associativas ?
- Como mapear Herança ?

Mapeamento das classes e seus tributos

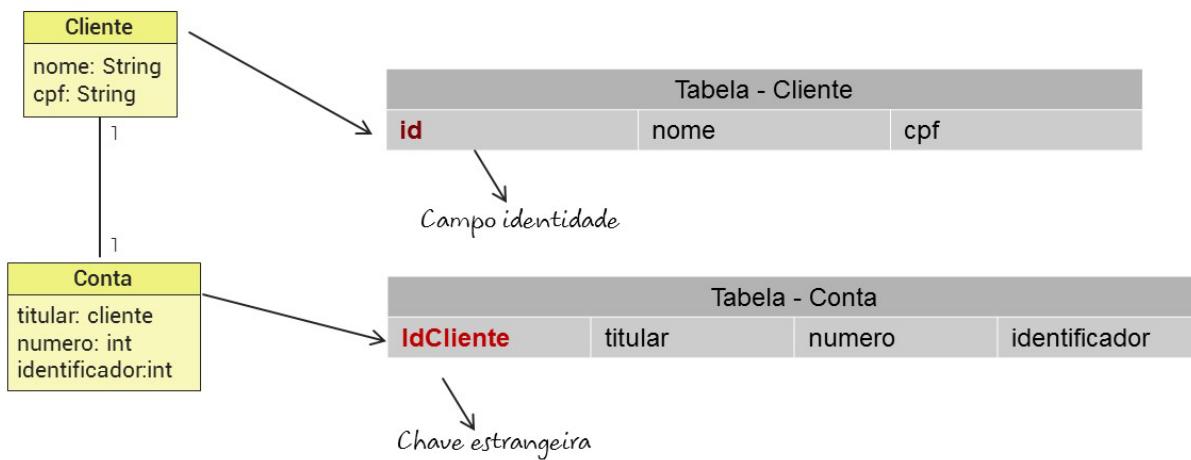
Mapear cada classe como uma tabela



17

A forma de mapeamento mais direta implica que cada classe torna-se um tabela no banco de dados e cada atributo não transiente da classe representa uma coluna nesta tabela. Isso fica muito óbvio para mapeamento de novos sistemas, porém para sistemas que utilizam dados de um banco dados legado (banco de dados que já existem), podem haver mais de uma tabela armazenando os dados de uma determinada classe.

Mapeamento de Associação 1-1

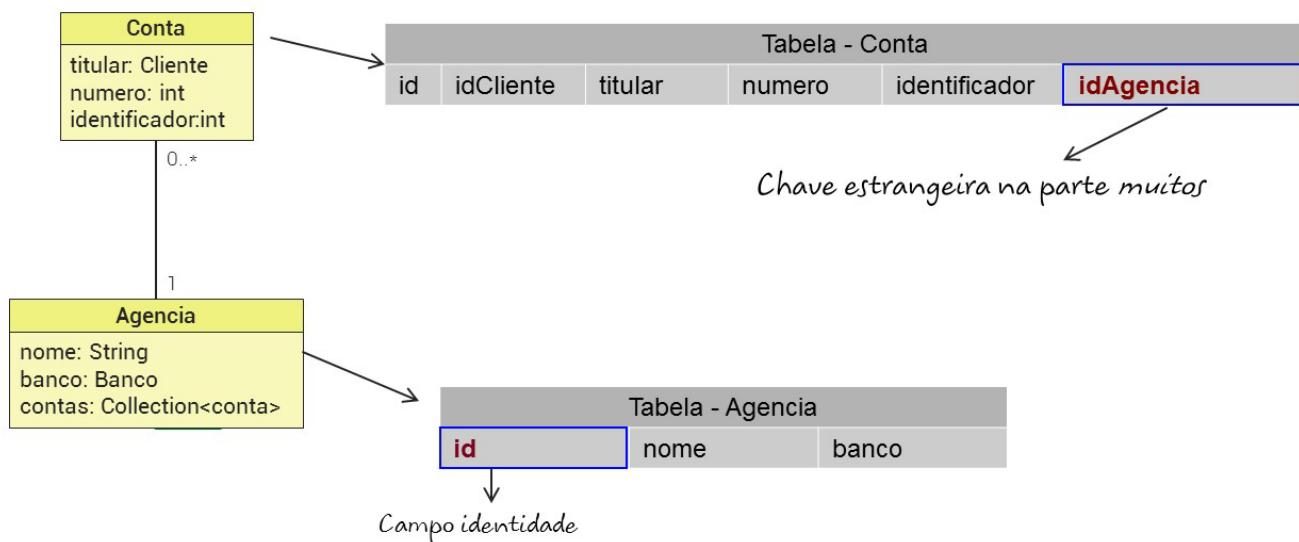


O mapeamento de associação utiliza o conceito de chave estrangeira, havendo 3 tipos, cada um correspondente a um tipo de conectividade.

O primeiro tipo é o mapeamento de associação 1 – 1.

Nesse tipo de mapeamento devemos adicionar uma chave estrangeira em um dos lados da relação, para que essa chave estrangeira refcrcie a chave primária do outro lado.

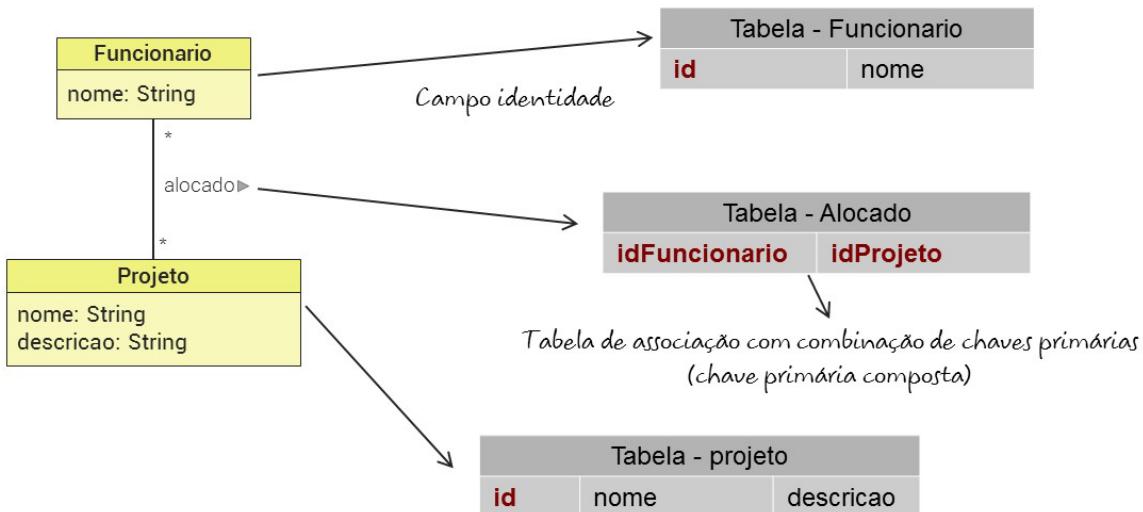
Mapeamento de Associação 1 - muitos



O segundo tipo é o mapeamento de associações 1 – muitos.

Nesse tipo de mapeamento, temos um objeto da classe A que se associa com vários objetos da classe B, considerando que ambas as classes possuam tabelas que as representem, devemos adicionar uma chave estrangeira na tabela A para referenciar a chave primária na tabela B. Isso significa que, se temos as classes Conta e Agencia, podemos afirmar que uma Agencia possui várias contas. Nas tabelas correspondentes, devemos adicionar uma chave estrangeira na tabela Conta referenciando a qual Agencia ela pertence.

Mapeamento de Associação muitos - muitos

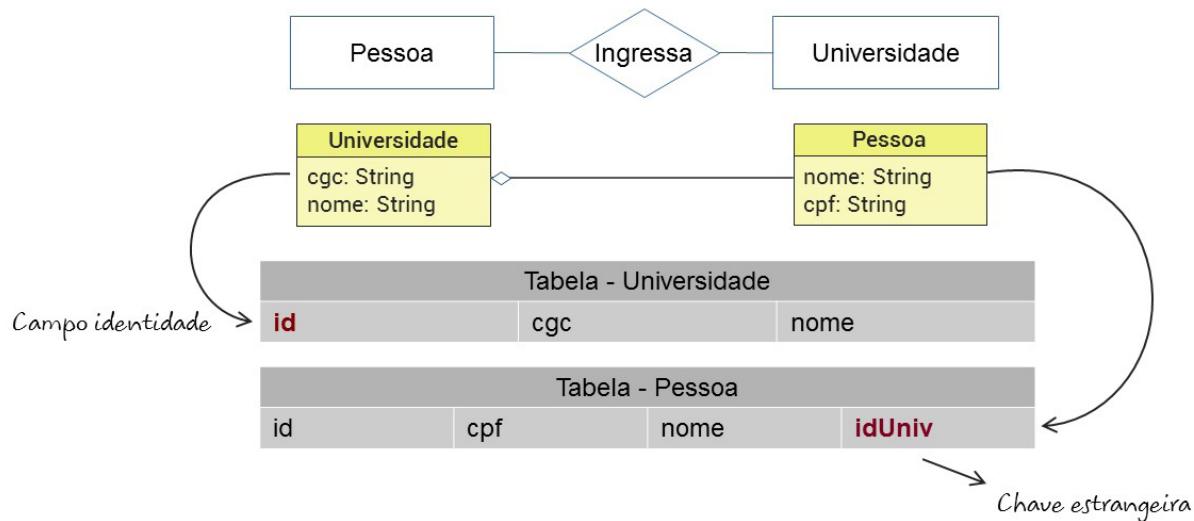


O terceiro tipo é o mapeamento de associações muitos – muitos.

Nesse tipo de mapeamento, devemos criar uma tabela de associação onde a chave primária dessa tabela é uma combinação das chaves primária das tabelas envolvidas na associação, resultando em uma chave primária composta.

Mapeamento de Agregação

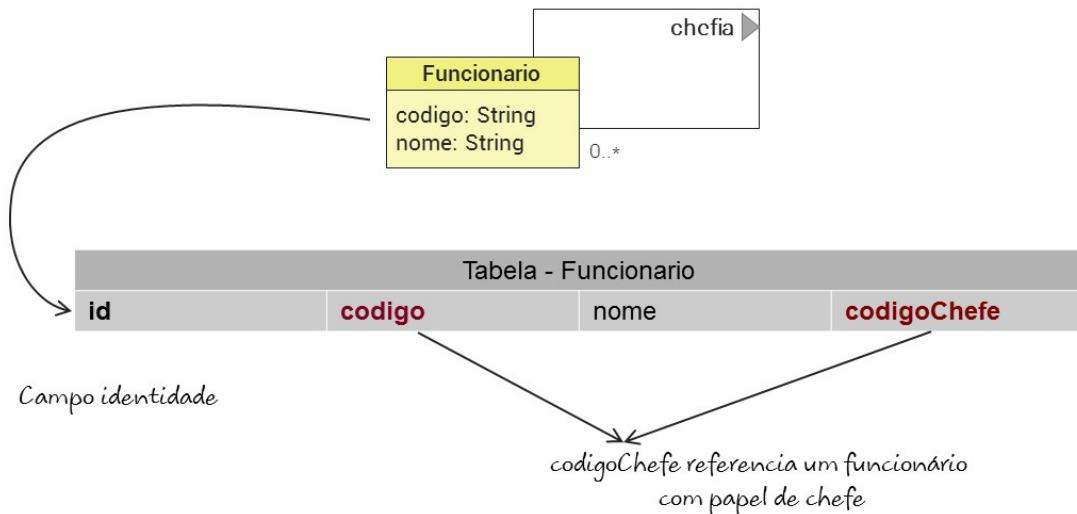
Alterações na tabela são feitas em cascata



Agregações são associações e são mapeadas da mesma forma. O mapeamento de agregações se diferencia na forma como o SGBD se comporta quando um registro da relação corresponde deve ser excluído ou atualizado. Isso implica que a remoção ou atualização do dado deve acontecer em cascata.

Mapeamento de Associações Reflexivas

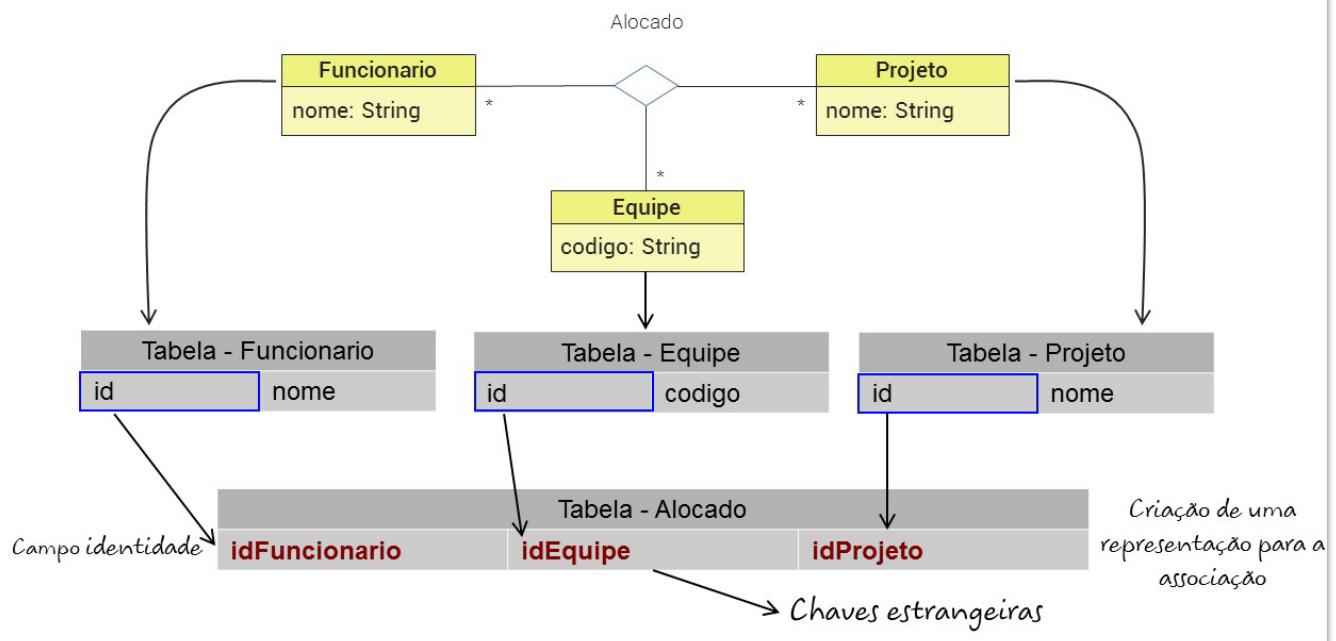
Mesmo procedimento de mapeamento de associação



O mapeamento de associações reflexivas é uma forma especial de associação, portanto utilizamos o mesmo procedimento de mapeamento.

Nesse tipo de mapeamento, devemos criar uma chave estrangeira que referencia a chave primária de um elemento específico da tabela, ou seja, suponha uma tabela de funcionários e dentre estes funcionários temos um chefe. Todo funcionário possui um código, então cada funcionário deve ter uma chave estrangeira que refencie o código do funcionário chefe.

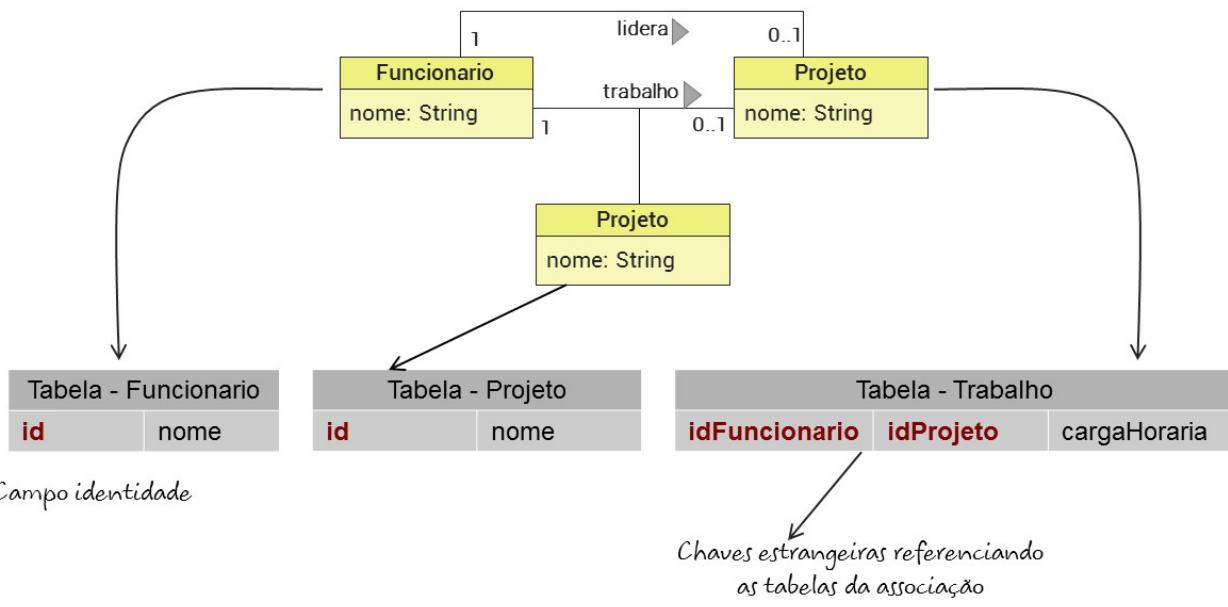
Mapeamento de Associações n-árias



Associações n-árias são semelhantes às associações muitos-muitos.

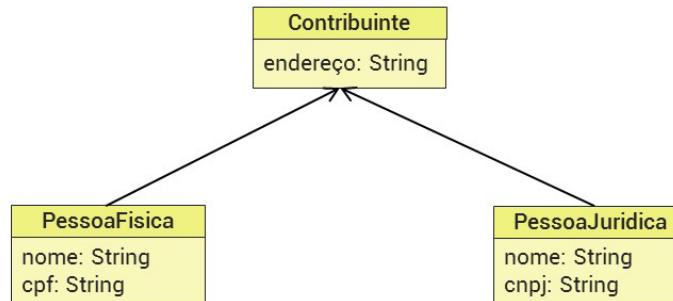
Devemos criar uma relação para representar a associação que é criada, adicionando chaves estrangeiras referenciando as classes associadas na relação.

Mapeamento de Classes Associativas



Para cada um dos casos de mapeamento de associações, há uma variante onde uma classe associativa é utilizada. O mapeamento é feito através da criação de uma relação para representá-la, onde os atributos da classe associativa são mapeados para colunas dessa relação, que também deve conter chaves estrangeiras que referenciam as relações correspondentes às classes que participam da associação.

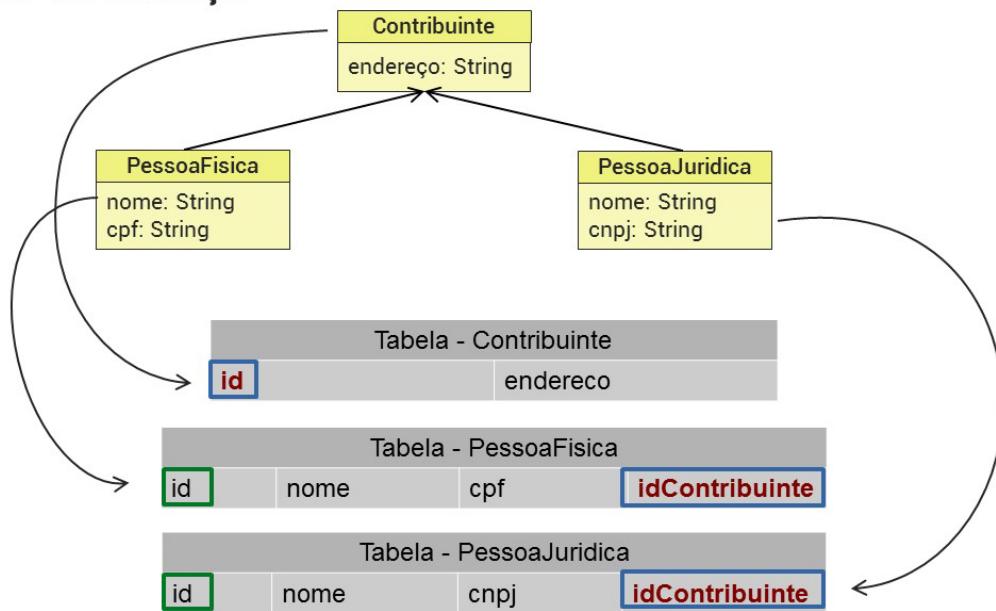
Mapeamento de Herança



3 Possíveis Soluções

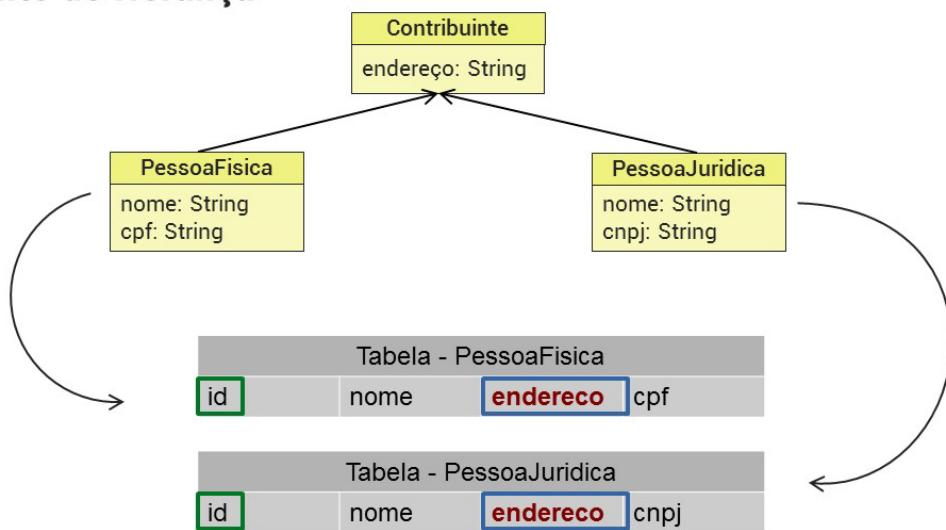
No mapeamento de herança (generalização) temos três formas alternativas de mapeamento:

Mapeamento de Herança



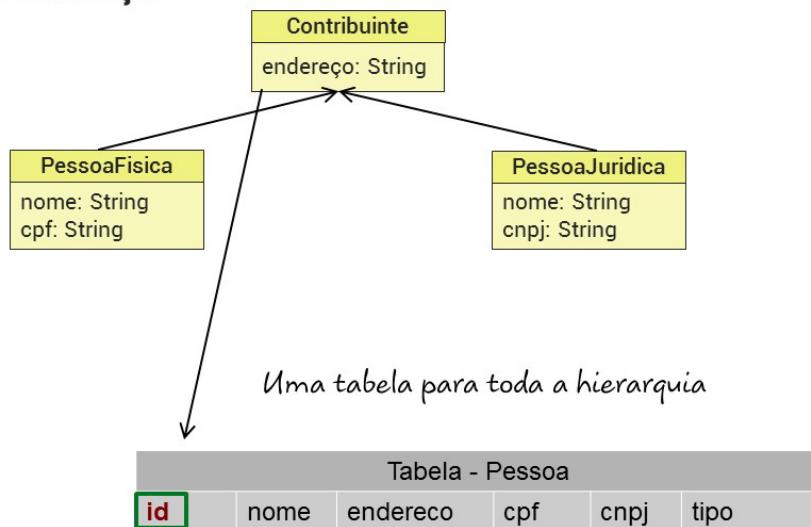
1. Consiste em criar uma tabela para cada classe da hierarquia. As classes são mapeadas para uma relação, sendo que as colunas desta relação são correspondentes aos atributos da classe. Devemos utilizar uma chave estrangeira para cada tabela da subclasse. A desvantagem é que o desempenho da manipulação das tabelas diminui. Essa alternativa é a que melhor reflete o modelo OO.

Mapeamento de Herança



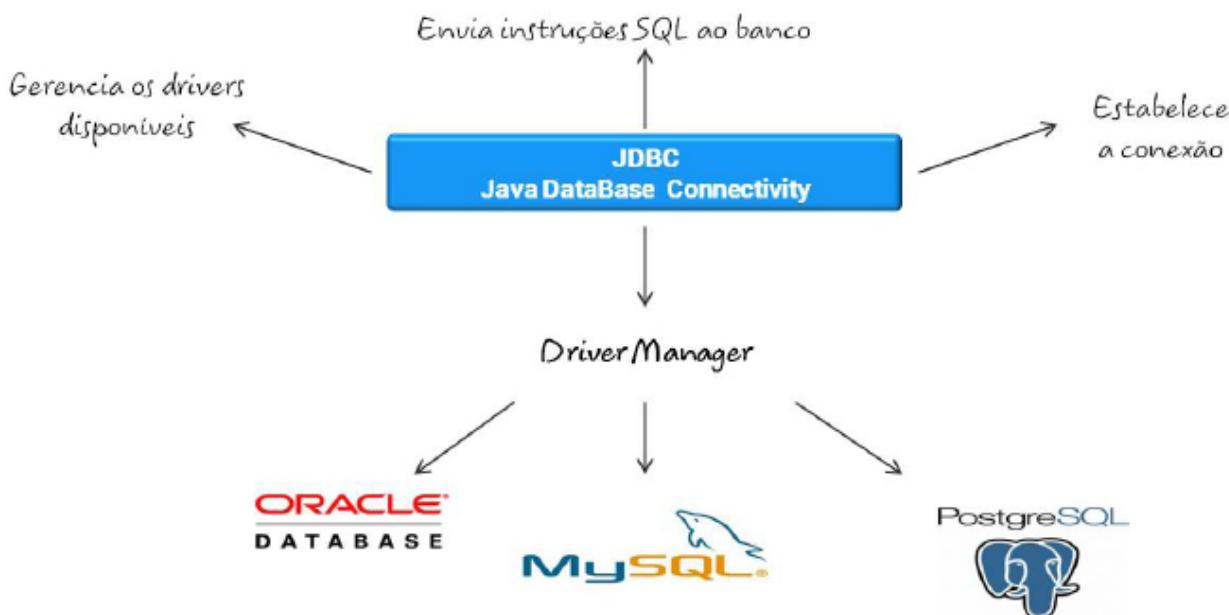
2. Consiste em criar uma tabela para cada classe concreta da hierarquia, tendo uma implementação bem simples. Nesse caso devemos replicar os atributos da superclasse. As desvantagens são o potencial desperdício de espaço de armazenamento, já que teremos uma hierarquia com várias classes irmãs e objetos pertencentes a somente ma classe da hierarquia.

