



tópicos seleccionados de java ee



Helder da Rocha

Conteúdo

1. Introdução
2. WebServlets
3. CDI, Interceptadores, Bean Validation
4. Java Persistence API (JPA)
5. Enterprise JavaBeans (EJB)
6. SOAP Web Services
7. JavaServer Faces (JSF)

1: Introdução

1	Java Enterprise Edition	1
1.1	Containers e componentes	2
1.2	Serviços e especificações Java EE	2
1.3	Arquitetura de aplicações Java EE	3
2	Serviços essenciais	4
2.1	JNDI	4
2.2	Objetos remotos	4
2.3	DataSources	4
2.4	UserTransaction	5
2.5	Conexões JMS	5
2.6	Serviços de autenticação e autorização	5

1 Java Enterprise Edition

O termo *enterprise* refere-se a aplicações complexas, que requerem grande investimento em arquitetura. São aplicações que precisam de *serviços* como segurança e distribuição, que requerem robustez e integridade dos dados compartilhados, escalabilidade e facilidade de manutenção. Esses serviços exigem conhecimentos e recursos que vão além da lógica de negócios da aplicação. São serviços proporcionados por *servidores de aplicação*, que oferecem ambientes controlados para execução de componentes, com serviços de middleware implícito e uma arquitetura container-componente. Serviços típicos dos servidores de aplicações incluem transações, autenticação e autorização, gerenciamento de recursos (ciclo de vida), agendamento, persistência transparente, sistema de nomes e diretórios e injeção de dependências.

Java EE - Java Enterprise Edition, fornece uma interface padrão comum para componentes e containers em diferentes servidores de aplicação. Java EE também descreve uma série de serviços padrão que podem ser usados pelos componentes. Essa arquitetura e serviços são descritos em especificações desenvolvidas pelos usuários e principais fabricantes, através de processos públicos da comunidade Java (Java Community Process), através de JSRs (*Java Specification Request*). Todas as especificações do Java EE são JSRs.

1.1 Containers e componentes

As especificações do Java EE descrevem um contrato entre *containers*, criados pelos fabricantes de ferramentas, e *componentes*, que irão executar nos containers, criados pelos programadores. Containers funcionam como pequenos sistemas operacionais, e os componentes são como suas aplicações.

Containers são fornecidos por servidores de aplicação que implementam as especificações Java EE e podem executar componentes que aderem ao contrato descrito nessas especificações. Eles controlam a execução e ciclo de vida de seus componentes, que pode ser ajustada através de configuração declarativa, em arquivos XML ou anotações. Toda a comunicação container-componente ocorre através de chamadas (métodos de *callback*) que são geralmente identificados no código através de anotações.

A instalação de um componente em um container configura a implantação (*deployment*) de um serviço, que muitas vezes envolve a geração e compilação automática de código, criação ou alteração de tabelas em bancos de dados e inclusão de registros em serviços globais de localização.

Servidores Java EE fornecem dois tipos de containers:

- Container Web
- Container para Enterprise Java Beans

Além disso, usando um *runtime* gerado pelo servidor, clientes de aplicação pode rodar nas plataformas do cliente e ter acesso, remotamente, a serviços implantados pelos componentes no servidor. Este *runtime* funciona como um *container cliente*.

1.2 Serviços e especificações Java EE

Serviços como registro de nomes, bancos de dados, transações distribuídas, autenticação e autorização fazem parte de qualquer servidor Java EE. Esses serviços são acessíveis aos componentes através de APIs e frameworks, descritos em diversas especificações.

As APIs do Java EE 7 são:

- EJB, usada para construir componentes Session Bean ou Message-driven Bean
- Java Servlet, usada para construir WebServlets e como infraestrutura básica para frameworks Web
- JSF, framework para aplicações Web
- JSP, tecnologia antiga para construir páginas Web
- JSTL, biblioteca de tags para páginas JSP
- JPA, framework ORM para criar objetos persistentes
- JTA, API para demarcação de transações distribuídas
- JAX-RS, framework para desenvolver aplicações RESTful (Web Services)
- Managed Beans, tecnologia antiga para criar componentes gerenciados leves
- CDI, serviços de contextos, injeção de dependências e componentes gerenciados
- DI, serviços de injeção de dependências
- Bean Validation, framework para validação automática de beans
- JMS, API para acesso a serviços de mensageria
- JCA, API para construir adaptadores para serviços externos
- JavaMail, API para envio de email
- JACC, API para serviço de autorização
- JASPIC, API para serviços de autenticação

- WebSocket API, suporta o protocolo WebSocket
- JSON-P, API para processamento JSON
- Utilitários de concorrência
- API para aplicações em lote

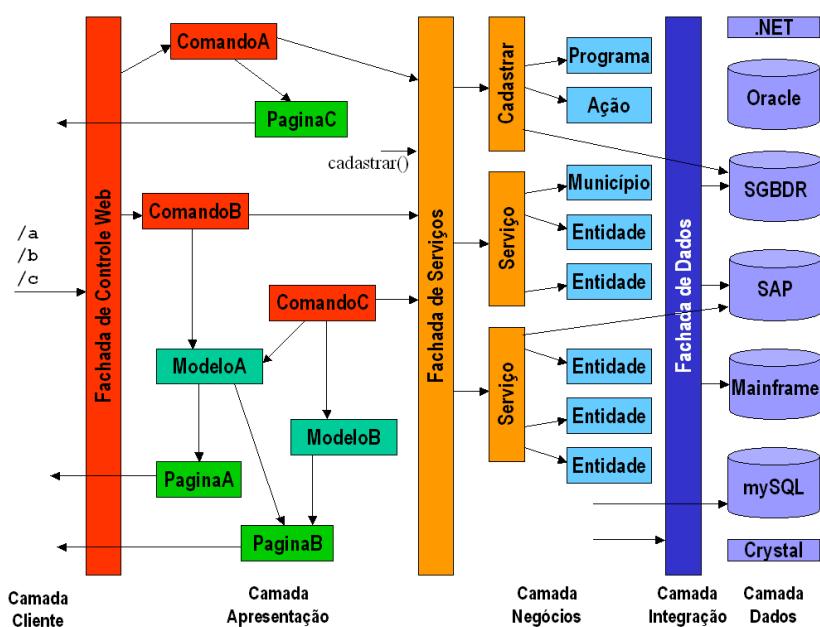
Além dessas APIs, servidores também oferecem serviços que são acessíveis através de APIs do Java SE 8:

- JDBC
- JNDI – para acesso a serviços de nomes e registro de serviços
- JavaBeans Activation Framework
- JAXP – API para processamento XML
- JAXB – API de mapeamento Objeto-XML
- JAX-WS – API para construção de Web Services SOAP
- SAAJ – API para construir mensagens SOAP
- JAAS – Serviço de autenticação e autorização
- Common Annotations – Anotações comuns

Vários desses serviços precisam ser configurados e ativados através de ferramentas do servidor, para que possam ser usados pelos componentes através de interfaces independentes de fabricante. Este tutorial assume que existe um servidor de aplicações completamente compatível com todas as especificações Java EE 7 instalado e configurado para implantar aplicações. Os módulos seguintes não exploram detalhes de um servidor específico, mas restringem-se às especificações Java EE 7.

1.3 Arquitetura de aplicações Java EE

Aplicações Java EE se dividem em pelo menos três camadas independentes: apresentação, negócios e integração. A camada de apresentação é controlada pelo Web container, e a camada de negócios, pelo EJB container. Adaptadores diversos viabilizam a camada de integração, que permitem plugar serviços como banco de dados, transações, etc. As camadas se comunicam entre si através de registros que são usados para localizar objetos e serviços. A ligação entre objetos e serviços geralmente é feita de forma declarativa usando injeção de dependências. Vários diferentes componentes são usados em cada camada. O desenho abaixo ilustra uma visão da arquitetura de uma aplicação Java EE:



Em Java EE, entidades são tipicamente modeladas com objetos e JPA. Serviços são modelados com EJB, e a camada de apresentação em JSF (com páginas XHTML como views, e beans gerenciados como modelos e controladores).

2 Serviços essenciais

Esta seção descreve alguns serviços essenciais de servidores de aplicação. Eles serão detalhados nas seções que abordam componentes e serviços que os utilizam dentro dos containers. O uso desses serviços em aplicações externas ao servidor muitas vezes utiliza protocolos proprietários para acesso fora do container, por clientes standalone.

2.1 JNDI

JNDI é uma interface padrão para registrar serviços e objetos que podem ser localizados por clientes. Para clientes internos ao servidor (ex: clientes de aplicação e aplicações Web) geralmente usa-se injeção de dependências (DI) ou CDI. Para clientes externos é necessário ter um cliente JNDI disponível, e pode haver configuração específica do servidor.

Através de JNDI, um cliente externo poderá se comunicar com um EJB disponível no servidor, ter acesso a um datasource com conexões de banco de dados, e iniciar e cometer uma transação distribuída. Esses serviços podem estar liberados ou serem negados a clientes externos.

A configuração do cliente geralmente requer que a aplicação cliente tenha acesso a um ou mais JARs que implementam o cliente de acesso, e defina pelo menos duas propriedades para inicializar o contexto JNDI.

JNDI será explorado no capítulo sobre EJB para acesso remoto a EJBs e serviços JMS.

2.2 Objetos remotos

EJBs e objetos distribuídos disponibilizados no servidor usam protocolos de objetos distribuídos como Java RMI e IIOP para transmissão, o que possibilita inclusive o uso por clientes escritos em outras linguagens. Muitas vezes é preciso fazer uma conversão desses objetos para que possam ser usados remotamente.

Objetos remotos também podem ser distribuídos através da exportação de interfaces de Web Services. Serviços SOAP exportam WSDL que permite clientes em qualquer linguagem. Serviços RESTful exportam interfaces HTTP a partir de EJBs.

Objetos remotos serão abordados nos capítulos sobre EJB e Web Services.

2.3 DataSources

DataSources são usados pelos servidores para prover serviços de persistência e podem também ser usadas por outros serviços que dependem de armazenamento, como mensageria. Se o servidor permitir o acesso remoto a um DataSource, ele deve ter um nome JNDI global publicado no serviço. Sabendo-se este nome, ele pode ser usado para fazer lookup e obter um DataSource do servidor, que permite realizar conexões e enviar declarações SQL via JDBC.

DataSources serão usados para conectar JDBC a partir de servlets e para configurar serviços de persistência JPA.

2.4 UserTransaction

Em servidores JBoss ou WildFly um proxy para UserTransaction pode ser obtido através de uma chamada JNDI para `java:/UserTransaction`. Para outros servidores, se habilitado, é necessário descobrir qual o nome usado, para que, de forma semelhante a DataSource, seja feito o lookup para obter um proxy, que permitirá abrir e cometer transações.

UserTransaction será explorado em mais detalhes nos capítulos sobre EJB e servlets.

2.5 Conexões JMS

Todo servidor Java EE é obrigado a suportar JMS. Clientes externos podem obter proxies para conexões JMS e para canais de mensageria através de JNDI. Assim como todos os outros acessos externos, isto é dependente de servidor.

JMS será introduzido no capítulo sobre EJB onde é usado para configurar Message-Driven beans e para construir componentes que disponibilizam operações assíncronas.

2.6 Serviços de autenticação e autorização

Java EE 7 não fornece uma interface uniforme para serviços de segurança. Cada componente possui uma API e configuração diferente para serviços de autorização. Embora exista uma API para autenticação (JASPI), ela não é utilizável sem recorrer a recursos proprietários, e a API de autorização (JAAC) é usada, de forma independente de fabricante, apenas declarativamente. Também não há uma forma padrão para autenticação Single-Sign-On ou OAuth.

A solução, sem recorrer a componentes proprietários, é usar uma camada de segurança proporcionada por terceiros, como por exemplo, Spring Security.

Este tutorial não irá abordar serviços de autenticação e autorização.

2: WebServlets

1	Introdução	2
1.1	Arquitetura Web	2
1.2	Aplicações Web (WAR)	3
1.3	Configuração	4
2	WebServlets	4
2.1	Criando um WebServlet	6
2.2	Mapeamento do servlet no contexto	6
2.2.1	Configuração de inicialização	7
2.3	Requisição	8
2.3.1	Parâmetros de requisição	8
2.4	Resposta	9
2.4.1	Geração de uma resposta	10
2.5	Acesso a componentes da URL	10
3	O Contexto	11
3.1	Inicialização do contexto	11
3.2	Resources	12
3.3	Atributos no escopo de contexto	12
3.4	Listeners	12
4	Sessões	13
4.1	Atributos de sessão	13
4.2	Validade e duração de uma sessão	14
5	Escopos	14
6	Cookies	15
7	Filtros	16
8	JSP (JavaServer Pages) e Taglibs	17
8.1	JSTL	17
8.2	EL (Expression Language)	18

1 Introdução

WebServlets são a base das aplicações Web em Java EE. Embora a principal API para criação de aplicações Web seja *JavaServer Faces*, *WebServlets* é a plataforma básica sobre a qual *JSF* é construído, e ainda serve de suporte a vários outros serviços fornecidos via Web, como *WebServices REST* e *SOAP*, *WebSockets*, *Filtros*, além de ser a base para frameworks Web de outros fabricantes (Spring MVC, Wicket, GWT, etc.)

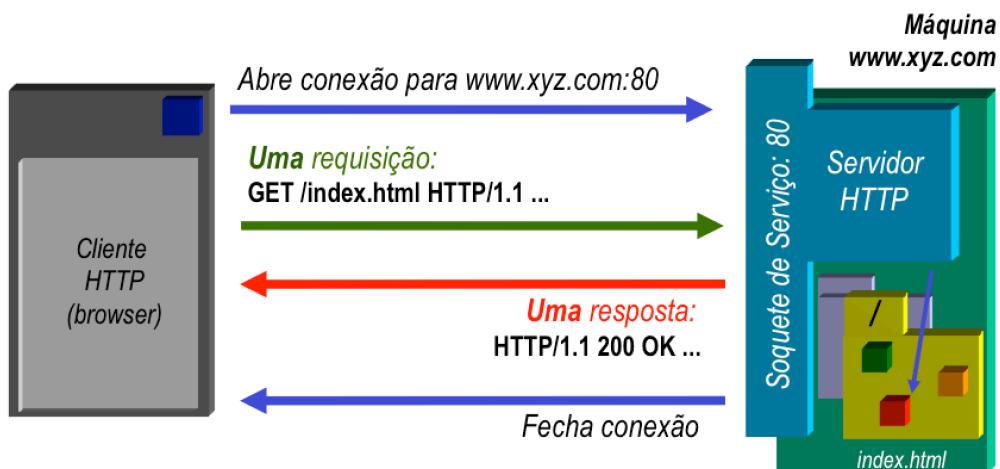
1.1 Arquitetura Web

WebServlets são classes que representam uma *requisição* e *resposta HTTP* em uma aplicação Web. A arquitetura Web é baseada em *cliente* (geralmente um browser ou um script), *protocolo HTTP* e *servidor Web* (que pode estar sob controle de um servidor de aplicações) que na maior parte das vezes serve arquivos estáticos (paginas, imagens, etc.) mas que pode ser configurado para executar comandos e servir dados dinâmicos.

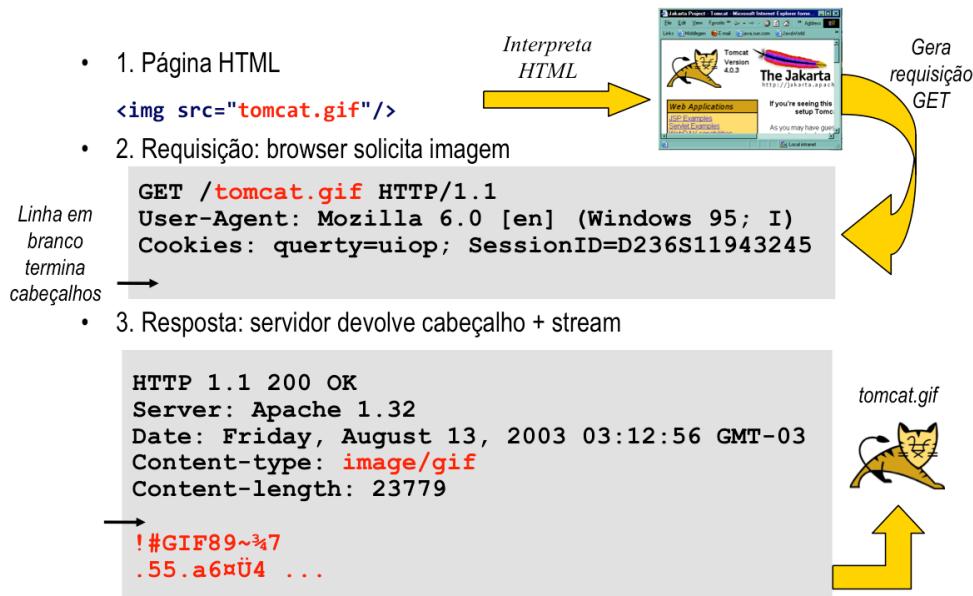
A arquitetura Web é centrada no protocolo de transferência de arquivos HTTP (padrão Internet, RFC 2068), que possui como uma das principais diferenças entre outros protocolos de transferência de arquivos (como FTP e WEBDAV) a característica de não manter o estado da sessão do cliente.

O servidor representa um sistema de arquivos virtual e responde a comandos que contém URLs de localização para cada recurso disponibilizado. Os comandos HTTP (requisições) e as respostas contém cabeçalhos com meta-informação sobre a comunicação.

O diagrama abaixo ilustra uma requisição e resposta HTTP realizada por um browser a um servidor Web:



A figura a seguir mostra detalhes dos cabeçalhos de requisição e resposta gerados quando um browser interpreta um tag HTML que requer a criação de uma requisição GET, e a correspondente resposta do servidor retornando a imagem solicitada:



1.2 Aplicações Web (WAR)

Ha várias especificações Java que envolvem serviços Web. A principal é a especificação de *WebServlets*, que descreve não apenas como construir *servlets*, mas como empacotar aplicações Web para implantação em containers Web Java EE.

Uma *aplicação Web* em Java (no Java EE 7) consiste de pelo menos um arquivo HTML empacotado em um *arquivo WAR*, contendo uma pasta *WEB-INF*. Geralmente consiste de vários arquivos XHTML, JSP ou HTML, junto com imagens, arquivos JavaScript e CSS, arquivos XML de configuração, arquivo *web.xml*, arquivos *.properties*, JARs e classes Java.