Mapeamento de Herança



3. Consiste em criar uma única tabela para toda a hierarquia, tendo a vantagem de agrupar os objetos em uma única relação. Devemos criar um atributo para diferenciar o conceito de cada elemento da tabela.

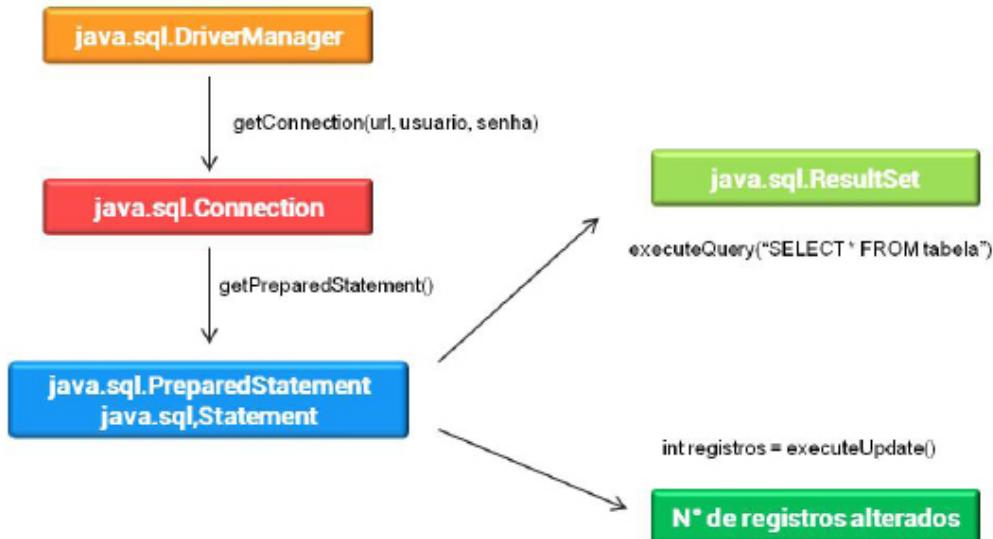
JDBC Java DataBase Connectivity



As seguintes classes estão na API JDBC:

- **Java.sql.DriverManager** – Gerencia os drivers JDBC utilizados pela aplicação. Em conjunto com o endereço e a autenticação fornece objetos de conexão através do método estático `getConnection([url],[usuario],[senha])`.
- **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.Statement** – Fornece meios ao desenvolvedor para que se possa executar comandos SQL.
- **Java.sql.PreparedStatement** – Um Statement aprimorado, permite usar parametros de substituição nas SQL preparadas.
- **Java.sql.ResultSet** – Representa o resultado de um comando SQL. Estes objetos normalmente são retornados por métodos.

Prepared Statement



```

import java.sql.*;
public class FabricaConexao {
    static String url = "jdbc:postgresql://localhost:5432/3way";
    static String usuario = "3way";
    static String senha = "123";
    public static Connection getConexao() throws SQLException{
        try{
            Class.forName("org.postgresql.Driver");
            return DriverManager.getConnection(url,usuario,senha);
        }catch(ClassNotFoundException e){
        {
            throw new SQLException(e.getMessage());
        }
    }
}
  
```

Para realizarmos operações com o SGDB, usando a API JDBC, precisamos executar alguns passos

1. Obter uma conexão com o Banco de Dados, devemos utilizar a classe **`java.sql.DriverManager`**, invocando o método estático **`getConnection()`**, esse método recebe o parâmetro URL, observe o padrão da string url, sempre começa com '**`jdbc`**' seguido pelo nome do banco de dados '**`postgresql`**' (este nome é definido pelo implementador do driver, por exemplo '`mysql`', '`oracle:thin`', '`sqlserver`', etc), seguidos pelo DNS ou IP e porta de conexão do servidor SGDB, e ao final o nome do banco de dados.

Exemplos de URL para outros bancos de dados:

Microsoft Sql Server

jdbc:sqlserver://localhost:1433;databaseName=3way

Oracle (nome instância orcl)

jdbc:oracle:thin:@localhost:1521:orcl

MySql

jdbc:mysql://localhost:3360/3way

Firebird

jdbc:firebirdsql://localhost:3050//diretorio/firebird/3way.fdb

Nota: a partir de JDBC 4 com Java 6, não precisamos mais carregar o driver usando **Class.forName()** como no exemplo, a partir dessa versão a classe do driver “auto registrável”, basta estar no CLASSPATH da sua aplicação. Nas versões inferiores ou usando JAVA 5 é preciso utilizar o método **Class.forName()**, que carrega a classe do driver para a memória.

2. o método **getConnection()** retorna um objeto **java.sql.Connection**, usamos os métodos **getPreparedStatement()** ou **getStatement()** para obtermos um objeto do tipo **java.sql.Statement**, que nos permite enviar comandos SQL ao SGDB através dos métodos **executeUpdate()** para comandos SQL Insert, Update, Delete e **executeQuery()** para comandos SQL Select

3. o método **executeQuery()** de **java.sql.Statement**, retorna um objeto do tipo **java.sql.ResultSet** contendo todas as linhas com as repectivas colunas especificadas no comando SQL Select

4. o método **executeUpdate()** de **java.sql.Statement**, retorna um número inteiro que corresponde à quantidade de linhas que foram inseridas, alteradas ou removidas da tabela referênciada.

Podemos perceber que o método **getConexao()** é uma fábrica de conexões, isto é, ele fabrica conexões para nós, não importando de onde elas vieram. Portanto, nada mais natural do que chamar a classe de **FabricaConexao** e o método **getConexao()**.

Padrão Fábrica – Conexão Com BD

```

package threeway.projeto.service.Dao;
import java.sql.Connection;
public class FabricaConexao {
    public static String url = "jdbc:postgresql://localhost:5432/threewayweb";
    public static String usuario = "postgres";
    public static String senha = "123456";

    public static Connection getConexao() throws SQLException {
        try{
            Class.forName("org.postgresql.Driver");
            return DriverManager.getConnection(url,usuario,senha);
        }catch(ClassNotFoundException e){
            throw new SQLException(e.getMessage());
        }
    }
}

```

Dados da conexão:

- Nome da base de dados
- Usuário
- Senha

Retorna a conexão usando
o DriverManager



www.3way.com.br

Observe o método DriverManager.getConnection(), ele constroi objetos de conexão de diferentes tipos de banco de dados baseado no parâmetro URL. No exemplo acima, o objeto retornado será do tipo org.postgresql.jdbc.PgConnection que é uma implementação da interface java.sql.Connection, do mesmo modo se você utilizar a URL para o banco de dados MySql, teríamos um objeto de conexão do tipo com.mysql.jdbc.Connection que também é uma implementação de java.sql.Connection. Ou seja o método é um fábrica conexões para diferentes tipos de drivers.

Quando estamos trabalhando com interfaces e temos mais que uma implementação dessa interface, podemos utilizar o padrão de programação denominado Factory Method Pattern para criar esses objetos sem nos preocuparmos com a classe concreta.

A classe FabricaConexao desenvolve uma técnica denominada de “**método fábrica estático**” (diferente do Design Pattern Factory Method anterior), através do método getConexao(). Neste caso o objetivo é não ter que criar um novo objeto FabricaConexao toda vez que precisarmos de um objeto java.sql.Connection. Além disso, damos garantia de que toda aplicação estará sempre trabalhando com a mesma configuração do objeto java.sql.Connection.

SQL no Código

Pegue a conexão antes de fazer o PreparedStatement

Cria o objeto PreparedStatement com a SQL parametrizada

executeQuery() executa a SQL presente no objeto PreparedStatement e retorna um objeto ResultSet como resultado

PreparedStatement retorna um objeto ResultSet como resultado

```

private static final String SELECT_SQL = "SELECT NOME, SENHA, LOGIN, " SQL
+ "ENDERECO, CIDADE, BAIRRO, ESTADO, "
+ "CEP, COD_CLIENTE FROM CLIENTE WHERE SENHA = ? and LOGIN = ? ";

public Cliente obter(Cliente cliente) {
    try (Connection conexao = FabricaConexao.getConexao();
        PreparedStatement consulta = conexao.prepareStatement(SELECT_SQL)) {
        consulta.setString(1, cliente.getSenha());
        consulta.setString(2, cliente.getLogin());
        ResultSet resultado = consulta.executeQuery();

        if (resultado.next()) {
            cliente.setNome(resultado.getString("NOME"));
            cliente.setSenha(resultado.getString("SENHA"));
            cliente.setLogin(resultado.getString("LOGIN"));
            cliente.setEndereco(resultado.getString("ENDERECO"));
            cliente.setCidade(resultado.getString("CIDADE"));
            cliente.setBairro(resultado.getString("BAIRRO"));
            cliente.setEstado(resultado.getString("ESTADO"));
            cliente.setCep(resultado.getString("CEP"));
            cliente.setCodigo(resultado.getInt("COD_CLIENTE"));
            cliente.setAutenticacao(true);
        } else {
            cliente = null;
        }
    } catch (SQLException e) {
        log.severe(e.getMessage());
    }
    return cliente;
}

```

Como submeter comandos SQL para o banco de dados?

Tomemos como exemplo a tarefa de inserir dados em uma tabela de um banco de dados. Já sabemos que basta usar a cláusula INSERT, especificar quais os campos e os seus respectivos valores que desejamos atualizar. Em java temos que montar a string SQL como segue:

```
String sql = "insert into contatos (nome,email,endereco) "+ "values ('"+ nome +
"','"+ email + "','" + endereco +"')";
```

O exemplo acima o uso da concatenação de strings, pode te trazer alguns transtornos como: saber se faltou uma vírgula, ou um apóstrofo, ou um parênteses, talvez. Outro problema é o de segurança, como SQL Injection. Suponha que o valor para o campo nome contenha string “Joana D’ Arc” o apóstrofo presente no texto vai quebrar seu código SQL, ou ainda pior, se o atacante conseguir inserir código SQL de modo a ter acesso ao que ele deseja (injeção de SQL).

Por esses motivos vamor utilizar o comando SQL parametrizados, usando o caracter “?” como marca de substituição:

```
String sql = "insert into contatos (nome,email,endereco) values (?,?,?)";
```

Os pontos de interrogação são mesmo algo desconhecido para nossa aplicação, não sabemos ainda quais são os parâmetros que serão utilizados nesse código SQL, chamado de statement.

Para submetermos esse statement para execução no banco de dados precisamos de um objeto do tipo da interface PreparedStatement, obtido através método preparedStatement de java.sql.Connection, passando como argumento o comando SQL.

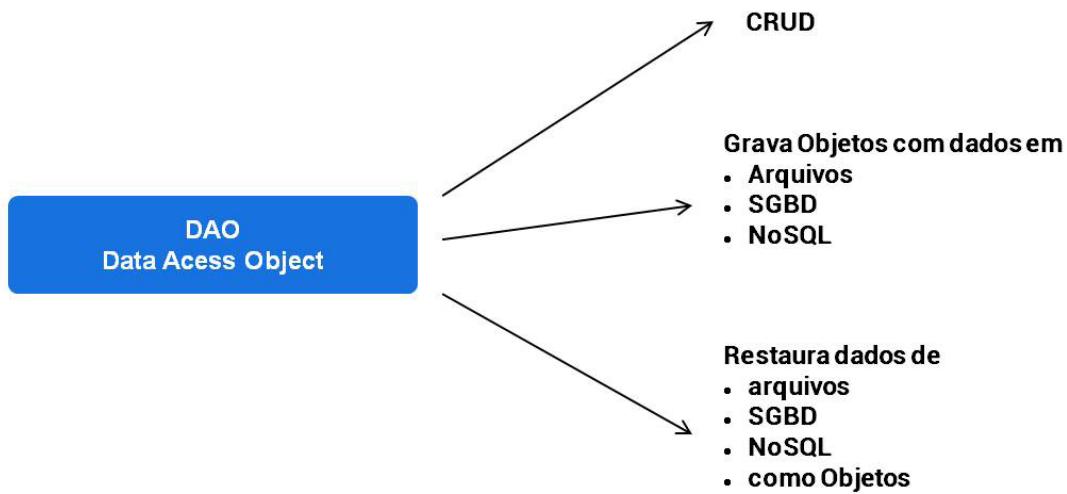
```
PreparedStatement stmt = con.prepareStatement  
        ("insert into contatos(nome,email,endereco) values(?, ?, ?)");
```

Em seguida, invocamos o método setString do PreparedStatement para substituir cada “?” pelo seu respectivo valor, começão pela primeira “?” (posição 1).

```
//preenche os valores  
stmt.setString(1, "3Way Networks");  
stmt.setString(2, "contato@3way.com.br");  
stmt.setString(3, "Av. 4a Rádial, 1952 Milão Shopping Cénter");
```

E finalmente, invocamos o método executeUpdate() de java.sql.preparedStatement para submetermos a SQL ao banco de dados.

Outra vantagem em usarmos preparedStatement é que grande parte dos SGDBs são otimizados para reaproveitar todo estrutura criada para execução dos commando SQL, fazendo somente a susbstiuuição dos valores dos parâmetros, otimizando a execução de comando SQL que sejam submetidos em série.



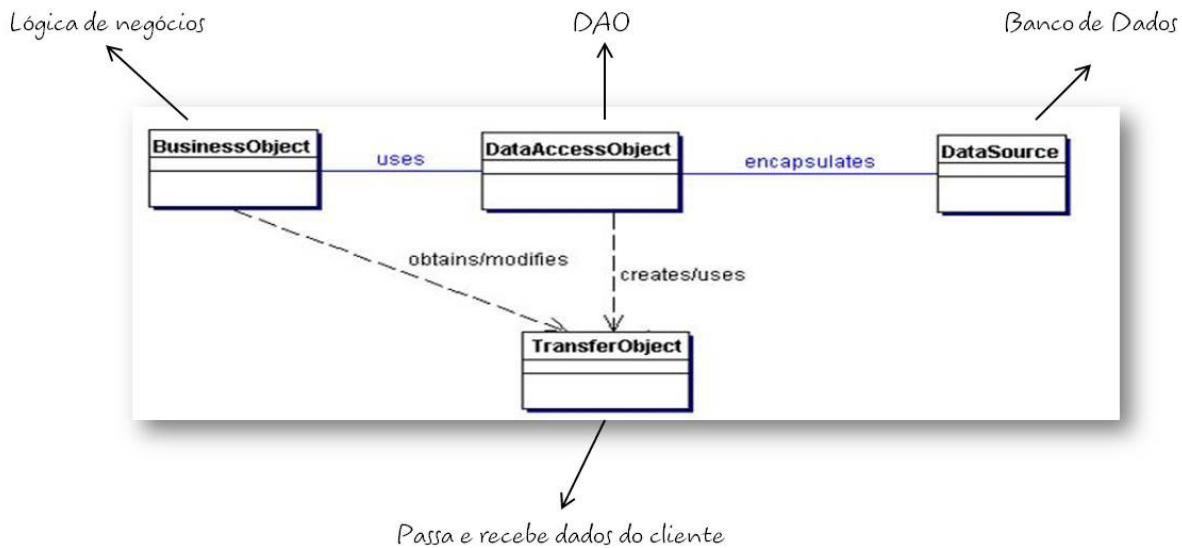
Desing Pattern DAO

Padrões de projeto são soluções conhecidas para problemas recorrentes. O padrão DAO é um modelo de como separar a lógica de negócios da lógica de persistência de dados, permitindo mudar a forma de persistência sem influenciar na lógica de negócios.

As classes DAO serão responsáveis por transformar os dados em formato relacional do SGBD em objetos java a serem utilizados em nossa aplicação.

A principal vantagem do DAO é ter um local onde todo acesso aos dados será concentrado, ao invés de ter várias classes que manipulam dados espalhadas pela aplicação.

Padrão DAO



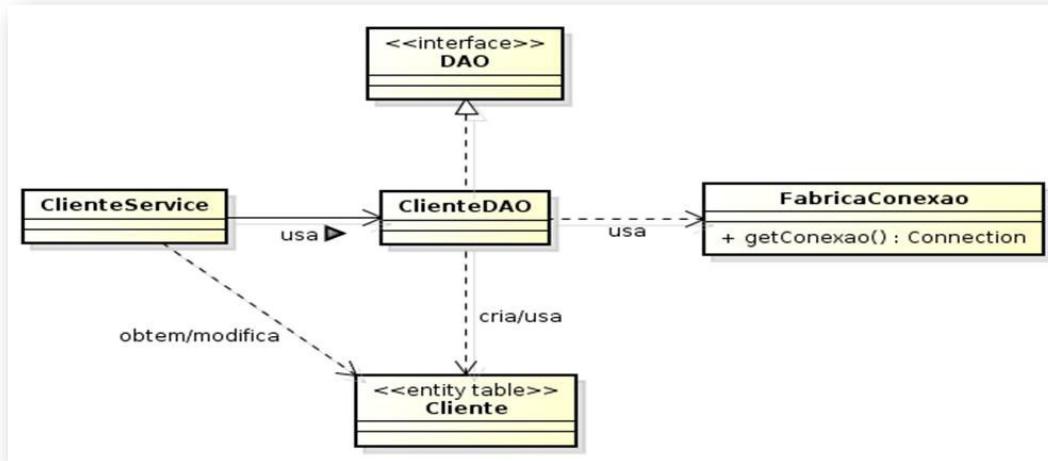
Padrão DAO

No diagrama de classes temos quatro classes se relacionando para atingir o objetivo de criar objetos de domínio, independentemente da fonte de dados ser um BD relacional, um arquivo de texto, um arquivo XML, etc.

O diagrama descata quatro entidades dentro do padrão:

- **DataSouce**: é nosso fonte de dados, origem dos dados, dentro do que temos visto seria nos so objeto de conexão com o banco de dados, um java.sql.Connection.
- **TransferObject**: é o objeto que contém os dados vindos do DataSource, normalmente são objetos de domínio que necessitam ser persistidos.
- **BusinessObject**: são as classes de serviço de mais alto nível, representam os repositórios de objetos de domínio da aplicação.
- **DataAccessObject**: coordena o encapsulamento dos dados vindos do DataSource em objetos TranferObject e os entregam ao objeto BussinessObject.

Padrão DAO



Padrão DAO Aplicado

O diagrama mostra a implementação do padrão DAO para um objeto de domínio da classe **Cliente** que precisa ser persistido na tabela de um banco de dados relacional. Assim:

ClienteDAO <=> DataAccessObject

FabricaConexao <=> DataSource

Cliente <=> TransferObject

ClienteService <=> BusinessObject

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 feito através das classes DAO de forma a se ter o encapsulamento.
 - cada instância da DAO é responsável por um objeto de domínio.
 - o não deve ser responsável por transações, sessões ou conexões que devem ser tratados fora do DAO.
 - o DAO é responsável pelas operações Create(criação), Recover(recuperação), Update(atualização), Delete (remoção) no domínio da aplicação.

CRUD CRIAR

```
public void salvar(Cliente entidade) {  
    StringBuffer sql = new StringBuffer();  
    sql.append("INSERT INTO cliente");  
    sql.append("(endereco, nome, telefone, cpf, rg) VALUES (?, ?, ?, ?, ?)");  
    try {  
        PreparedStatement consulta = conexao.prepareStatement(sql.toString());  
        consulta.setString(1, entidade.getEndereco());  
        consulta.setString(2, entidade.getNome());  
        consulta.setString(3, entidade.getTelefone());  
        consulta.setString(4, entidade.getCpf());  
        consulta.setString(5, entidade.getRg());  
        consulta.executeUpdate();  
    } catch (SQLException e) {  
        System.out.println("Erro ao realizar Insert: " + e.getMessage());  
    }  
}
```

CRUD RECUPERAR

```
public Cliente obter(Serializable identificador) {  
    Cliente cliente = null;  
    String sql = "SELECT * FROM cliente WHERE identificador = ?";  
    try {  
        PreparedStatement consulta = conexao.prepareStatement(sql);  
        consulta.setLong(1, (Long) identificador);  
        ResultSet resultado = consulta.executeQuery();  
        if(resultado.next()){  
            cliente = new Cliente();  
            cliente.setIdentificador(resultado.getLong("identificador"));  
            cliente.setEndereco(resultado.getString("endereco"));  
            cliente.setNome(resultado.getString("nome"));  
            cliente.setTelefone(resultado.getString("telefone"));  
            cliente.setCpf(resultado.getString("cpf"));  
            cliente.setRg(resultado.getString("rg"));  
        }  
    } catch (SQLException e) {  
        System.out.println("Erro ao realizar Select: " + e.getMessage());  
    }  
    return cliente;  
}
```

CRUD UPDATE

```
public void alterar(Cliente entidade) {  
    StringBuilder sql = new StringBuilder();  
    sql.append("UPDATE cliente SET ");  
    sql.append("endereco = ?, " );  
    sql.append("nome = ?, " );  
    sql.append("telefone = ?, " );  
    sql.append("cpf = ?, " );  
    sql.append("rg = ? " );  
    sql.append("WHERE identificador = ?");  
  
    try {  
        PreparedStatement consulta = conexao.prepareStatement(sql.toString());  
        consulta.setString(1, entidade.getEndereco());  
        consulta.setString(2, entidade.getNome());  
        consulta.setString(3, entidade.getTelefone());  
        consulta.setString(4, entidade.getCpf());  
        consulta.setString(5, entidade.getRg());  
        consulta.setLong(6, entidade.getIdentificador());  
        consulta.executeUpdate();  
  
    } catch (SQLException e) {  
        System.out.println("Erro ao realizar Update: " + e.getMessage());  
    }  
}
```

CRUD DELETAR

```
public void remover(Cliente entidade) {  
    String sql = "DELETE FROM cliente WHERE identificador = ?";  
  
    try {  
        PreparedStatement consulta = conexao.prepareStatement(sql);  
        consulta.setLong(1, entidade.getIdentificador());  
        consulta.executeUpdate();  
    } catch (SQLException e) {  
        System.out.println("Erro ao realizar Delete: " + e.getMessage());  
    }  
}
```

HTML5

- Novos atributos de formulários, como number, date, time, search, e range.



- Novos elementos de semântica como <header>, <footer>, <section> e <nav>

- Novos elementos gráficos e de multimédia

HTML5



HTML	<div id="header">
	<div id="nav">
	<div class="article">
	<div class="section">
	<div id="sidebar">
	<div id="footer">

HTML5	<header>
	<nav>
	<article>
	<section>
	<aside>
	<footer>

Tags Semântica

Article:	Aside:	Footer:	Header:	Nav:	Section:
Este é um elemento autocontido. Pode ser um post, um artigo entre outros!	Este elemento indica um seção do documento com conteúdo relacionado. Pode ser uma sidebar!	Este elemento representa um rodapé para o elemento pai, ou mais próximo a ele.	Este é um elemento de cabeçalho. Pode ser utilizado para logo, navegação, pesquisa e outros.	Este elemento indica uma navegação. Pode ser utilizado para apontar os principais blocos de navegação do site.	Este elemento representa uma seção genérica de conteúdo do site. Realizando um agrupamento temático.

Esta nova versão traz consigo importantes mudanças quanto ao papel do HTML no mundo da Web, através de novas funcionalidades como semântica e acessibilidade. Sua essência tem sido melhorar a linguagem com o suporte para as mais recentes multimídias, enquanto a mantém facilmente legível por seres humanos e consistentemente compreendida por computadores.

Em particular, HTML5 adiciona várias novas funções sintáticas. Elas incluem as tags de `<video>`, `<audio>` e elementos `<canvas>`. Novos elementos, como `<section>`, `<article>`, `<header>` e `<nav>`, são projetados para enriquecer o conteúdo semântico dos documentos. Novos atributos têm sido introduzidos com o mesmo propósito, enquanto alguns elementos e atributos têm sido removidos. Alguns elementos, como `<a>`, e `<menu>` têm sido mudados, redefinidos ou padronizados.

CSS Container e Background Imagem

The screenshot shows a web application interface with a header, a central container, a body section, and a footer. Arrows point from specific CSS rules to their corresponding visual elements:

- header{ background-image: url(..../imagens/header-back.jpg); clear: both; opacity: 0.8; padding: 20px 0; margin-bottom: 10px; }** points to the header bar.
- .container{ position: relative ; margin: 0 auto; width: 1200px; }** points to the main content area.
- body{ background-image: url(..../imagens/background.jpg); }** points to the body section below the header.
- footer{ background-image: url(..../imagens/fundo-rodape2.png); }** points to the footer area at the bottom of the page.

Uma regra CSS é uma declaração que segue uma sintaxe própria e que define como será aplicado estílo a um ou mais elementos HTML. Um conjunto de regras CSS formam uma Folha de Estilos. A estrutura de uma regra CSS compõe-se de três partes: um seletor, uma propriedade e um valor, e tem a seguinte sintaxe:

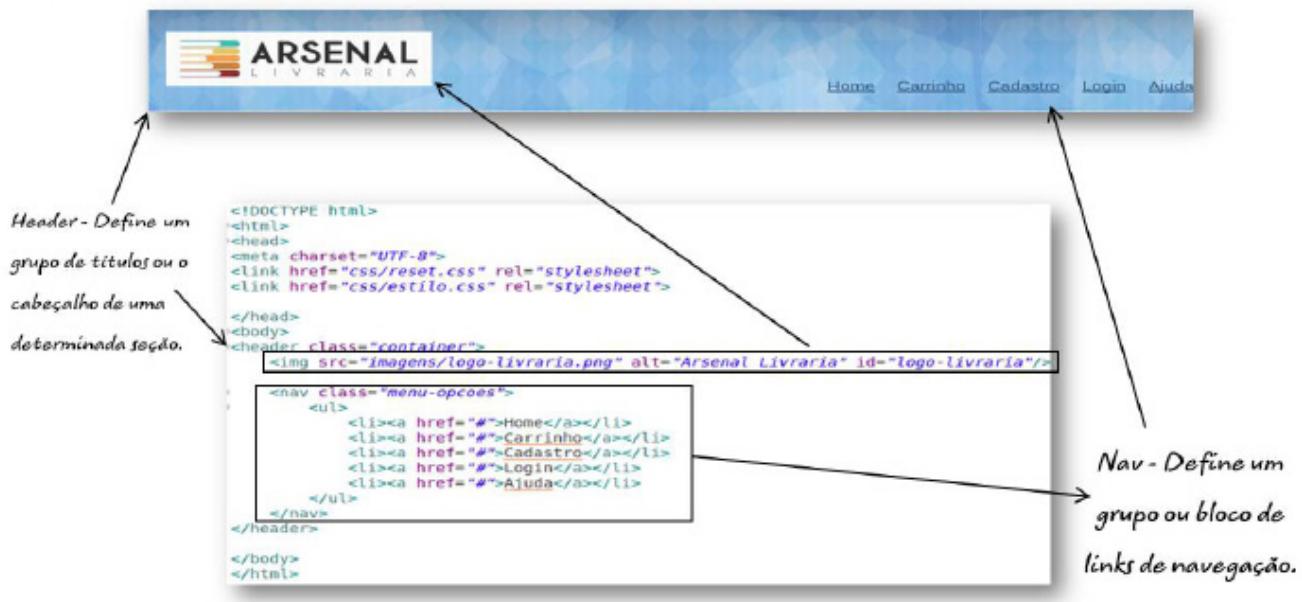
seletor { propriedade: valor; }, onde o Seletor é o elemento HTML identificado por sua tag (body), ou por uma classe (.container), ou por uma ID (#logo-livraria), e para o qual a regra será válida; a Propriedade é o atributo do elemento HTML ao qual será aplicada a regra (margin, width, height); e o Valor é a característica específica a ser assumida pela propriedade. Como exemplo, temos a class="container" :

```
.container{
margin: 0 auto;
width: 1200px;
}
```

Repare nas diferenças entre a imagem de cabeçalho e a de rodapé. A tag header está associada ao seletor “container”, tendo assim suas limitações laterais, o que não ocorre com a tag footer, mas com alguns elementos que se encontram nela.

Podemos usar o CSS também para acrescentar uma imagem de fundo (background-image) na página toda ou somente no local desejado, como fizemos com o fundo da nossa página, o cabeçalho e o rodapé. Como o valor da propriedade de imagem de fundo, devemos especificar o caminho da imagem desejada.

Elementos de Cabeçalho



Como é necessário incluir o mesmo cabeçalho, o rodapé e uma barra de menu lateral em varias páginas, cada um deles foi criado um html separado, podendo assim ser incluído em qualquer página com somente uma linha de comando.

No código, separamos o cabeçalho do corpo da página simplesmente com a tag semântica `<header>` (cortesia do HTML 5), que representa um grupo de suporte introdutório ou navegacional.

Logo no começo é disposto uma imagem da logo da loja com a tag ``, contendo as propriedades `src` (source), responsável por informar onde a imagem se encontra; `alt` (alternative), que define o texto que será informado caso a imagem não apareça; e `id`, que nos oferece a total liberdade de alterar a altura, largura, disposição da imagem, tudo através de regras CSS no arquivo `estilo.css`.

Ao extremo lado direito do cabeçalho, temos a tag `<nav>` que é nada mais que uma seção de links de navegação, nesse caso, nos fornecendo a navegação a outras páginas, como a página inicial “home”. Classificada pela class =”menu-opcoes”, a lista de links de navegação dispõe do arquivo `estilo.css` para organização e design.

Elementos de Rodapé

The diagram shows the footer section of a web page. It includes a logo ('ARSENAL LIVRARIA'), a social media link section ('social'), and a copyright notice ('Todos os direitos reservados'). Arrows point from the text labels to their corresponding parts in the code and visual representation.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<link href="css/reset.css" rel="stylesheet">
<link href="css/estilo.css" rel="stylesheet">
</head>
<body>
<div class="container">

<ul class="social">
<li><a href="http://facebook.com/">Facebook</a></li>
<li><a href="http://twitter.com/">Twitter</a></li>
<li><a href="http://plus.google.com/">Google+</a></li>
</ul>
</div>
</body>
</html>

```

Footer - Define o rodapé das seções ou da página.

social a

social a{background-image: url(../imagens/facebook.png);}

social a{background-image: url(../imagens/twitter.png);}

social a{background-image: url(../imagens/googleplus.png);}

social a{height: 32px; width: 32px; display: block; text-indent: -9999px;}

Já no arquivo rodape.html, temos a tag <footer> para dividir a seção de rodapé do resto do código. Temos a tag para incluir a logo da loja também no fim da página.

Diferente do cabeçalho, ao extremo lado direito do rodapé temos uma lista com vários elementos com a tag <a>, que define um hiperlink. Porém, a intenção foi trocar o hyperlink das páginas sociais pelos botões com a logo de cada página.

No css, incluímos as imagens das logos como imagem de fundo (background-image) mas na propriedade indentação de texto (text-indent) ao setá-lo em um número infinitamente negativo (-9999px), o hiperlink passa para atrás da página, deixando as imagens na frente com a mesma funcionalidade. Para acertar o link na imagem, é setado no css a altura (height) e largura (width) do hiperlink para o tamanho certo da imagem (32px), ou seja, todo o espaço da imagem funcionará como um hiperlink.

Barra Lateral

```

<!DOCTYPE html>
<html>
<body>
<div class="container_destaque">
<section class="busca">
<h2>Busca</h2>
<form action="Pesquisa" id="form-busca">
<input type="search" name="titulo" id="q">
<input type="image" src="imagens/busca.png">
</form>
</section>
<section class="menu-departamentos">
<h2>Departamentos</h2>
<nav>
<ul>
<li><a href="#">Livros</a>
<ul>
<li><a href="#">Auto-ajuda</a></li>
<li><a href="#">Bibliografias</a></li>
<li><a href="#">Teen</a></li>
<li><a href="#">Romance</a></li>
<li><a href="#">Ficção</a></li>
<li><a href="#">Gibis e HQs</a></li>
</ul>
</li>
<li><a href="#">Filmes</a></li>
<li><a href="#">Games</a></li>
<li><a href="#">Música</a></li>
<li><a href="#">Acessórios</a></li>
</ul>
</nav>
</div>
</body>
</html>

```

Section - Define um grupo ou bloco de por seção de um assunto específico

Barra Lateral

```

<section class="menu-departamentos">
<h2>Departamentos</h2>
<nav>
<ul>
<li><a href="#">Livros</a>
<ul>
<li><a href="#">Auto-ajuda</a></li>
<li><a href="#">Bibliografias</a></li>
<li><a href="#">Teen</a></li>
<li><a href="#">Romance</a></li>
<li><a href="#">Ficção</a></li>
<li><a href="#">Gibis e HQs</a></li>
</ul>
</li>
<li><a href="#">Filmes</a></li>
<li><a href="#">Games</a></li>
<li><a href="#">Música</a></li>
<li><a href="#">Acessórios</a></li>
</ul>
</nav>
</div>
</body>
</html>

```

```

busca,
menu-departamentos{
  float:left;
}

menu-departamentos li ul{
  display: none;
}

menu-departamentos li:hover ul{
  display: block;
}

```

Float - O painel flutua ao lado de uma imagem ou texto

Display:none - A segunda lista não irá aparecer

Hover - A lista irá aparecer somente se for invocada ao passar o mouse em cima de "Livros".

Para a construção da barra de menu ao lado esquerdo da página, foi usado a tag semântica <section>, que divide as seções de busca e o menu de departamentos da barra lateral e o elemento css float, que fará com que nossa barra flutue ao lado esquerdo da pagina.

Na seção de Busca, usamos um input do tipo busca (search), um atributo do HTML 5 que tem a mesma funcionalidade de uma caixa de texto, porem pode apresentar alguns controles novos dependendo do navegador, como no Google Chrome, aparece a letra “x”

ao lado direito do input, assim que tem o foco, para apagar o campo. Ao lado do campo de texto, temos um input do tipo imagem (image) , que define uma imagem no lugar do botão submit, porém com a mesma funcionalidade de submeter o formulário.

Já na seção de Departamentos, utilizamos a tag <nav> para estabelecer os links de navegação e em seguida, organizamos esses links em uma lista não ordenada (- “unordered list”). Veja que dentro de um dos elementos da lista, podemos acrescentar outra lista. No nosso caso, acrescentamos uma lista dentro do item livros e ocultamos essa lista, para que seja visualizada somente se o usuário passar o mouse por cima do item Livros , ação chamada de HOVER.

JavaScript e Janelas de Alerta

```

<section class="busca">
    <h2>Busca</h2>
    <form onsubmit=" return buscarProduto(titulo)">
        <input type="search" name="titulo">
        <input type="image" src="imagens/busca.png">
    </form>
</section>

<script src="js/busca.js"></script>

localhost:8080 diz:
Busca Vazia! Procure entre nossas seleções de LIVROS, FILMES, GAMES e MÚSICAS!
OK

localhost:8080 diz:
Branca de Neve,A Bela Moça,O Guia,Dr. Ozzy,Pequeno Irmão
OK

function buscarProduto(titulo){
    var livros = ["Branca de Neve", "A Bela Moça", "O Guia", "Dr. Ozzy", "Pequeno Irmão"];
    var filmes = ["Deadpool", "Godzilla", "Lagoa Azul", "Loucademia de Polícia"];
    var musicas = ["Ex Top", "Black Sabbath", "Lorranna Santos"];
    var games = ["Mario Kart", "Fifa 17", "Battlefield 4", "Undertale", "SuperHot"];
    if(titulo.value == ""){
        alert("Busca Vazia! Procure entre nossas seleções de LIVROS, FILMES, GAMES e MÚSICAS!");
    }
    else if(titulo.value=="livros" || titulo.value=="livro"){
        window.alert(livros);
    }
    else if(titulo.value=="filmes"){
        window.alert(filmes);
    }
    else if(titulo.value=="musicas"){
        window.alert(musicas);
    }
    else if(titulo.value=="games"){
        window.alert(games);
    }
    else{
        window.alert("Procure entre nossas seleções de LIVROS, FILMES e MÚSICAS!");
    }
}

```

The diagram illustrates the interaction between an HTML form and a JavaScript function. The form has an 'onsubmit' event that calls the 'buscarProduto' function. This function contains logic to check if the input field is empty or contains specific search terms ('livros', 'filmes', 'musicas', 'games'). Depending on the input, it displays an alert window containing a list of items from an array. Two alert windows are shown: one for an empty search and another for a valid search term like 'livros'.

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.

O JavaScript pode ser usado para fazer cálculos matemáticos, monitorar o horário, enfim, funções simples de java (scripts) que podem ser executados do lado do cliente e interagir com o usuário sem a necessidade deste script passar pelo servidor, controlando o navegador, realizando comunicação assíncrona e alterando o conteúdo do documento exibido.

No caso do nosso projeto, fizemos uso de uma função js (JavaScript) que irá abrir uma janela de alerta de acordo com o que será pesquisado na seção de busca da página. Para ficar mais organizado, criamos um documento busca.js fora do nosso documento html, mas que foi chamado pela tag <script> (É sempre necessário chamar o documento js se ele for criado fora do html da página).

Perceba o elemento <form onSubmit=" return buscarProduto(titulo)">. Nele, chamamos a função buscarProduto() assim que submetemos a busca. O busca.js vai pegar o título da busca, ou seja, o que foi escrito no input de busca, e dependendo do que o usuário digitar, o js nos enviará uma resposta como uma janela de alerta.

Formulário Bootstrap

The diagram illustrates the mapping between a Bootstrap-based user interface and the underlying HTML code. On the left, a screenshot of a web page titled 'Cadastro' shows a form with fields for 'Nome Completo', 'Usuário' (with a validation message 'Preencha este campo.'), 'Senha', 'Repite a senha', and a 'Confirmar' button. On the right, the corresponding HTML code is shown:

```

<link href="css/bootstrap.css" rel="stylesheet">



<div class="panel-heading">Cadastro</div>
  <div class="panel-body">

    <form method="post" action="EditarCliente?acao=cadastrar">
      <div class="row">
        ${mensagem}
        <fieldset>

          <div class="form-group">
            <label for="name">Nome Completo</label>
            <input type="text" class="form-control" id="name" name="name" autofocus required>
          </div>

          <div class="form-group">
            <label for="login">Usuário</label>
            <input type="text" class="form-control" id="login" name="login" required>
          </div>

          <div class="form-group">
            <label for="senha">Senha</label>
            <input type="password" class="form-control" id="senha" name="senha" required>
          </div>

          <div class="form-group">
            <label for="senha2">Repite a senha</label>
            <input type="password" class="form-control" id="senha2" name="senha2" required>
          </div>

        </fieldset>
        <input type="submit" class="btn btn-primary" value="Confirmar">
      </form>
    </div>
  </div>


```

Annotations with arrows point from specific UI elements to their corresponding HTML code. A red arrow points from the 'Nome Completo' field to the `<input type="text" id="name" name="name" autofocus required>` line. A green arrow points from the 'Usuário' field to the `<input type="text" id="login" name="login" required>` line. A purple arrow points from the 'Senha' field to the `<input type="password" id="senha" name="senha" required>` line. A black arrow points from the 'Repite a senha' field to the `<input type="password" id="senha2" name="senha2" required>` line.

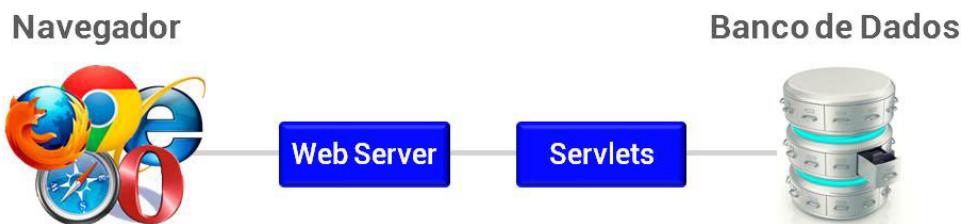
Para facilitar a criação de formulários, utilizamos o bootstrap, uma ferramenta CSS Framework que permite a criação fácil e rápida de uma interface web, sem precisar escrever uma linha CSS sequer.

Na nossa página de cadastro, usamos a tag `<form>` para empacotar as informações do formulário e enviar para outra página assim que confirmar com um botão no final do `<form>`. Para pegar essas informações, utilizamos vários campos de texto, porém na parte de senha o html já me oferece um tipo password de input que esconde o que foi escrito na caixa de texto, para proteger a senha.

O bootstrap aqui é usado como o design desse formulário. As classes `panel` e `panel-primary` são classes existentes no `bootstrap.css` que já definem o formato e as cores do painel que envolve o formulário. `panel-heading` define o título do painel e `panel-body`, a estrutura interior.

Já no corpo do painel, cada etiqueta e caixa de texto faz uso da classe `form-group`, uma classe do bootstrap criada para organizar os campos de texto do formulário em um padrão de espaçamentos. Os inputs, cada um tem a classe `form-control` para organizar os tamanhos, margens, efeitos das caixas de texto. Além da classe, foram incluídas regras como `required` e `autofocus`, que definem que o campo não pode ficar vazio e ao entrar na pagina, o campo já será selecionado automaticamente.

Servlets e JSP



Servlets

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

São tipicamente usados no desenvolvimento de sites dinâmicos, onde algumas páginas são construídas no momento do atendimento de uma requisição HTTP, sendo possível criar páginas com conteúdo variável, de acordo com perfil do usuário ou a partir de informações armazenadas em um banco de dados. Também são utilizadas para gerenciar informações de estado que não existem no protocolo HTTP, como retirar ou inserir produtos de um carrinho de compras de um cliente.

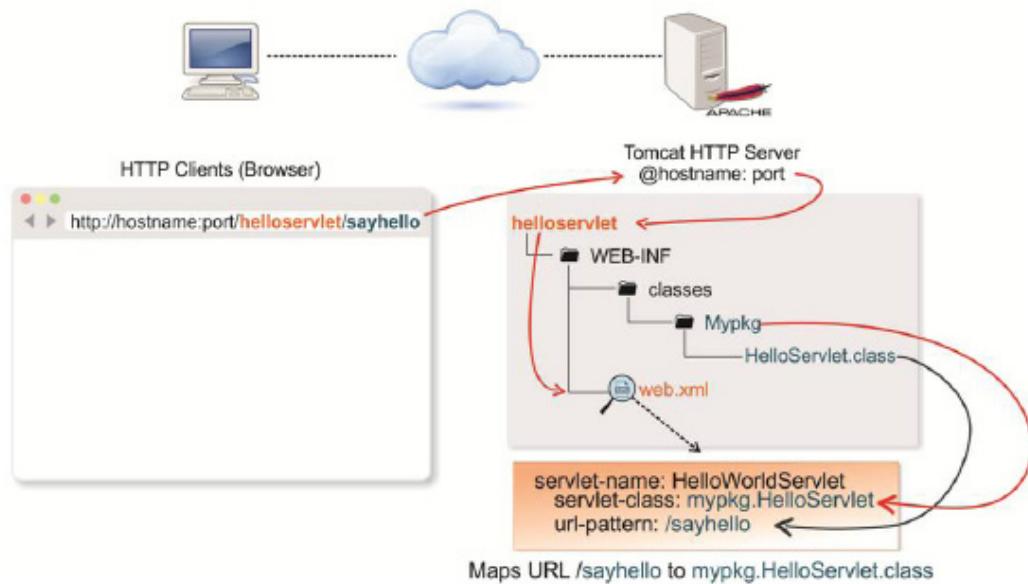
O HTTP é um protocolo **stateless (sem estado)**, que permite comunicação cliente-servidor.

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

Existem diversos métodos HTTP que podem ser especificados em requisições, sendo os métodos GET e POST os mais comuns. GET normalmente é utilizado para obter conteúdo de um arquivo no servidor enquanto POST é utilizado para enviar dados de formulários HTML ao servidor. Além desses métodos, o protocolo HTTP versão 1.1 admite os seguintes métodos:

- **HEAD:** permite que o cliente obtenha somente os headers da resposta (informações sobre a entidade requisitada).
- **PUT:** transfere um arquivo do cliente para o servidor (editar informações de um recurso).
- **DELETE:** remove um arquivo do servidor (remove um recurso).
- **OPTIONS:** obtém a lista dos métodos HTTP suportados pelo servidor.
- **TRACE:** retorna o conteúdo da requisição enviada de volta para o cliente.

Estrutura de uma Aplicação Web



2

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

Servlets dependem de um contêiner de execução, mas sua utilização não de uma única implementação de 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, Javascripts e outros a mais. Estes recursos podem acessados pelo usuário diretamente através do navegador web.

No diretório WEB-INF estão todas as classes da aplicação e armazenamento de código java (JavaBeans, servlets, entre outros). É criado por convenção e padrão da especificação o diretório “classes”, as classes da sua aplicação (*.class e *.java) e seus packages (pacotes), devem estar presentes nesse diretório.

O arquivo web.xml, denominado deployment descriptor, descreve a aplicação segundo a especificação de Servlets e JavaServer pages. Veremos mais detalhes sobre esse arquivo logo a frente.

As bibliotecas (pacotes .jar) são colocados dentro do diretório WEB-INF/lib, por padrão.

Criação de uma Servlet

Devemos anotar a classe com `@WebServlet` para indicar qual a URL

```
→ @WebServlet("/HelloServlet")
  public class HelloServlet extends HttpServlet {
```

Uma classe servlet deve estender de `HttpServlet`

5

O comportamento dos servlets que iremos ver 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 a seguir:

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

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

Implementação de uma classe Servlet

```
@WebServlet("/ServletExemplo")
public class ServletExemplo extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public ServletExemplo() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
     *      response)
     */
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
     *      response)
     */
    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }
}
```

4

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.

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 do 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.

Mapeamento no web.xml (versões < 3.0)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>Hello</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/HelloServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Nome do servlet

Caminho para a classe servlet

URL de acesso

Anotação @WebServlet faz a mesma coisa

Deployment Descriptor

É utilizado para informar ao contêiner os servlets a serem executados, ele permite mapear uma URL pública, conhecida pelo seu cliente, para 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, 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 deployment descriptor.

<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

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

Mapeamento de Servlets no web.xml

Após a criação de um servlet, precisamos identificar a classe como um servlet. Para isso podemos mapear no web.xml através das seguintes tags:

```
<servlet>
    <servlet-name>  </servlet-name>
    <servlet-class>  </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>  </servlet-name>
    <url-pattern>  </url-pattern>
</servlet-mapping>
```

Assim, quando acessarmos a página que utiliza esse servlet, teremos certeza que as informações e ações a serem executadas serão feitas pelo servlet correto.

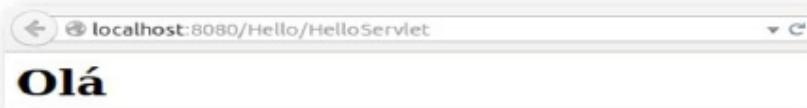
A partir da versão 3.0 de servlets, podemos fazer o mesmo mapeamento de uma forma mais simples: através de anotações no código. Para isso, basta anotarmos a classe servlet com a anotação **@WebServlet(name, urlPatterns, asyncSupported, initParams)**.

```
@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
    ...
}
```

Implementação

O método doGet é responsável por manipular requisições GET.

```
@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>" +
                    "<head><title>Hello Servlet</title></head>" +
                    "<body>" +
                    "<h1>Olá</h1>" +
                    "</body></html>");
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    }
}
```



Interface HttpServletRequest

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

Veja no exemplo uma página HTML que permite ao usuário enviar dois parâmetros ao servidor:

```
<form action="CaminhoDoServlet" method="post">
    <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>
    <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 para o servidor usando uma requisição HTTP POST.

Uma vez que a requisição tenha sido enviada ao servidor, o servlet que está mapeado no web.xml é invocado. Veja como fica o método `doPost` do servlet:

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    //retorna o 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 getParameters() 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.

Interface HttpServletResponse

Usada para enviar respostas ao cliente, normalmente texto HTML, usando os objetos Writer ou OutputStream obtidos através dos métodos getWriter() e getOutputStream(), respectivamente.

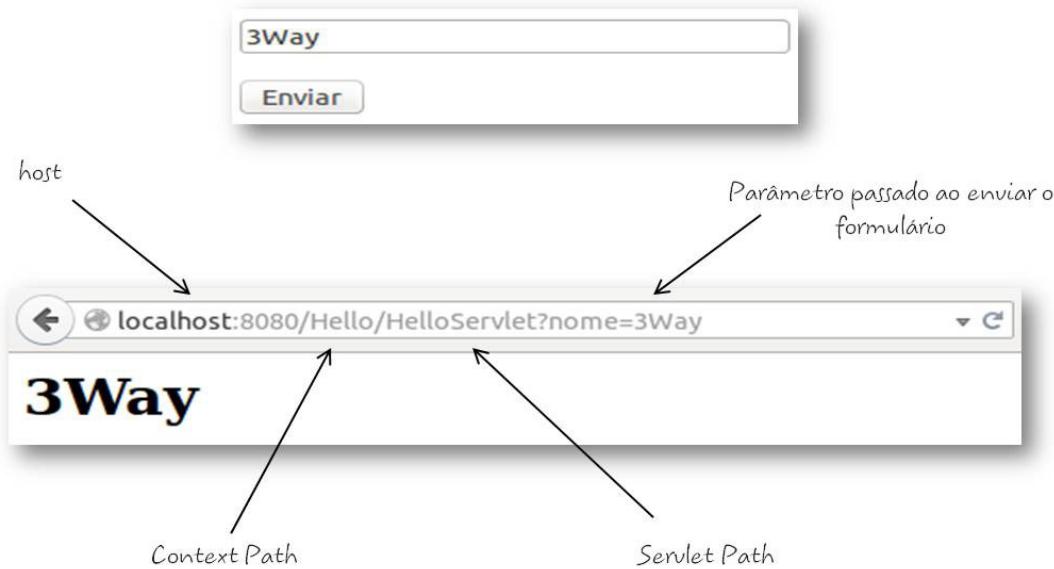
A interface ServletResponse fornece métodos relevantes para qualquer protocolo, enquanto HttpServletResponse estende ServletResponse e adiciona métodos específicos para o protocolo HTTP, ServletResponse declara vários métodos genéricos, incluindo getWriter(), getOutputStream(), setContentType entre outros.