O arquivo WAR é um ZIP que contém uma estrutura de arquivos e diretórios. A raiz do arquivo corresponde a raiz do contexto da aplicação, e quaisquer arquivos ou pastas lá contidas serão por default acessíveis aos clientes Web. A pasta WEB-INF é privativa e pode conter arquivos que não serão expostos na URL de acesso à aplicação. Dentro da pasta WEB-INF fica o arquivo *web.xml*, se existir, e se houver classes Java, como *servlets*, *managed beans*, etc. elas ou são localizadas dentro de WEB-INF/classes (que é o classpath) ou empacotadas em um JAR dentro de WEB-INF/lib:

```

biblioteca.war
  index.xhtml
  login.xhtml
  logo.png
  css/
    biblioteca.css
  js/
    jquery.js
    app.js
  WEB-INF/
    web.xml
    classes/
      br/com/unijava/biblioteca/
      CadastroServlet.class
      EmprestimoServlet.class
    lib/
      ojdbc7.jar
  
```

O contexto da aplicação geralmente é mapeado como um subcontexto da raiz de documentos do servidor (que geralmente é `/`). Por default, o nome do contexto é o nome do WAR, mas é comum

que esse nome seja alterado na configuração da aplicação. Por exemplo, a aplicação `biblioteca.war`, publicada em um servidor Tomcat localizado em `http://localhost:8080` será acessível, por default, em

`http://localhost:8080/biblioteca`

1.3 Configuração

O arquivo `web.xml` é o *Web Deployment Descriptor*. Em aplicações Java EE 7 muito simples ele é opcional. Se estiver presente deve aparecer dentro de WEB-INF. A maior parte das configurações realizadas no `web.xml` podem também ser realizadas via anotações, no entanto a configuração em `web.xml` tem precedência e sobrepõe as configurações em anotações.

Dependendo do servidor usado, e do tipo e complexidade da aplicação, poderá ser necessário incluir outros arquivos de configuração além do `web.xml` em uma aplicação Web. A estrutura mínima do `web.xml` é:

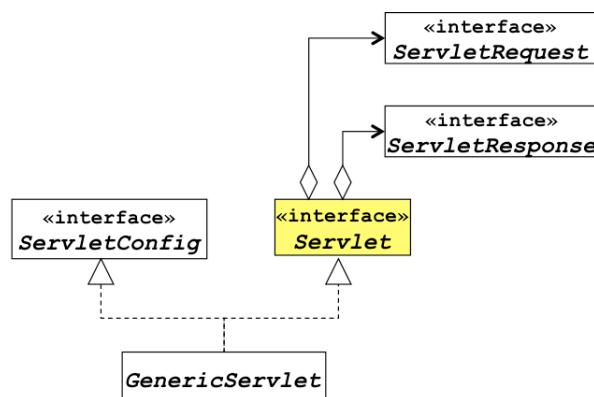
```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

</web-app>
```

2 WebServlets

WebServlets são um tipo de `javax.servlet.Servlet` – componente que encapsula um serviço em um servidor. A API de servlets contém interfaces genéricas que permite a construção desses componentes para qualquer tipo de servidor, mas a única implementação disponível é para servlets HTTP ou WebServlets.

Todo servlet possui um método `service()` que representa o serviço realizado pelo servidor. Este método recebe um par de objetos que representam a requisição (`ServletRequest`), feita pelo cliente, e a resposta (`ServletResponse`), feita pelo servidor. Dentro do método é possível extrair informações do objeto da `ServletRequest`, e usar o `ServletResponse` para construir uma resposta. A hierarquia de interfaces está mostrada abaixo:

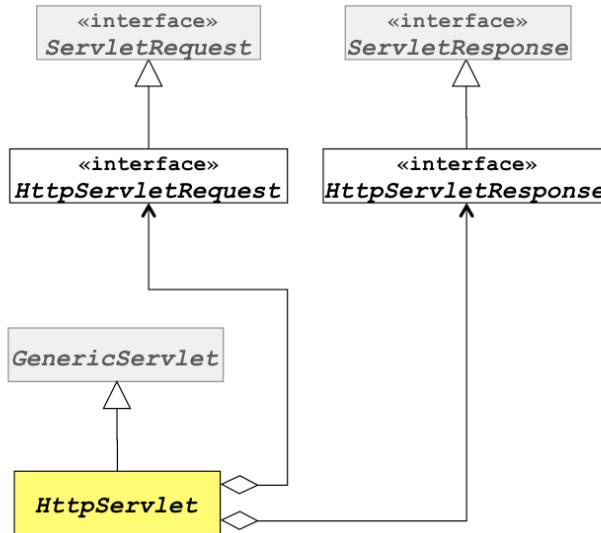


O *WebServlet* estende essas interfaces no pacote `javax.servlet.http` e implementa `GenericServlet` para lidar com os protocolos da Web. Em vez de um método `service()`, o `HttpServletRequest` disponibiliza métodos Java para cada um dos métodos HTTP: `GET, POST, PUT, HEAD, DELETE, OPTIONS`. Sua

implementação de `service()` redireciona para um método `doXXX()` correspondente (`doGet()`, `doPost()`, etc).

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {...}
public void doPost(HttpServletRequest request,
                    HttpServletResponse response) {...}
...
...
```

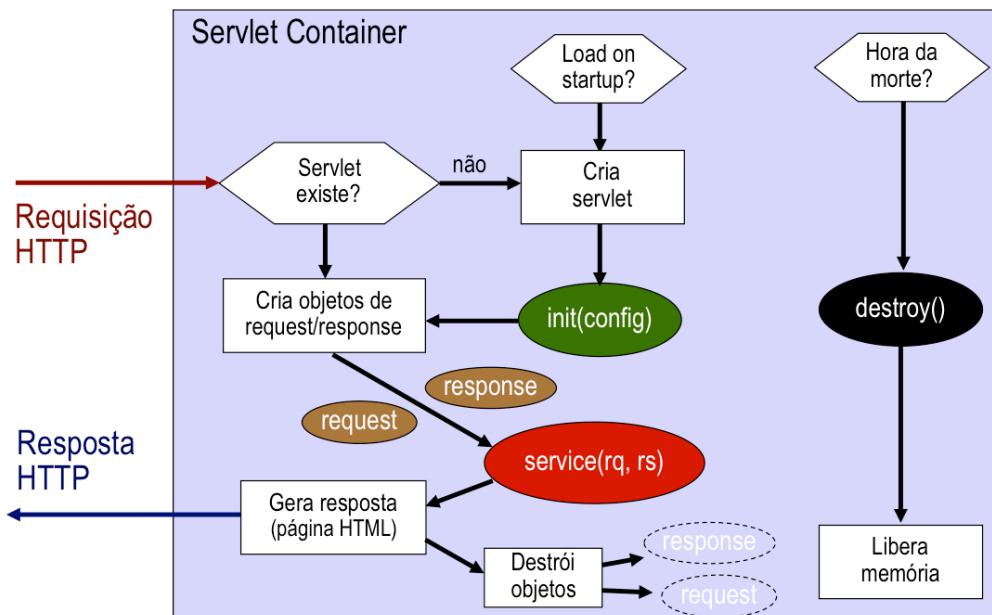
A hierarquia de `HttpServlet` está mostrada abaixo:



O container Web controla o ciclo de vida de um `WebServlet`. Após a implantação do WAR, o servlet pode ser carregado e inicializado (ou inicializado com a primeira requisição). Enquanto estiver ativo, pode receber requisições.

A cada requisição objetos `request` (`HttpServletRequest`) e `response` (`HttpServletResponse`) serão criados, configurados e passados para o método `service()`, que o repassa a um método `doGet()`, `doPost()`, ou outro de acordo com o protocolo HTTP recebido. Ao final da execução do método, os dois objetos são destruídos.

O servlet permanece ativo até que seja destruído (isto depende de política do container ou servidor). A ilustração abaixo mostra as etapas do ciclo de vida:



2.1 Criando um WebServlet

Para criar um *WebServlet* é necessário criar uma classe que estenda a classe *HttpServlet* e implementar pelo menos um dos métodos *doGet()* ou *doPost()*. Só um dos dois será chamados automaticamente quando ocorrer uma requisição HTTP. Cada um deles recebe objetos *HttpServletRequest* e *HttpServletResponse* que representam, respectivamente, a requisição e a resposta HTTP. O *request* vem preenchido com dados da requisição e cabeçalhos, e possíveis parâmetros e cookies enviados pelo cliente, que podem ser lidos dentro do método. A *resposta* deve ser configurada e preenchida, obtendo seu stream de saída para produzir uma página de resposta.

A classe deve ser anotada com `@WebServlet` e o caminho para executar o servlet dentro do contexto da aplicação. Depois de compilada, deve ser incluída em *WEB-INF/classes*, empacotada no WAR e instalada no servidor.

Abaixo um servlet simples que apenas gera uma resposta em HTML, acessível em `http://servidor:porta/contexto/HelloServlet`:

```
@WebServlet("/HelloServlet")
public class ServletWeb extends HttpServlet {
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {

        Writer out = response.getWriter();
        response.setContentType("text/html");
        out.println("<h1>Hello</h1>");
    }
}
```

2.2 Mapeamento do servlet no contexto

Para ser acessado, o *WebServlet* precisa ser mapeado a um caminho acessível no contexto. Por exemplo, se na raiz contexto há uma página HTML, ela pode ter um link para o servlet ilustrado anteriormente através de uma URL relativa:

```
<a href="HelloServlet">Link para o servlet</a>
```

O nome escolhido pode ser qualquer um, incluindo caminhos dentro do contexto como `/caminho/para/servlet` ou mesmo curingas (ex: mapear a `*.xyz` irá redirecionar qualquer URL terminada em `.xyz` para o servlet). Exemplos de mapeamentos possíveis incluem:

- **/nome** – mapeamento exato (relativo a contexto)
- **/nome/subnome** – mapeamento exato (relativo a contexto)
- **/** - servlet default (chamado se nenhum dos outros mapeamentos existentes combinar com a requisição)
- **/nome/*** - mapeamento de caminho (relativo a contexto). Aceita texto adicional (que é tratado como *path info*) após nome do servlet na requisição.
- **/nome/subnome/*** - outro exemplo de mapeamento de caminho
- ***.ext** - mapeamento de extensão. Qualquer URL terminando nesta extensão redireciona a requisição para o servlet.

O mapeamento pode ser declarado usando a anotação `@WebServlet`, contendo pelo menos um mapeamento no atributo `value`, que é o default:

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

Ou no atributo *urlPatterns*, que aceita vários mapeamentos:

```
@WebServlet(urlPatterns = {"/listar", "/detalhar"})
public class ProdutoServlet extends HttpServlet { ... }
```

O servlet também pode ser mapeado no arquivo *web.xml*, usando dois conjuntos de tags (um para associar a classe do servlet a um nome, e o outro para associar o nome a um mapeamento de URL):

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.ServletWeb</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>umServlet</servlet-name>
    <url-pattern>/HelloServlet</url-pattern>
</servlet-mapping>
```

2.2.1 Configuração de inicialização

Parâmetros de inicialização podem ser definidos para cada instância de um servlet usando o elemento *<init-param>* dentro de *<servlet>*:

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.Meuservlet</servlet-class>
    <init-param>
        <param-name>dir-imagens</param-name>
        <param-value>c:/imagens</param-value>
    </init-param>
    <init-param> ... </init-param>
</servlet>
```

Esses parâmetros podem ser recuperados dentro do servlet através de seu método de inicialização:

```
private java.io.File dirImagens = null;

public void init() throws ServletException {

    String dirImagensStr = getInitParameter("dir-imagens");

    if (dirImagensStr == null) {
        throw new UnavailableException("Configuração incorreta!");
    }

    dirImagens = new File(dirImagensStr);

    if (!dirImagens.exists()) {
        throw new UnavailableException("Diretório de imagens não existe!");
    }
}
```

Também é possível definir parâmetros de inicialização no próprio servlet usando *@WebInitParam* (embora não pareça ter tanta utilidade declará-los no mesmo arquivo onde ele é recuperado):

```
@WebServlet(
    urlPatterns = "/HelloServlet",
    initParams = @WebInitParam(name = "dir-imagens", value = "C:/imagens")
)
public class ServletWeb extends HttpServlet { ... }
```

2.3 Requisição

Uma requisição HTTP feita pelo browser tipicamente contém vários cabeçalhos RFC822 (padrão de cabeçalhos de email):

```
GET /docs/index.html HTTP/1.0
Connection: Keep-Alive
Host: localhost:8080
User-Agent: Mozilla 6.0 [en] (Windows 95; I)
Accept: image/gif, image/x-bitmap, image/jpg, image/png, /*
Accept-Charset: iso-8859-1, *
Cookies: jsessionid=G3472TS9382903
```

Os métodos de *HttpServletRequest* permitem extrair informações de qualquer um deles. Pode-se também identificar o método e URL. Estas e outras informações sobre a requisição podem ser obtidas através dos métodos do objeto *HttpServletRequest*. Alguns deles estão listados abaixo:

- Enumeration **getHeaderNames()** - obtém nomes dos cabeçalhos
- String **getHeader("nome")** - obtém primeiro valor do cabeçalho
- Enumeration **getHeaders("nome")** - todos os valores do cabeçalho
- String **getParameter(param)** - obtém parâmetro HTTP
- String[] **getParameterValues(param)** - obtém parâmetros repetidos
- Enumeration **getParameterNames()** - obtém nomes dos parâmetros
- Cookie[] **getCookies()** - recebe cookies do cliente
- HttpSession **getSession()** - retorna a sessão
- void **setAttribute("nome", obj)** - define um atributo obj chamado "nome".
- Object **getAttribute("nome")** - recupera atributo chamado nome

2.3.1 Parâmetros de requisição

Parâmetros são pares nome=valor que são enviados pelo cliente concatenados em strings separados por &:

```
nome=Jo%E3o+Grand%E3o&id=agente007&acesso=3
```

Parâmetros podem ser passados na requisição de duas formas.

Se o método for GET, os parâmetros são passados em uma única linha no query string, que estende a URL após um "?"

```
GET /servlet/Teste?id=agente007&acesso=3 HTTP/1.0
```

Se o método for POST, os parâmetros são passados como um stream no corpo na mensagem (o cabeçalho *Content-length*, presente em requisições POST informa o tamanho):

```
POST /servlet/Teste HTTP/1.0
Content-length: 21
Content-type: x-www-form-urlencoded

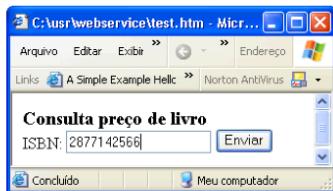
id=agente007&acesso=3
```

Caracteres reservados e maiores que ASCII-7bit são codificados seguindo o padrão usado em URLs: (ex: à = %E3). Formulários HTML codificam o texto ao enviar os dados automaticamente (formato *text/x-www-form-urlencoded*).

Seja o método POST ou GET, os valores dos parâmetros podem ser recuperados pelo método *getParameter()* de *ServletRequest*, que recebe seu nome:

```
String parametro = request.getParameter("nome");
```

Por exemplo, o seguinte formulário HTML recebe um código em um campo de entrada:

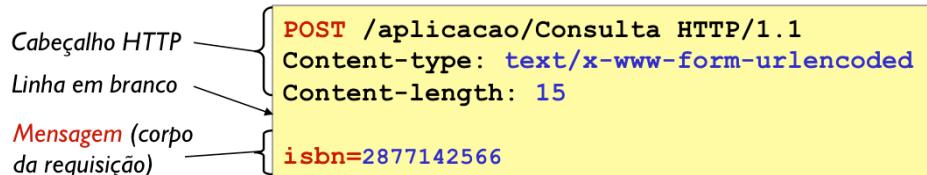


```

<form action="/aplicacao/Consulta"
      method="post">
  <h3>Consulta preço de livro</h3>
  <p>ISBN: <input type="text" name="isbn">
  <input type="submit" VALUE="Enviar">
</form>

```

Quando o usuário clicar em enviar, o browser criará a seguinte requisição POST para o servidor:



O servlet abaixo irá receber a requisição no seu método `doPost()`, extrair a informação da requisição e enviar para um serviço (DAO) para recuperar as informações necessárias para que possa gerar uma resposta:

```

@WebServlet("/Consulta")
public class ServletWeb extends HttpServlet {
    @Inject LivroDAO dao;

    public void doPost (HttpServletRequest request,
                       HttpServletResponse response) throws IOException {

        String isbn = request.getParameter("isbn");
        Livro livro = dao.getLivro(isbn);
        Writer out = response.getWriter();
        response.setContentType("text/html");
        out.println("<h1>" + livro.getTitulo() + "</h1>");
    }
}

```

Parâmetros de mesmo nome podem estar repetidos. Isto é comum e pode acontecer no caso de checkboxes ou menus de seleção múltipla. Na URL gerada eles não serão sobrepostos, mas concatenados. Neste caso uma chamada a `getParameter()` retornará apenas a primeira ocorrência. Para obter *todas* e preciso usar `String[] getParameterValues()`.

Por exemplo os parâmetros da URL:

`http://servidor/aplicacao?nome=Fulano&nome=Sicrano`

Podem ser recuperados usando:

```
String[] params = request.getParameterValues("nome");
```

2.4 Resposta

Uma resposta HTTP é enviada pelo servidor ao browser e contém informações sobre os dados anexados. O exemplo abaixo mostra uma resposta contendo uma página HTML simples:

```

HTTP/1.0 200 OK
Content-type: text/html
Date: Mon, 7 Apr 2003 04:33:59 GMT-03
Server: Apache Tomcat/4.0.4 (HTTP/1.1 Connector)
Connection: close
Set-Cookie: jsessionid=G3472TS9382903

<HTML>
  <h1>Hello World!</h1>
</HTML>

```

Os métodos de *HttpServletResponse* permitem construir um cabeçalho. Alguns dos principais métodos estão listados abaixo:

- void **addHeader**(String nome, String valor) - adiciona cabeçalho HTTP
- void **setContentType**(tipo MIME) - define o tipo MIME que será usado para gerar a saída (text/html, image/gif, etc.)
- void **sendRedirect**(String location) - envia informação de redirecionamento para o cliente (mesmo que enviar o cabeçalho Location: url)
- Writer **getWriter()** - obtém um Writer para gerar a saída. Ideal para saída de texto.
- OutputStream **getOutputStream()** - obtém um OutputStream. Ideal para gerar formatos diferentes de texto (imagens, etc.)
- void **addCookie**(Cookie c) - adiciona um novo cookie
- void **encodeURL**(String url) - envia como anexo da URL a informação de identificador de sessão (sessionid)
- void **reset()** - limpa toda a saída inclusive os cabeçalhos
- void **resetBuffer()** - limpa toda a saída, exceto cabeçalhos

2.4.1 Geração de uma resposta

Para gerar uma resposta, primeiro é necessário obter, do objeto *HttpServletResponse*, um fluxo de saída, que pode ser de caracteres (*Writer*) ou de bytes (*OutputStream*).