No exemplo, o método getWriter() retorna 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 utilizadas para adicionar Strings ao stream de saída do Servlet; como a saída é mantida em 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 gerar páginas HTML dinamicamente ou enviar qualquer outro recurso aceito pelo navegador do cliente.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    String nome = request.getParameter("nome");
    out.println("<html>");
    out.println("<body>");
    out.println("<h1>" + nome + "<h1>");
    out.println("</body>");
    out.println("</html>");
}
```

Recebendo Parâmetros



Invocando um servlet

O servlet tem um nome público de URL – nome que o usuário conhece. Ou seja, o nome codificado no HTML de modo que, quando o usuário clicar em um link sua navegação será redirecionada a aquele servlet, esse nome público é enviado ao servidor na requisição HTTP.

Rotear a requisição para um servlet é um processo feito pelo servlet contêiner em duas etapas. Primeiro 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:

1. Caminho contexto (context path)
2. Caminho do Servlet (servlet path)
3. Informação de caminho (path info)
4. String Query, tudo que vem depois do caracter “?”

A estrutura da URI ficaria desta forma:

`http://host/contextPath/servletPath/pathInfo`

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 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. No exemplo 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 após a determinação do servlet path é chamado de path info. No exemplo da URL /banco/ContaServlet/pessoafisica, então /pessoafisica é o path info.

Query String – tudo o que vier após o caracter “?”, na URL

/banco/ContaServlet?nome=Elizeu&nascimento=10/12/2001&rg=2003456

a query string contem um ou mais conjutos de chave e valor associados separados pelo caracter “&”. No exemplo ([nome=>Elizeu], [nascimento=>10/12/2001], [rg=>2003456]).

Dados de Formulário

localhost:8080/Hello/HelloServlet?nome=3Way

3Way

```

@.WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String nome = request.getParameter("nome");
        out.println("<html>" +
                    + "<body>" +
                    + "<h1>" + nome + "</h1>" +
                    + "</body></html>");
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    }
}
  
```

The screenshot shows a Java Servlet code snippet for handling GET requests. It prints an HTML page with a single input field named "nome" and a submit button labeled "Enviar". The browser window displays the resulting page with the title "3Way" and the name "3Way" entered into the input field.

Lendo Formulários de Dados usando Servlet

Servlets stratam dados vindos de formulários Html automaticamente usando um dos seguintes métodos de HttpServletRequest, dependendo da situação:

No exemplo o parâmetro nome foi recuperado pela invocação do método **request.getParameter()**, para obter o valor digitado no formulário html.

Já o método getParameterValues() é usado para obter parâmetros que aparecem mais de uma vez no formulário como valores retornados de campos CheckBox ou Multi-Select, como nesse formulário:

```

<form action="CheckBox" method="POST" target="_blank">
<input type="checkbox" name="materia" checked="checked" />Matematica
<input type="checkbox" name="materia" />Fisica
<input type="checkbox" name="materia" checked="checked" />
Quimica
<input type="submit" value="Select Subject"/>
<form/>
  
```

no exemplo a invocação de `request.getParameterValues("materia")` retorna o array de String {"Matematica", "Quimica"}

Ainda temos `getParameterNames()`, para recuperar todos os nomes de parâmetros enviados pelo formulário html.

Expression Language \${ expressão }

Operador	Função
.	Acessar alguma propriedade
[]	Acessar algum elemento de uma lista
()	Agrupamento de subexpressões
+	Adição
-	Subtração
*	Multiplicação
/ ou div	Divisão
% ou mod	Resto
== ou eq	Igualdade
!= or ne	Diferença
< ou lt	Menor que
> ou gt	Maior que
<= ou le	Menor ou igual a
>= ou ge	Maior ou igual a
&& ou and	E lógico
ou or	OU lógico
! ou not	Boleano
empty	Testa por variáveis vazias

Expression Language (EL) é uma linguagem de script que permite o acesso a atributos salvos em todos os escopos JSP (a partir da versão 2.0) sem a necessidade de inserir scriplets de código java nas mesmas.

Por exemplo, se você invocar a página `parametros.jsp?nome=3way`, você pode recuperar o valor do parâmetro nome usando a expressão `${param.nome}` ou `${param['nome']}`), compare com expressão em scriplet JSP `<% request.getParameter('nome') %>` bem mais simples e menos trabalhoso de se escrever.

A sintaxe básica da EL no JSP é **\${ expressão }**, onde expressão é o componente/atributo a ser resgatado do escopo Servlet. O caracter (\$) é usado em JSP enquanto o caracter (#) é usado com JSF.

A EL define os seguintes literais:

Boolean: true e false.

Integer: 42, -5 (como em Java)

String: “uma string” com aspas simples ou duplas;
 aspas duplas (“ ”) é escapada com \”, exemplo \${“uma aspa duplo \” dentro da string”}
 e aspas simples (‘ ’) com \’, exemplo \${‘uma aspa simple \’ dentro da string’}
 e \ é escapada com \\ , , exemplo \${‘uma barra \\ dentro da string’}

Null: null.

Acessando variáveis

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

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 do ponto deve ser uma propriedade do bean ou uma chave do Map e deve obedecer aos padrões de nomenclatura de variáveis em Java.

Veja exemplos:

```
 ${pessoa.nome} //atributo do bean pessoa invoca getName()
 ${map.chave} //chave do map
 ${pessoa.1} //não funciona
```

Operador []

Este operador é o 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:

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

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:

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

Expression Language \${ expressão }

Números

Expressão	Resultado
<code>\$(1 < 2)</code>	true
<code>\$(1 lt 2)</code>	true
<code>\$(1 > (4/2))</code>	false
<code>\$(4.0 >= 3)</code>	false
<code>\$(4.0 ge 3)</code>	true
<code>\$(4 <= 3)</code>	true
<code>\$(4 <= 3)</code>	false
<code>\$(4 le 3)</code>	false
<code>\$(100.0 == 100)</code>	true
<code>\$(100.0 eq 100)</code>	true
<code>\$((10*10) != 100)</code>	false
<code>\$((10*10) ne 100)</code>	false

Alfanuméricos

Expressão	Função
<code>\$('a' < 'b')</code>	true
<code>\$('4' > '3')</code>	true

Expressão	Resultado
<code>\$(1)</code>	1
<code>\$(1.2 + 2.3)</code>	3.5
<code>\$(1.2E4 + 1.4)</code>	12001.4
<code>\$(-4 -2)</code>	-6
<code>\$(21 * 2)</code>	42
<code>\$(3/4)</code>	0.75
<code>\$(3 div 4)</code>	0.75
<code>\$(3/0)</code>	Infinity
<code>\$(10%4)</code>	2
<code>\$(10 mod 4)</code>	2
<code>\$((1==2) ? 3:4)</code>	4

Expression Language \${ expressão }

Objeto Implícito	Mapeia
pageScope	Atributo no escopo da página
requestScope	Atributo no escopo da requisição
sessionScope	Atributo no escopo da sessão
param	Parâmetro da requisição
paramValues	Parâmetro da requisição
header	Cabeçalho da requisição
headerValues	Cabeçalho da requisição
cookie	Cookie
initParam	Parâmetro de inicialização de contexto

Alterar Parâmetro

foo =

EL Expression	Result
<code>\$(param.foo)</code>	3way
<code>\$(param["foo"])</code>	3way
<code>\$(header["host"])</code>	localhost:8080
<code>\$(header["accept"])</code>	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
<code>\$(header["user-agent"])</code>	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Firefox/39.0

Os objetos implícitos já estão instanciados e prontos para serem usados. Os objetos implícitos existentes:

- **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.
- **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).

Página JSP

```

<% page language="java" contentType="text/html; charset=UTF-8" %>
<% page import="java.util.Date" %>
<!DOCTYPE html>
<%!
    int contador = 0;
    Date agora;
%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSP Essencial</title>
</head>
<body>
    <form method="get">
        <label for="nome">Nome</label>
        <input type="text" id="nome" name="nome" size="25">
        <label for="numero">Número</label>
        <input type="text" id="numero" name="numero" size="10">
        <input type="submit" value="Enviar">
    </form>
    <%
        String pNome = request.getParameter("nome");

        if(pNome != null && ! pNome.isEmpty()){
            contador++;
            agora = new Date();
%>
        <h3>Número: ${param.numero}</h3>
        <h2>${param.nome}</h2>
        <h4>${contador} em ${agora}</h4>
    <% } %>
    <jsp:include page="rodape.html" />
</body>
</html>

```

Directive: A tag <%> que define as regras para a página JSP.

Declaração: A tag <%!> que define variáveis e métodos para serem usados na página.

Scriptlet: São pedaços de código Java em sua página JSP. Deve estar entre as tags <%>.

Expressão EL: Expressões de linguagem de script, usadas para inserir valores dinâmicos no HTML.

Expressão JSP: Expressões JSP, usadas para inserir valores dinâmicos no HTML.

Ação JSP: Ação JSP, usada para incluir conteúdo de outra página.

JSP é uma linguagem de script com especificação aberta que tem como objetivo facilitar geração de conteúdo dinâmico para páginas da internet, mesclando código HTML estático com códigos JSP embutidos.

Uma página JSP possui extensão de arquivo “.jsp”, você mescla HTML com codificação Java inserindo as tags <%>, denominada scriptlets. Dessa forma o container Java Web recebe uma requisição para uma página JSP, interpreta esta página gerando a codificação HTML e retorna ao cliente o resultado de sua solicitação. Scriptlets permitem 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.

```

<%
    out.println("Isso é um Scriptlet");
    out.println("Podemos adicionar qualquer código Java!");
%>

```

Uma vez que você tem desenvolvido as páginas JSP basta disponibilizá-las no Servlet Container (Tomcat, por exemplo). O trabalho restante será realizado pelo servidor que faz a tradução e compilação em tempo execução, e disponibiliza o servlet gerado.

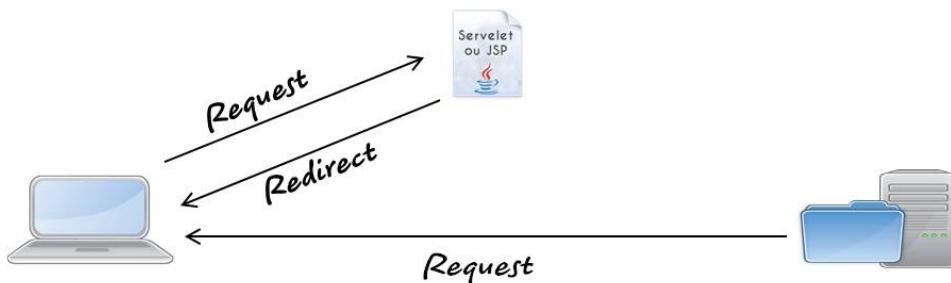
NAVEGAÇÃO

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, verifica se o usuário já está cadastrado em sua base de dados de clientes, se não estiver, ele invoca o servlet CadastroClienteServlet, que ao encerrar cadastro encaminha requisição novamente para FinalizaCompraServlet, que confirma o usuário e finaliza a compra emitindo fatura pedido.

Redirecionar (redirect)



```
Public [doPost | doGet] (...) throws ...{  
  
    response.sendRedirect("http://www.google.com.br");  
    return;  
  
}
```

Redirecionamento

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.

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 redirecionamento para uma página HTML. Esse método envia um código 304 e header `location`, do protocolo HTTP, de volta para o cliente informando que o recurso foi transferido para outra URL e o web browser envia uma nova requisição para a URL informada.

Redirecionando response para um servlet:

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

Redirecionando response para um servidor web diferente:

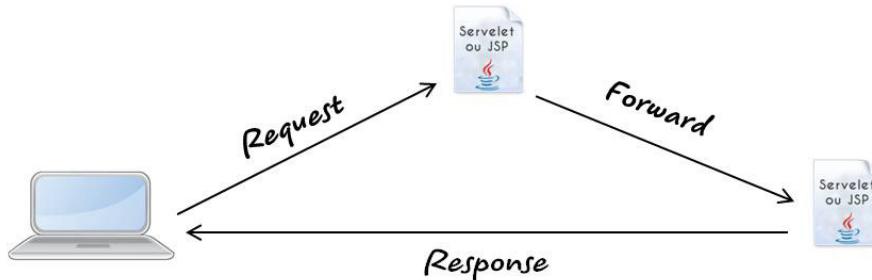
```
response.sendRedirect("http://www.oracle.com");
```

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 enviada ao navegador. Se você fizer a chamada sob esta condição, o método irá disparar um exceção do tipo **`java.lang.IllegalStateException`**.

```
protected void doGet(HttpServletRequest request, HttpServletResponse)
    throws ServletException, IOException {

    PrintWriter pw = response.getWriter();
    pw.println("<html><body>Dispara exception</body></html>");
    //envia resposta
    pw.flush();
    //Tenta redirecionar
    response.sendRedirect("http://www.apache.org");
}
```

Encaminhar (forward)



```
Public [doPost | doGet] (...) throws ...{  
    Request.getRequestDispatcher("/resultListagem.jsp")  
        .forward(request, response);  
}
```

Reencaminhamento

Diferente de sendRedirect(), no reencaminhamento, a requisição é encaminhada diretamente para a nova URL mantendo todos os objetos associados e evitando uma volta ao web browser. O uso de reencaminhamento é mais eficiente do que o uso de redirecionamento. O reencaminhamento é obtido usando o método forward() de uma instância 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)  
        throws ServletException, IOException {  
  
        ServletContext sc = getServletContext();  
        RequestDispatcher rd = sc.getRequestDispatcher("outroServletMapeado");  
        rd.forward(request, response);  
    }  
}
```

Redirect versus Forward

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

Cookies

Cria um novo cookie com o valor do parâmetro nomeCookie (nomeDoCookie, valor)

```
@WebServlet("/CookieServlet")
public class CookieServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
    }

    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        Cookie novoCookie = new Cookie("nome", request.getParameter("nome"));
        novoCookie.setMaxAge(60 * 60 * 24);
        response.addCookie(novoCookie);
        response.sendRedirect("cookies.jsp");
    }
}
```

Adiciona o cookie na resposta HTTP

Informa o tempo
máximo de vida do cookie

Em muitos sites existentes na internet, as configurações principais, como idioma e modo de exibição, são mantidos 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 computador armazena dados para um endereço web específico.

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 navegador do usuário. Posteriormente, a cada requisição enviada, o navegador anexa também as informações desses cookies, permitindo que o servidor recupere os valores definidos anteriormente.

Enquanto o cookie estiver salvo em seu computador, 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 o protocolo TTP, uma vez que este protocolo, como já dissemos, é em estado ou não orientado a 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 de 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 navegador do usuário estiver configurado para não aceitar cookies. Usuários preocupados com sua privacidade normalmente bloqueiam o armazenamento de cookies por seus navegadores. Podemos citar os seguintes motivos:

- 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);

A classe javax.servlet.http.Cookie

Manipulamos um cookie através de instâncias de 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
String getName()	Retorna o nome do cookie.
String getValue()	Retorna o valor armazenado no cookie
String getDomain()	Retorna o servidor ou domínio do qual o cookie pode ser acessado.
Boolean getSecure()	Indica se o cookie acompanha solicitação HTTP ou HTTPS
Void setValue(String newValue)	Atribui um novo valor para o cookie
Void setDomain(String pattern)	Define o servidor ou domínio 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.

Cookies

```

<body>
    <%
        Cookie[] cookies = request.getCookies(); ← Retorna um array de objetos do tipo Cookie.

        for (Cookie cookie : cookies) {
            out.println("Nome do Cookie: " + cookie.getName()); ← Acessa o nome do cookie
            out.println("<br>");
            out.println("Valor do Cookie: " + cookie.getValue()); ← Acessa o valor do cookie
            out.println("<br><br>");
        }
    %>

    <table border="1">
        <thead>
            <tr>
                <th>Nome do Cookie</th>
                <th>Valor do Cookie</th>
            </tr>
        </thead>
        <tbody>
            <c:forEach var="cookies" items="${cookie}"> ← Podemos acessar a lista de cookies em
            <tr> uma página JSP através da EL.
                <td>${cookies.value.name}</td>
                <td>${cookies.value.value}</td>
            </tr>
        </c:forEach>
    </tbody>
</table>

```

Nome do Cookie: JSESSIONID
Valor do Cookie: EDF43706AEAED4C982412D217818E11A

Nome do Cookie: nome
Valor do Cookie: Novo Cookie

Nome do Cookie	Valor do Cookie
JSESSIONID	EDF43706AEAED4C982412D217818E11A
nome	Novo Cookie

Para criar um cookie, devemos seguir os seguintes passos:

Crie um objeto do tipo javax.servlet.http.Cookie.

Chame o construtor do cookie passando como parâmetros um nome do cookie e o valor, ambos strings.

Cookie c = new Cookie("nome", "valor");

Atribua o tempo de vida máximo do cookie, isso faz com que o navegador armazene o cookie em disco ao invés de apenas na memória.

c.setMaxAge(60*60*60*7); //Uma semana

Insira o cookie na resposta HTTP usando:

response.addCookie(c);

Só assim o cookie será salvo.

Para ler Cookies do cliente:

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.

Em java script, ao receber o array de cookies, ele terá a seguinte estrutura:

“chave1=valor1;chave2=valor2;chaveN=valorN;expires=data”

Então precisamos separá-los pelos pares. Isso pode ser feito através do método split(); , depois separamos cada par pelo “=”, assim sabemos que a posição 0 do array é a chave (nome do cookie) e a posição 1 é o seu valor. Seguindo essa lógica, podemos manipular os cookies:

```
var allcookies = document.cookie;

// Pega todos os pares de cookies
cookiearray = allcookies.split(';');

// Agora separa os pares entre suas chaves e valores
for(var i=0; i<cookiearray.length; i++){
    name = cookiearray[i].split('=')[0];
    value = cookiearray[i].split('=')[1];
    // Faz alguma coisa com o valor encontrado
}
```



Paginas de Erro

→ Adicionar diretiva <%@ page isErrorPage = "true" %>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ page isErrorPage="true"%>
<!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=UTF-8">
<title>ERRO 404</title>
</head>
<body>
<h1>Ocorreu um erro! Tente outra vez.</h1>
A página que você está tentando acessar não foi encontrada.
<br> Verifique se o endereço de acesso está correto.
</body>
</html>
```

Permitir que um usuário visualize a pilha de exceções quando algo dá errado ou uma mensagem de erro padrão 404 Not Found não será um bom cartão de visitas para sua página.

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

```
<%@ page isErrorPage="true" %>
<html>
<h1>Ocorreu um erro! Por favor, tente outra vez.</h1>
</html>
```

Diretiva page - definirá as diretivas da página. Essa diretiva permite importação de classes, customização de super classes Servlet, etc.

isErrorPage="true | false" - define se é uma página de controle de erro.

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 através da diretiva:

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

Nesse caso teremos um grande inconveniente ao utilizarmos essa abordagem, caso você tenha muitas páginas que precisam deste tratamento de erro, será necessário modificar cada uma para que contenham a diretiva isErrorPage. Se precisarmos de um tratamento diferente para exceções de tipos diferentes devemos utilizar um abordagem mais abrangente com a tag <error-page> do web.xml.

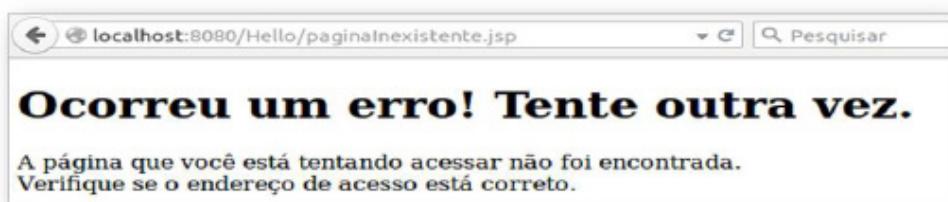
Paginas de Erro

Declare a página de erro no web.xml

```
<error-page>
    <error-code>404</error-code>
    <location>/errorPages/erro404.jsp</location>
</error-page>
```

Código do erro HTTP
Caminho para a página de erro

Acessando uma página inexistente na aplicação



É possível declarar páginas de erro no web.xml para uma aplicação web inteira, podendo 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 `<exception-type>` e `<error-code>` são usados para indicar o tipo da exceção ou código de erro HTTP que será interceptado pelo contêiner. Veja um exemplo:

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

Diretiva `page` - definirá as diretivas da página. Essa diretiva permite importação de classes, customização de super classes Servlet, etc.

`isErrorPage="true | false"` - define se é uma página de controle de erro.

```
<error-page>
    <exception-type>java.lang.NullPointerException</exception-type>
    <location>/erroNullPointer</location>
<error-page>
```

JSTL JavaServer Pages Standard Tag Library

Adicionar os jars da JSTL no build path do projeto



Download da Implementação

<http://jstl.java.net/>

usa

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

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

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

Para instalar a JSTL você precisa:

- Fazer o download da ultima versão no site <http://jstl.java.net/>
- Copiar os JARs das bibliotecas para o diretório WEB-INF/lib e adiciona-las no build path

JSTL Core Tags

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

Podemos importar outras páginas jsp (cabecalhos, rodapés).

```
<c:import url="http://www.google.com.br"/>
<c:import url="foreach.jsp"/>
```

Testa se o parâmetro nome está vazio e exibe uma mensagem.

```
<c:if test="${empty param.nome}">
    Você não preencheu o seu nome!
</c:if>
<c:if test="${not empty param.nome}">
    O seu nome é ${param.nome}
</c:if>
```

Funciona como um for em java.

```
<c:forEach var="item" begin="1" end="10">
    ${item}
</c:forEach>
```

1 2 3 4 5 6 7 8 9 10

Podemos pegar um atributo passado na sessão.

```
<table border="1">
    <tbody>
        <c:forEach items="${listPessoas}" var="pessoa">
            <tr>
                <td>${pessoa.nome}</td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

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) 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**.

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

A URI para a Core Librar é essa:

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

Internationalization Library: tags para internacionalização e formatação.

A URI para a Internationalization Library é essa:

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

Lista de tags da JSTL 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
C:set	Criação de variável
C:url	Veja adiante
C:when	Teste para o switch

C:set e c:remove

O conjunto de tags que permite configurar valores de variáveis EL ou propriedades de uma variável EL em qualquer escopo 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}" />
```

Pode ser usado 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:

```
<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" />
```

Condicional <c:if> e <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.

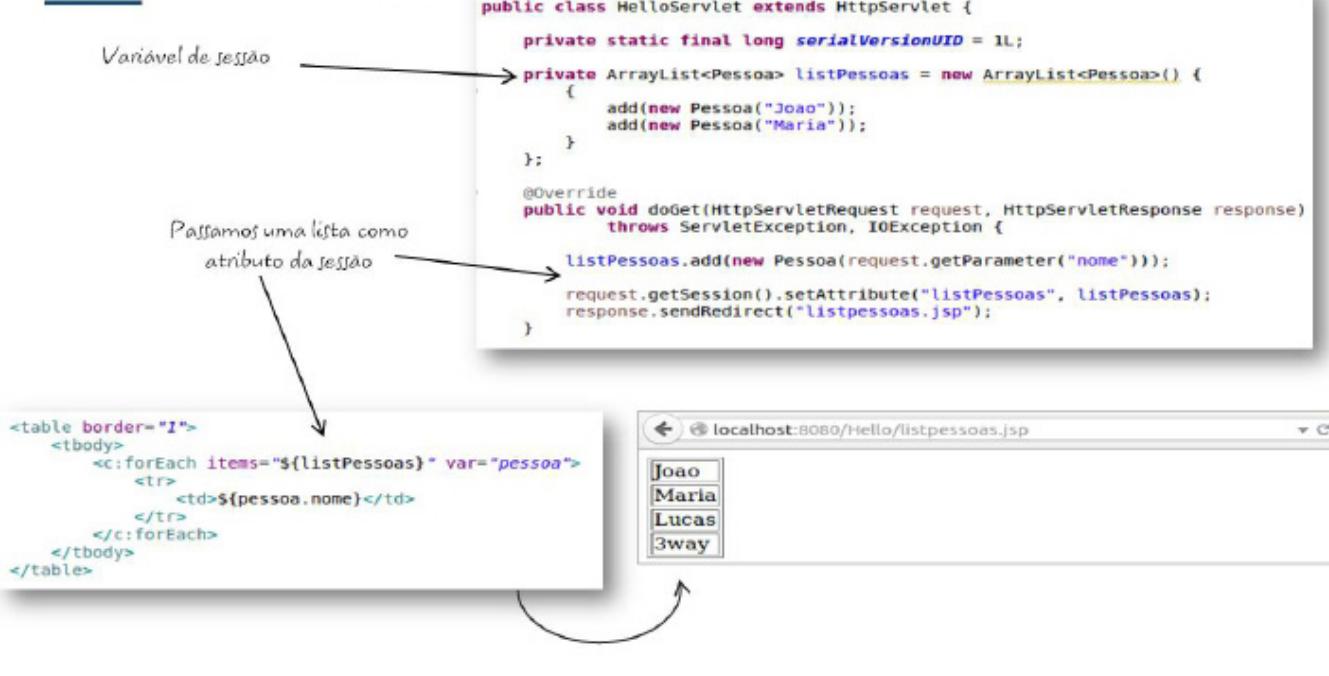
```
</c:if>
```

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

```
<c:choose>
  <c:when test="#{param.userpref eq 'volvo'}">
    A volvo é uma empresa sueca fundada em 1927
  </c:when>
  <c:when test="#{param.userpref eq 'saab'}">
    Saab Automobile AB é uma subsidiária da General Motors
  </c:when>
  <c:when test="#{param.userpref eq 'mercedes'}">
    A Mercedes-Benz é uma marca alemã de automóveis pertencente ao grupo
    Daimler AG criada em 1924.
  </c:when>
  <c:otherwise>
    Escolha um dos veículos da lista.
  </c:otherwise>
<c:choose>
```

Servlet com <c:forEach>



Iteradores <c:forEach>

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.

```

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

```

Iterador <c:forTokens>

Nessa tag, o atributo `items` é uma String constituída por tokens separados por algum delimitador. Se você imaginar uma String como uma 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="${pageScope.nomes}" var="nomeAtual" varStatus="status">
    Membro familia #<c:out value="${status.count}" /> is
    <c:out value="${nomeAtual}" /> <br/>
</c:forTokens>

```

Internacionalização de Páginas



A internacionalização de 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.

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 navegador do cliente.

```
<fmt:setLocale value="${param.lingua}" scope="session"/>
<fmt:setLocale value="pt_BR" 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 na sessão do usuário:

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

JSTL Tags de Formatação

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

Internacionalização de páginas

```

<fmt:setLocale value="en_US" scope="session"/>
<table border="1">
  <thead>
    <tr>
      <th><fmt:message key="form.pessoa.nome"/></th>
    </tr>
  </thead>
  <tbody>
    <c:forEach items="${listPessoas}" var="pessoa">
      <tr>
        <td>${pessoa.nome}</td>
      </tr>
    </c:forEach>
  </tbody>
</table>

```

Chave para uma mensagem em um arquivo .properties

Após a execução deste fragmento JSP as preferências de idioma especificadas pelo usuário para seu navegador serão ignorada

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 de dados), enquanto `<fmt:parseDate>` é usada para fazer análise de valores de datas e horas (entrada de dados).