```
Writer out = response.getWriter(); // se for texto
OutputStream out = response.getOutputStream(); // se outro formato
```

Apenas um deve ser usado. Os objetos correspondem ao mesmo stream de dados.

Deve-se também definir o tipo de dados a ser gerado. Isto é importante para que o cabeçalho Content-type seja gerado corretamente e o browser saiba exibir as informações

```
response.setContentType("image/png");
```

Depois, pode-se gerar os dados, imprimindo-os no objeto de saída (out):

```
byte[] image = gerarImagenPNG();
out.write(buffer);
```

2.5 Acesso a componentes da URL

A não ser que seja configurado externamente, o *nome do contexto* aparece por default na URL absoluta após o nome/porta do servidor:

```
http://serv:8080/contexto/subdir/pagina.html
http://serv:8080/contexto/servletMapeado
```

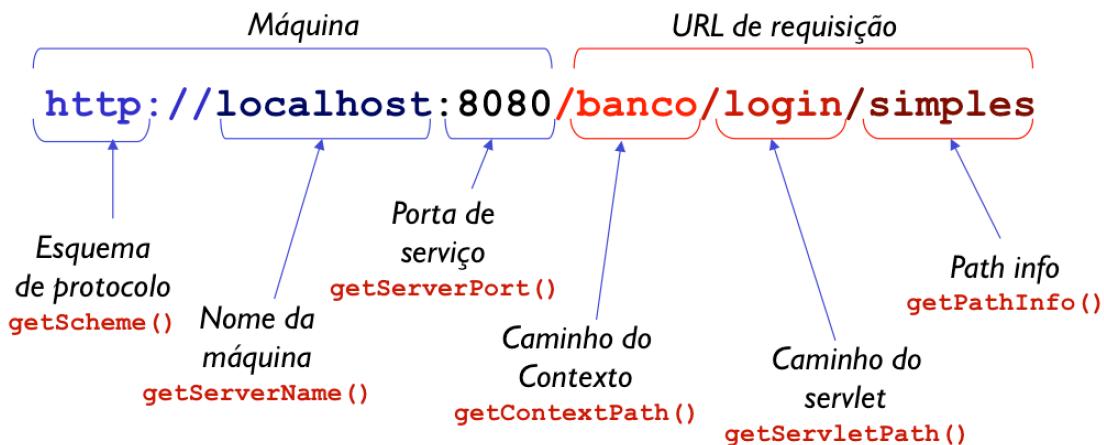
Se um contexto contiver páginas HTML estáticas, e essas páginas contiverem links para outros recursos e arquivos do contexto, elas devem usar, preferencialmente, URLs relativas. A raiz de referência para páginas estáticas não é o contexto, mas a raiz de documentos do servidor, ou *DOCUMENT_ROOT*. Por exemplo: *http://servidor:8080/*. Em links de páginas estáticas documentos podem ser achados relativos ao DOCUMENT_ROOT:

```
/contexto/subdir/pagina.html
/contexto/servletMapeado
```

Para a configuração do contexto (*web.xml*), a raiz de referência é a raiz de documentos do contexto (aplicação). Por exemplo: *http://servidor:8080/contexto/*. Componentes são identificados relativos ao contexto:

```
/subdir/pagina.html
/servletMapeado
```

Diferentes partes de uma URL usada na requisição podem ser extraídas usando métodos de `HttpServletRequest` listados na figura abaixo:



3 O Contexto

A interface `ServletContext` encapsula informações sobre o contexto ou aplicação. Cada servlet possui um método `getServletContext()` que devolve o contexto atual. A partir de uma referência ao contexto pode-se interagir com a aplicação inteira e compartilhar informações entre servlets.

Os principais métodos de `ServletContext` são:

- `String getInitParameter(String)`: lê parâmetros de inicialização do contexto (não confunda com o método similar de `ServletConfig`)
- `Enumeration getInitParameterNames()`: lê lista de parâmetros
- `InputStream getResourceAsStream()`: lê recurso localizado dentro do contexto como um `InputStream`
- `void setAttribute(String nome, Object)`: grava um atributo no contexto
- `Object getAttribute(String nome)`: lê um atributo do contexto
- `void log(String mensagem)`: escreve mensagem no log do contexto

3.1 Inicialização do contexto

E possível configurar parâmetros de configuração para o contexto usando o arquivo `web.xml`. Parâmetros consistem de nome e valor armazenados em `<context-param>`:

```
<context-param>
  <param-name>tempdir</param-name>
  <param-value>/tmp</param-value>
</context-param>
```

Para ler um parâmetro no servlet, é preciso obter acesso à instância de `ServletContext` usando o método `getServletContext()`:

```
ServletContext ctx = this.getServletContext();
String tempDir = ctx.getInitParameter("tempdir");

if (tempDir == null) {
    throw new UnavailableException("Configuração errada");
}
```

3.2 Resources

O método `getResourceAsStream()` permite que se localize e se carregue qualquer arquivo no contexto sem que seja necessário saber seu caminho completo. Resources podem ser arquivos de configuração (XML, properties), imagens, ou quaisquer outros arquivos necessários que estejam armazenados dentro do WAR.

O exemplo abaixo mostra como ler um arquivo XML que está dentro da pasta WEB-INF:

```
ServletContext ctx = getServletContext();
String arquivo = "/WEB-INF/usuarios.xml";
InputStream stream = ctx.getResourceAsStream(arquivo);
InputStreamReader reader =
    new InputStreamReader(stream);
BufferedReader in = new BufferedReader(reader);
String linha = "";
while ( (linha = in.readLine()) != null) {
    // Faz alguma coisa com linha de texto lida
}
```

3.3 Atributos no escopo de contexto

Objetos podem ser armazenados no contexto. Isto permite que sejam compartilhados entre servlets. O exemplo abaixo mostra como gravar um objeto no contexto:

```
String[] vetor = {"um", "dois", "tres"};
ServletContext ctx = getServletContext();
ctx.setAttribute("dados", vetor);
```

Para recuperar a chave deve ser usada em `getAttribute()`:

```
ServletContext ctx = getServletContext();
String[] dados = (String[])ctx.getAttribute("dados");
```

3.4 Listeners

Não existem métodos `init()` ou `destroy()` globais para realizar operações de inicialização e destruição de um contexto. A forma de controlar o ciclo de vida global para um contexto é através da implementação de um `ServletContextListener`, que é uma interface com dois métodos:

- `public void contextInitialized(ServletContextEvent e)`
- `public void contextDestroyed(ServletContextEvent e)`

Eles são chamados respectivamente depois que um contexto é criado e antes que ele seja destruído. Para isto é preciso ou anotar a classe da implementação com `@WebListener`:

```
@WebListener
public class OuvinteDeContexto implements ServletContextListener {...}
```

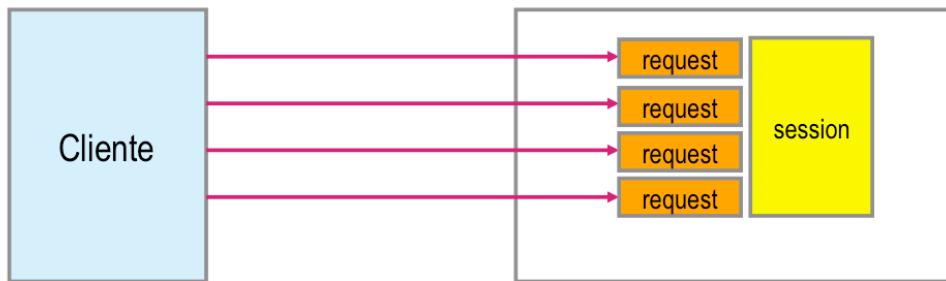
ou registrá-lo no web.xml usando o elemento `<listener>`

```
<listener>
    <listener-class>ex01.OuvinteDeContexto</listener-class>
</listener>
```

O objeto `ServletContextEvent`, recebido em ambos os métodos, possui um método `getServletContext()` que permite obter o contexto associado.

4 Sessões

Uma sessão HTTP representa o tempo que um cliente acessa uma página ou domínio. Uma sessão é iniciada com uma requisição, é única para cada cliente e persiste através de várias requisições.



Como o HTTP não mantém estado de sessão de forma nativa, as aplicações Web precisam cuidar de mantê-lo quando necessário. Soluções padrão incluem cookies, re-escrita de URLs, etc. A especificação WebServlet permite mais de uma solução, mas implementa por default (e recomenda) uma solução baseada em cookies.

Sessões são representadas por objetos *HttpSession* e são obtidas a partir de uma requisição. O objeto pode ser usado para armazenar objetos que serão recuperados posteriormente pelo mesmo cliente.

Para criar uma nova sessão ou para recuperar uma referência para uma sessão existente usa-se o mesmo método:

```
HttpSession session = request.getSession();
```

Para saber se uma sessão é nova, use o método *isNew()*:

```
BusinessObject myObject = null;
if (session.isNew()) {
    myObject = new BusinessObject();
    session.setAttribute("obj", myObject);
}
myObject = (BusinessObject)session.getAttribute("obj");
```

4.1 Atributos de sessão

Dois métodos da classe *HttpSession* permitem o compartilhamento de objetos na sessão.

- void **setAttribute("nome", objeto);**
- Object **getAttribute("nome");**

Por exemplo, o código abaixo irá gravar um atributo (um array) em uma requisição:

```
String[] vetor = {"um", "dois", "tres"};
HttpSession session = request.getSession();
session.setAttribute("dados", vetor);
```

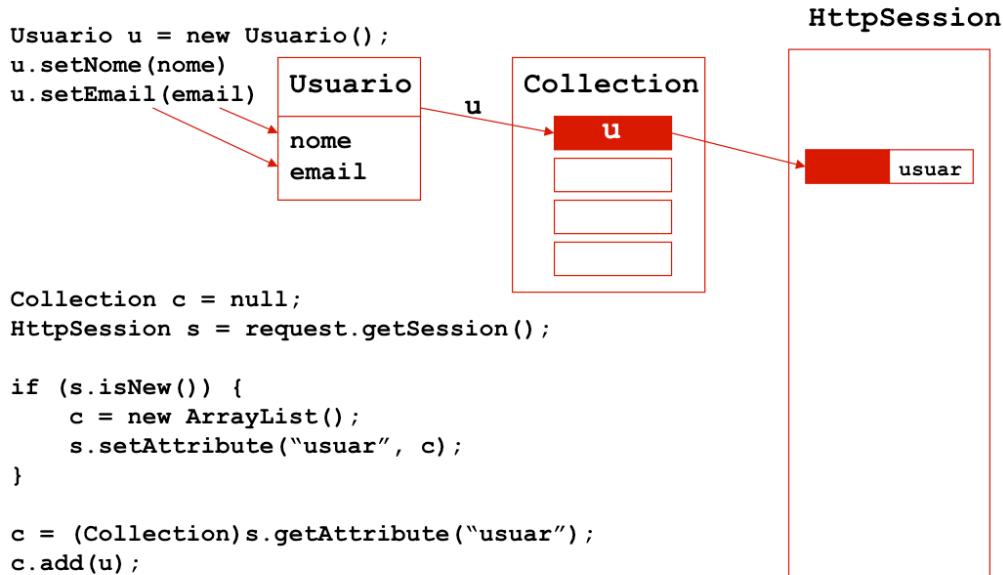
Em outro servlet que ainda executa na mesma sessão, pode-se obter o array usando:

```
HttpSession session = request.getSession();
String[] dados = (String[])session.getAttribute("dados");
```

Como a sessão pode persistir além do tempo de uma requisição, é possível que a persistência de alguns objetos não sejam desejáveis. Eles podem ser removidos usando:

```
removeAttribute("nome")
```

O exemplo abaixo ilustra o uso de sessões para guardar uma coleção de objetos. Um uso típico seria um carrinho de compras, onde cada item da coleção é um item selecionado pelo cliente. Para esvaziar o carrinho ele pode esvaziar a coleção ou invalidar a sessão (que remove não apenas a coleção da sessão atual, mas quaisquer outros objetos que tenham sido armazenados).



4.2 Validez e duração de uma sessão

Não há como saber que cliente não precisa mais da sessão. Normalmente estabelece-se um tempo de validade contado pela inatividade do cliente e assim definir um timeout em minutos para a duração de uma sessão desde a última requisição. Há quatro métodos em *HttpSession* para lidar com duração de uma sessão:

- void **setMaxInactiveInterval(int)** – define novo valor para timeout
- int **getMaxInactiveInterval()** – recupera valor de timeout
- long **getLastAccessedTime()** – recupera data em UTC
- long **getCreationTime()** – recupera data em UTC

Timeout default pode ser definido no web.xml para todas as sessões. Por exemplo, para que dure ate 15 minutos:

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

Para destruir imediatamente uma sessão usa-se:

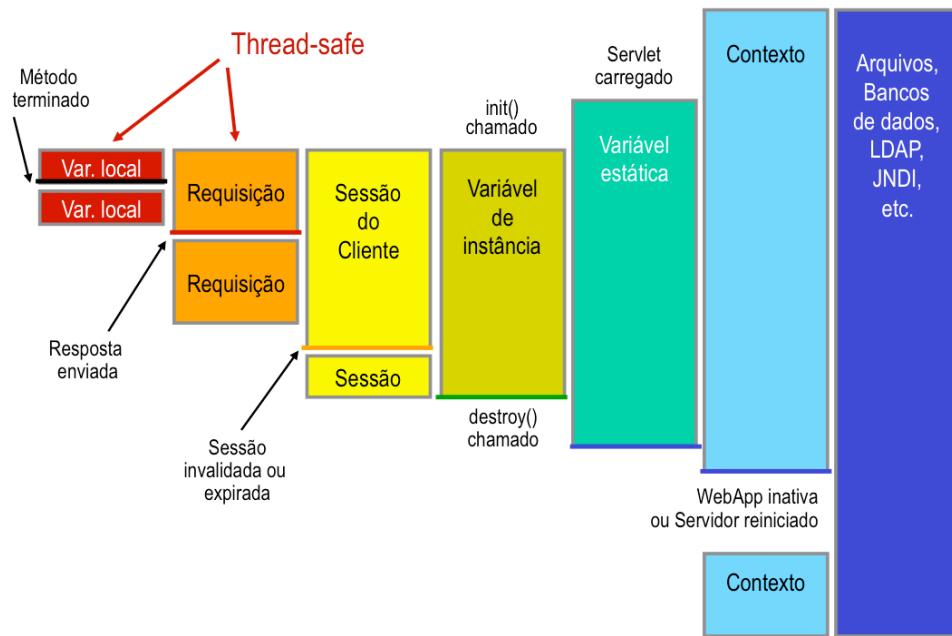
```
session.invalidate();
```

5 Escopos

Servlets podem compartilhar informações de várias maneiras

- Usando meios persistentes (bancos de dados, arquivos, etc)
- Usando objetos na memória por escopo (requisição, sessão, contexto)
- Usando variáveis estáticas ou de instância

Esses escopos estão ilustrados na figura abaixo:



Devido à natureza dos servlets e sua forma de execução, não é recomendado o compartilhamento usando variáveis estáticas e de instância. A forma recomendada consiste em usar os métodos `get/setAttribute` contidos nos três objetos de escopo:

- `javax.servlet.ServletContext` – que representa o contexto da aplicação e existe enquanto a aplicação estiver executando.
- `javax.servlet.http.HttpSession` – que representa o contexto da sessão do cliente e existe enquanto o cliente estiver conectado.
- `javax.servlet.ServletRequest` – que representa o contexto da requisição e existe enquanto o método `service()` não terminar.

Portanto, para gravar dados em um objeto de persistência na memória, deve-se usar:

```
objeto.setAttribute("nome", dados);
```

E para recuperar ou remover os dados:

```
Object dados = objeto.getAttribute("nome");
objeto.removeAttribute("nome");
```

6 Cookies

Sessões geralmente são implementados com cookies, mas são cookies gravados na memória que deixam de existir quando o browser é fechado. A especificação de cookies define outro tipo de cookie que é gravado em disco, no cliente, e persiste por tempo indeterminado. Eles são usados principalmente para definir “preferências” (nem sempre o usuário preferiu receber um cookie desse tipo) que irão durar além do tempo da sessão.

O cookie é sempre gerado no cliente. O servidor cria um cabeçalho HTTP que instrui o browser a criar um arquivo guardando as informações do cookie. Para criar um cookie persistente é preciso:

- Criar um novo objeto `Cookie`
- Definir a duração do cookie com o método `setMaxAge()`
- Definir outros métodos se necessário
- Adicionar o cookie à resposta

Por exemplo, para definir um cookie que contenha o nome do usuário recebido como parâmetro na requisição, com duração de 60 dias:

```
String nome = request.getParameter("nome");
Cookie c = new Cookie("usuario", nome);
c.setMaxAge(1000 * 24 * 3600 * 60); // 60 dias
```

O cookie é adicionado à resposta:

```
response.addCookie(c);
```

Para recuperar o cookie na requisição, usa-se

```
Cookie[] cookies = request.getCookies();
```

Um HttpCookie tem name e value. Para extrair cookie para um objeto local pode-se usar:

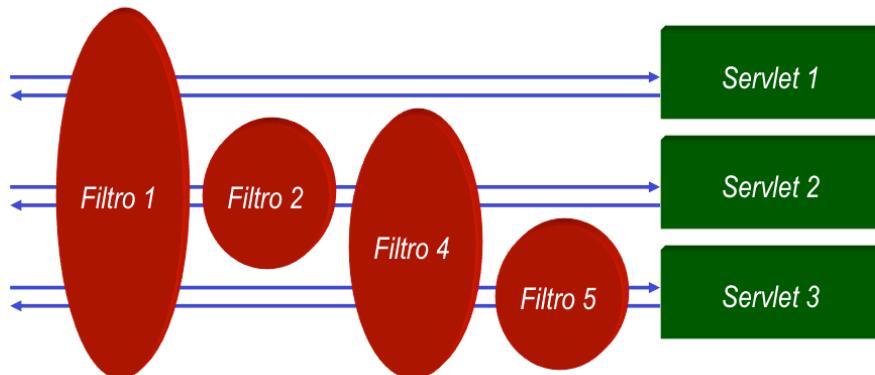
```
for (int i = 0; i < cookies.length; i++) {
    if (cookies[i].getName().equals("nome")) {
        usuario = cookies[i].getValue();
    }
}
```

7 Filtros

Filtros não são abordados neste tutorial. Esta seção contém apenas uma visão geral sobre o assunto.

Um filtro é um componente Web que reside no servidor e intercepta as requisições e respostas no seu caminho até um servlet, e de volta ao cliente. Sua existência é ignorada por ambos. É totalmente transparente tanto para o cliente quanto para o servlet.

Filtros podem ser concatenados em uma corrente. Neste cenário, as requisições são interceptadas em uma ordem e as respostas em ordem inversa. Filtros são independentes dos servlets e podem ser reutilizados.



Um filtro pode realizar diversas transformações, tanto na resposta como na requisição antes de passar esses objetos adiante (se o fizer). Aplicações típicas envolvem

- Tomada de decisões: podem decidir se repassam uma requisição adiante, se redirecionam ou se enviam uma resposta interrompendo o caminho normal da requisição
- Tratamento de requisições e respostas: podem empacotar uma requisição (ou resposta) em outra, alterando os dados e o conteúdo dos cabeçalhos

Aplicações típicas envolvem autenticação, tradução de textos, conversão de caracteres, MIME types, tokenizing, conversão de imagens, compressão e decompressão e criptografia.

Quando o container recebe uma requisição, ele verifica se há um filtro associado ao recurso solicitado. Se houver, a requisição é roteada ao filtro.

O filtro, então, pode gerar sua própria resposta para o cliente, repassar a requisição, modificada ou não, ao próximo filtro da corrente, se houver, ou ao recurso final, se ele for o último filtro, ou rotear a requisição para outro recurso. Na volta para o cliente, a resposta passa pelo mesmo conjunto de filtros em ordem inversa.

Em aplicações JSF, que são interceptadas por um único servlet, o filtro, que é mapeado a um servlet, afeta todas as requisições, portanto é necessário que ele use outras informações da URL (subcaminhos do contexto) para decidir se deve ou não processar a requisição ou resposta.

Filtros são criados implementando a interface `javax.servlet.Filter` e mapeados a servlets através de anotações `@WebFilter` ou via `web.xml`.

8 JSP (JavaServer Pages) e Taglibs

JSP e Taglibs não serão abordadas neste tutorial. Esta seção contém apenas uma visão geral sobre o assunto.

JSP é uma tecnologia padrão, baseada em templates para servlets. Em JSF 1.x JSP era usada para construir templates de páginas dinâmicas, mas desde JSF 2.x o seu uso não é mais recomendado e deve ser substituído por XHTML. Pode ainda ser usada para construir páginas simples que não precisam da arquitetura de componentes proporcionada pelo JSF. Mas uma arquitetura MVC é recomendada, evitando usar scriplets `<% ... %>` que podem ser embutidos em JSP.

Uma página JSP equivale a e efetivamente é um servlet que gera o HTML da página. A forma mais simples de criar documentos JSP, é

1. Mudar a extensão de um arquivo HTML para `.jsp`
2. Colocar o documento em um servidor que suporte JSP

Fazendo isto, a página será transformada em um servlet. A compilação é feita no primeiro acesso. Nos acessos seguintes, a requisição é redirecionada ao servlet que foi gerado a partir da página.

Transformado em um JSP, um arquivo HTML pode conter blocos de código (scriptlets): `<% ... %>` e expressões `<%= ... %>` que são os elementos mais frequentemente usados (mas que hoje são desaconselhados em favor de JSTL e Expression Language, usados em arquiteturas MVC):

```
<p>Texto repetido:<br/>
<% for (int i = 0; i < 10; i++) { %>
    <p>Esta é a linha <%=i %>
<% }%>
```

8.1 JSTL

JSTL é uma biblioteca de tags que permite que o autor de páginas controle a geração de código HTML usando laços e blocos condicionais, transformações e comunicação com objetos externos, sem a necessidade de usar scripts. JSTL (junto com EL) são a forma recomendada de usar JSP.

A adoção de JSTL estimula a separação da apresentação e lógica e o investimento em soluções MVC.

Para usar JSTL em uma página JSP é preciso declarar taglib em cada página (não é necessário declarar em `web.xml` ou incluir TLDs em WEB-INF, como ocorre com outras bibliotecas, porque a distribuição faz parte do container Java EE):

```
<%@ taglib prefix="c"
   uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head><title>Simple Example</title></head>
  <body>
    <c:set var="browser"
           value="${header['User-Agent']}"/>
    <c:out value="${browser}"/>
  </body>
</html>
```

8.2 EL (Expression Language)

Expression Language (EL) é uma linguagem declarativa criada para viabilizar a comunicação entre views (páginas JSP) e controllers (beans e outros objetos Java) em aplicações Web que usam arquitetura MVC. Usando EL é possível embutir expressões simples dentro de delimitadores \${...} diretamente na pagina ou em atributos de tags JSTL.

Através de EL e JSTL é possível eliminar completamente os scriptlets das páginas JSP. Com EL é possível ter acesso a objetos implícitos e beans. Por exemplo, em vez de usar, para ler o atributo de uma requisição, o scriptlet

```
<% request.getAttribute("nome") %>
```

pode-se usar simplesmente:

```
${nome}
```

E em vez de

```
<% bean.getPessoa().getNome() %>
```

usa-se

```
${bean.pessoa.nome}.
```

EL suporta também expressões simples com operadores aritméticos, relacionais e binários:

```
${!empty biblioteca.livros}
${livro.preco * pedido.quantidade}
```

Também converte tipos automaticamente e suporta valores default.

```
<tag item="${request.valorNumerico}" />
<tag value="${abc.def}" default="todos" />
```

Exemplos usando JSTL e EL:

```
<c:out value='${param.emailAddress}'/>
<c:if test='${not empty param.email}'>
  ...
</c:if>
```


3: CDI, Interceptadores e Bean validation

1 CDI	1
1.1 Características do CDI	2
1.1.1 Injeção de dependências (DI)	2
1.1.2 Contextos	2
1.1.3 Contextos e Injeção de Dependências	2
1.2 Design para injeção de dependências	3
1.1 Injeção de dependências com CDI	4
1.3 Como usar CDI	5
1.3.1 Instalação em ambiente standalone	5
1.3.2 Ativação	6
1.3.3 Uso do CDI	6
1.3.4 Exemplo de uma aplicação usando CDI	6
1.4 JavaBeans, Enterprise JavaBeans e Managed Beans	7
1.4.1 Quando usar um Session Bean em vez de um Managed Bean?	8
1.4.2 Injeção de Managed Beans	8
1.5 Escopos do CDI	8
1.6 Qualificadores	8
1.7 Métodos produtores	9
2 Interceptadores	10
2.1 Tipos de interceptadores	10
2.2 Como criar um interceptador @AroundInvoke	11
2.3 Interceptor Binding	11
2.4 Aplicações para interceptadores	12
3 Bean Validation	12
3.1 Como usar	12

1 CDI

A especificação CDI define serviços que permitem mapear objetos a contextos de um ciclo de vida, serem injetados em componentes e associados a interceptadores e decoradores, e

interagirem com outros componentes de forma fraca mente acoplada através da observação e disparo de eventos assíncronos.

Objetos injetáveis são chamados de *beans*, e incluem EJBs, *managed beans* e recursos Java EE. Instâncias de beans que pertencem a contextos (instâncias contextuais) podem ser injetados em outros objetos através do serviço de injeção de dependências.

Os serviços são configurados através de *metadados*. No bean, através de anotações. Na aplicação através de descritores XML.

CDI simplifica a construção de aplicações Java EE. A integração entre as camadas Web e serviços é facilitada. Componentes EJB podem ser usados automaticamente como *managed beans* do JSF. É possível também criar componentes personalizados para interagir com frameworks de terceiros.

1.1 Características do CDI

1.1.1 Injeção de dependências (DI)

Injeção de Dependências (DI) é um padrão de design de aplicações orientadas a objeto que consiste em inverter o controle usual que é do próprio objeto localizar ou construir suas dependências para tê-las inseridas no contexto por um agente externo. DI remove do componente a responsabilidade de localizar e configurar suas dependências.

DI tornou-se popular inicialmente através de um framework de Java Enterprise que hoje é uma alternativa concorrente ao Java EE: o Spring.

Em Java EE a injeção de dependências é realizada através de anotações. Uma anotação **@EJB** antes da declaração de um atributo do tipo de um Session Bean, faz o sistema procurar, localizar um bean compatível, instanciá-lo e disponibilizá-lo no contexto onde foi chamado. **@Resource** permite injetar recursos (ex: fábricas de objetos, conexões de bancos de dados, filas de mensageria, etc.) e **@PersistenceContext** é usado para injetar contextos de persistência JPA (ex: EntityManager). DI injeta os componentes através do seu tipo, mas às vezes é necessário identificar uma instância específica através de propriedades.

1.1.2 Contextos

Contextos definem um *escopo* no qual estão disponíveis serviços injetados. Podem durar o tempo da execução de um método, ou uma requisição, ou uma sessão do usuário, ou mesmo um período arbitrário estabelecido pela aplicação ou enquanto o serviço estiver no ar. Contextos são largamente utilizados em aplicações Web e JSF.

1.1.3 Contextos e Injeção de Dependências

O CDI praticamente torna obsoletos os mecanismos anteriores para injeção de serviços, declaração de escopos e beans gerenciados (**@ManagedBean**), oferecendo um framework integrado para configuração desses mecanismos. Proporciona a comunicação entre camadas de aplicações Java EE permitindo que serviços sejam injetados nos componentes que os requisitam através de uma API mínima baseada em anotações e tipo dos componentes. CDI reduz consideravelmente o acoplamento entre essas camadas.

A implementação de referência do CDI é JBoss Weld e sua especificação está publicada no grupo de trabalho do JSR 346. CDI é padrão obrigatório em servidores de aplicação que suportam Java EE 7.

1.2 Design para injeção de dependências

A injeção de dependências (DI) é um padrão de design que contribui para diminuir o acoplamento entre um componente e suas dependências, já que transfere a responsabilidade por criar e instanciar a dependência para a camada que é responsável por criar o componente. Portanto, em vez de um DAO instanciar um *DataSource*, ele pode ser injetado automaticamente pela classe que o criou, ou pelo container. Em vez do componente persistente localizar um DAO via JNDI, ele disponibiliza uma referência ou método *setter*, e deixa que outra classe ou o container forneça um.

DI não depende de nenhum framework. É um padrão de design clássico para aplicações orientadas a objeto (também chamado de inversão de controle, e indireção) e pode ser implementado usando Java puro. Os frameworks fornecem meios adicionais de desacoplamento (ex: anotações, XML, micro-containers). Os exemplos abaixo mostram como o padrão DI pode ser aplicado.

Considere as seguintes interface e classe. Elas representam um objeto que deverá ser armazenado em um mecanismo de persistência.

```
public interface Biblioteca {
    void emprestar(Livro livro);
    void devolver(Livro livro);
}

public class BibliotecaImpl implements Biblioteca {
    private BibliotecaStorage dao; // uma dependência!

    public void emprestar(Livro livro) {
        livro.status(Livro.EMPRESTADO);
        dao.atualizarStatus(livro);
    }

    public void devolver(Livro livro) {
        livro.status(Livro.DISPONIVEL);
        dao.atualizarStatus(livro);
    }
}
```

A interface abaixo, que é uma dependência de *BibliotecaImpl*, representa o objeto que cuidará do armazenamento:

```
public interface BibliotecaStorage {
    void atualizarStatus(Livro livro);
}
```

A implementação da dependência não precisa saber dos detalhes de *como* uma biblioteca armazena os dados sobre livros. Diferentes implementações poderiam existir.

```
public class BibliotecaNoSql implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando banco NoSQL");
    }
}
public class BibliotecaRemote implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando dados via rede");
    }
}
public class BibliotecaJdbc implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando banco JDBC");
    }
}
```

Pode até existir uma implementação para testes:

```
public class BibliotecaTeste implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando na Mock database");
    }
}
```

Sem usar injeção de dependências, a própria classe *Biblioteca* teria que escolher e instanciar o mecanismo de storage explicitamente (usando new ou JNDI, por exemplo):

```
public class BibliotecaImpl implements Biblioteca {
    private BibliotecaStorage dao = new BibliotecaNoSql();
    ...
}
```

Para usar injeção de dependências, é preciso que a classe ofereça uma interface que permita que se passe a referência externamente. Pode ser um *construtor* que receba o *dao* como parâmetro; pode ser um método *setter*, por exemplo:

```
public class BibliotecaImpl implements Biblioteca {
    private BibliotecaStorage dao; // dependencia

    public void setBibliotecaStorage(BibliotecaStorage ref) {
        this.dao = ref;
    }
    ...
}
```

Desta maneira, uma classe de controle (que cria a *Biblioteca*) poderia também instanciar uma ou mais dependências e injetar na *Biblioteca* a dependência desejada, em tempo de execução:

```
public static void main(String[] args) {
    BibliotecaImpl biblioteca = new BibliotecaImpl(); // criando o componente
    BibliotecaStorage dao = new BibliotecaNoSql(); // criando a dependência
    biblioteca.setBibliotecaStorage(dao); // injetando a dependencia no componente
}
```

Isto permite um *acoplamento menor*, já que a classe *Biblioteca* não precisa conhecer a implementação dos métodos de *storage*, e está acoplada apenas à interface. Também garante *extensibilidade*, já que novas formas de *storage* que forem inventadas no futuro poderiam ser usadas; e *flexibilidade*, permitindo que o método de *storage* seja trocado durante a execução.

Este design não é novidade. É um dos usos clássicos de interfaces, aparece em diversos padrões de design, e muitas vezes é o padrão de acoplamento em projetos. A diferença em relação a DI usando algum framework é que a ligação entre objetos é realizada através de metadados como anotações ou arquivos XML, em vez de Java.

1.1 Injeção de dependências com CDI

CDI oferece um *framework* que inclui o uso de injeção de dependências. O seu uso não é limitado apenas à servidores de aplicação e pode ser usado em aplicações *standalone*. Se o ambiente de execução estiver configurado para usar CDI, é preciso apenas criar um arquivo chamado *beans.xml* (não precisa ter conteúdo) e coloca-lo na pasta *META-INF* do *Classpath* da aplicação para sinalizar que CDI deve ser usado. Depois disso, o ambiente de execução será ativado e as anotações do CDI serão interpretadas.

Para injetar o serviço na classe *Biblioteca*, anotamos o método ou o atributo com *@Inject*.

```
@Inject public void setBibliotecaStorage(BibliotecaStorage ref) {
    this.dao = ref;
}
```

CDI identifica a instância pelo tipo, ou seja, pela classe. Se houver apenas uma implementação, ela será localizada e usada. Se houver mais de uma é preciso escolher uma que será a implementação default. Ela *não precisa de anotação* (é opcional – pode-se usar `@Default`):

```
@Default // anotação sempre opcional
public class BibliotecaNoSql implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando banco NoSQL");
    }
}
```

Outras implementações da interface, se existirem no classpath *precisam* ser anotadas com `@Alternative`:

```
@Alternative // anotação obrigatória havendo outras implementações desta interface
public class BibliotecaTeste implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando na Mock database");
    }
}
```

Se a biblioteca representar um *Model* em uma arquitetura MVC, ou se ela precisar ser acessada via expressões EL, ou se for armazenada no registro de um container, deve-se anota-la com `@Named`:

```
@Named("biblioteca")
public class BibliotecaImpl implements Biblioteca {}
```

O nome default é o próprio nome (não qualificado) da classe, de acordo com as regras de formação de JavaBeans. Sem a propriedade acima, o nome seria “*bibliotecaImpl*”.

1.3 Como usar CDI

CDI é nativo a qualquer servidor Java EE 7. Caso o CDI seja usado de forma standalone, fora de um servidor, é preciso instalar uma implementação. JBoss Weld é a implementação de referência mas há outras implementações: Apache OpenWebBeans, Caucho CanDI.

1.3.1 Instalação em ambiente standalone

Para instalar, baixe o Weld em

<http://seamframework.org/Weld/WeldDistributionDownloads>

copie o *weld-servlet.jar* para a pasta *WEB-INF/lib*, ou use o Maven e importe as dependências da API e do Weld. Importe a API para poder compilar o código (e para ambiente standalone ou servidor que não suporta CDI):

```
<dependencies>
    <dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <version>1.1</version>
    </dependency>
    ...
</dependencies>
```

O pacote acima contém apenas a interface. Se o ambiente não suporta CDI, será preciso também incluir a implementação. Esta é a implementação de referência Weld:

```
<dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se</artifactId>
    <version>2.2.6.Final</version>
</dependency>
```

Que requer a configuração de um *listener* no *web.xml*:

```
<listener>
    <listener-class>
        org.jboss.weld.environment.servlet.Listener
    </listener-class>
</listener>
```

Se o CDI estiver sendo usado em um servidor de aplicações Java EE 7, esta etapa não é necessária.

1.3.2 Ativação

O próximo passo é criar um arquivo *beans.xml* (vazio) e armazená-lo na pasta META-INF dentro do classpath da aplicação. Em Java EE 7 o *beans.xml* não precisa conter nada, mas se contiver XML, deve conter pelo menos o elemento raiz com o seguinte código:

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="annotated">
</beans>
```

Isto garante que o servidor irá processar anotações CDI. Agora pode-se anotar beans, serviços, etc. com as anotações CDI.

1.3.3 Uso do CDI

Uma vez configurado, o uso do CDI envolve principalmente anotações:

- Usar a anotação *@Inject* para anotar um atributo (ou método) de uma classe como *dependência injetável*
- Usar (opcionalmente) *@Default* para anotar a implementação default da dependência que será injetada
- Se houver outras implementações, usar (obrigatoriamente) *@Alternative* para anotá-las.
- Usar *@Named* para dar um nome à implementação do serviço no registro CDI. Este nome permitirá que o bean seja acessível via EL (ex: em JSF)

1.3.4 Exemplo de uma aplicação usando CDI

Exemplo de um serviço (*MailService*) representado por uma implementação *@Default* (*TestMailService*) em escopo CDI (*@ApplicationScoped*):

```
@ApplicationScoped
public class TestMailService implements MailService {
    public enviar(String de, String para, String texto) {
        ... // realiza o envio do email
    }
}
```

Exemplo de um bean em escopo CDI (*@SessionScoped*) e disponível em registro EL com o nome “usuario”:

```
@SessionScoped
@Named
public class Usuario {
    public String getEmail() {
        ...
    }
    ...
}
```

Exemplo de um bean em escopo CDI (`@RequestScoped`), disponível em registro EL com o nome “mail” que injeta os dois beans anteriores.

```
@RequestScoped
@Named
public class Mail {
    private @Inject MailService servico;
    private @Inject Usuario usuario;

    private String mensagem, destinatario;

    public String getEmail() { ... }
    public String getDestinatario() { ... }
    public void setEmail(String email) { ... }
    public void setDestinatario(String destinatario) { ... }

    public String enviarEmail() {
        servico.enviar(usuario.getEmail(), "fulano@mail.com", "Olá");
        return "enviada";
    }
}
```

Exemplo de um formulário JSF usando os managed beans declarados acima através de seus nomes EL (definidos com `@Named`), acessando seus métodos e propriedades:

```
<h:form>
    <h:outputLabel value="Usuário" for="usuario"/>
    <h:outputText id="usuario" value="#{usuario.email}" /><br/>

    <h:outputLabel value="Destinatario" for="destinatario"/>
    <h:inputText id="destinatario" value="#{mail.destinatario}" /><br/>

    <h:outputLabel value="Body" for="body"/>
    <h:inputText id="body" value="#{mail.mensagem}" /><br/>

    <h:commandButton value="Send" action="#{mail.enviarEmail}" />
</h:form>
```

1.4 JavaBeans, Enterprise JavaBeans e Managed Beans

Objetos injetáveis via CDI são JavaBeans, ou simplesmente Beans. Beans são objetos que seguem convenções mínimas de programação. Quase qualquer classe que tenha um construtor default (sem parâmetros) pode ser considerada um bean. CDI permite que beans sejam gerenciáveis (managed) pelo container.