```

<c:set var="brDateString">10/12/15 13:00 </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>

```

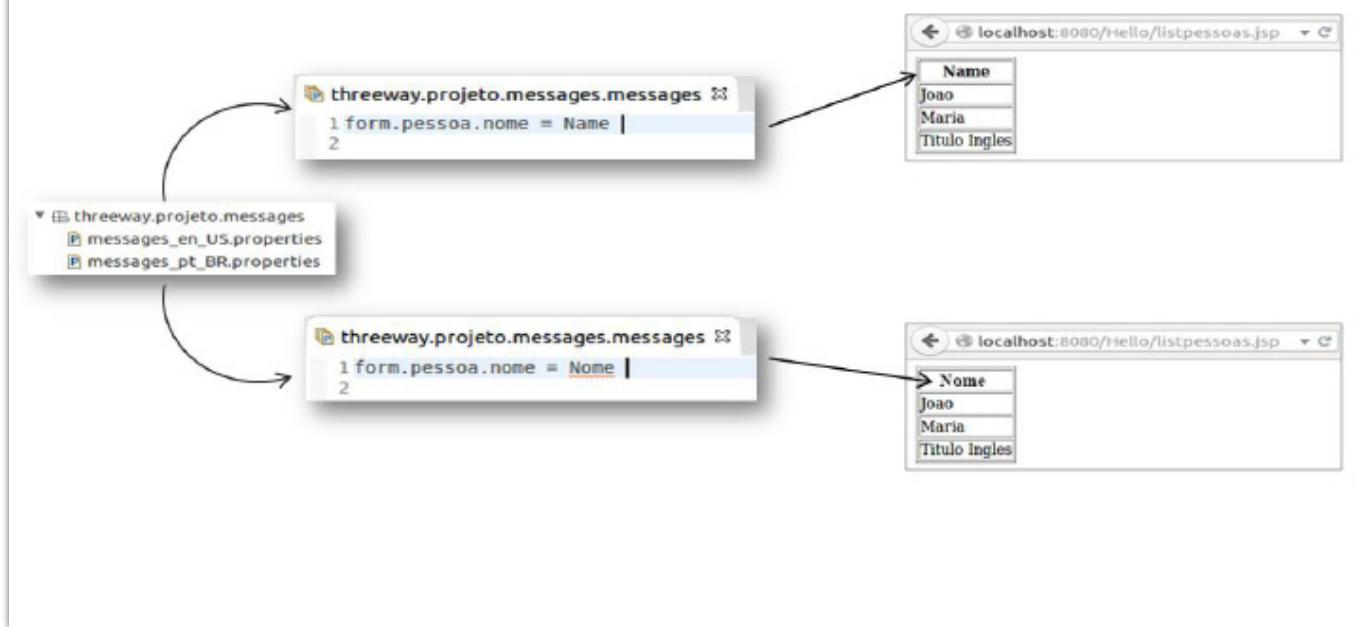
A tag `<fmt:formatNumber>` é usada para mostrar dados numéricos, incluindo valores monetários e percentuais, de acordo com uma localização específica.

```

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

```

Internacionalização de Páginas



Textos de localização de idiomas são utilizadas na JSTL com uso da tag <fmt:message>. Essa 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 o conteúdo de um arquivo **resource bundle e messages.properties**.

```
saudacao = Bem Vindo ao Sistema
form.pessoa.nome = Nome
form.pessoa.email = Email
erro.campo.obrigatorio = Por favor, preencha o campo.
```

Para usar esse arquivo contendo as mensagens do seu sistema, você precisa referenciá-lo em sua página para que a tag <fmt:message> saiba onde encontrar os recursos. Isto deve ser feito adicionando a seguinte configuração ao seu web.xml:

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

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 um pacote, a configuração no web.xml deverá conter o nome completo do arquivo:

```
<param-value>threeway.projeto.messages.message</param-value>
```

Para usar essas mensagens na sua página JSP você deve 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 navegador. Experimente mudar essas configurações no seu navegador e veja o que o sistema passa a ser exibido em idiomas diferentes.

Se o Locale associado ao usuário for pt_BR (português do Brasil), a tag <fmt:message> irá buscar as mensagens nos seguintes arquivos de mensagens, nessa 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.

JSF



JSF é mais do que um framework, é uma especificação Java que incorpora características de um framework MVC para WEB e de um modelo de interfaces gráficas baseado em eventos. Por basear-se no padrão de projeto MVC, uma de suas vantagens é a clara separação entre a visualização e regras de negócio.

O JSF oferece facilidade de uso das seguintes formas:

- Separação entre as camadas de apresentação e de aplicação.
- É uma especificação que faz parte do Java Enterprise Edition (Java EE). Por ser uma especificação, ela é mantida dentro do Java Community Process (JCP). Possui várias implementações: Oracle Mojarra (implementação de referência) disponível em <http://javaserverfaces.java.net>; Apache MyFaces em <http://myfaces.apache.org>
- Facilita a construção de uma interface usando um conjunto de componentes reutilizáveis.
- Ajuda a gerenciar o estado da interface nas solicitações do servidor.
- Oferece um modelo simples para conectar os eventos gerados pelo cliente ao código da aplicação do servidor.
- Permite personalizar os componentes de Interface para que sejam facilmente construídos e reutilizados.

TagLibs do JSF



JSF HTML tag reference

JSF Core tag reference

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

Para utilizar JSF em seu projeto, devemos adicionar as taglibs de referência:

Tags html - xmlns:h="http://xmlns.jcp.org/jsf/html"
ou
xmlns:h="http://java.sun.com/jsf/html"

A biblioteca **http://xmlns.jcp.org/jsf/html** possui tags básicos para renderização de telas em HTML. Ou seja, são componentes JSF que geram o código html.

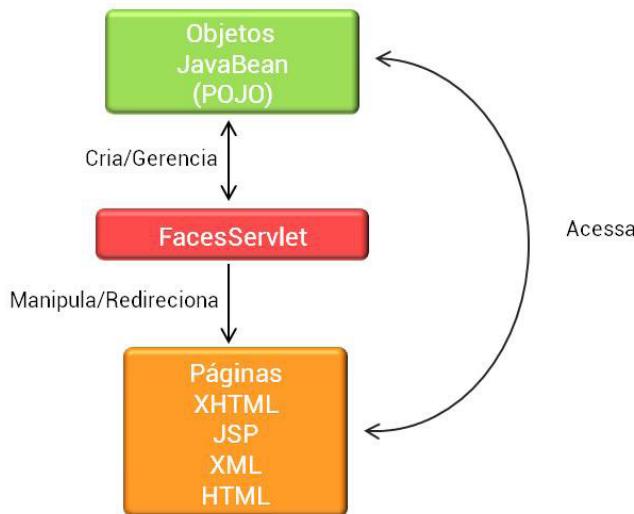
Core tags - xmlns:f="http://xmlns.jcp.org/jsf/core"
ou
xmlns:f="http://java.sun.com/jsf/core"

A biblioteca **http://xmlns.jcp.org/jsf/core** possui tags de propósito geral para conversão, validação , Ajax, I18N (internacionalização) e outros.

O controlador do JSF, o Faces Servlet precisa ser declarado no deployment descriptor (web.xml).

```
<!-- JSF -->
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
  <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

MVC



A arquitetura do JSF é baseado no Padrao MVC (Model View Controller). Permitindo uma clara separação de responsabilidades dentro da sua aplicação

Visão

- Componentes de Interfaces em páginas JSP/XHTML
- Kits renderizadores (HTML, XML, etc)

é a tecnologia de renderização do JSF, essa tecnologia define o leiaute e conteúdo da página. A tecnologia de renderização do JSF desde a versão 2.0 é o Facelets XHTML, entretando ainda podemos utilizar JSP como substituto, mas não é recomendável.

94

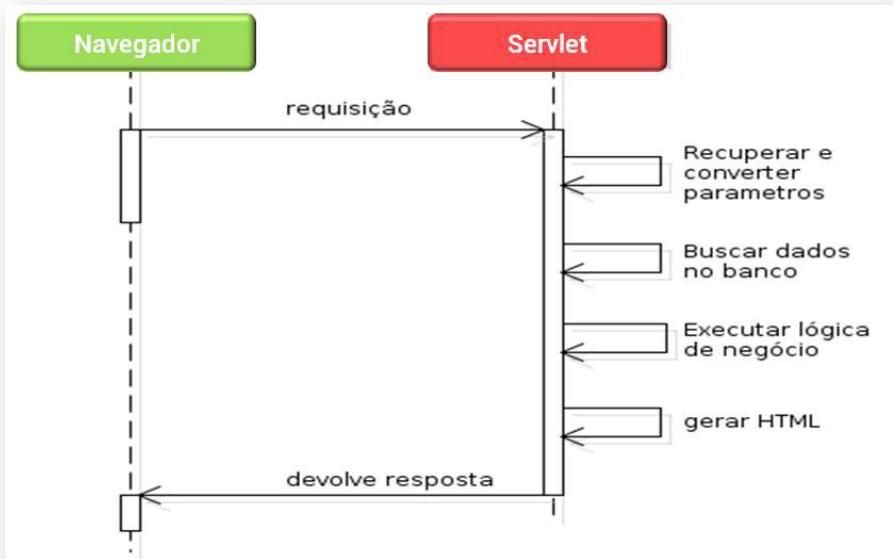
Controlador

- Faces Servlet
responsável pela manipulação das requisições e navegação entre as páginas. O Faces Servlet cuidado do ciclo de vida da requisição.

Modelo

- Managed Bean, um simples objeto java, mesmo convenção de nomes do JavaBeans, que é criado e gerenciado pelo JSF. Todo managed bean tem um escopo que controla seu tempo de vida; esse escopo pode estar em request, view, flow, session, application ou none.
- Backing Bean, objeto java contendo código de acesso a dados, regras de negócio, etc.

Fluxo com tudo no Servlet



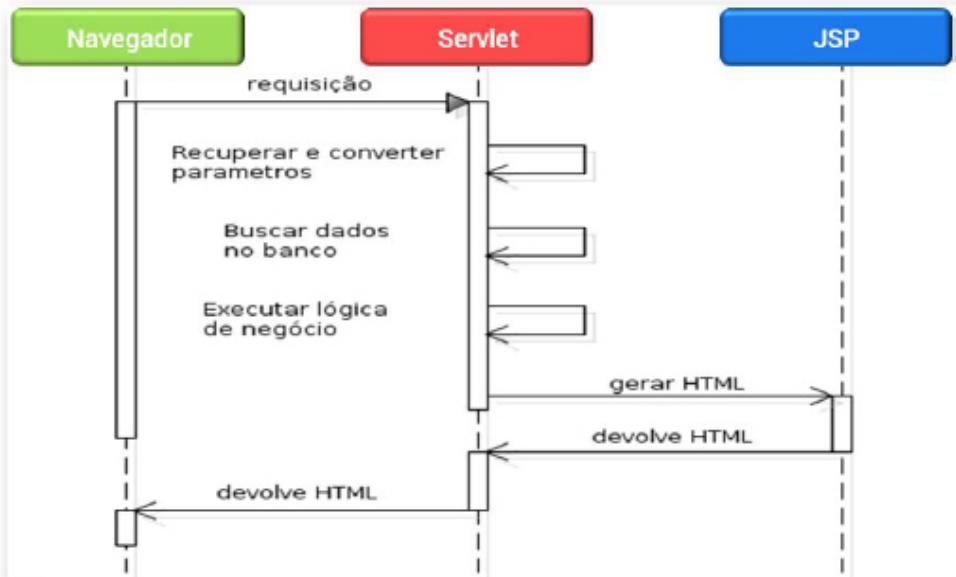
Para melhor compreender o benefício do padrão MVC da JSF observe o diagrama de eventos. No diagrama temos dois objetos o cliente (browser) submetendo requisições para um servidor, um servlet (tecnologia por traz de todo o conceito Java para Web).

Com um servlet podemos receber as requisições http, recuperar dados de um SGDB, aplicar regras de negócio, fazer conversões de dados, renderizar páginas Html e submete-la como resposta ao cliente (normalmente um browser web).

Porém ao colocarmos todas as coisas num único lugar, aumentamos sobremaneira a complexidade da aplicação, tornando o processo de manutenção bastante complexo, senão quase impossível dado a quantidade de código Html, CSS e Javascript que precisaríamos tratar com concatenações de strings, escapes de caracteres especiais, dentro de código java.

A tecnologia JSP tira um pouco desse trabalho maçante, facilitando a construção das páginas.

Fluxo com geração do HTML fora do Servlet



Como sabemos, com JSP podemos mesclar código java com Html, Css, Javascript.

Se procedermos com uma separação de responsabilidades, podemos usar o melhor do Servlet: código java para manipular requisições, realizar funções de backend, salvar dados no SGDB, gerenciar dados de sessão, etc; e utilizar o melhor do JSP que é renderizar páginas Html, Css, Javascript.

Nesse modelo, os servlet agem como um controlador da comunicação request/response, sendo responsável por receber a requisição, tratar os dados de formulários e salvar esses dados como atributos no escopo request, session ou application; e redirecionar/reencaminhar a requisição para o JSP responsável por renderizar a resposta.

```

protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException {
    AutomovelDAO dao = new AutomovelDAO();

    Automovel auto = dao.busca(request.getParameter("id"));

    // coloca as informações no request para seguirem adiante
    req.setAttribute("modelo", auto.getModelo());
    req.setAttribute("anoModelo", auto.getAnoModelo());

    RequestDispatcher rd =
        req.getRequestDispatcher("/visualiza.jsp");

    // encaminha para o JSP, levando o request com os dados
    rd.forward(req, res);
}
  
```

O Jsp, assim como o Servlet, utiliza os objetos implícitos request, session, application para recuperar os dados que foram salvos como atributos nestes escopos pelo Servlet. Fazendo uso da JSTL(Java Standard Tag Library) e da EL (expression Languague).

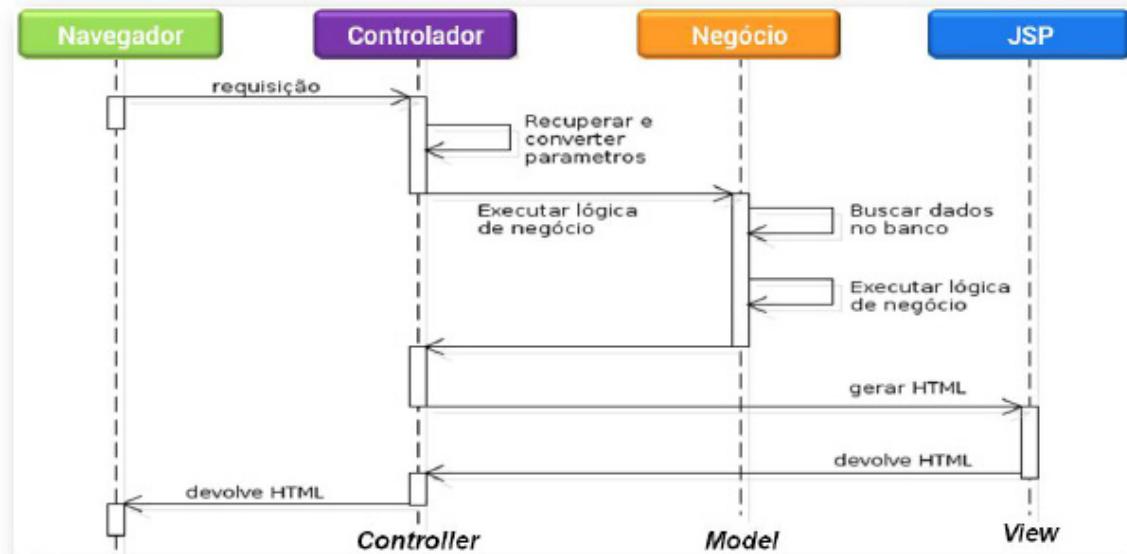
Essa separação de responsabilidades já agrega uma boa organização do nosso código. Porém, ainda temos algum trabalho repetitivo como recuperar informações do HttpServletRequest e entregar para o JSP.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <body>
        Modelo: <c:out value="${modelo}" />
        Ano do Modelo: <c:out value="${anoModelo}" />
    </body>
</html>
```

Os atributos “modelo” e “anoModelo” adicionados na camada de controle pelo Servlet são usados na camada de Visão (apresentação) JSP.

Utilizando EL no JSP podemos consumir os atributos dos escopos Request, Session e Application de forma transparente, não precisamos de muito código Java na camada de apresentação e nem precisamos saber em qual escopo foi salvo o atributo, a EL procura em todos os escopos pelo nome passado na expressão.

Fluxo com tarefas bem distribuídas

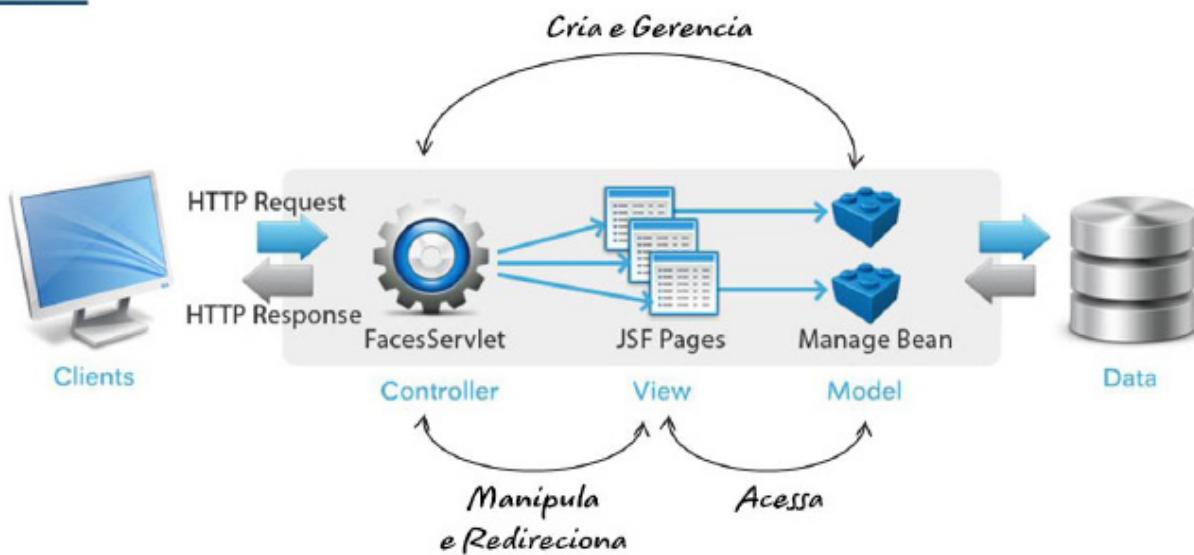


Ainda há muita responsabilidade no servlet, ele trata a informação vindo do HttpServletRequest, se necessário, precisa fazer conversão, ainda está responsável por obter informações do banco de dados, validar regras entre outras atividade.

Melhor seria que toda requisição recebida ficasse a cargo de um único componente, responsável por fazer todas as tarefas repetitivas. As tarefas de pegar parâmetros do HttpServletRequest, e de conversão e de navegação ficariam nesse componente especializado. Nesse novo modelo as regras negociais também teriam um componente especializado.

Isso é o que se propõe com um Framework MVC com o JSF, permitir que os desenvolvedores não percam tempo recriando código para cuidar daa parte repetitiva do processo de requisição, transformação dos dados, compartilhamento dos dados entre componentes de visualização e controle, e regras de navegação.

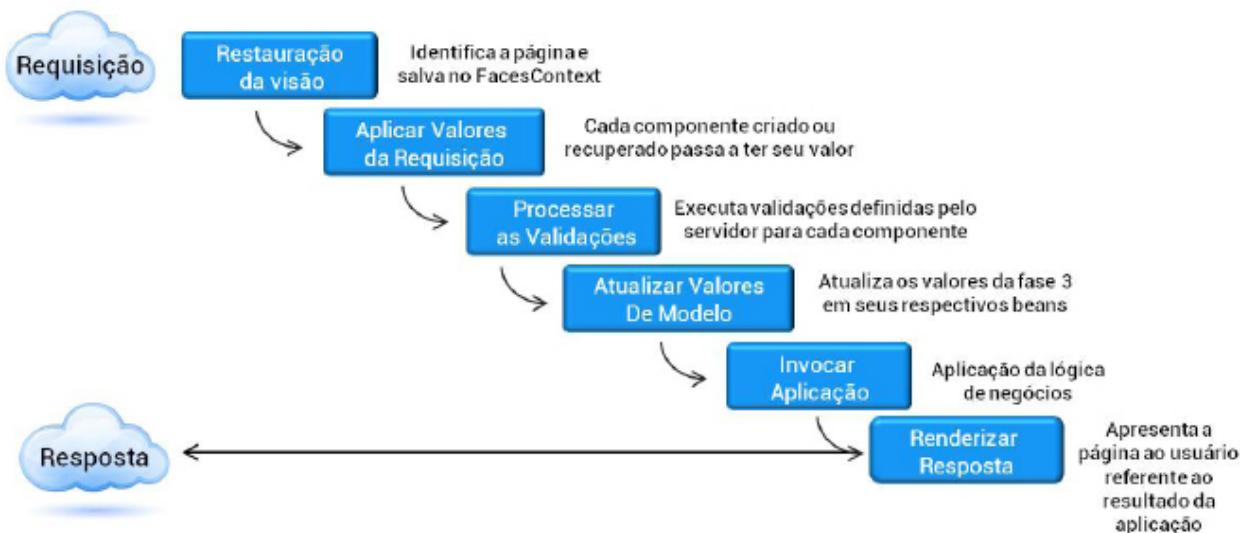
Arquitetura - MVC



No Java Server Faces, o Faces Servlet, é configurado no arquivo faces-config.xml em conjunto com anotações; são declarados Managed Beans, validadores e conversores.

O FacesServlet (Controller) é responsável por receber requisições da WEB, e fazer a ligação entre as propriedades das páginas JSF (View) com os Managed Beans (Model) e então atualizar a árvore de componentes UI e remeter uma página html de resposta. A navegação entre páginas e o mapeamento de ações, os validadores e conversores, os Managed Beans , todos são definidos no faces-config.xml ou por anotações.

Ciclo de Vida JSF



Restaurar a Visão

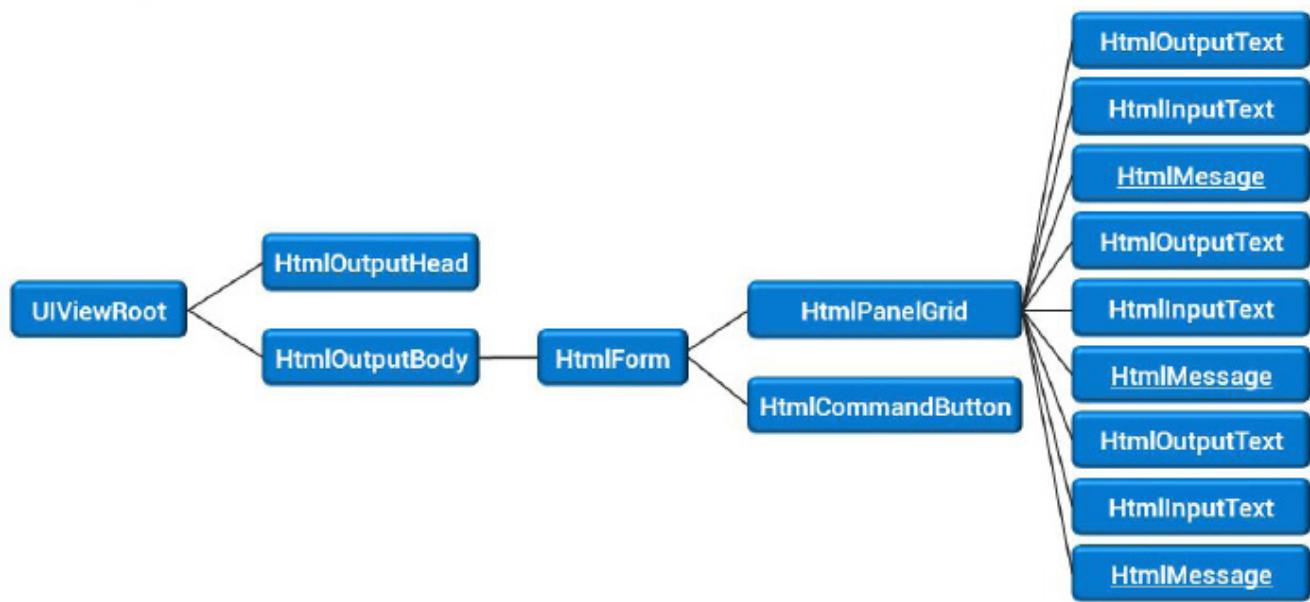
Toda página JSF é representada no servidor com uma árvore de componentes UI que será mapeado um-para-um com os elementos html presentes na página. Para uma melhor compreensão veja o xhtml abaixo:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>Seus Favoritos</title>
</h:head>
<h:body>
    <h:form id="favForm">
        <h:panelGrid columns="3">
            <h:outputText value="Comida Favorita"></h:outputText>
            <h:inputText id="favoritoComida" value="#{favorito.comida}"
                         required="true">
            </h:inputText>
            <h:message for="favoritoComida" />
            <h:outputText value="Bebida Favorita"></h:outputText>
            <h:inputText id="favoritoBebida" value="#{favorito.bebida}"
                         required="true">
            </h:inputText>
            <h:message for="favoritoBebida" />
            <h:outputText value="Esporte Favorito"></h:outputText>
            <h:inputText id="favoritoEsporte" value="#{favorito.esporte}"
                         required="true">
            </h:inputText>
            <h:message for="favoritoEsporte" />
        </h:panelGrid>
        <h:commandButton value="Salve meus favoritos" action="#{favorito.salvar}" />
        <br />
        <br />
    </h:form>
</h:body>
</html>

```

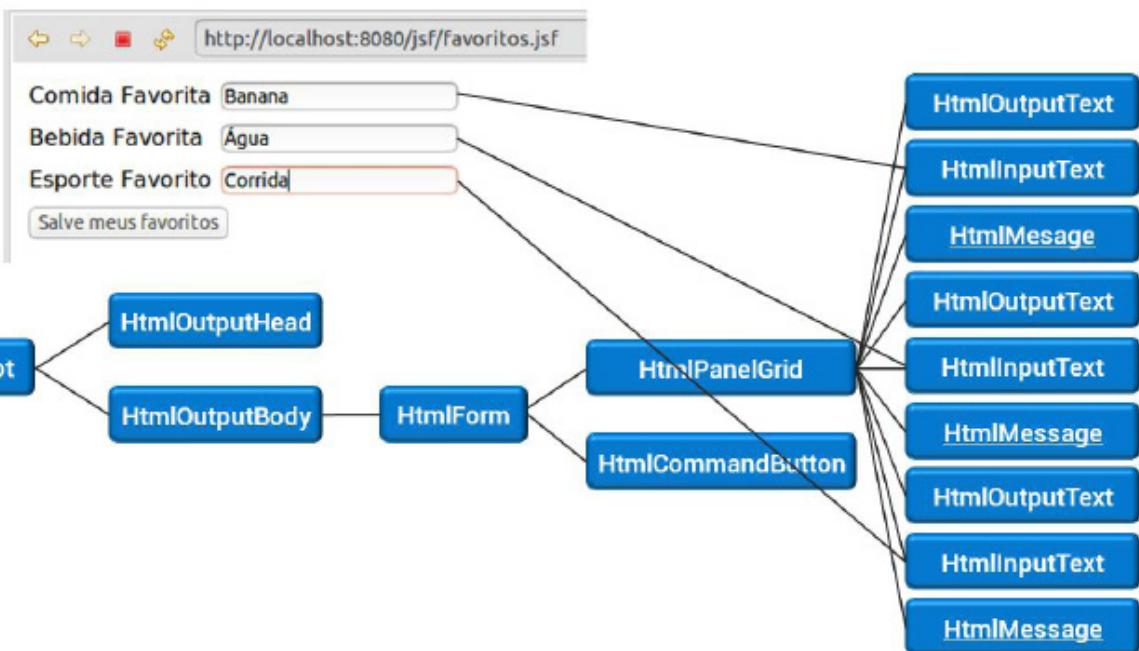
Árvore de Componentes



O código anterior é representado como uma árvore de componentes UI. Nessa há duas situações:

1. É uma nova requisição para página, neste caso é criado uma árvore de componentes UI vazia e a mesma é salva na instância atual do FacesContext (Context do Faces Servlet). Neste caso o ciclo de vida avança direto para a fase “Renderizar Resposta”, que popula os componentes UI vazios da árvore com os componentes JSF desta página. Além disso o estado da árvore de componentes UI é salvo no estado da View JSF, para a próxima requisição.
2. Quando o conteúdo do formulário é enviado para a mesma página usando o método HTTP POST. Neste caso, a fase “Restaurar Visão”, restaura a árvore de componentes UI a partir do estado da visão que foi gerado na requisição prévia desta página.