A especificação CDI afirma que um JavaBean é qualquer classe Java que

- Não é classe interna (a menos que seja estática)
- Seu construtor não tem parâmetros ou é anotado com `@Inject`

Um Enterprise JavaBean (EJB) é um tipo de JavaBean que possui uma anotação que o qualifica como EJB (`@Remote`, `@Local`, `@Stateless`, etc.), ou é declarado como EJB em um arquivo de configuração ejb-jar.xml.

O CDI define um *Managed Bean* como qualquer JavaBean que não seja um Enterprise JavaBean e que:

- É uma classe concreta (ou anotada com `@Decorator`)
- Não implementa a interface Extension (javax.enterprise.inject.spi)

Na prática, tanto Session Beans quanto Managed Beans são beans gerenciáveis (ou seja, managed). Pode-se injetar um session bean em um managed bean e vice-versa. Message-driven

beans são objetos não contextuais e não podem ser injetados em outros objetos. Tanto EJBs como Managed Beans suportam callbacks de ciclo de vida `@PostConstruct` e `@PreDestroy`.

1.4.1 Quando usar um Session Bean em vez de um Managed Bean?

No Java EE 7 é melhor usar um Session Bean quando houver necessidade de:

- Transações e segurança com granularidade de método
- Controle de concorrência em singletons
- Passivação de nível de instância para Stateful beans e *pooling* para Stateless
- Chamadas remotas ou web services
- Timers e métodos assíncronos

Embora possam ser usados em escopos longos, Managed Beans são mais fáceis de gerenciar em escopos de duração menor, como RequestScoped, ou com limites bem definidos (*ViewScoped*, *ConversationScoped*). Em geral, beans usados em contextos *SessionScoped* ou *ApplicationScoped* devem (preferencialmente) ser EJBs.

1.4.2 Injeção de Managed Beans

Um Managed Bean possui um *conjunto de tipos*, pelos quais pode ser identificado. Seus tipos incluem sua classe, superclasse e interfaces visíveis pelo cliente.

Um Managed Bean também tem uma coleção de *qualificadores*. Qualificadores são declarados como anotações. Todo bean possui pelo menos o qualificador `@Default`. Se um bean possui várias implementações, elas podem receber qualificadores diferentes para que possam ser selecionadas durante a injeção.

Todo managed bean pertence a um *escopo*, que determina a sua duração. O escopo default é `@Dependent` (herda o escopo da classe que o injetou)

O contrato que precisa ser satisfeita pelo bean que é injetado requer que seja possível distinguir uma implementação única através da combinação do seu *tipo* mais *qualificadores*.

1.5 Escopos do CDI

Todo bean em CDI existe em um escopo. Escopos podem ser declarados antes da classe. A maioria são semelhantes aos do JSF e têm o mesmo nome:

- `@ApplicationScoped`
- `@SessionScoped`
- `@ConversationScoped`
- `@RequestScoped`

Se um bean não declara nenhum escopo, ele pertence ao escopo do objeto onde foi injetado. Isto corresponde ao pseudo-escopo `@Dependent`. É recomendado anotar um bean como `@Dependent` (mesmo sendo opcional) se esse comportamento for intencional. Os escopos listados acima são escopos normais (são meta-anotados, na sua definição, como `@NormalScope`). Outros frameworks podem usar `@NormalScope` para definir escopos adicionais. Alguns outros são definidos no framework JSF (`@ViewScoped`, `@FlowScoped`).

1.6 Qualificadores

CDI injeta tipos, e não instâncias. Se uma aplicação precisa manter de múltiplas implementações da mesma interface e elas têm o mesmo nome, não será possível selecionar ambas. Para isto

podem definir um `@Qualifier`, que é uma anotação específica para qualificar diferentes implementações.

Por exemplo, se houver dois diferentes `EntityManagers`, pode-se definir um `@Qualifier` para cada um deles. Cada um será uma anotação, como esta:

```
@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface CustomerDb {}
```

Suponha que exista também outro qualificador `@AdminDb`. Agora os dois `EntityManagers` podem ser identificados:

```
public @ApplicationScoped class MyService {
    private @Inject @CustomerDb EntityManager customerEm;
    private @Inject @AdminDb EntityManager adminEm;
    ...
```

Neste outro exemplo temos uma implementação diferente para o serviço de storage da biblioteca:

```
@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface ServicoRemoto {}
```

Uma vez definido, o qualificador pode ser usado para remover a ambiguidade do *tipo*:

```
@Inject @ServicoRemoto BibliotecaStorage storage;
```

O container vai procurar por beans do tipo `PaymentProcessor` que tenham qualificador `@ServicoRemoto`. Portanto é preciso anotar o bean com essa anotação:

```
@ServicoRemoto
public class BibliotecaRemote implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
        System.out.println("Atualizando dados via rede");
    }
}
```

Qualquer bean ou ponto de injeção que não especificar um qualificador possui o qualificador default `@Default`.

1.7 Métodos produtores

Muitas vezes o objeto para ser injetado não pode ser simplesmente criado com `new`. Às vezes é necessário determinar uma implementação específica em tempo de execução. Às vezes o objeto é obtido através de um outro meio e precisa ser injetado. Para essas situações pode-se usar métodos produtores. Um método produtor deve retornar uma instância do objeto que produz e ser anotado com `@Produces`. Usando produtores é possível injetar praticamente qualquer coisa. Este exemplo (da documentação oficial do Weld) mostra como criar um produtor de inteiros aleatórios, que pode ser injetado em qualquer bean:

```
import javax.enterprise.inject.Produces;
@ApplicationScoped
public class RandomNumberGenerator {
    private java.util.Random random =
        new java.util.Random(System.currentTimeMillis());
    @Produces @Named @Random int getRandomNumber() {
        return random.nextInt(100);
    }
}
```

Foi especificado um qualificador (para distinguir o inteiro aleatório de outros componentes que produzem Integer) e seu escopo é o default `@Dependent`, ou seja, herda o escopo do objeto onde é usado. Como foi usado `@Named`, o método também possui um *nome* para uso em EL: `randomNumber`. Agora pode-se injetar um número aleatório da forma:

```
@Inject @Random int numeroAleatorio;
```

Ou chamá-lo em uma expressão EL:

```
<p>Número da sorte: #{randomNumber}</p>
```

O produtor é um método concreto de uma classe de managed bean ou session bean.

Um método produtor pode conter parâmetros. Neste caso o container irá buscar um bean instanciado que combine com o tipo e qualificadores de cada parâmetro e injetar automaticamente. Se a instância não for encontrada, ou se houver mais de uma que combine com tipo e qualificador, uma exceção será lançada.

```
@Produces List<Livro> acervo(@Central Biblioteca biblioteca) {
    return biblioteca.getLivros();
}
```

Campos produtores podem ser usados anotando um atributo que declara e instancia uma variável diretamente:

```
@Produces Storage storage = new BDStorage();
```

O container escolhe o objeto a ser injetado com base no *tipo do ponto de injeção*.

2 Interceptadores

Interceptadores são operações que podem ser injetadas em outros métodos durante a chamada. Representam interesses ortogonais e são uma abstração da programação orientada a aspectos (AOP).

Um interceptador pode ser aplicado a vários métodos através de anotações, permitindo injetar operações adicionais em métodos que serão aplicados em pontos de junção: antes ou depois dos métodos interceptados.

2.1 Tipos de interceptadores

Interceptadores são callbacks especiais que são chamados quando acontecem eventos relacionados ao ciclo de vida e execução dos objetos. Existem cinco anotações. Duas delas são chamadas ou antes ou depois de um evento, e as outras três são chamadas em volta do evento.

- `@AroundConstruct` - Marca o método como um interceptador que recebe um callback depois que a classe-alvo é construída
- `@AroundInvoke` - Marca o método como um interceptador
- `@AroundTimeout` - Interceptador de timeout
- `@PostConstruct` - Interceptador para eventos PostConstruct
- `@PreDestroy` - Interceptador para eventos PreDestroy

Vimos aplicações de `@PostConstruct` e `@PostDestroy` em vários managed e enterprise beans. `@AroundInvoke` é talvez o mais usado. `@AroundTimeout` é usado em timers, e `@AroundConstruct` é usado raramente.

2.2 Como criar um interceptador @AroundInvoke

Para usar é preciso criar uma classe contendo um método anotado com `@AroundInvoke` recebendo parâmetro `InvocationContext`. A classe não precisa ter nenhuma anotação (`@Interceptor` é opcional):

```
@Interceptor
public class Intercep1 {
    @AroundInvoke
    public Object metodo(InvocationContext ctx) throws Exception {...}
}
```

Chame `invocationContext.proceed()` no ponto do código em que o método a ser interceptado deverá ser chamado:

```
@Interceptor
public class XmlLogger {
    @AroundInvoke
    public Object xmlLog(InvocationContext ctx) throws Exception {
        System.out.println("<log>");
        Object resultado = ctx.proceed();
        System.out.println("</log>");
        return resultado;
    }
}
```

Use a anotação `@Interceptors` nas classes cujos métodos deverão ser interceptados. A anotação pode receber uma, ou uma lista de interceptadores, para que todos os seus métodos sejam interceptados:

```
@Stateless
@Interceptors({XmlLogger.class, Intercept2.class, ...})
public class MyBean implements MyBeanIF { ... }
```

A declaração também pode ser feita em cada método.

2.3 Interceptor Binding

Também é possível configurar um interceptador usando `<interceptor-binding>` no ejb-jar.xml ou web.xml ou usado a anotação `@Priority` na classe do interceptador para associá-lo a um qualificador especial chamado de `@InterceptorBinding`. Desta forma é possível criar anotações que representam a injeção dos interceptadores.

Por exemplo, para em vez de usar `@Interceptors` nos métodos e classes poderíamos usar `@XmlLogger`, primeiro criando um qualificador:

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
public @interface XmlLogger { ... }
```

E usando essa anotação para anotar o interceptador da forma:

```
@XmlLogger
@Priority(Interceptor.Priority.APPLICATION)
public class XmlLogger {
    @AroundInvoke
    public Object xmlLog(InvocationContext ctx) throws Exception {...}
    ...
}
```

Assim, é possível anotar os métodos e classes com o interceptador usando apenas `@XmlLogger`:

```
@Stateless @XmlLogger
public class MyBean implements MyBeanIF { ... }
```

2.4 Aplicações para interceptadores

Aplicações típicas de interceptadores incluem loggers, segurança e transações. Várias anotações desse tipo existem em CDI e foram construídas com interceptadores. Por exemplo, uma anotação para inserir um contexto transacional pode ser feita com um interceptador, definindo um `@InterceptorBinding`:

```
@Retention(RetentionPolicy.RUNTIME)
@Target( { ElementType.TYPE, ElementType.METHOD })
@InterceptorBinding
public @interface ContextoTransacional {}
```

Depois o interceptador:

```
@ContextoTransacional
@Priority(Interceptor.Priority.APPLICATION)
public class ContextoTransacionalInterceptor {

    @AroundInvoke
    public Object xmlLog(InvocationContext ctx) throws Exception {
        Object resultado;
        try {
            ut.begin();
            resultado = ctx.proceed();
            ut.commit();
        } catch (Exception e) {
            ut.rollback();
        }
        return resultado;
    }
}
```

Agora o interceptador pode ser usado para tipos (classes) ou métodos:

```
@ApplicationScoped
public class FilmeService {
    private @Inject EntityManager em;

    @ContextoTransacional
    public gravarFilme(Filme f) {
        em.persist(f);
    }
}
```

3 Bean Validation

Bean Validation é uma coleção de anotações para configurar validação em beans. Essas validações são usadas automaticamente em frameworks como JSF. As anotações fazem parte do pacote `javax.validation.constraints` e representam restrições (*constraints*) que são aplicadas em classe, atributos, métodos, parâmetros e que provocam exceções se violadas.

3.1 Como usar

Para usar *Bean Validation* basta anotar uma propriedade, um método ou seus parâmetros com anotações. O sistema verificará se os valores são compatíveis com as restrições de validação e lançará uma exceção se não forem.

Por exemplo:

```
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
}
```

As anotações default da API de Bean Validation estão listadas abaixo:

<code>@AssertFalse @AssertTrue @NotNull @Null</code>	Propriedade deve ser true ou false, nula ou não
<code>@DecimalMax @DecimalMin @Max @Min</code>	Especifica um valor máximo ou mínimo
<code>@Digits(integer=n, fraction=m)</code>	Dígitos (ex: 12.345 = integer=2, fraction=3)
<code>@Future @Past</code>	Data tem que ser no futuro ou no passado
<code>@Pattern(regexp="pattern")</code>	Expressão regular deve combinar com o string
<code>@Size(min=n, max=m)</code>	Tamanho do string

Anotações podem ser combinadas:

```
public class Name {
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
}
```

Anotações também podem ser estendidas e customizadas.

4: Java Persistence API (JPA)

Table of Contents

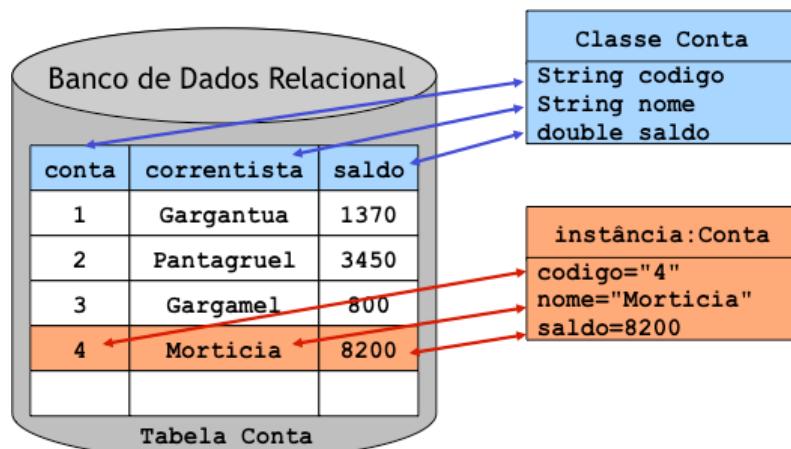
1	Introdução	2
1.1	<i>Unidade de persistência e persistence.xml</i>	3
1.2	<i>Entidades e mapeamento</i>	5
1.3	<i>Configuração do ambiente</i>	6
1.3.1	Configuração local (standalone)	7
1.3.2	Configuração em ambiente Java EE	9
2	Persistência	11
2.1	<i>Operações CRUD</i>	11
2.2	<i>Ciclo de vida e operações de persistência</i>	13
2.3	<i>Listeners</i>	14
3	Mapeamento	15
3.1	<i>Mapeamento de relacionamentos entre entidades</i>	15
3.1.1	Relacionamentos @ManyToOne	16
3.1.2	Relacionamentos @ManyToMany	17
3.1.3	Relacionamentos @OneToOne	18
3.2	<i>Coleções</i>	19
3.2.1	Lists, Sets, Collections	19
3.2.2	Mapas e @MapKey	19
3.3	<i>Mapeamento de objetos embutidos (@Embeddable)</i>	20
3.3.1	Coleções	21
3.3.2	Chaves compostas	21
3.4	<i>Persistência transitiva, cascading, lazy loading</i>	21
3.4.1	Configuração de CascadeType	22
3.4.2	Lazy loading	22
3.5	<i>Mapeamento de Herança</i>	23
3.5.1	MappedSuperclass	23
3.5.2	Tabela por classe concreta	24
3.5.3	Tabela por hierarquia	25
3.5.4	Tabela por subclasse	26
3.6	<i>Outros mapeamentos</i>	27
3.6.1	Mapeamento de enumerações	27
3.6.2	Mapeamento de uma instância a duas tabelas	28
4	Queries	28
4.1	<i>Cláusulas e joins</i>	29
4.2	<i>JPQL</i>	29
4.2.1	Sintaxe essencial do JPQL	30
4.2.2	Exemplos de queries simples JPQL	32
4.2.3	Exemplos de queries usando relacionamentos	32
4.2.4	Exemplos de queries usando funções, group by e having	33
4.2.5	Exemplos de subqueries	33
4.2.6	Queries que retornam múltiplos valores	34

4.2.7	Named Queries	35
4.3	<i>Criteria</i>	36
4.3.1	Sintaxe essencial de Criteria	36
4.3.2	Exemplos de queries simples usando Criteria	38
4.3.3	Exemplos de queries usando relacionamentos	40
4.3.4	Exemplos de queries usando funções, group by e having	41
4.3.5	Exemplos com subqueries	41
4.3.6	Queries que retornam múltiplos valores	42
4.3.7	Typesafe Criteria com static metamodel	42
5	Tuning em JPA	43
5.1	<i>Transações</i>	43
5.1.1	Resource-local javax.persistence.EntityTransaction	43
5.1.2	JTA javax.transaction.UserTransaction	44
5.2	<i>Cache</i>	45
5.2.1	Cache de primeiro nível (L1, EntityManager)	45
5.2.2	Cache de segundo nível (L2)	47
5.3	<i>Locks</i>	48
5.3.1	Locks otimistas	48
5.3.2	Locks pessimistas	48
5.4	<i>Operações em lote</i>	49

1 Introdução

JPA – Java Persistence API é uma especificação do Java EE que define um mapeamento entre uma estrutura relacional e um modelo de objetos em Java. JPA pode ser também usado de forma *standalone*, em aplicações Java SE, utilizando um provedor de persistência independente, mas é um componente obrigatório e nativo de qualquer servidor Java EE. O Java EE 7 adota a especificação JPA 2.1 (cuja especificação foi criada através do JSR 338).

JPA permite a criação de objetos persistentes, que retém seu estado além do tempo em que estão na memória. Oferece uma camada de persistência que permite pesquisar, inserir, remover e atualizar objetos, além de um mecanismo de mapeamento objeto-relacional (ORM) declarativo. Mapeamento objeto-relacional em aplicações JPA consiste na declaração de mapeamentos entre classes e tabelas, e atributos e colunas em tempo de desenvolvimento, e da sincronização de instâncias e registros durante o tempo de execução.



1.1 Unidade de persistência e persistence.xml

Para configurar JPA em uma aplicação é necessário que essa aplicação tenha acesso, através do seu *Classpath*, aos seguintes componentes (geralmente empacotados em JARs):

- Módulo contendo a *API* do JPA (as classes, interfaces, métodos, anotações do pacote *javax.persistence*)
- Um provedor de persistência JPA que implementa a especificação (ex: Hibernate, EclipseLink)
- Drivers necessários para configuração do acesso aos datasources usados (e pools de conexão, se houver)

Além disso, a aplicação deverá incluir um arquivo *persistence.xml* contendo a configuração da camada de persistência.

Dependendo do provedor de persistência usado, pode haver necessidade de incluir outros JARs e arquivos de configuração. Um arquivo *orm.xml* pode também ser incluído para configuração de mapeamento (como alternativa, ou para sobrepor os mapeamentos declarados nas próprias classes através de anotações).

O *contexto de persistência*, usado para sincronizar as instâncias persistentes com o banco de dados é chamado de *Entity Manager*. Ele controla o ciclo de vida das instâncias. O Entity Manager pode ser instanciado pela própria aplicação (comum em aplicações JPA standalone) ou obtido através de JNDI ou injeção de dependências (padrão em ambientes Java EE).

O conjunto de entity managers e as entidades que eles gerenciam configura uma *unidade de persistência* (*Persistence Unit*). Essa configuração é declarada no arquivo *persistence.xml* e referenciada nas classes que *usam* os objetos persistentes, como DAOs, fachadas, session beans, servlets, etc.

O arquivo *persistence.xml* deve estar no Classpath acessível pelas classes Java usadas na aplicação em */META-INF/persistence.xml*, e possui a seguinte configuração *mínima*:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="MyPU" transaction-type="JTA">
  </persistence-unit>

</persistence>
```

Este exemplo é utilizável em um servidor Java EE que tenha sido configurado para oferecer um datasource default (*java:comp/DefaultDataSource*). Normalmente aplicações usam datasources selecionadas de forma explícita. Este segundo exemplo é típico para uma aplicação JPA simples em um servidor Java EE 7:

```
<persistence version="2.1" ...>
  <persistence-unit name="com.empresa.biblioteca.PU"
    transaction-type="JTA">

    <jta-data-source>jdbc/Biblioteca</jta-data-source>
    <class>br.empresa.biblioteca.entity.Livro</class>
    <properties>
      <property name="eclipselink.deploy-on-startup" value="true" />
    </properties>

  </persistence-unit>
</persistence>
```

Ele declara o *tipo de transações utilizada* (JTA – gerenciada pelo container), o nome JNDI de um datasource específico (que precisa ser previamente configurado no servidor de aplicações), uma entidade mapeada, e uma propriedade de configuração do provedor usado (tipicamente, e principalmente quando usa-se um provedor que não é nativo do servidor, há várias outras propriedades específicas do provedor de persistência que precisam ser configuradas).

Neste segundo exemplo temos um *persistence.xml* usado para acessar os mesmos dados através de uma camada de persistência configurada *localmente* (transações gerenciadas pela aplicação), usando provedor *EclipseLink* e banco *PostgreSQL*:

```
<persistence version="2.1" ...>
    <persistence-unit name="com.empresabiblioteca.PU"
                      transaction-type="RESOURCE_LOCAL">

        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>br.com.agonavis.javaee7.jpa.intro.Livro</class>

        <properties>
            <property name="javax.persistence.jdbc.driver"
                     value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.url"
                     value="jdbc:postgresql://localhost:5432/postgres" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
            <property name="javax.persistence.jdbc.password" value="admin" />
            <property name="eclipselink.ddl-generation"
                     value="drop-and-create-tables" />
            <property name="eclipselink.deploy-on-startup" value="true" />
        </properties>
    </persistence-unit>

</persistence>
```

Cada provedor de persistência poderá requerer propriedades específicas. Há quatro propriedades *padrão* para informar os dados de conexão a um banco de dados (*driver*, *url*, *user*, *password*). Essa configuração é mais comum em ambientes de testes, que rodam em Java SE.

A configuração do *persistence.xml* também depende do provedor de persistência usado. Este outro *persistence.xml* configura uma unidade de persistência com transações locais usando um provedor *Hibernate* acessando localmente um banco de dados *MySQL*. Diferentemente do exemplo anterior, este passa as propriedades de configuração do banco via propriedades do *Hibernate*. O resultado é o mesmo.

```
<persistence ...>
    <persistence-unit name="LojaVirtual" transaction-type="RESOURCE_LOCAL">

        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>lojavirtual.Produto</class>

        <properties>
            <property name="hibernate.dialect"
                     value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.connection.driver_class"
                     value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password" value="" />
            <property name="hibernate.connection.url"
                     value="jdbc:mysql://localhost:3306/test"/>
        </properties>
    </persistence-unit>
</persistence>
```

1.2 Entidades e mapeamento

A arquitetura do JPA baseia-se no mapeamento de tabelas a classes, colunas a atributos. O objetivo do mapeamento é a construção de uma *hierarquia de entidades*.

Entidade (Entity) é o nome dado a um objeto persistente que em JPA é representado por um JavaBean (ou POJO) mapeado a uma tabela (um JavaBean/POJO é basicamente uma classe Java com atributos privativos acessíveis via métodos get/set e que contém um construtor default sem argumentos.) Uma entidade JPA pode ser mapeada a uma tabela de duas formas:

- Através de elementos XML no arquivo *orm.xml*
- Através de anotações antes de declarações de classes, métodos, atributos, construtores.

Instruções de mapeamento podem ser incluídas associando classes a tabelas, atributos a colunas, etc. A listagem abaixo ilustra um exemplo de mapeamento declarado em *orm.xml*. Existem defaults para praticamente tudo. Os únicos tags obrigatórios são *<entity>* e *<id>*, e os atributos *name*. Pode-se incluir um nível maior de detalhamento (ex: tipos de dados, constraints) ou menor (ex: omitir tags com nomes das tabela e colunas, se forem iguais).

```
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm/orm_2_1.xsd"
  version="2.1">
  <entity name="Livro" class="br.com.agonavis.Livro" access="FIELD">
    <table name="LIVRO"/>
    <attributes>
      <id name="id">
        <column name="ID"/>
        <generated-value strategy="TABLE" generator="LIVRO_SEQ"/>
      </id>
      <basic name="titulo">
        <column name="TITULO"/>
      </basic>
      <basic name="paginas">
        <column name="PAGINAS"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

Não é necessário usar o arquivo *orm.xml* nem mapeamentos em XML para criar entidades. Os mapeamentos podem ser declarados através de anotações nas próprias classes Java que representam os objetos persistentes. Esse é o procedimento mais comum e recomendado. Como o mapeamento em XML tem *precedência* sobre o mapeamento realizado via anotações, ele geralmente só é usado quando é necessário sobrepor ou refinar o mapeamento default em uma aplicação existente.

Para declarar uma entidade com anotações JPA é necessário no mínimo anotar a classe com *@Entity* e anotar pelo menos um atributo (ou método) com *@Id*, indicando sua chave primária.

```
@Entity
public class Livro implements Serializable {

  @Id
  private Long id;
  private String titulo;
  private int paginas;

  public Long getId() {
    return id;
  }
}
```

```

public void setId(Long id) {
    this.id = id;
}

public String getTitulo() {
    return this.titulo;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}
public int getPaginas() {
    return this.paginas;
}

public void setPaginas(int paginas) {
    this.paginas = paginas;
}
}

```

Por default, todos os atributos da classe, expostos via métodos get/set serão mapeados a colunas de mesmo nome. Ou seja, no exemplo acima seriam mapeadas as colunas: *ID*, *TITULO* e *PAGINAS*. O nome da classe será usado por default como nome da tabela. Se o nome da tabela e o nome da classe coincidir, e se coincidirem os nomes das colunas com os nomes dos atributos, e seus tipos forem compatíveis, é possível que o mapeamento seja suficiente, e não haja a necessidade de configuração adicional.

O valor e o tipo usado no *ID* é importante para a operação do mecanismo de persistência transitiva no JPA, portanto, é comum que seja um valor sequencial gerado automaticamente. A estratégia de geração da seqüencia, que muitas vezes depende do banco usado, também pode ser declarada em XML ou com anotações.

Se houver necessidade adicional de configuração, é possível informar nomes de colunas, tabelas, tipos, limites, relacionamentos, estratégias e diversos outros mecanismos para viabilizar o mapeamento usando anotações como *@Table*, *@Column*, *@CollectionTable*, *@JoinTable*, *@JoinColumn*, etc. No exemplo abaixo foram usadas anotações para mapear a classe e seus atributos a tabelas e colunas com nomes diferentes:

```

@Entity
@Table(name="BOOK")
public class Livro implements Serializable {

    @Id
    private Long id;

    @Column(name="TITLE")
    private String titulo;

    @Column(name="PAGES")
    private int paginas;

    ...
}

```

1.3 Configuração do ambiente

E critico para o uso do JPA que o ambiente de desenvolvimento e implantação esteja configurado corretamente. Nas seções a seguir descreveremos os detalhes de algumas configurações através de exemplos completos que incluem operações de persistência com um objeto simples.

1.3.1 Configuração local (standalone)

A configuração *standalone* pode ser usada em projetos Java EE para rodar testes unitários sobre as instâncias sem a necessidade de realizar *deploy* no servidor. Pode também ser usada em ferramentas e aplicações que rodam fora do servidor em comunicação direta com o banco de dados.

É preciso incluir explicitamente no *Classpath* os JARs da API, do provedor JPA usado e do driver do banco de dados (e pool de conexões se houver). O exemplo abaixo ilustra a configuração das dependências *Maven* para baixar os JARs para um acesso JPA usando *EclipseLink 2.5* e banco de dados *PostgreSQL*:

```
<dependencies>
    <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>javax.persistence</artifactId>
        <version>2.1.0</version>
    </dependency>

    <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>eclipselink</artifactId>
        <version>2.5.0</version>
    </dependency>

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>9.4.1208</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Pode-se usar outros mecanismos (IDEs, Ant, etc.) para incluir essas dependências. O importante é que a primeira (API) esteja disponível em tempo de desenvolvimento e execução, e todas estejam no *Classpath* em tempo de execução.

No modo *standalone*, o arquivo *persistence.xml* precisa declarar a configuração JDBC para que a camada de persistência possa se comunicar com o banco. Como as transações serão controladas pela aplicação e serão locais, a unidade de persistência declara o tipo de transações como *RESOURCE_LOCAL*:

```
<persistence version="2.1">
    <persistence-unit name="tutorial-jpa" transaction-type="RESOURCE_LOCAL">

        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>br.com.argonavis.javaee7.jpa.intro.Livro</class>
        <properties>
            ...
            <property name="eclipselink.ddl-generation" value="drop-and-create-tables" />
            <property name="eclipselink.deploy-on-startup" value="true" />
        </properties>

    </persistence-unit>
</persistence>
```

Dependendo do provedor usado, poderá ser necessário informar propriedades adicionais, dialetos, opções, etc. Neste exemplo incluímos uma propriedade do EclipseLink que irá causar a geração automática de esquemas a partir das instâncias (útil em tempo de desenvolvimento e ambiente de testes). Essa geração será genérica e baseada em vários defaults (para ter um controle maior é necessário incluir dados adicionais no mapeamento via anotações ou XML).

As classes que irão instanciar o *EntityManager* podem obtê-lo via consulta JNDI global a um servidor, se houver um container cliente configurado para tal, ouy instancia-lo diretamente. No exemplo abaixo, a configuração foi feita localmente, instanciando diretamente uma fábrica local para obter o *EntityManager* da unidade de persistência especificada.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class LivroTest {
    public static void main(String[] args) {

        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("tutorial-jpa");
        EntityManager em = factory.createEntityManager();

        ...
    }
}
```

Observe que o parâmetro passado ao método de fabrica do *EntityManagerFactory* corresponde ao nome da unidade de persistência exatamente como foi declarada no *persistence.xml*.

A entidade a ser sincronizada com a tabela é instanciada e configurada normalmente em Java puro.

```
public class LivroTest {
    public static void main(String[] args) {

        ...

        Livro livro = new Livro();
        livro.setTitulo("Meu Livro");
        livro.setPaginas(100);

        ...
    }
}
```

Como é um objeto novo será usado o método *persist()* para sincronizar o objeto com o banco. Dependendo de como foi configurada a persistência transitiva, a criação do objeto novo poderá gerar um ID numérico que será a chave primária do registro. Nesse tipo de configuração também poderia ser usado o método *merge()*, que distingue objetos novos (inserções) de antigos (atualizações) através do ID. Tanto *merge()* como *persist()* precisam ser chamados dentro de um contexto transacional, e como as transações são locais, é preciso obtê-las do *EntityManager*:

```
public class LivroTest {
    public static void main(String[] args) {

        ...
        try {
            em.getTransaction().begin();

            em.persist(livro); // irá criar um registro novo no commit()

            em.getTransaction().commit();
        } catch (Exception e) {
            em.getTransaction().rollback();
        } finally {
            em.close();
        }
    }
}
```

O método `getTransaction()` retorna uma `EntityTransaction`, que possui os métodos `begin()`, `commit()` e `rollback()`.

1.3.2 Configuração em ambiente Java EE

Em Java EE o suporte a JPA é *nativo*, e a configuração do banco de dados deve ser preferencialmente realizada através de um datasource *previamente configurado no servidor*, e que possa ser acessado dentro de um contexto JTA. Portanto, as dependências em um projeto Maven não devem ser incluídas no componente (JAR) que será implantado (devem ser marcadas como *provided*):

```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
    ...
</dependencies>
```

O arquivo `persistence.xml` declara estratégia transacional JTA (gerenciada pelo container) e em vez de informar dados para acesso ao driver do banco, informa-se o caminho JNDI para o datasource que corresponde ao banco usado.

```
<persistence version="2.1" ...>
    <persistence-unit name="tutorial-jpa" transaction-type="JTA">
        <jta-data-source>jdbc/TutorialJPA</jta-data-source>
        <class>br.com.agonavis.javaee7.jpa.intro.ejb.Livro</class>
    </persistence-unit>
</persistence>
```

Dependendo do provedor usado, poderá ser necessário informar propriedades adicionais, dialetos, opções, etc. Neste exemplo *não* estamos usando a geração automática de esquemas, assumindo que as tabelas já existem e estão mapeadas corretamente.

Em Java EE o `EntityManager` poderá ser obtido via JNDI ou via injeção de dependências através de `@PersistenceContext` (ou ainda via CDI, se previamente configurado). Neste exemplo de acesso em servidor Java EE através de um `WebServlet`, identificamos a unidade de persistência usada e injetamos também um `UserTransaction`, necessário para delimitar o contexto transacional para cada operação de alteração (não é permitido usar o `EntityTransaction` – ou qualquer outro tipo de transação gerenciada pela aplicação – pois declararamos o uso de transações gerenciadas pelo container).

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

    @PersistenceContext(unitName="tutorial-jpa")
    EntityManager em;

    @Resource
    private UserTransaction ut; // Transação JTA

    protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {
        Writer out = response.getWriter();
        try {
            ut.begin();

            List<Livro> livros =
                em.createQuery("select livro from Livro livro")
                    .getResultList();
```

```

        for(Livro livro : livros) {
            out.write("<p>"+livro.getTitulo()+"</p>");
        }

        ut.commit();

    } catch (...) {...} // rollback e finalização

}

```

Aqui construímos um query usando a linguagem JPQL para selecionar *todas* as entidades do tipo *Livro*. Assumindo que o exemplo anterior (*standalone*) foi executado, devemos ter um objeto persistente no banco. A query irá retornar a lista (contendo um elemento) que será exibida na página retornada pelo servlet, ao acessar <http://servidor:porta/aplicacao/LivroTestServlet> (onde *servidor*, *porta* e *aplicacao* são respectivamente o nome do servidor, número da porta, e o nome da webapp instalada)

Neste outro exemplo fizemos o acesso através de um EJB, um *Session Bean*, cujos métodos são *transacionais por default*. Neste caso não é necessário (nem permitido) escrever nenhum código de delimitação para transações, já que ela será iniciada antes do método começar, e será cometida após o término do método, provocando um *rollback* se houver exceção (as políticas transacionais podem posteriormente ser configuradas via anotações do EJB.) Se houver apenas uma unidade de persistência, o nome dela não precisa ser informado ao injetar o `@PersistenceContext`.

```

@Stateless
public class LivroDAOsessionBean {

    @PersistenceContext
    EntityManager em;

    public Livro findByID(Long id) {
        Query query = em.createNamedQuery("selectById");
        query.setParameter("id", id);
        return (Livro)query.getSingleResult();
    }

    public List<Livro> findAll() {
        return (List<Livro>)em.createNamedQuery("selectAll").getResultList();
    }

    public void delete(Livro livro) {
        em.remove(livro);
    }

    public void update(Livro livro) {
        em.merge(livro);
    }

    public Livro insert(Livro livro) {
        return em.merge(livro);
    }

}

```

No exemplo acima, o JPQL foi declarado na classe que define a entidade usando `@NamedQuery`. Essa forma permite que os queries sejam referenciados pelo nome:

```

@Entity
@NamedQueries({
    @NamedQuery(name="selectAll", query="SELECT livro FROM Livro livro"),
    @NamedQuery(name="selectById", query="SELECT livro FROM Livro livro WHERE livro.id=:id")
})
public class Livro { ... }

```

Por fim, neste último exemplo a seguir, usamos um POJO comum com CDI. Diferentemente do EJB, o uso de CDI não garante métodos transacionais por default. Como o POJO roda dentro do servidor Java EE, podemos usar o *UserTransaction* (como no exemplo com *WebServlet*) para fornecer contexto transacional para as operações de persistência, ou usar a anotação CDI *@Transactional* antes de cada método. No servidor Java EE isto garante que o método será protegido por transação JTA e terá comportamento igual ao EJB.

```

@Named
public class LivroDAOManagedBean {

    @Inject
    EntityManager em;

    public Livro findByID(Long id) {
        TypedQuery<Livro> query =
            em.createNamedQuery("selectById", Livro.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }

    public List<Livro> findAll() {
        return em.createNamedQuery("selectAll", Livro.class)
            .getResultList();
    }

    @Transactional
    public void delete(Livro livro) {
        em.remove(livro);
    }

    @Transactional
    public void update(Livro livro) {
        em.merge(livro);
    }

    @Transactional
    public Livro insert(Livro livro) {
        return em.merge(livro);
    }
}

```

No exemplo acima usamos um *TypedQuery* em vez de *Query* para construir a consulta. *TypedQuery* é recomendado pois evita o uso de *cast* ao declarar o tipo dos objetos retornados.