Aplicar Valores Requisição



Aplicar Valores de Requisição

Nessa fase, cada componente UI da árvore, que possua um atributo 'value', é associado aos valores submetidos pelo formulário html.

Processar as Validações

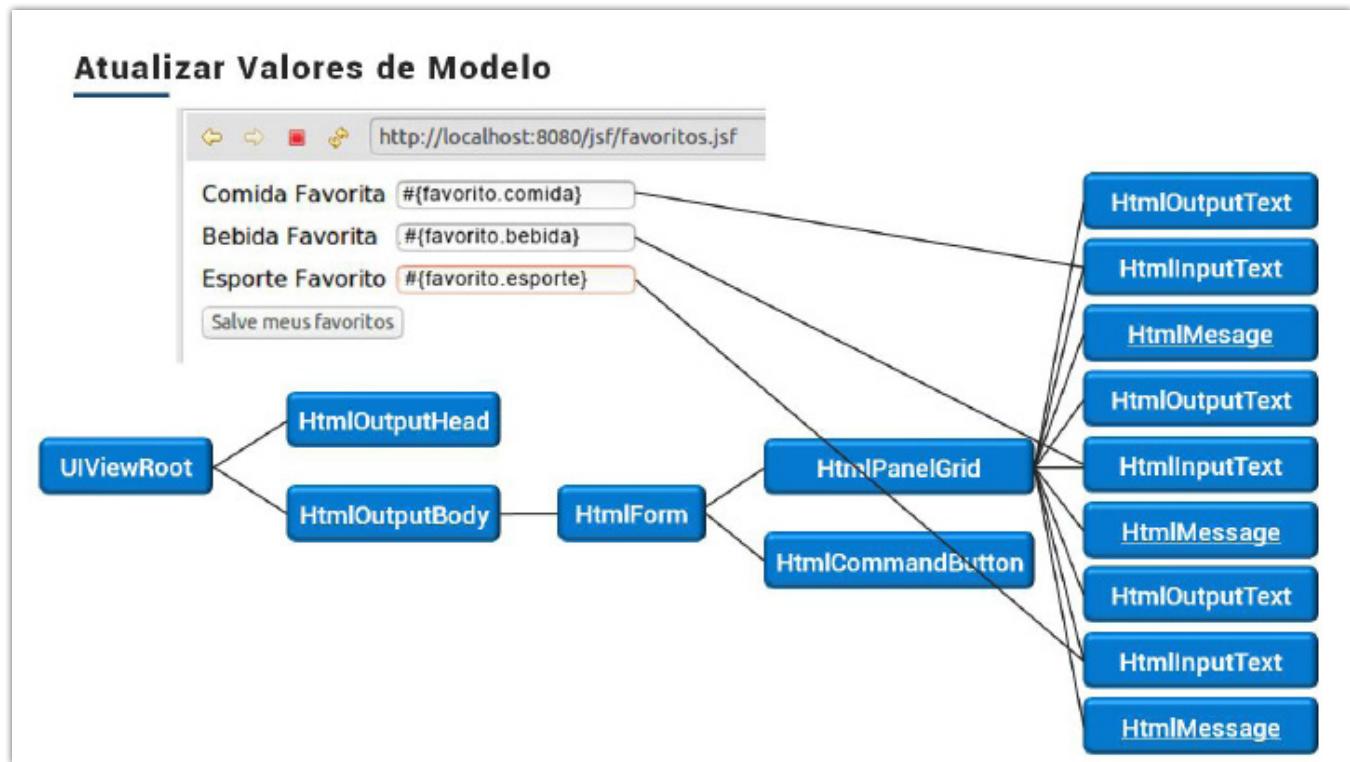
Nessa fase conversão e validação são aplicados em ordem. As validações serão executados todos os campos que possuem o atributo ‘required’ igual a true.

Conversões ocorrem sobre os parâmetros de requisição HTTP ao seu correspondente tipo em Java. O JSF possui vários conversores internos e ainda fornece a possibilidade de criar conversores personalizados.

```
<h:inputText id="aniversario" value="#{usuario.nascimento}">
    <f:convertDateTime pattern="dd/MM/yy"/>
</h:inputText>
```

Neste exemplo, será garantido que o campo ‘aniversario’ seja corretamente preenchido com o formato “dd/MM/yy” e convertido para o tipo java.util.Date e associado à propriedade ‘nascimento’ do Managed Bean Usuario.

Quando uma validação ou conversão falhar, será criado uma instância do componente FacesMessage, o mesmo será enfileirado numa instância de FacesContext. Havendo erros de validação (ou conversão), o ciclo de vida do JSF vai para a fase “Renderizar Resposta”, e fila de mensagens de erro será mostrada no <h:message> ou <h:messages> associado ao componente UI.



Atualizar Valores de Modelo

Nesta fase é realizada a ligação entre os valores, convertidos e validados, dos componentes UI e os Managed Beans do Modelo JSF.

Invocar Aplicação

Nesta fase, as ações são executadas, no exemplo, o código da ação está vinculado ao componente `<h:commandButton ... action=[código de ação] />`, neste caso, a invocação do método `salvar()` no managed bean `Usuario`. Código de ação no JSF podem estar em métodos de ação ou listeners de ação, ambos são métodos em um objeto java vinculado ao atributo ‘action’ dos componentes `<h:commandButton>`, `<h:commandLink>`.

Após a execução do código de ação é realizado a navegação pelo JSF NavigationHandler. O atributo ‘action’ deve receber uma valor literal, normalmente o retorno do código de ação, uma String ou void. Mas o atributo ‘action’ também pode ter um valor estático predefinido. Esse literal é tratado com uma regra de navegação dentro faces-config.xml, caso haja uma regra explícita, a navegação será redirecionada para página na regra. Se não houver regra, o literal é tratado implicitamente como sendo a página de destino. Se não houver regra ou página com o mesmo nome do literal, então, o NavigationHandler fica na mesma página.

Renderizar Resposta

A última fase do ciclo de vida do JSF, nessa fase os resultados são renderizados para o usuário. A redenrização da árvore de compontes UI, invoca os método encodeBegin(), encodeEnd(), encodeChildren() ou encodeAll() de cada componente.

Adicionalmente à renderização, esta fase também armazena o estado da árvore de componentes UI no estados da visão, permitindo assim que a visão possa ser restaurada na próxima requisição.

O atributo Immediate

Há situações em que você desejará escapar da ordem de conversão e saltar a fase de validação com intuito de navegar para outra página. Imagine que você tem um botão do tipo “Voltar para Página Principal”:

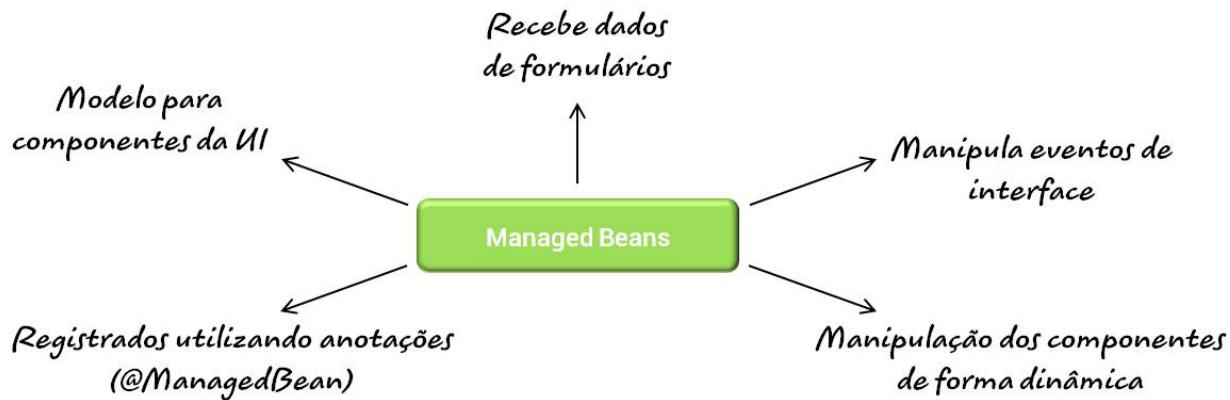
```
<h:form id="favForm">
    ...
    <h:inputText id="favoritoComida" value="#{favorito.comida}" required="true"/>
    ...
    <b><h:commandButton values="Página Principal" action="index" /></b>
</h:form>
```

Ao clicar no botão “Página Principal”, se você deixar o campo obrigatório vazio, você vai se deparar com mensagens de erro de validação não podendo completar a operação “Voltar para Página Principal”. Isto ocorre porque o botão faz uma submissão do tipo HTTP POST do formulário que dispara o ciclo de vida do JSF perfazendo a fase de validação e conversão que gera o erro.

Em casos assim o JSF te permite utilizar o atributo immediate, que te permite escapar das fase de conversão e validação. Este atributo permite que os eventos de ação sejam processados na fase de “Aplicar Valores de Requisição”

```
<h:form id="favForm">
    ...
    <h:inputText id="favoritoComida" value="#{favorito.comida}" required="true"/>
    ...
    <b><h:commandButton values="Página Principal" action="index"
        immediate="true"/></b>
</h:form>
```

Managed Beans



Managed Beans são classes Java que são configuradas para interagir com páginas JSF intermediando a comunicação entre nossas páginas e o nosso modelo. Eles são os responsáveis por receber os dados preenchidos nos formulários das páginas e executar alguma ação relacionada a métodos de ação.

O JSF é responsável por criar destruir os Managed Beans de acordo com a sua definição de escopo.

Managed Beans são originalmente definidos no arquivo faces-config.xml, exemplo:

```

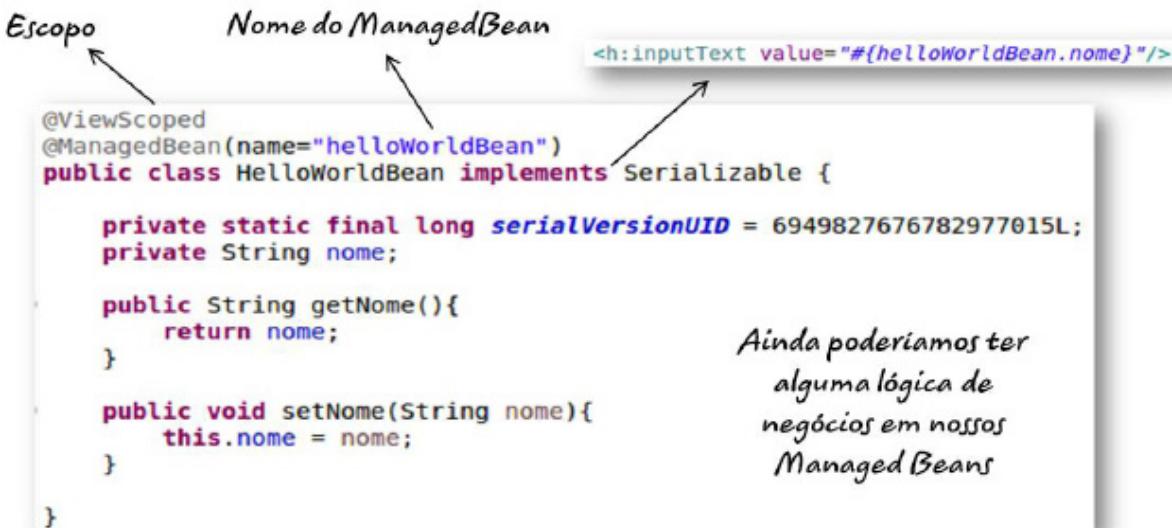
public class Usuario{
    private String nome;
    public String getName(){ return nome; }
    ...
}
  
```

```

<managed-bean>
    <managed-bean-name>usr</managed-bean-name>
    <managed-bean-class>br.com.triway.model.Usuario</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
  
```

Usamos a EL `#{{usr.nome}}` em nossas páginas JSF para vincular os campos de dados da visão ao modelo.

Managed Bean



Disponível a partir da versão 2.0 do JSF, a anotação `@ManagedBean` é utilizada para indicar que a classe é um Managed Bean, nela temos duas propriedades:

- **name** – que indica o nome do nosso Managed Bean (quando não informado essa propriedade, o Managed Bean recebe o mesmo nome da classe com o primeiro caractere minúsculo). Esse nome é a referência usada na EL, exemplo:

- **eager** - quando seu valor for true e o escopo do Managed Bean for Application indica que este bean deve ser inicializado quando a aplicação for iniciada.

Anotações Java EE

Várias anotações JEE trabalham perfeitamente com Managed Beans no JSF. Os métodos marcados com `@PostConstruct` devem ser invocados após a construção do Managed Bean, te dando uma chance de inicializar o Managed Bean, que só usa o construtor default. Outras anotações JEE que trabalham com Managed Beans: `@Resource`, `@PersistenceUnit`, `@PersistenceContext`, `@EJB` e `@WebServiceRef`.

Anotações CDI

As anotações `@Named` e `@Inject` fazem parte da CDI (framework de injecção de dependência). Essas anotações não são gerenciadas pelo JSF, mas em futuras implementações elas irão substituir `@ManagedBean` <=> `@Named` e `@ManagedProperty` <=> `@Inject` em especificação futuras da JSF. Veremos mais sobre essas anotações em nosso estudo sobre CDI (Context Dependency Injection).

Escopos de Navegação



Todo Managed Bean tem um ciclo de vida definido pelo seu escopo. As anotação de tipo de escopo devem vir antes da declaração da classe. Se não colocarmos nenhuma declaração de escopo, será atribuído por padrão a escopo de requisição (@RequestScoped).

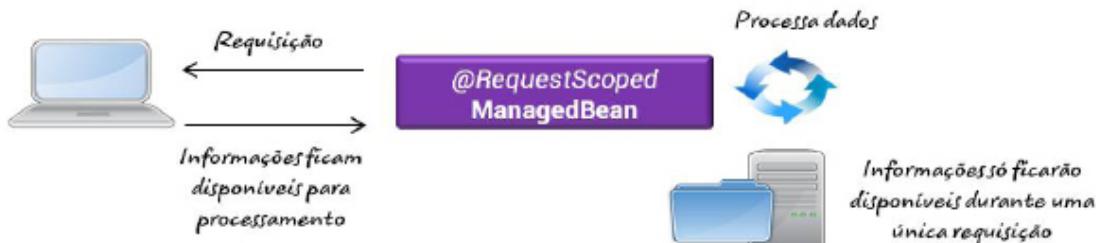
Abaixo temos os escopos permitidos para os Managed Beans:

- Request Scope (@RequestScoped)
- View Scope (@ViewScoped)
- Session Scope (@SessionScoped)
- Application Scope (@ApplicationScoped)
- None scope (@NoneScoped)
- Flow Scope" (@FlowScoped), novo na JSF 2.2

RequestScoped

RequestScoped (@RequestScoped)

O Managed-Bean não manterá seu estado entre as chamadas do usuário.



@RequestScoped – Escopo de requisição. Existe durante uma única solicitação/requisição HTTP.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped

public class RequestManager { }
```

SessionScoped

SessionScoped (@SessionScoped)

Todo atributo de um ManagedBean SessionScoped terá seu valor mantidos até o fim da sessão do usuário.



@SessionScoped – Escopo de sessão são usados por objetos através de uma sessão HTTP, permanece ativo em várias requisições HTTP ou enquanto durar a sessão do usuário. Por exemplo, detalhe de usuários logados são o tipo de informação candidatas a permanecer neste escopo.

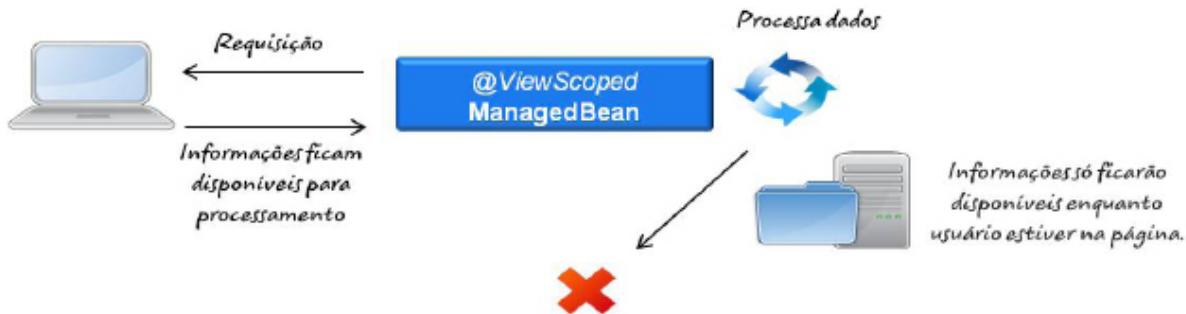
```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean
@ViewScoped
public class SessionManager {
```

ViewScoped

ViewScoped (@ViewScoped)

Existe na memória enquanto o usuário permanecer na página exibida.



@ViewScoped – Introduzido no JSF 2.0, podemos dizer que é o escopo de página, persiste durante a interação do usuário com uma única página. Este escopo é particularmente útil quando você precisa editar algum objeto enquanto permanece na mesma página.

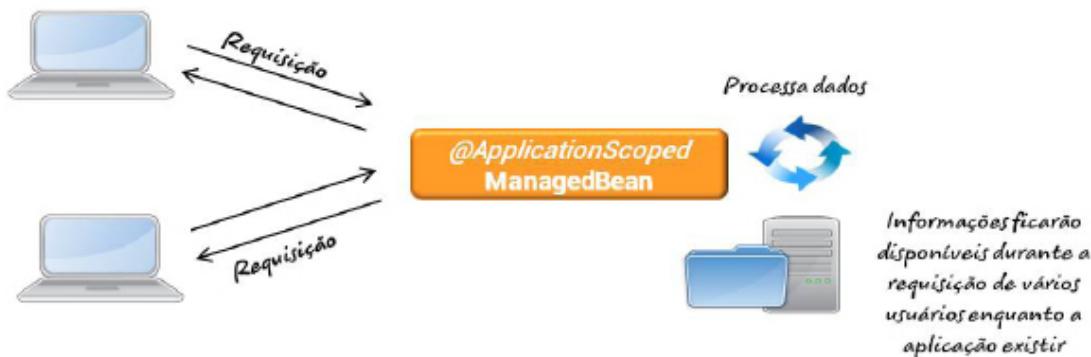
Você deve perceber que este escopo é mais longo que @RequestScoped e menor do @SessionScoped, uma vez que este escopo termina quando você navega para outra página.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean
@ViewScoped
public class SessionManager {  
}
```

ApplicationScoped

ApplicationScoped (@ApplicationScoped)
Existe enquanto a aplicação estiver sendo executada.



@ApplicationScoped – Escopo de aplicação, o bean permanece entre todas as interações de todos os usuários, ou seja, possui vida útil durante todo o tempo em que a aplicação estiver funcionando.

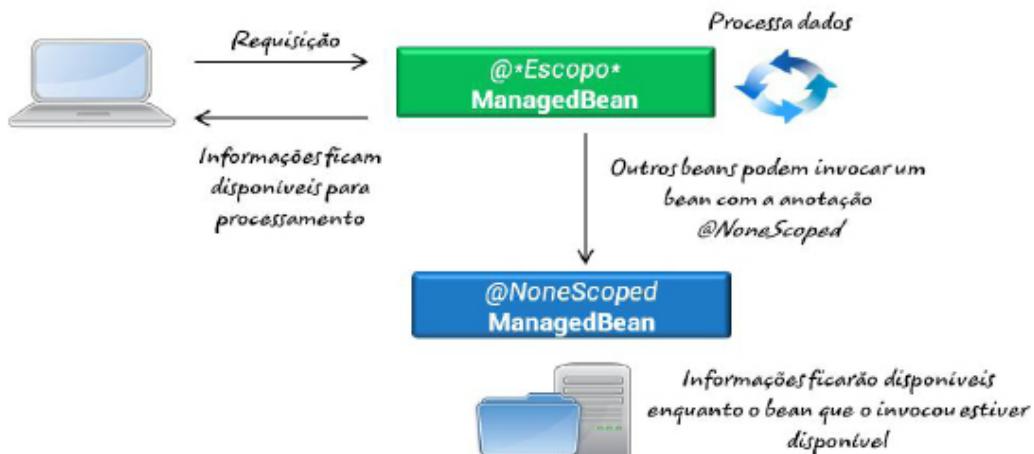
```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ApplicationScoped;
```

```
@ManagedBean
@ApplicationScoped
public class SessionManager {  
}
```

NoneScoped

NoneScoped(@NoneScoped)

O Managed Bean será invocado por outros beans e o seu tempo de vida vai ser de acordo com o escopo de quem o invocou



@NoneScoped – Beans com este escopo não serão instanciados e nem salvos em nenhum escopo, estes são instanciados sobre demanda, por outros beans. Ou seja, quando são injetados em outro Managed Bean, estes seguem o escopo do bean principal.

@CustomScoped – Indica que o escopo do Managed Bean terá um tempo de vida personalizado, criado pelo próprio desenvolvedor.

Nota: beans injetados devem estar no mesmo escopo do bean injetor.

Componentes UI

h → Contém tags de componentes JavaServer Faces para todas as combinações UIComponent + HTML RenderKit Renderer definidas na especificação JavaServer Faces

`xmlns:h="http://xmlns.jcp.org/jsf/html"
xmlns:f="http://xmlns.jcp.org/jsf/core"`

f → A JSF Core tag library contém tags para fins de conversão, validação, Ajax e alguns outros usos.

Componentes UI podem ser estilizados por CSS e também interagem com Managed Beans

As tags h e f são bibliotecas de componentes para construir a UI, esses componentes também podem ser estilizados por CSS e também interagem com Managed Beans. Esses componentes separam a lógica de negócios da apresentação e toda view possui uma árvore de componentes.

h – contém tags de componentes Java Server Faces para todas as combinações UIComponent + Renderizador HTML RenderKit definidas na especificação Java Server Faces.

f – A JSF Core tag library contém tags para fins de conversão e validação de dados, Ajax e alguns outros usos.