Experimente rodar todos esses exemplos. Se necessário adapte-os para as configurações do seu ambiente. É importante rodar exemplos simples para garantir que o JPA esteja bem configurado. A configuração é uma das principais fontes de problemas em sistemas JPA e é bem mais fácil solucioná-los antes de construir aplicações mais complexas.

Consulte também os JavaDocs e a API para conhecer outras classes e anotações não abordadas aqui. O objetivo deste material não é ser completo e abrangente mas proporcionar um roteiro prático de introdução ao JPA.

2 Persistência

2.1 Operações CRUD

Operações CRUD (Create, Retrieve, Update, Delete – Criar, Recuperar, Atualizar, Remover) são as principais tarefas realizadas pela camada de persistência. Podem envolver uma única entidade

ou uma rede de entidades interligadas via relacionamentos. JPA oferece várias maneiras de realizar operações CRUD:

- Através dos métodos de persistência transitiva: *persist*, *merge*, *delete*, *find*
- Através do *Java Persistence Query Language (JPQL)*
- Através da API *Criteria*
- Através de SQL nativo

Neste curso trataremos apenas das três primeiras formas.

A recuperação de uma entidade, o “R” de “Retrieve” do acrônimo CRUD, pode ser feita conhecendo-se sua chave primária através de *find*:

```
Livro livro = em.find(Livro.class, 1234);
```

Normalmente a recuperação envolve uma pesquisa no estado da entidade, então pode-se usar JPQL através de um *Query* (ou *TypedQuery*):

```
TypedQuery query =
    em.createQuery("select m from Livro m where m.id=:id", Livro.class);
query.setParameter("id", 1234);
Livro livro = query.getSingleResult();
```

Criteria é um query que é construído dinamicamente, através da construção de relacionamentos entre objetos. É ideal para pesquisas que são construídas em tempo de execução:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Livro> criteria = builder.createQuery(Livro.class);
Root<Livro> queryRoot = criteria.from(Livro.class);
criteria.where(builder.equal(queryRoot.get(Livro.id), 1234));
TypedQuery<Livro> q = em.createQuery(criteria);
Livro livro = q.getSingleResult();
```

Para usar a API desta forma, uma classe *Livro.class* foi gerada (veja: Criteria Metamodel API) automaticamente pelas ferramentas da IDE (ou plug-in Maven).

Operações que envolvem atualização, “Create”, “Update” e “Delete”, geralmente são executadas através dos métodos de persistência transitiva *persist*, *merge* e *delete*. Por exemplo, para alterar o título do livro dentro de um contexto transacional pode-se usar:

```
Livro livro = em.find(Livro.class, 1234);
livro.setTitulo("Novo título");
em.merge(livro);
```

A mesma alteração pode ser realizada via JPQL:

```
Query query = em.createQuery("UPDATE Livro m"
    + "SET m.titulo = :titulo WHERE m.id = :id");
query.setParameter("titulo", "Novo título");
query.setParameter("id", "1234");
query.executeUpdate();
```

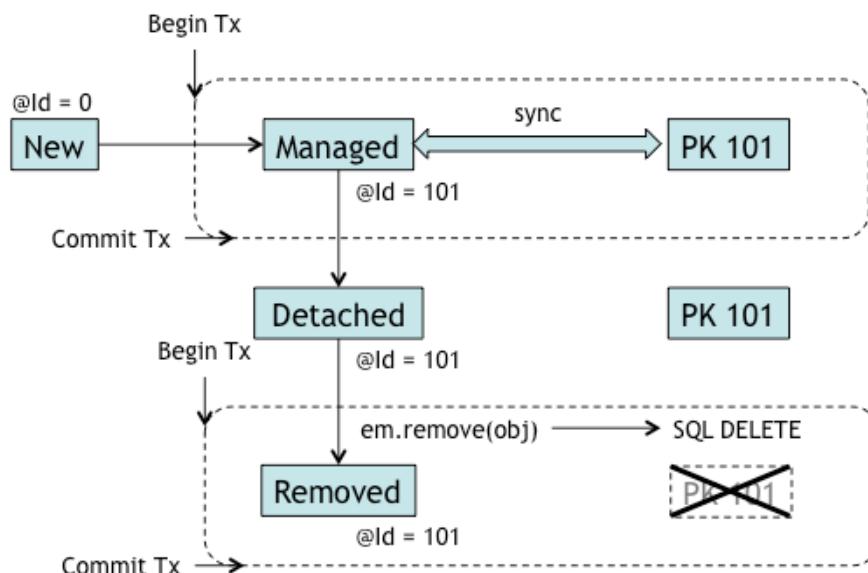
Ou *Criteria*:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaUpdate updateCriteria = builder.createCriteriaUpdate(Livro.class);
Root<Livro> updateRoot = updateCriteria.from(Livro.class);
updateCriteria.where(builder.equal(updateRoot.get(Livro.id), "1234"));
updateCriteria.set(updateRoot.get(Livro.titulo), "Novo título");
Query q = em.createQuery(updateCriteria);
q.executeUpdate();
```

2.2 Ciclo de vida e operações de persistência

Uma entidade pode estar em quatro diferentes estados durante o seu ciclo de vida: transiente (*new*), persistente (*managed*), desligado (*detached*) ou removido (*removed*). O estado irá determinar o comportamento do objeto quando forem chamadas operações de persistência transitiva sobre ele. Toda entidade tem uma chave primária indicada pela anotação `@Id` cujo valor geralmente é determinado pelo sistema de persistência. Quando uma entidade é nova e ainda não foi inserida no banco, o ID é nulo. Uma vez atribuído um ID a uma entidade, esse valor não muda mais.

O diagrama abaixo ilustra o ciclo de vida em entidades JPA:



Uma entidade é *transiente* quando não está associada a uma tabela no banco, e portanto não é persistente. Objetos mapeados são transientes logo depois que são criados e antes de serem persistidos através de uma operação *persist/merge* ou um *cascade* (*merge* disparado por um relacionamento). O identificador de um objeto transiente contém o valor null ou zero. Esse valor é usado pelo mecanismo de persistência para identificar objetos transientes.

A operação *new* cria uma instância no estado *transiente*:

```
Entidade novo = new Entidade();
```

Isto é Java puro. Ainda não usamos JPA. Para tornar a instância *persistente* é preciso utilizar a API do EntityManager dentro de um contexto transacional. Esta API possui vários métodos que controlam a persistência:

- `void persist(objeto)` – Gera um SQL INSERT e transforma o objeto passado como parâmetro em uma entidade. Causa *EntityExistsException* se o ID já existir no banco. Se o ID não existir no banco ou se for `null` a entidade será criada.
- `Object merge(objeto)` – Gera um SQL UPDATE ou um SQL INSERT e devolve a entidade construída. O INSERT será gerado se o ID do objeto for `null`. Caso contrário, o ID será usado como chave primária para realizar um UPDATE. Causa *IllegalArgumentException* se o ID não for `null` e não existir no banco.
- `void remove(objeto)` – Remove o registro do banco, desligando a entidade permanentemente. Ela não mais poderá ser ligada ao banco (um SQL UPDATE não é mais possível)
- `void detach(objeto)` – Separa a instancia do seu registro tornando a entidade desligada (*detached*). O mesmo ocorre com as entidades que são usadas fora do contexto

transacional do EntityManager. O método `clear()` causa um `detach` em todas as entidades. Um `merge()` ou `refresh()` dentro do contexto transacional religa a entidade tornando-a persistente.

- `void refresh(objeto)` – Sincroniza a entidade com dados obtidos do banco, sobrepondo quaisquer alterações que tenham sido feitas no estado do objeto.
- `void flush()` – Sincroniza as entidades do contexto de persistência com o banco (oposto de `refresh`).

Pode-se criar uma nova entidade passando um objeto novo para o método `persist()`, que cria um registro novo através de um SQL `INSERT` e o sincroniza com a instância transiente, tornando-a persistente:

```
entityManager.persist(novo);
```

Também é possível criar uma entidade através do método `merge()`, se o objeto tiver um ID `null`. Se o ID não for `null`, `merge()` considera que o objeto não é novo (é transiente) e irá usar a chave primária para gerar um SQL `UPDATE`. Diferentemente do `persist()`, o método `merge()` não altera a instância passada como argumento, mas devolve uma cópia persistente dela.

```
Entidade persistente = entityManager.merge(novo);
```

Uma entidade é considerada *desligada (detached)* quando já possui uma associação com uma tabela no banco (tem um `@Id` diferente de zero ou `null`), mas não está, no momento, sincronizada com ela por não se encontrar dentro de um contexto transacional, ou porque a transação ainda não foi cometida, ou porque ela foi desligada explicitamente através do método `detach()` ou `clear()`. A entidade poderá ser religada se entrar em outro contexto transacional com outro `merge` dentro de um contexto transacional:

```
Entidade persistente = entityManager.merge(detached);
```

Uma entidade desligada é útil, pois contém estado que pode ser exibido em uma página Web e alterado. Se não houver nenhuma alteração no registro correspondente durante o tempo que o objeto estiver desligado, o objeto poderá posteriormente sincronizado com o banco sem problemas, mas neste período ele também corre risco de se tornar obsoleto e conter dados inconsistentes com o banco, caso outro processo o tenha alterado. Para lidar com essas o JPA oferece travas (ótimistas e pessimistas) que podem ser configuradas no mapeamento.

Um objeto pode ser *removido* do banco usando o método `remove()`:

```
Entidade removida = entityManager.remove(persistente);
```

A remoção elimina o registro do banco, no entanto o objeto ainda pode ser usado. Ele não é considerado transiente porque possui um `@Id`, mas não é mais considerado desligado ou persistente porque não há mais uma tabela associada a ele. Não é possível mais sincronizá-lo com o banco. Um `merge()` causaria uma exceção. Mas é possível enviá-lo para que seus dados sejam lidos por uma interface do usuário, por exemplo, e seus dados poderão retornar ao banco se forem copiados para uma nova instância, o que fará com que seus dados sejam reinseridos no banco em um novo registro, tornando-se assim novamente persistente (mas com outro `@Id`). Isto pode ser feito com um `persist()`.

2.3 Listeners

Para lidar com as mudanças de estado de uma entidade, existe em JPA um conjunto de anotações para definir *callbacks* que respondem a eventos de mudanças:

- `@PostLoad` - executado depois que uma entidade foi carregada no contexto de persistência atual (ou `refresh`)
- `@PrePersist` - executado antes da operação `persist` ser executada (ou `cascade` - sincronizado com `persist` se entidade for sincronizada via `merge`)

- `@PostPersist` - executado depois que entidade foi persistida (inclusive via *cascade*)
- `@PreUpdate` - antes de operação UPDATE
- `@PostUpdate` - depois de operação UPDATE
- `@PreRemove` - antes de uma operação de remoção

Os métodos que implementam os listeners podem ter quaisquer nomes. Não devem ser *static* nem *final*. Podem ser declarados na própria instância (nesse caso não devem receber parâmetros) ou em uma classe separada (devem receber um objeto como parâmetro, que corresponde à instância monitorada.) Para declarar os listeners em uma classe separada, pode-se usar a anotação `@EntityListener` na entidade informando o nome da classe que os contém:

```
@Entity
@EntityListeners(LivroListener.class)
public class Livro implements Serializable {
    ...
}
```

Neste caso, os métodos devem receber a entidade como parâmetro:

```
public class LivroListener {
    @PostLoad
    public void livroLoaded(Livro livro) { ... }
    ...
}
```

Além desses métodos, entidades também podem usar os *callbacks* do CDI, `@PostConstruct` e `@PreDestroy`, que são habilitados por default em ambientes Java EE. Listeners também podem ser declarados em *orm.xml*. Essa configuração é ideal se houver mais de um listener para as entidades.

Rode os exemplos que ilustram o funcionamento de listeners usando apenas um objeto. Na seção a seguir apresentaremos outros exemplos com relacionamentos e *cascade*.

3 Mapeamento

3.1 Mapeamento de relacionamentos entre entidades

Associações no JPA funcionam da mesma maneira que associações de objetos em Java. São naturalmente unidirecionais e *não são* gerenciadas pelo container, o que significa que é preciso escrever e chamar explicitamente o código Java necessário para atualizar os campos correspondentes de cada uma das instâncias que participam do relacionamento, quando dados são adicionados, removidos ou alterados (o container, porém, pode gerenciar a propagação da persistência de forma transitiva através de configuração de operações de cascata).

Os relacionamentos são sempre associações entre *entidades* (diretamente, ou indiretamente através de coleções). Pode-se também configurar associações entre objetos que não são relacionamentos. Neste caso, para que o estado desses objetos seja persistente, é preciso que estejam dependentes de uma entidade.

Para mapear relacionamentos, JPA utiliza informações da estrutura do código Java e das anotações. As anotações podem ser mínimas. Serão usados parâmetros *default* até onde for possível. Assim como `@Column` pode ser usado para informar um nome de coluna diferente do nome da propriedade escalar, `@JoinColumn` pode ser usado quando a coluna que contém a chave estrangeira da associação tiver um nome ou configuração diferente da propriedade mapeada.

A *cardinalidade* define o número de entidades que existe em cada lado do relacionamento. Podem ser mapeados dois valores: “*muitos*” e “*um*”, resultando em *quatro* possibilidades:

- Um para muitos
- Muitos para um
- Um para um
- Muitos para muitos.

As duas primeiras são equivalentes, portanto há três diferentes associações:

- `@ManyToOne / @OneToMany`
- `@OneToOne`
- `@ManyToMany`

Dentro de cada contexto, é importante também levar em conta a *direção* dos relacionamentos, que pode ser *unidirecional* ou *bidirecional*.

`@ManyToOne` é a associação mais comum. O modelo relacional é naturalmente um-para-muitos e uma referência simples de objeto é um-para-muitos. Em associações unidirecionais, apenas um lado possui referência para o outro (que não tem conhecimento da ligação). A anotação é usada apenas em um dos lados. Em associações bidirecionais, do outro lado deve possuir uma anotação `@OneToMany` e é preciso informar o nome da referência que determina a associação.

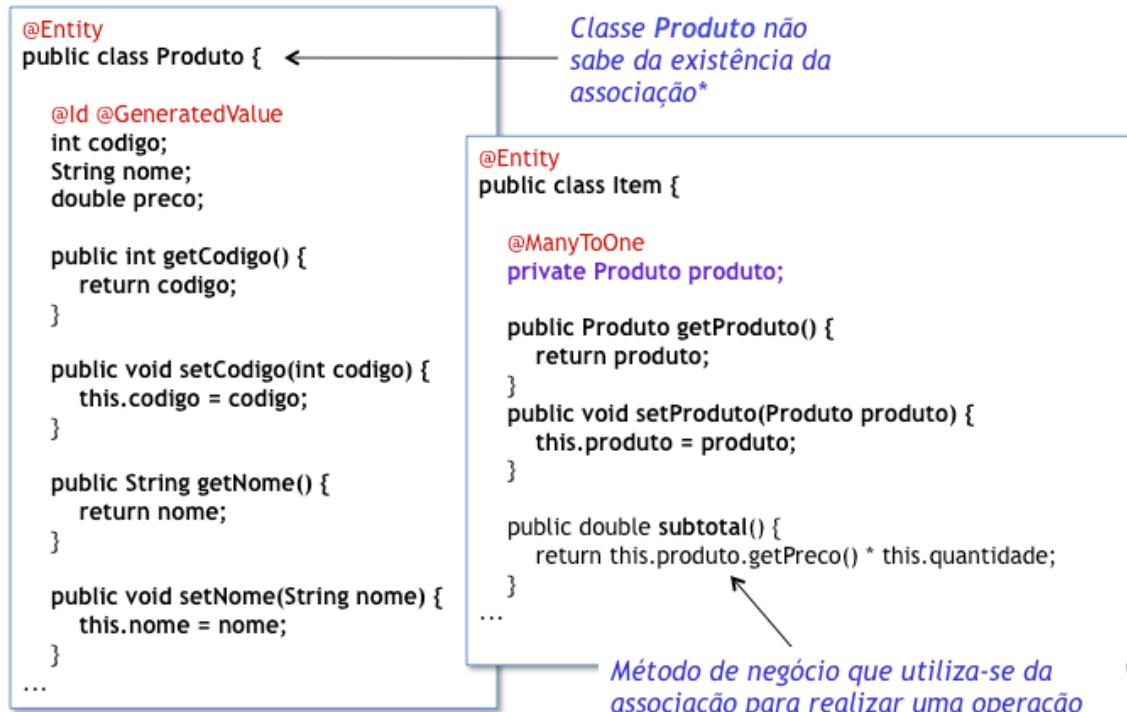
`@OneToOne` requer constraints (que são gerados) nas tabelas para garantir a consistência da associação. É também possível estabelecer uma relação um-para-um entre entidades usando `@ManyToOne`, desde que seja unidirecional o lado “many” seja limitado a um único elemento.

`@ManyToMany` mapeia três tabelas a dois objetos. Também é possível estabelecer uma relação muitos-para-muitos entre entidades usando `@ManyToOne`, desde que haja três objetos, dois contendo associações `@ManyToOne` para um terceiro. Isto pode ser usado se for necessário representar a associação como uma entidade, mapeada a uma tabela via `@JoinTable`.

Várias anotações e atributos permitem configurar ajustes finos dos mapeamentos, como regras de cascade, estratégias de carga, caches, transações, direção, etc.

3.1.1 Relacionamentos `@ManyToOne`

O exemplo a seguir ilustra um relacionamento *unidirecional* usando `ManyToOne`.