Exemplo de utilização de algumas tags

Página xhtml de exemplo

```
<h:head>
<title>JSF 2 Hello World</title>
</h:head>

<h:body>
<h3>JSF 2 Hello World Exemplo - hello.xhtml</h3>
<h:form id="form">
    <h:inputText value="#{helloWorldBean.nome}" />
    <h:commandButton value="Bem Vindo">
        <f:ajax execute="form" render="form" />
    </h:commandButton>
    <br/><br/>
    <h:outputLabel rendered="#{helloWorldBean.nome != isEmpty}" value="Bem Vindo #{helloWorldBean.nome}" />
</h:form>
</h:body>

</html>
```

Menu de seleção única

```
<h:selectOneMenu id="list">
    <f:selectItem itemLabel="Opção 1" itemValue="1" />
    <f:selectItem itemLabel="Opção 2" itemValue="2" />
</h:selectOneMenu>
```

Desde JSF 2.0 é aconselhável utilizar o Facelets como tecnologia de renderização. Como regra geral seus arquivos de página JSF devem ter extensão “.xhtml” e declarar os namespaces das tags JSF:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
```

ao invés de usar a extensão “.jsp” e as taglibs

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

Você sempre precisará de uma tag `<h:form>` para submeter seus dados ao JSF, todas as outras tags do formulário html estarão contidas dentro dela.

Os dados do Modelo são associados às tags através da EL, no exemplo `<h:inputText value="#{helloWorldBean.nome}" />`, “helloWorldBean” é o nome do managed bean e “nome” refere-se ao método público `getNome()` definido na classe do Bean.

As tags `<h:commandButton>` e `<h:commandLink>` disparam a submissão do formulário ao JSF. A submissão pode ser síncrona - segue o modelo request/response do protocolo HTTP, o browser envia a requisição e aguarda pela resposta. Ou pode ser assíncrona - o browser submete a requisição e continua a processar até que seja avisado da resposta. Para tanto, você deve usar a tag `<f:ajax>` e informar quais dados devem ser submetidos - atributo `execute`, e quais campos devem ser atualizados com a resposta - atributo `render`.

O atributo `rendered` informa ao JSF que a tag deve (true) ser renderizada ou não deve (false) ser renderizada.

Exemplo de utilização de algumas tags=

Conversor

```
<h:outputText id="balance" value="#{accountBean.balance}">
    <f:convertNumber currencySymbol="$" groupingUsed="true"
        maxFractionDigits="2" type="currency" />
</h:outputText>

<h:inputText id="nascimento">
    <f:convertDateTime type="date" pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo"/>
</h:inputText>
```

Validador

```
<h:inputSecret id="senha" value="#{validadoresBean.atributoObrigatorio}" required="true">
    <f:validateLength minimum="5"/>
</h:inputSecret>
```

```
<h:inputText id="withdraw" value="#{accountBean.withdraw.amount}">
    <f:validateDoubleRange minimum="20.00" maximum="1000.00" />
</h:inputText>
```

Conversores e validadores, e validadores podem ser aplicados aos campos dos formulários de dados. Há vários conversores e validadores pré-definidos na JSF:

- f:convertDateTime, f:converter,f:convertNumber
- f:validateBean, f:validateDoubleRange, f:validateLength, f:validateLongRange, f:validateRegex, f:validateRequired, f:validator

Você pode criar seu próprio conversor e invocar a classe de conversão através da tag <f:converter>.

Você pode criar seu próprio validador e invocar a classe de validação através da tag <f:validator>.

Vale lembrar que tanto conversão quanto validação ocorrem no servidor, após a submissão dos dados, na Fase “Processar Validação” do ciclo de vida JSF.

Muitas vezes é desejável validar dados no browser antes de submeter, daí você vai precisar usar JavaScript na página para atingir esse propósito.

Nota: todo componente UI tem um identificador único gerado pelo JSF, exemplo:

```
<h:form id="frm">
    <h:inputText id="email"/>
```

para o campo email será gerado os seguinte id “frm:email”.

Exemplo de utilização de algumas tags

Lista de seleção

```
<!-- Lista criada na mão -->
<h:selectOneListbox id="Opcoes">
    <f:selectItem id="opcao1" itemLabel="Opção 1" itemValue="1"/>
    <f:selectItem id="opcao2" itemLabel="Opção 2" itemValue="2"/>
    <f:selectItem id="opcao3" itemLabel="Opção 3" itemValue="3"/>
</h:selectOneListbox>

<!-- Lista criada usando um array de objetos -->
<h:selectOneListbox id="Opcoes" value="Lista usando Array">
    <f:selectItems var="cliente" itemLabel="#{cliente.nome}"
                   itemValue="#{cliente.codigo}" value="#{clienteData.clienteList}"/>
</h:selectOneListbox>

<!-- Lista criada usando um HashMap -->
<h:selectOneListbox id="Opcoes" value="Lista usando HashMap">
    <f:selectItems value="#{clienteData.criarMap()}" />
</h:selectOneListbox>
```

Podemos utilizar o atributo `value` passando como parâmetro uma array de objetos ou um Map



A tag `<f:selectItems/>` dentro da tag `<f:selectOneListBox/>`, representa os itens da lista.

No primeiro exemplo temos uma lista estática.

No segundo exemplo temos uma lista de itens dinâmica onde cada item é um objeto dentro `clientList` do Managed Bean.

No terceiro exemplo estamos utilizando o `HashMap`, da mesma forma como utilizados o array de objetos.

Exemplo de utilização de algumas tags

Lista de seleção única

```
private ClienteTable[] clienteList = new ClienteTable[] {
    new ClienteTable("Cliente 01", 1),
    new ClienteTable("Cliente 02", 2),
    new ClienteTable("Cliente 03", 3)
};

private Map<String, Integer> clienteMap = new HashMap<String, Integer>();

public Map<String, Integer> criarMap() {

    clienteMap.put("Cliente 01", 1);
    clienteMap.put("Cliente 02", 2);
    clienteMap.put("Cliente 03", 3);

    return clienteMap;
}
```

Tabela de dados

```
@ManagedBean(name = "clienteData")
@SessionScoped
public class ClienteTableData implements Serializable {
    private static final long serialVersionUID = 1L;
    private ClienteTable[] clienteList = new ClienteTable[] {
        new ClienteTable("Cliente 01", 1),
        new ClienteTable("Cliente 02", 2)
    };
    public ClienteTable[] getClienteList() {
        return clienteList;
    }
}
```

Lista de Clientes	
Cliente	Código
Cliente 01	1
Cliente 02	2

Criamos uma coluna para cada atributo que desejamos mostrar e utilizamos um outputText para acessar o atributo no bean.

Passamos uma lista de clientes para a dataTable

```
<h:body>
    <h:dataTable id="table1" value="#{clienteData.clienteList}"
        var="clienteTable" border="1">
        <f:facet name="header">
            <h:outputText value="Lista de Clientes" />
        </f:facet>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Cliente" />
            </f:facet>
            <h:outputText value="#{clienteTable.nome}" />
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Código" />
            </f:facet>
            <h:outputText value="#{clienteTable.codigo}" />
        </h:column>
    </h:dataTable>
</h:body>
```

Tabelas ou Grid de dados são dos componentes mais comuns na maioria das interfaces que você vai criar. A Jsf tem a tag `<h:dataTable/>` para esse fim, ela te permite mostrar dados num formato de tabela Html.

Assim como um componente `<c:forEach/>` da JSTL o componente `<h:dataTable>` percorre todos o conjunto de dados associados ao atributo ‘value’ e salva cada instância deste conjunto no atributo ‘var’. Cada elemento em ‘var’ representa uma linha da tabela.

O componente `h:dataTable` pode iterar sobre os seguintes tipos de dados java:

- Um Objeto Java
- Um array
- Uma instância de `java.util.List`
- Uma instância de `java.sql.ResultSet`
- Uma instância de `javax.servlet.jsp.jstl.sql.Result`
- Uma instância de `javax.faces.model.DataModel`

A tag `<f:facet/>` define o cabeçalho da tabela e das colunas.

Command Link e Command Button

<h:commandButton>

Gera um submit button que pode ser associado a um bean ou a uma classe ActionListener para lidar com eventos.

```
<h:form>
    Command Button
    <h:commandButton value="Confirmar" action="" />
    Command Link
    <h:commandLink value="Redirecionar" action="" />
</h:form>
```

Command Button
Confirmar

Command Link
[Redirecionar](#)

<h:commandLink>

Gera uma tag de link <a> que se comporta como um submit button e que pode ser associada a um bean ou uma classe ActionListener para lidar com eventos. Utiliza Java Script para enviar o formulário.

A tag **<h:commandButton>** gera um submit button (um elemento HTML `<input type="submit">`) que pode ser associado a um bean ou a uma classe ActionListener para lidar com eventos.

A tag **<h:commandLink>** gera uma tag link `<a>` que se comporta como um submit button e que pode ser associada a um bean ou uma classe ActionListener para lidar com eventos.

As duas tags aceitam a tag `<f:ajax/>`, para enviar o formulário de dados de forma assíncrona.

Ajax com HTML e Java Script

```

<body>
    <button type="button" onclick="acionarAjax()">Ajax!</button>
    <p id="output" />

    <script type="text/javascript">
        function acionarAjax() {
            var xhttp = new XMLHttpRequest();
            xhttp.onreadystatechange = function() {
                if (xhttp.readyState == 4 && xhttp.status == 200) {
                    document.getElementById("output").innerHTML = xhttp.responseText;
                }
            };
            xhttp.open('GET', "texto.txt", true);
            xhttp.send(); ←
        }
    </script>
</body>

```

Envia qualquer dado adicional caso a requisição seja um post.

Método da requisição HTTP, URL e se a chamada será assíncrona.

Elemento com id "output" irá receber a resposta.

Ajax permite que requisições e atualizações de páginas sejam realizados de forma assincronamente, através da troca de uma pequena quantidade de informações com o servidor, ou seja, possibilita a atualização de partes de uma página web sem ter que recarregar toda a página.

A JSF consegue simplificar o código acima, ativando comportamento Ajax nos componentes UIComponents, basta apenas ativá-los usando a tag <f:ajax/>

Ajax com JSF

`execute="form"` indica que o formulário com esse ID será enviado para o servidor para ser processado.

`render="output"` indica que depois da requisição Ajax, o componente com esse ID será atualizado.

```
<h:form id="form">
    <h:inputText value="#{helloWorldBean.nome}" />
    <h:commandButton value="Bem Vindo">
        <f:ajax execute="form" render="form" />
    </h:commandButton>
    <br/><br/>
    <h:outputLabel rendered="#{helloWorldBean.nome != isEmpty}" value="Bem Vindo #{helloWorldBean.nome}" />
</h:form>
```

JSF 2 Hello World Exemplo - hello.xhtml

Botão aciona a requisição Ajax

A tag `<f:ajax>` é responsável por executar a chamada assíncrona. O atributo ‘execute’ indica o componente que terá seu ‘value’ enviado enviado e processado no servidor, sendo informado o seu ID ou ainda podemos usar `@this` quando for o próprio componente. O atributo ‘render’ contém uma lista de IDs de componentes, separados por espaço, que devem ser atualizados (renderizados) quando a resposta da requisição assíncrona for devolvida.

Valores especiais para o atributo ‘execute’ e ‘render’:

- **@all** - executa ou renderiza todos os componentes
- **@none** - não executa ou renderiza qualquer dos componentes
- **@this** - executa ou renderiza o componente que dispara a requisição
- **@form** - executa ou renderiza todos os componentes no formulário do componente que dispara a requisição ajax

Outro importante atributo é ‘listener’ que dispara a execução de um método no servidor.

```
<f:ajax listener="{ajaxBean.listenerTeste()}" execute="@form" render="@form"/>
```

```
public void listenerTest(AjaxBehaviorEvent event){ .... }
```

Tome cuidado com requisições AJAX para Managed Beans com escopo RequestScoped, cada requisição precisa enviar todos os dados necessários para o processamento, já que JSF não guardará os dados da última requisição. Por exemplo, em aplicações com filtros de múltiplos níveis do tipo mestre-detalhe. Neste caso o melhor é usar ViewScoped.

Para ter suporte a ajax na sua página a tag `<h:head/>` deverá ser adicionada à sua página.

Conversor Padrão

```
<h:inputText id="nascimento">
    <f:convertDateTime type="date" pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo"/>
</h:inputText>
```

Converte entradas do usuário em valores do tipo `java.util.Date` com um formato padrão.

```
<h:inputText id="peso">
    <f:convertNumber integerOnly="true"/>
</h:inputText>
```

Converte entradas do usuário em valores do tipo `java.lang.Number`.

Converter é o componente do JSF responsável por fazer a conversão de um objeto para uma String e de uma String para um objeto.

Ele atua entre a página JSF e o Managed Bean, convertendo dados do formulário da página automaticamente em objetos no Managed Bean. Você pode definir como a conversão deve ocorrer através de conversores pré-fabricados ou você pode personalizar um para seu uso específico.

Conversores Padrão

Temos conversores implícitos e explícitos. Implícitos são aplicados automaticamente de acordo com o tipo da propriedade no Managed Bean. Veja os ID dos conversores:

- **Boolean** - conversor implícito aplicado para valores do tipo `java Boolean` ou `boolean`
- **Byte** - conversor implícito aplicado para valores do tipo `java Byte` ou `byte`
- **Character** - conversor implícito aplicado para valores do tipo `java Character` ou `char`
- **Short** - conversor implícito aplicado para valores do tipo `java Short` ou `short`
- **Integer** - conversor implícito aplicado para valores do tipo `java Integer` ou `int`
- **Long** - conversor implícito aplicado para valores do tipo `java Long` ou `long`
- **Float** - conversor implícito aplicado para valores do tipo `java Float` ou `float`
- **Double** - conversor implícito aplicado para valores do tipo `java Double` ou `double`
- **BigDecimal** - conversor implícito aplicado para valores do tipo `java BigDecimal`
- **BigInteger** - conversor implícito aplicado para valores do tipo `java BigInteger`
- **Number** - conversor explícito aplicado para valores do tipo `java Number`.

- **DateTime** - conversor explícito aplicado para valores do tipo java Date com um formato padrão.

- **Enum** - conversor explícito aplicado para valores do tipo java Enum

De forma geral você pode usar qualquer conversor com qualquer tag com o atributo ‘converter’ passando ID do da classe de conversão:

```
<h:inputText convert="Integer" value="#{mBean.propiedadeInt}" />
```

ou

```
<h:inputText value="#{mBean.propiedadeInt}">
    <f:converter converterId="Integer"/>
</h:inputText>
```

Os conversores DateTimeConverter e NumberConverter possuem tags próprias

A tag **<f:convertDateTime>**, veja um exemplo:

```
<h:inputText id="dateField" value="#{dateBean.date}" required="true">
    <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
```

você pode usar os seguintes atributos para configurar a conversão DateTimeConverter:

- **dateStyle** – Um dos estilos de data definidos pelo javax.text.DateFormat. Os valores possíveis são: default, short, long, medium ou full.
- **ParseLocale** – A localidade a ser usada como referência durante a conversão.
- **TimeStyle** – Um dos estilos de data definido pelo javax.text.DateFormat. Os valores possíveis são: default, short, long, medium ou full.
- **TimeZone** – A time zone utilizada.
- **Type** – Uma String que define quando a saída será uma data, uma instância do tipo time, ou ambos. Os valores possíveis são: date, time, both. O valor padrão é date.
- **Pattern** – O padrão de formatação a ser usado na conversão. Se um valor para esse atributo for definido, o sistema irá ignorar qualquer valor para o dateStyle, timeStyle e type.

A tag **<f:convertNumber>**, veja exemplos:

```

<h: outputText value ="#{produto.preco}">
    <f:convertNumber minFractionDigits="2" />
</h:outputText>

<h: outputText value ="#{produto.preco}">
    <f:convertNumber type="currency" currencySymbol="R$" />
</h:outputText>

<h: outputText value ="#{produto.codigo}">
    <f:convertNumber pattern="#0.00" />
</h:outputText>

```

Os seguintes atributos são usados para podermos controlar o comportamento desse componente de conversão NumberConverter:

- **currencyCode** – Código monetário do tipo ISO 4217 a ser utilizado ao formatar.
currencySymbol – O símbolo monetário a ser usado ao formatar.
- **groupInUsed** – Indica se o valor utilizará separadores de grupo (por exemplo: ponto depois de cada 3 dígitos).
- **integerOnly** – Indica que apenas a parte inteira de um número deve ser mostrada. Isso significa que a parte decimal será truncada antes do dado ser armazenado em uma propriedade escondida (bound).
 - **locale** – A localidade usada para referência de formação.
 - **maxIntegerDigits** – O número dígitos mostrados ou usados na parte inteira.
 - **maxFractionDigits** – O número máximo de dígitos mostrados ou usados na parte decimal do número.
 - **minFractionDigits** - O número mínimo de dígitos mostrados ou usados na parte decimal do número.
 - **minIntegerDigits** - O número mínimo de dígitos mostrados ou usados na parte inteira do número.
 - **pattern** - O padrão a ser utilizado quando formatamos ou mostramos o número. Para mais detalhes sobre os padrões permitidos, verifique no JavaDoc por `java.text.NumberFormat`.
 - **type** - Indica quando o número a ser convertido deve ser tratado como moeda, percentual ou número. Para mais detalhes, verifique no JavaDoc por `java.text.NumberFormat`.

Ao utilizar NumberConverter, assim como no DateTimeConverter, sua entrada de dados será validada pelo padrão ou símbolo indicado nos atributos. Se os dados de entrada do usuário não seguirem o padrão ou não apresentarem os símbolos solicitados, um erro de conversão irá ocorrer e o processamento dos dados será paralisado.

Conversor Customizado

```

@FacesConverter("celsiusToFahrenheitConverter")
public class CelsiusToFahrenheitConverter implements Converter {
    Anotação indicando que a classe é um converter
    + nome do converter

    Interface Converter

    @Override
    public Object getAsObject(FacesContext context, UIComponent component, String value) {
        Método que
       recebe a string
        e converte
        para objeto
        Float resultado = 0F;
        try {
            Float celsius = Float.parseFloat(value);
            resultado = (celsius * 9/5) + 32;
        } catch (Exception e) {
            FacesMessage msg = new FacesMessage(
                "Erro de conversão em CelsiusToFahrenheitConverter", "Entrada inválida, tente novamente.");
            msg.setSeverity(FacesMessage.SEVERITY_ERROR);

            throw new ConverterException(msg);
        }
        return resultado; ← Objeto do tipo Float
    }

    @Override
    public String getAsString(FacesContext context, UIComponent component, Object value) {
        Método que
       recebe o objeto
        e converte
        para String
        return value.toString();
    }

}

<h:inputText id="celsius" value="#{conversoresBean.celsiusToFahrenheit}">
    <f:converter converterId="celsiusToFahrenheitConverter"/>
</h:inputText>

```

Componentes de conversão customizados podem ser criados implementando a interface Converter e configurando com a anotação @FacesConverter("celsiusToFahrenheitConverter"), onde "celsiusToFahrenheitConverter" é o ID do conversor. Originalmente podemos configurar conversores no faces-config.xml. A interface Converter define dois métodos:

```

public Object getAsObject (FacesContext ctxt, UIComponent component, String
input)
public Object getAsString (FacesContext ctxt, UIComponent component, Ob-
ject source)

```

O método getAsObject() é chamado pelo JSF nas Fases “Aplicar Valores de Requisição” e “Processar Validação”, este método recebe o componente UI, recupera o valor da string de requisição HTTP, parâmetro input, e deve retornar um objeto Java.

O método getAsString() é invocado na fase “Renderizar Resposta”, convertendo o objeto Java associado componente UI, parâmetro source e retornando uma String a ser renderizada no código html gerado.

No exemplo, o método getAsObject() recebe um objeto String, parâmetro value, que é a temperatura em Celsius digitada pelo usuário, fazemos a conversão com Float.parseFloat(), convertemos da escala Celsius para Fahrenheit e retornamos um objeto do tipo Float. Caso ocorra uma exceção na conversão adicionamos um FacesMessage na fila de mensagens JSF e lançamos uma exceção do tipo ConverterException, dessa forma a mensagem de erro será mostrada por <h:messages> ou <h:messages>.

Já o método getAsString(), nesse caso, é mais simples, já que nosso objeto é do tipo Float, apenas precisamos convertê-lo para String com `toString()`.

Validador Padrão

```
<h:inputSecret id="senha" value="#{validadoresBean.atributoObrigatorio}" required="true">
  <f:validateLength minimum="5"/>
</h:inputSecret>
```

Verifica o comprimento de um valor e o limita no intervalo especificado.

```
<h:inputText id="withdraw" value="#{accountBean.withdraw.amount}">
  <f:validateDoubleRange minimum="20.00" maximum="1000.00" />
</h:inputText>
```

Valida entradas do usuário quando a entrada esperada for do tipo ponto flutuante (float).

Vimos que durante a fase de Processar Validações do ciclo de vida JSF são feitas todas validações atribuídas aos componentes UI da árvore da página. Aqueles em que ocorrer erro na validação ou forem considerados inválidos, será adicionado a mensagem na fila de mensagens do JSF, que retornará a página aplicando as mensagens às tags `<h:message>` ou `<h:messages>` associados.

Validadores JSF padrão:

- **<f:validateRequired>** – usado para validar campos de entrada tipo input text como requerido. Colocar atributo required=true no componente ativa a validação.
- **<f:validateLongRange>** – usado para validar campos de entrada do tipo ‘long’ com limites de valores mínimos e máximos.
 - **<f:validateDoubleRange>** – usado para validar campos de entrada do tipo ‘double’ com limites de valores mínimos e máximos.
 - **<f:validateLength>** – usado para validar tamanho do campo de entrada.
 - **<f:validateRegex>** – usado para validar campos de entrada que satisfação uma expressão regular.
- **<f:validateBean>** – usado para validar grupos de propriedades JavaBeans conforme JSR 303 Bean Validation introduzido no Java EE 6, a especificação define os seguintes validadores @NotNull, @Min, @Max, @Past, @Future, @Size, etc.

Você pode sobrepor as mensagens de validação usando os atributos requiredMessage para campos obrigatórios e o atributo validatorMessage para as mensagens de validação.

```
<h:inputText id="campoNumerico"
    value="#{bean.campoNumerico}"
    required="true"
    requiredMessage="Obrigatório digitar um número inteiro"
    validatorMessage="Numeros devem ser no mínimo 10 e no máximo
100">
    <f:validateLongRange minimum="10" maximum="100"/>
</h:inputText>
```

O lado ruim da solução acima é que ele só funciona para esse componente, se quiser alterar a mensagens de uma maneira global, você precisa seguir o seguintes passos:

1. Registre um recurso de mensagem na sua aplicação usando o faces-config.xml:

```
<application>
    <message-bundle>resources.messages</message-bundle>
</application>
```

*OBS: o arquivo messages.properties estará dentro de WEB-INF/classes/resources/messages.properties

2. Sobrescreva o campo de mensagem que se quer modificar, por exemplo

javax.faces.validator.LengthValidator.MAXIMUM = {1}: Falha ao validar: tamanho é maior do que o permitido de \u201C{o}\u201D.

Validador Customizado

Anotação indicando que a classe é um validator + nome do validator

```
@FacesValidator("emailValidator")
public class EmailValidator implements Validator {
    private static final String EMAIL_PATTERN =
        "^[_A-Za-z0-9]+(\\" . "+" [_A-Za-z0-9-]+)*@[A-Za-z0-9]+(\\" . [A-Za-z0-9]+)*" + "(\\.[A-Za-z]{2,})$";
    private Pattern pattern;
    private Matcher matcher;

    public EmailValidator(){
        pattern = Pattern.compile(EMAIL_PATTERN);
    }

    @Override
    public void validate(FacesContext context, UIComponent component, Object value) throws ValidatorException {
        matcher = pattern.matcher(value.toString());
        if(!matcher.matches()){
            FacesMessage msg = new FacesMessage("Validação de email falhou", "Email informado inválido");
            msg.setSeverity(FacesMessage.SEVERITY_ERROR);

            throw new ValidatorException(msg);
        }
    }
}
```

Interface Validator

```
<h:inputText id="email" value="#{validadoresBean.email}" required="true">
    <f:validator validatorId="emailValidator"/>
</h:inputText>
```

Provavelmente você terá um cenário que necessitará criar um validador específico na sua aplicação por exemplo: número de CPF, CNPJ, telefones, etc.

A criação de um validador implica na implementação da interface javax.faces.validator.Validator e registrá-lo seja pela anotação @FacesValidator("nomeDoValidador") ou no arquivo faces-config.xml

O método validate() repassa uma instância FacesContext da requisição atual, o Componente UIInput e um Object contendo o valor.

Se a validação falhar adicionamos uma objeto FaceMessage contendo a descrição da falha e adicionamos a mensagem à pilha de mensagens do JSF, lançamos uma exceção do tipo ValidatorException. Essa mensagem será apresentada na página por um componente <h:message> associado ao componente UIInput e/ou por <h:messages/>.

No exemplo temos um validador de e-mail, onde utilizamos uma expressão regular e comparamos com o e-mail passado pelo usuários. Para ativar o validador customizado, devemos utilizar a tag <f:validator> e o atributo validatorId passando o ID do validador que criamos, no caso ‘emailValidator’.

Mensagens

→ Mensagem será exibida caso exceção seja lançada.

```
FacesMessage msg = new FacesMessage
    ("Erro de conversão em celsiusToFahrenheitConverter", "Entrada inválida, tente novamente.");
msg.setSeverity(FacesMessage.SEVERITY_ERROR);

throw new ConverterException(msg);
```

```
FacesMessage msg = new FacesMessage("Validação de email falhou", "Email informado inválido");
msg.setSeverity(FacesMessage.SEVERITY_ERROR);

throw new ValidatorException(msg);
```

```
FacesMessage msg = new FacesMessage (severidade, mensagem, detalhes);
FacesContext context = FacesContext.getCurrentInstance();
Context.addMessage(client_id, msg);
```

`FacesMessage.Severity_ERROR`
`FacesMessage.Severity_FATAL`
`FacesMessage.Severity_INFO`
`FacesMessage.Severity_WARN`

Mensagens (FacesMessage) de erro, conversão, validação ou para qualquer outro propósito podem ser enfileiradas na instância do FacesContext.

No caso de erros de validações (ou conversões), o ciclo de vida JSF procede diretamente para a fase “Renderizar Resposta”, e a fila de FacesMessages será mostrada num componente `<h:message>` ou `<h:messages>`.

A severidade da mensagem (FacesMessage.Severity) indica a prioridade de exibição da mensagem, sendo essa a ordem, do menor para o maior:

1. INFO
2. WARN
3. ERROR
4. FATAL

Internacionalização

Arquivo **.properties**

nomeDoArquivo_abreviacao_local.properties

messages_en_US.properties
messages_pt_BR.properties

messages

```
1 msg.title = JSF 2 Internationalization
2 msg.boasVindas = Welcome to the extraordinary world of Java!
messages
1 msg.title = JSF 2 Internacionalização
2 msg.boasVindas = Bem Vindo ao mundo extraordinário do Java!
```

CHAVE = VALOR

As mensagens são identificadas pelas chaves. Somente seus valores devem ser alterados.

Quando você precisar mostrar mensagens em diferentes idiomas, você pode utilizar os mecanismos de Resource Bundle (pacotes de recursos) da JSF. Você simplesmente cria arquivos de propriedades (tem extensão .properties) para os diferentes idiomas.

messages_pt_BR.properties – Português do Brasil.

messages_en_US.properties – Inglês dos Estados Unidos

Cada arquivo conterá o mesmo conjuntos “chaves” enquanto os valores associados satisfazem a tradução desejada. Essas mensagens em diferentes idiomas, uso de símbolos monetários, separadores de casas decimais, formatação de datas, etc, é denominado LOCALIZAÇÃO (ou simplesmente LOCALE, do inglês).

Em Java uma localização é representada pela classe java.util.Locale

```
Locale local = new Locale("pt", "BR"); //localização Português Brasileiro
```

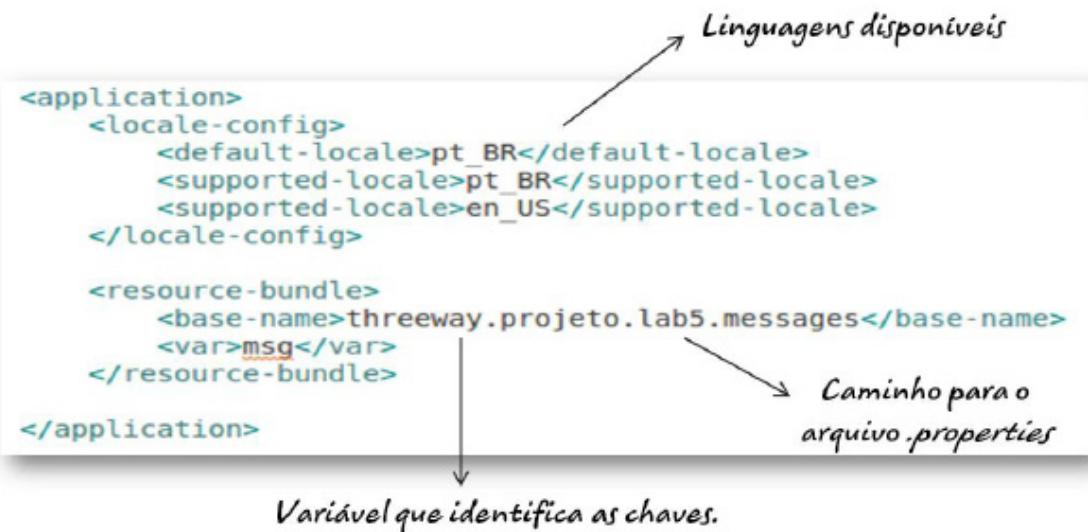
Para carregar o arquivo de localização desejado normalmente você faria assim :

```
Locale local = new Locale("pt", "BR");
ResourceBundle bundle = ResourceBundle.getBundle("messages", local);
```

E em um Managed Bean, após registrar o seu Resource Bundle:

```
FacesContext fctx = FacesContext.getCurrentInstance();
fctx.getViewRoot().setLocale(locale);
```

Mapeamento dos arquivos properties no faces-config.xml



Podemos configurar um locale padrão para a nossa aplicação, assim ela sempre será exibida caso não seja encontrado um locale definido pelo sistema do usuário. A tag <supported-locale> indica quais são os locales/línguas disponíveis no sistema.

Devemos definir o caminho para o arquivo ‘.properties’ e uma variável (“msg”) que guarda a referência para o objeto de recurso. Você pode usar a EL para referenciar as chaves na sua view:

```

<f:view locale="pt_BR">
    <h:outputText value="#{msg['boasVindas']}" />
</f:view>

```

Usamos os <f:view locale=”[código do locale]”>, te permite alterar o idioma na página.

O registro <resource-bundle> no faces-config.xml é uma configuração global, mas você ainda poderia querer utilizar em uma única página. Para isto use a tag <f:loadBundle>:

```
<f:loadBundle basename="threeway.projeto.lab5.messages" var="msg"/>
```

Para mudar as mensagens do sistema nós usamos a tag <message-bundle> no faces-config.xml

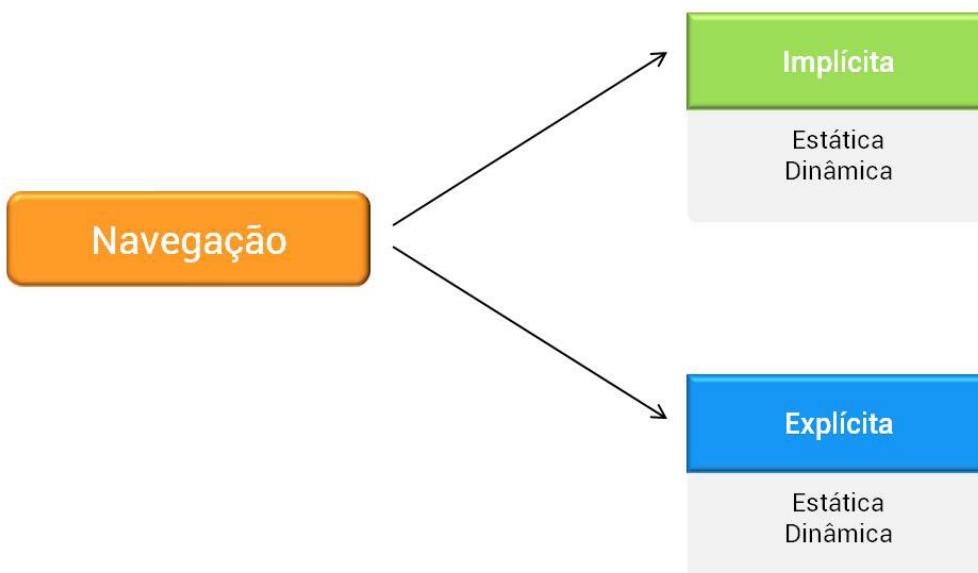
```

<application>
    <message-bundle>resources.messages</message-bundle>
</application>

```

então não confundir <resource-bundle> texto estático traduzido com <message-bundle>, personalizar mensagens são mostradas com <h:message> ou <h:messages>. O arquivo também pode ser carregado com a tag <f:loadBundle>

Navegação



No Framework JSF há dois tipos de navegação:

Navegação Implícita

Nas navegações implícitas, incluídas a partir de JSF 2.0. Veja o exemplo, para navegar da página index.xhtml para página cadastro.xhtml:

```
<h:commandButton action="cadastro"/>
```

No segundo exemplo usando `<h:link>`, que permite navegar da página cadastro.xhtml para a página index.xhtml

```
<h:link outcome="index"/>
```

Esse tipo de navegação é chamado de *implícito* porque não há regra de navegação definida no `faces-config.xml`; você só precisa definir o caminho relativo para a página de destino nos atributos '`action`' e '`outcome`', não há necessidade de informar a extensão do arquivo o sistema de navegação JSF vai anexa-lo para por você.

A grande vantagem da navegação implícita é a simplicidade, entretanto em sistemas com navegação complexa ela pode ser inconveniente se por exemplo você precisar renomear alguma página. Daí você vai ter que revisitar todos as páginas que têm como destino a página renomeada.

Navegação Explícita

O que definimos aqui como explícito são as regras de navegação declaradas dentro do arquivo faces-config.xml. São definidos conjuntos de regras de navegação <navigation-rule>. Cada regra de navegação pode ter um ou mais casos de navegação <navigation-case>:

```
<faces-config ...>
    <navigation-rule>
        <from-view-id>/index.xhtml</from-view-id>
        <navigation-case>
            <from-action>#{exemploBean.actionCadastrar}</from-action>
            <from-outcome>sucesso</from-outcome>
            <to-view-id>/cadastrar.xhtml</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-action>#{exampleBean.actionCadastrar}</from-action>
            <from-outcome>falha</from-outcome>
            <to-view-id>/erro.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Como mostrado no exemplo uma regra de navegação pode ter os seguintes elementos:

- <from-view-id> (Opcional), representa a página de onde começa a navegação.
- <navigation-case>, pode haver vários dentro de um <navigation-rule>.

Um caso de navegação pode conter os seguintes elementos:

- <from-action> (Opcional), envolve uma EL que se refere ao método de ação que retorna a String (outcome).
- <from-outcome>, representa a String literal (o outcome). Se o valor retornado de <from-action> for igual ao valor definido <from-outcome> a navegação é encaminhada para o <to-view-id>. Se <from-action> não estiver presente então o <from-outcome> é comparado com o atributo ‘action’ do componente que disparou a ação, da mesma forma, se os valores casarem, a navegação segue para <to-view-id>
- <to-view-id>, representa a página de destino.

Navegação Bean

Escopo deste Managed Bean → Nome do Managed Bean

```

@ViewScoped
@ManagedBean(name="navegacaoBean")
public class NavegacaoBean implements Serializable {

    private static final long serialVersionUID = -7622250974049755899L;

    public String navegacaoDinamicaImplicitaIndex() {
        return "index";
    }

    public String navegacaoDinamicaImplicitaPaginal() {
        return "pagina";
    }

    public String navegacaoDinamicaExplicitaIndex() {
        return "NavegacaoExplicitaIndex";
    }

    public String navegacaoDinamicaExplicitaPaginal() {
        return "navegacaoExplicitaPaginal";
    }
}

```

Métodos para realizar a navegação dinâmica

Métodos de ação associados aos atributos ‘action’, são métodos públicos sem parâmetros que retornam um literal String. Esses métodos estão associados aos componentes através da EL:

<h:commandButton action="#{navegacaoBean.navegacaoDinamicaImplicitaIndex()}" />

ou estarão associados a um navigation case também através da EL em uma tag <from-action>

```

<faces-config ...>
  <navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
    <navigation-case>
      <from-action>#{exemploBean.actionCadastrar}</from-action>
      <from-outcome>index</from-outcome>
    </navigation-case>
  </navigation-rule>
</faces-config>

```

faces-config.xml

Configura uma regra de navegação entre as páginas. Ainda é possível inserir a tag <from-view-id> para determinar a página de origem da navegação.

```
<navigation-rule>
    <navigation-case>
        <from-outcome> NavegacaoExplicitaIndex </from-outcome>
        <to-view-id> /lab4/index.xhtml </to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <navigation-case>
        <from-outcome> NavegacaoExplicitaPagina1 </from-outcome>
        <to-view-id> /lab4/pagina1.xhtml </to-view-id>
    </navigation-case>
</navigation-rule>
```

Valor de saída do atributo action definido na UI

Página a ser exibida

Cada caso de navegação determina um caminho diferente partindo de uma mesma página.

```

graph TD
    index((index.xhtml)) -- "NavegacaoExplicitaIndex" --> pagina1((pagina1.xhtml))
    index -- "NavegacaoExplicitaIndex" --> index
    
```

As regras de navegação definidas no faces-config.xml, permitem centralizar e manter mais facilmente o mapa de navegação da aplicação.

Como vimos, se <from-action> não estiver presente então o <from-outcome> é comparado com o atributo ‘action’ do componente que disparou a ação e a navegação segue para <to-view-id>. No caso há um componente, como um commandButton ou commandLink, definindo o atributo action de forma estática,

<h:commandButton action=”NavegacaoExplicitaIndex”/>

ou de forma dinâmica, através de um método de ação,

<h:commandButton action=”#{navegacaoBean.navegacaoDinamicaExplicitaPagina1()}”/>

o caminho relativo à página de destino está explicitamente declarada no <to-view-id>.

Navegação Explícita

```

<h:panelGrid cellpadding="2">
  Navegacao Estatica Explicita
  <h:commandButton value="Ir Para Index" action="NavegacaoExplicitaIndex"/>
  <h:commandButton value="Ir para Pagina 1" action="NavegacaoExplicitaPaginal"/>
</h:panelGrid>
<hr/>
<h:panelGrid cellpadding="2">
  Navegacao Dinamica Explicita
  <h:commandButton value="Ir Para Index"
    action="#{navegacaoBean.navegacaoDinamicaExplicitaIndex()}" />
  <h:commandButton value="Ir Para Pagina 1"
    action="#{navegacaoBean.navegacaoDinamicaExplicitaPaginal()}" />
</h:panelGrid>
  
```

Nome do outcome que define a página a ser exibida

Navegação é gerada dentro de um Managed Bean

```

public String navegacaoDinamicaExplicitaPaginal() {
  return "NavegacaoExplicitaPaginal";
}
  
```

Retorna o outcome que define a página a ser exibida.

```

public String navegacaoDinamicaExplicitaIndex() {
  return "NavegacaoExplicitaIndex";
}
  
```

Para realizar a navegação, devemos configurar o atributo action, sendo que na navegação estática, passamos um literal String diretamente e na navegação dinâmica, acessamos os métodos de ação criados no Managed Bean.

Como há um <navigation-case> dentro do faces-config.xml, definido para cada outcome associado ao atributo action, então a navegação é explícita e segue para o <to-view-id> associado. Porém se não houver um <navigation-case> definido no faces-config.xml a navegação é implícita, sendo que sistema de navegação do JSF vai tentar localizar uma página com caminho relativo similar ao outcome.

Navegação Implícita

```

<h:panelGrid cellpadding="2">
    Navegacao Estatica Implicita
    <h:commandButton value="Ir para index" action="index"/>
    <h:commandButton value="Ir para pagina 1" action="pagina1"/>
</h:panelGrid>
<hr/>
<h:panelGrid cellpadding="2">
    Navegacao Dinamica Implicita
    <h:commandButton value="Ir para Index"
        action="#{navegacaoBean.navegacaoDinamicaImplicitaIndex()}" />
    <h:commandButton value="Ir para Pagina 1"
        action="#{navegacaoBean.navegacaoDinamicaImplicitaPaginal()}" />
</h:panelGrid>

    public String navegacaoDinamicaImplicitaIndex() {
        return "index";
    }

    public String navegacaoDinamicaImplicitaPaginal() {
        return "pagina1";
    }
  
```

O nome da página a ser exibida é definido no próprio botão

A navegação é gerada dentro de um managed bean

Retorna o nome da página a ser exibida.

Se não houver um <navigation-case> definido no faces-config.xml a navegação é implícita, sendo que sistema de navegação do JSF vai tentar localizar uma página com caminho relativo similar ao outcome definido no atributo action.

Nos dois casos, atributo action definido de forma estática ou de forma dinâmica, a navegação vai para index.xhtml ou para pagina1.xhtml.

Action Handlers

Executam alguma ação e depois podem redirecionar o usuário para outra tela.

→ O método action executa alguma lógica de negócio, chamando um método em um bean.

```
<h:commandButton value="Ir para Index"  
action="#{navegacaoBean.navegacaoDinamicaImplicitaIndex()}">
```

```
public String navegacaoDinamicaImplicitaIndex() {  
    return "index";  
}
```

→ Método pode retornar uma String ou Void.

Quando um usuário clica em um botão ou link JSF, ou muda o valor de algum campo de texto, ele está executando uma ação, fazendo com que a UI do JSF dispare um evento que poderá ser devidamente capturado e tratado pelo código da sua aplicação.

Vimos anteriormente o atributo action, esse atributo é um Action Handler. Action Handlers executam alguma ação e depois podem redirecionar o usuário para outra tela. Um exemplo é mudança de página ao clicar em algum botão. São geralmente utilizados para realizar a navegação entre páginas.

Event Listeners

O ActionListener Executa uma lógica relacionada a view.

```
<h:commandButton value="Listener"  
actionListener="#{algumBean.myActionListener}"/>
```

```
public void myActionListener (ActionEvent event) {  
    //lógica  
}
```

Recebe um ActionEvent
como parâmetro

Se você não precisa de navegação você pode usar action listener. Um método action listener pode ser associado a um componente UICommand usando seu atributo actionListener. Um método action listener não retorna um outcome para o sistema de navegação JSF, seu tipo de retorno é void e tem um parâmetro do tipo ActionEvent.

Como boa prática sempre use métodos de ação para lógica de negócio que possa envolver navegação para uma nova página e use métodos action listener para inicializar o trabalho antes de executar um método de ação.

```
<h:commandButton actionListener="#{favorito.logFavoritoCorrente}"  
action="#{favorito.alterarFavorito}" />
```

Os métodos actionListener são sempre executados antes dos métodos action.

Facelets

XHTML

- Alta reutilização de código
- Até 50% mais rápido """"""

Facelets

Definição de custom tags

Criação de templates

Reuso de telas

Facelets é um framework para templates web, utilizado como view padrão do JSF 2. Pode ser definido como uma seção reutilizável de conteúdo ou uma sub-árvore de componentes em uma página JSF que pode ser por si mesma composta de outros Facelets.

Facelets possui várias vantagens que vão desde a facilidade na criação e reutilização de páginas e componente, melhor depuração de erros, Ajax nativo, uma melhor compatibilidade entre XHTML, JSTL e os componentes, ele é independente de web container, Facelets é mais rápido que JSP.

Outro ponto positivo é que Facelets tem suporte a Unified Expression Lanaguage (EL), incluindo suporte para funções EL e validação EL em tempo de compilação.

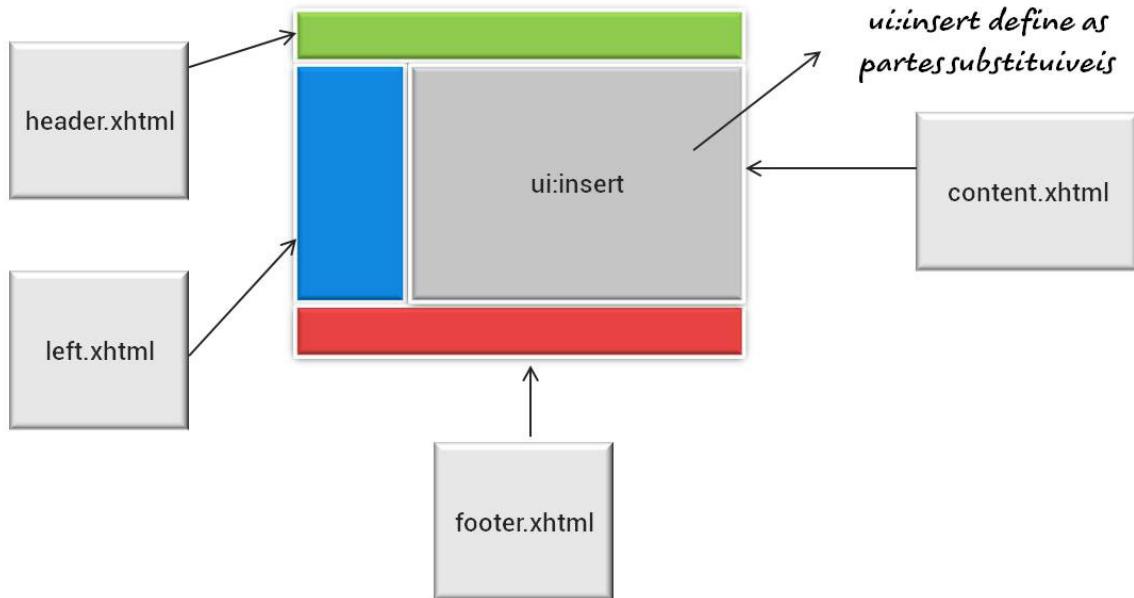
Para utilizar os elementos do framework Facelets declare a taglib:

xmlns:ui="http://xmlns.jcp.org/jsf/facelets"

ou

xmlns:ui="http://java.sun.com/jsf/facelets"

Templates



Template

Ao longo do processo de desenvolvimento de uma aplicação web, percebe-se que várias páginas têm uma estrutura semelhante, o uso de template vai permitir ao desenvolvedor web a reutilização de código das telas, utilizando-se uma página padrão como um esqueleto e que possua pontos para que sejam substituídos posteriormente. Em JSP o recurso utilizado para o reaproveitamento de código é o include, porém este tem várias desvantagens comparado com o templates.

Facelets Template

Templates permitem a reutilização de código, reduzindo assim, o tempo de desenvolvimento e os custos de manutenção de uma aplicação. Além disso, templates nos auxiliam na obtenção de uma aparência comum em páginas que possuem uma estrutura semelhante. Um template define pontos onde o conteúdo pode ser substituído. O conteúdo a ser usado nesses locais, é definido pelo cliente do template (Template Client). O Template define as diferentes áreas usando a tag <ui:insert>, e os clientes utilizam os modelos com as tags: <ui:component>, <ui:composition>, <ui:fragment>, ou <ui:decorate>.

Página de template criada layout.xhtml

ui:include inclui o conteúdo de uma página XHTML em outra página XHTML

O componente incluso deve estar contido em um **ui:composition**

A tag **ui:insert** serve como um container para definir que a área do template pode ser substituída

```
<h:body>
    <div id="pagina">

        <div id="header" class="border">
            <ui:insert name="header">
                <ui:include src="header.xhtml" />
            </ui:insert>
        </div>

        <div class="content">
            <div id="Left" class="Col_left border">
                <ui:insert name="left">
                    <ui:include src="left.xhtml" />
                </ui:insert>
            </div>

            <div id="content" class="Col-center border">
                <ui:insert name="content">
                    <h2>Conteúdo a ser substituído</h2>
                </ui:insert>
            </div>
        </div>

        <div id="footer" class="border">
            <ui:insert name="footer">
                <ui:include src="footer.xhtml" />
            </ui:insert>
        </div>
    </div>
</h:body>
```

Essa template divide uma página em quatro áreas: header, left (painel esquerdo), content (painel central) e footer (rodapé ou base da página).

Dentro de header, left, footer e content há uma tag **<ui:insert>**, que é utilizada para declarar um conteúdo inicial que pode ser substituído por páginas que usem a template. Em header, left e footer esse conteúdo inicial é incluído pela tag **<ui:include src="..." />** que inclui o conteúdo de outros arquivos XHTML.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<h:html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head></h:head>

<h:body>
 <ui:composition template="layout.xhtml">
 <ui:define name="content">
 
 <ui:include src="content/content.xhtml" />
 
 </ui:define>
 </ui:composition>
</h:body>
</h:html>
```

pagina1.xhtml

A tag `ui:define` define qual região do template será substituída.

Qualquer conteúdo incluso dentro da tag `ui:composition` será incluso quando outra página Facelets incluirá página contendo a tag.

content.xhtml

Aqui fica o conteúdo da página. Podemos adicionar um link para uma segunda página que irá substituir a página1.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<h:html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head></h:head>

<h:body>
 <ui:composition>
 <h1>Conteúdo inserido e substituível</h1>
 <p><a href="pagina2.xhtml">Next</a></p>
 </ui:composition>
</h:body>
</h:html>
```

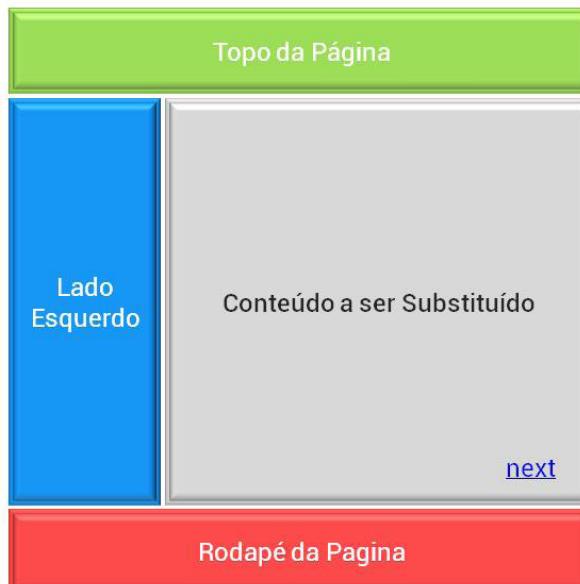
A esquerda desse exemplo temos a página (pagina1.xhtml) que faz o uso da página template (layout.xhtml) que acabamos de mostrar.

Através da tag `<ui:composition template="[relative path]">` definimos qual será o template utilizado. Na página template (layout.xhtml) definimos a tag `<ui:insert name="content">`, essa tag define uma área substituível no template.

Para sobrepor o conteúdo atual da template pelo novo conteúdo desejado, usamos a tag `<ui:define name="content">`, onde o atributo name refere-se ao mesmo nome usado na definição da tag `<ui:insert>`. Então todo no conteúdo criado dentro da tag `<ui:define>` será apresentado na área de conteúdo da página.

Nesse caso temos a página content.xhtml que contém um texto e um link para uma segunda página, e está sendo incluído em pagina1.xhtml através da tag `<ui:include>`.

Executando a página1.xhtml



```
<body>
<h:form>
<ui:include src="ola.xhtml">
    <ui:param name="saudacoes" value="#{bean.message}" />
</ui:include>
</h:form>
</body>
```

OLA.XHTML

```
<body>
<ui:composition>
    <h2><h:outputText value="#{saudacoes}" /></h2>
</ui:composition>
</body>
```

A tag `<ui:param>` é um tag que permite passar parâmentros para arquivos de inclusão (`<ui:include>`) ou para template (`<ui:composition>`). No exemplo você está enviando o parâmetro “saudacoes” para a página de inclusão (ola.xhtml), que recupera o valor através da EL `#{{saudacoes}}`.

O mesmo processo pode ser usado com as páginas de template na tag `<ui:composition>`

```
<ui:composition template="/WEB-INF/templates/layout.xhtml">
    <ui:param name="saudacoes" value="#{bean.message}" />
</ui:composition>
```