* Esses dados são insuficientes para a geração automática do esquema (as tabelas devem existir antes)

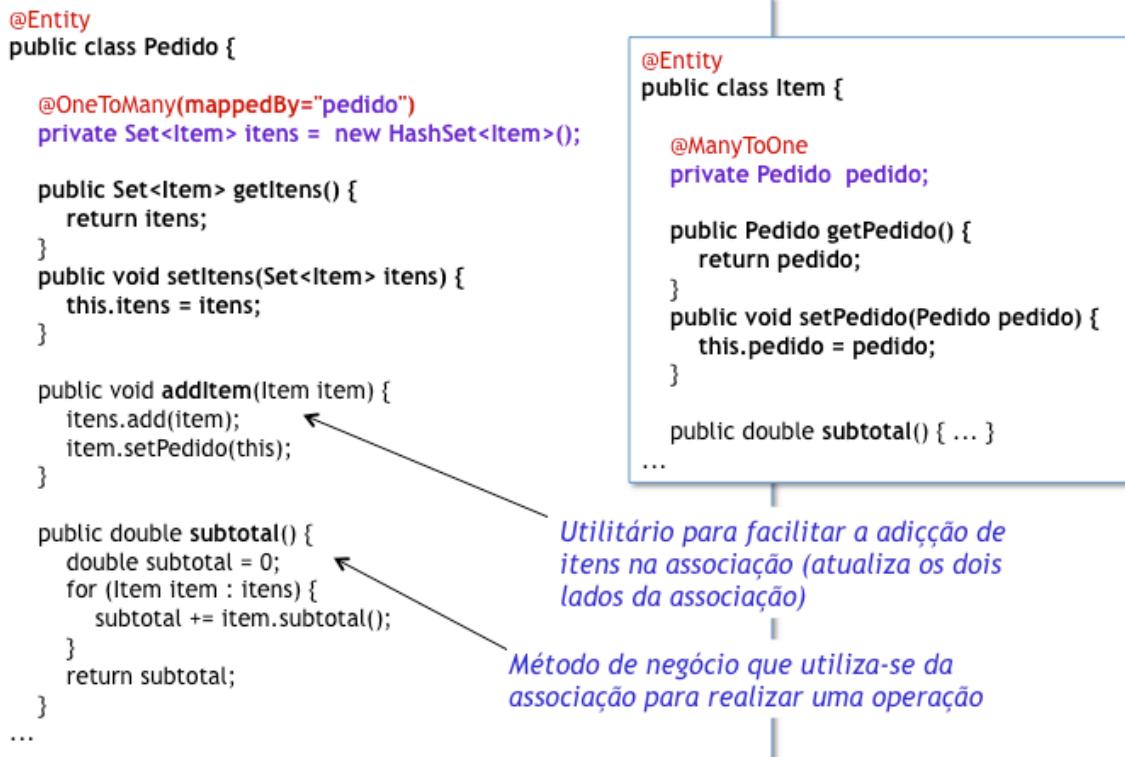
Se as anotações `@OneToOne`, `@ManyToMany` e `@ManyToOne` forem usadas em associações *bidirecionais* é necessário especificar anotações de ambos os lados e utilizar o atributo `mappedBy` (em um dos lados apenas) para informar o nome do atributo da outra classe que faz a associação.

O diagrama abaixo ilustra algumas associações bidirecionais.

<code>@OneToOne CPF cpf;</code>	→	<code>@OneToOne(mappedBy="cpf") Cliente cliente;</code>
<code>@ManyToMany Set<Turma> turmas;</code>	→	<code>@ManyToMany(mappedBy="turmas") Set<Aluno> alunos;</code>
<code>@ManyToOne Item item;</code>	→	<code>@ManyToOne(mappedBy="item") Set<Produto> produtos;</code>

Caso sejam usadas ferramentas de geração automática de esquemas e tabelas, poderá ser necessário incluir anotações bidirecionais, pois as anotações unidirecionais poderão ser insuficientes para gerar os esquemas corretamente.

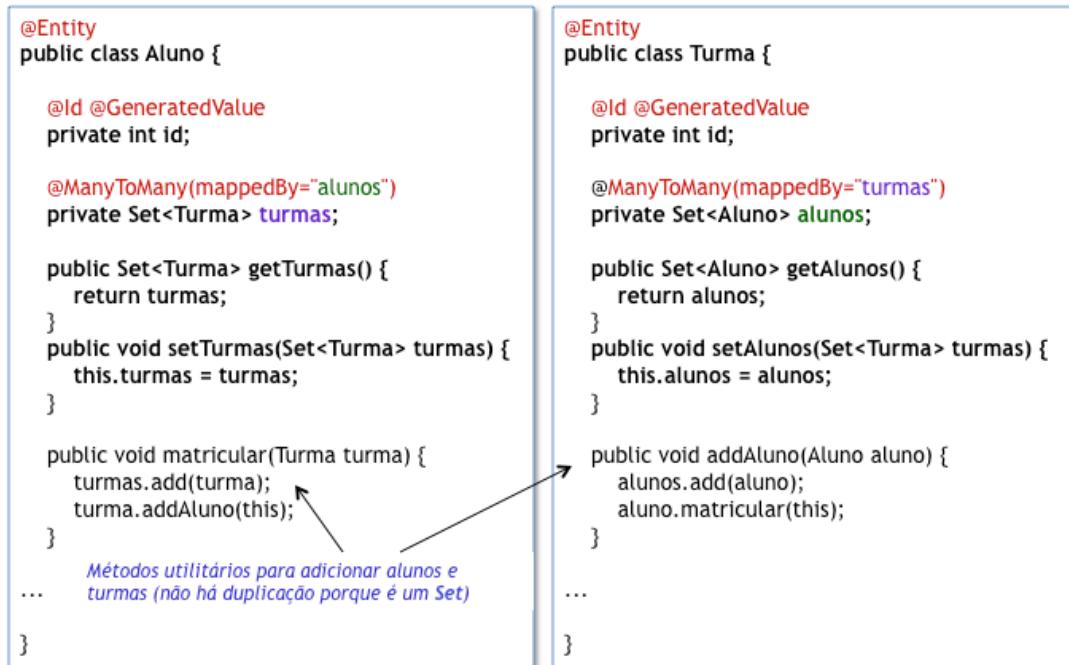
Exemplo de um relacionamento *ManyToOne bidirecional*:



3.1.2 Relacionamentos `@ManyToMany`

Use `@ManyToMany` nas coleções de cada lado da associação, informando o campo de mapeamento (`mappedBy`) em um dos lados. Usando-se um `Set` e `mappedBy`, garante-se a não duplicação de dados quando objeto é adicionado duas vezes na coleção.

Abaixo um exemplo simples ilustrando o mapeamento *Many to Many*.



3.1.3 Relacionamentos @OneToOne

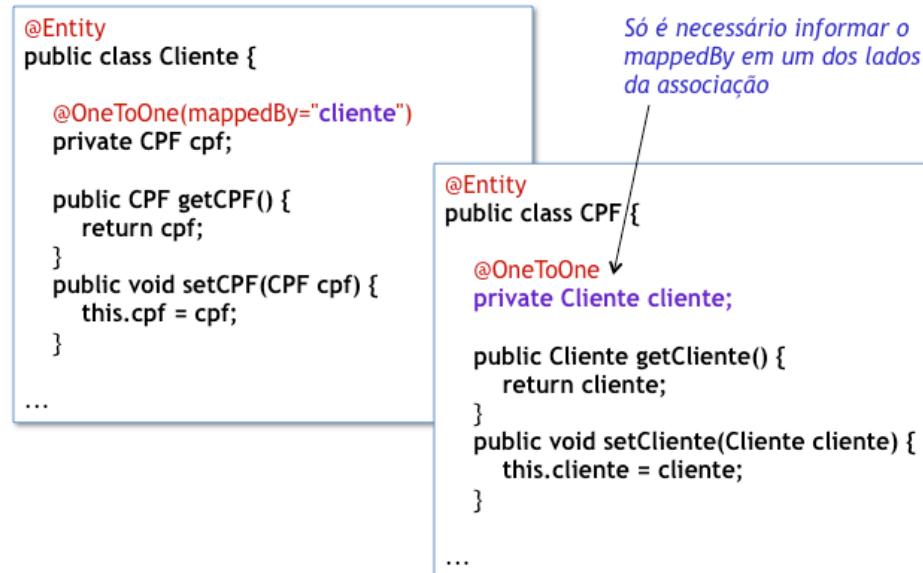
Um relacionamento *um para um* considera que os objetos em um relacionamento têm alto grau de dependência. No modelo relacional isto pode ser implementado de três maneiras:

1. Dois objetos, uma única tabela (dados em colunas da mesma tabela)
2. Um objeto, duas tabelas (dados em colunas de outra tabela)
3. Dois objetos, duas tabelas unidas por associação

Usando JPA, a terceira estratégia pode ser implementada usando relacionamentos *@ManyToOne* (unidirecional) ou *@OneToOne* (com mapeamento via chave estrangeira ou chave primária).

As duas primeiras estratégias não são relacionamentos entre entidades, mas associações com instâncias ou coleções de objetos. Em JPA, a primeira estratégia é implementada usando componentes *@Embeddable* e a segunda através de *@SecondaryTable*.

Para mapear associações um-para-um, inclua *@OneToOne* em um dos lados, se unidirecional. Se for bidirecional, o campo correspondente do outro lado é informado através de *mappedBy*:



3.2 Coleções

3.2.1 Lists, Sets, Collections

Relacionamentos `@ManyToOne` e `@ManyToMany` envolvem coleções. O mapeamento natural para registros em uma tabela é (`java.util.Set`), porque *Sets* não possuem uma ordem específica e não podem ser duplicados, mas isto não é uma regra. Existem situações onde pode haver pequenos ganhos de performance usando *Sets* em certos tipos de relacionamento e *Lists* em outros, mas em geral a escolha é motivada pela finalidade dessas estruturas. Basicamente:

- Use `java.util.Set` na maior parte dos casos, para listas de itens não ordenados e que não contém duplicatas (na verdade *Sets* *podem* ser ordenados no query usando `@OrderBy` e na instância usando uma implementação ordenada como `TreeSet`)
- Use `java.util.List` para listas de itens que podem ser duplicadas e que tem uma ordem indexada (onde o índice é importante); Uma alternativa é usar `@MapKey` com uma lista ordenada para as chaves.
- Use `java.util.Collection` quando a coleção representar uma coleção qualquer, na qual não faz diferença se há duplicações, ou se é ordenada.

Um outro motivo para escolher *List* em vez de *Set* ou vice-versa (quando não faz diferença para a aplicação a escolha de uma ou outra) é devido a compatibilidade com outros componentes (ex: componentes JSF, Primefaces, etc) que esperam *List*, ou *Set*, ou *Collection*. Isto evita a necessidade de criar um adaptador para fazer a conversão.

3.2.2 Mapas e @MapKey

E possível mapear uma coleção a um mapa usando `@MapKey` e representando a coleção como um `java.util.Map`. Um *Map* contém um *Set* de chaves e uma *Collection* de valores, e é indicado situações nas quais é mais eficiente recuperar um objeto através de uma chave.

A chave default é a chave-primária dos objetos da coleção:

```
@Entity
public class LojaVirtual {
    ...
    @OneToMany(mappedBy="lojaVirtual")
    @MapKey // indexado pela chave primaria
    public Map<Long, Cliente> clientes = new HashMap <>();
    ...
}
```

Usando o atributo *name* é possível indexar por outras propriedades dos objetos da coleção:

```
@Entity
public class LojaVirtual {
    ...
    @OneToMany(mappedBy="lojaVirtual")
    @MapKey(name="cpf") // indexado pelo CPF
    public Map<String, Cliente> clientes = new HashMap <>();
    ...
}
```

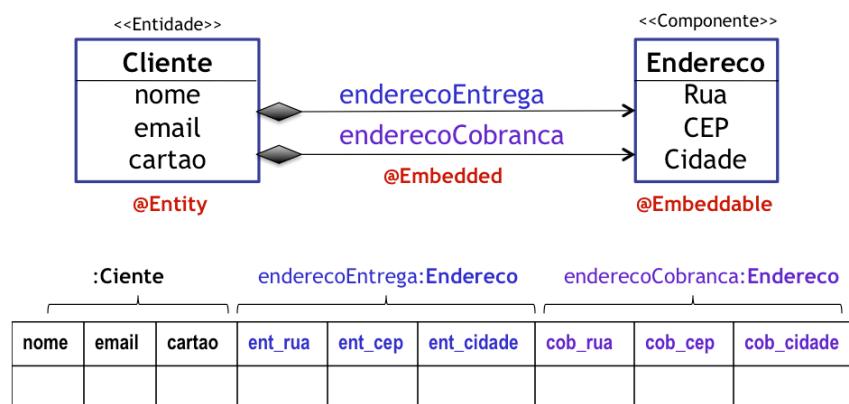
O mapeamento da chave também pode ser feito através das colunas, em vez dos atributos, usando `@MapKeyColumn` e `@MapKeyJoinColumn`. Existem ainda mais duas anotações para `MapKey`: `@MapKeyEnumerated`, se a chave for uma enumeração, e `@MapKeyTemporal`, se a chave for uma data ou `Timestamp`.

3.3 Mapeamento de objetos embutidos (@Embeddable)

Entidades são objetos persistentes que têm identidade no banco. Objetos que contém dados persistentes mas não têm identidade unívoca no banco são identificados pelo seu estado, ou valor (*value objects*). Em JPA são chamados de *embutíveis* e anotados como `@Embeddable`. Como atributo de uma entidade são anotados com `@Embedded` e são dependentes dela.

No *modelo de domínio* um objeto embutido representa um objeto, mas no *modelo relacional* representa apenas algumas colunas de uma tabela. O objeto embutido é *propriedade* da entidade. Muitas associações 1-1 podem ser implementadas de maneira mais eficiente como objetos embutidos.

Na ilustração abaixo *Cliente* é um objeto e é uma entidade, e *Endereço* é um objeto mas não é uma entidade. Um *Cliente* está associado a dois objetos *Endereco*, mas como não são entidades separadas, ocupam a mesma tabela:



A remoção do registro no banco remove a entidade e seus objetos dependentes. O exemplo abaixo ilustra como utilizar `@Embeddable` e `@Embedded` em JPA.

```

@javax.persistence.Entity
@javax.persistence.Table(name="CLIENTE")
public class Cliente {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long codigo;

    private String nome;
    ...

    @Embedded
    private Endereco enderecocobranca; <-->

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="rua", column=@Column(name="ENT_RUA")),
        @AttributeOverride(name="cep", column=@Column(name="ENT_CEP")),
        @AttributeOverride(name="cidade", column=@Column(name="ENT_CIDADE")),
    })
    private Endereco enderecotravel; <-->

    ...
}
    
```

```

@javax.persistence.Embeddable
public class Endereco {
    @Column(name="COB_RUA")
    private String rua;
    @Column(name="COB_CEP")
    private String cep;
    @Column(name="COB_CIDADE")
    private String cidade;
    ...
}
    
```

reusa mesma classe Endereco, mas redefine mapeamentos

O uso de `@AttributeOverrides` para redefinir os nomes das colunas é inevitável se houver mais de um *objeto embutido* do mesmo tipo na entidade, já que os nomes das colunas, por default, são gerados a partir dos atributos da classe, e não já uma estratégia default para gerar nomes diferentes automaticamente.

3.3.1 Coleções

Se uma entidade contém atributo com uma coleção de objetos embutidos, ou tipos básicos (que não sejam entidades) ele deve ser anotado com `@ElementCollection`. Por exemplo, a entidade abaixo contém uma coleção de objetos *Endereco*, que são `@Embeddable`, e uma coleção de Strings:

```
@Entity
public class Representantes {
    ...
    @ElementCollection
    private Set<Endereco> enderecos = new HashSet<>();

    @ElementCollection
    private Set<String> emails = new HashSet<>();
    ...
}
```

3.3.2 Chaves compostas

`@Embeddable` pode ser usado na criação de chaves compostas. Por exemplo, uma chave primária que consiste de dois campos do tipo String:

```
@Embeddable
public class DependenciaId {
    @Column(name = "appID")
    private String appID;

    @Column(name = "groupID")
    private String groupID;
    //...
}
```

Ela pode ser usada em uma entidade anotada com `@EmbeddedId`:

```
@Entity
@Table(name = "Dependencia")
public class Dependencia {

    @EmbeddedId
    private DependenciaId id;
    // ...
}
```

É responsabilidade da aplicação gerenciar o estado da chave composta, para que a persistência transitiva ocorra corretamente.

3.4 Persistência transitiva, cascading, lazy loading

Os relacionamentos em JPA são implementados em Java puro, e não são gerenciados pelo container. Por exemplo, se um *Item* possui um relacionamento com um *Pedido* da forma “um Pedido contém muitos Itens”, para acrescentar um item em um pedido, seria preciso sincronizar dois objetos: *Pedido* e *Item*:

```
Pedido p = new Pedido();
Item i = new Item();
p.addItem(i);      // atualizando o pedido
item.setPedido(p); // atualizando o item (se bidirectional)
```

Além disso, é necessário que essas entidades sejam persistentes. A tentativa de adicionar uma entidade transiente ou nova a uma entidade persistente irá causar uma *TransientObjectException*. É necessário que a entidade transiente ou nova seja antes sincronizada com o banco. Isto pode ser feito explicitamente, dentro de um contexto transacional, através de *persist()*/*merge()*:

```
em.persist(p);
em.persist(i);
```

Mas isto também pode ser realizado automaticamente através da configuração de persistência transitiva, que pode ser configurada automaticamente no JPA usando o atributo cascade das anotações de relacionamentos. Para propagar a inserção de entidades pode-se usar *CascadeType.PERSIST*. Por exemplo, na classe *Pedido* pode-se mapear os itens da forma:

```
@OneToMany(cascade={CascadeType.PERSIST}, mappedBy="item")
private Set<Item> itens = new HashSet<Item>();
```

Isto fará que quando um *Pedido* for persistido no banco, quaisquer itens que estiverem na sua hierarquia de objetos também serão persistidos automaticamente.

3.4.1 Configuração de CascadeType

O atributo cascade recebe um array {} de opções que podem ser combinadas. Há várias opções para *CascadeType*:

1. PERSIST – propaga operações *persist()* e *merge()* em objetos novos
2. MERGE – propaga operações *merge()*
3. REMOVE – propaga operações *remove()*
4. DETACH – propaga operações *detach()*
5. REFRESH – propaga operações *refresh()*
6. ALL – propaga todas as operações. Declarar *cascade=ALL* é o mesmo que declarar *cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}*

A escolha vai depender do design da aplicação. Um uso típico seria PERSIST com MERGE, para garantir que inserts e updates possam ser feitos automaticamente.

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
```

Talvez a remoção de um *Pedido* exija também a remoção de seus *Itens*. Neste caso um *CascadeType.REMOVE* seria necessário. Pode-se usar *CascadeType.ALL* para que todas as operações sejam transitivas (mas poderá haver impactos de performance no uso desnecessário de *CascadeType.ALL*).

3.4.2 Lazy loading

Lazy loading é um padrão de design usado para adiar a inicialização de um objeto para quando ele for realmente necessário. É uma técnica que contribui para a eficiência da aplicação se usado corretamente.

ORMs usam estratégias de lazy loading para otimizar as pesquisas. A navegação em uma coleção de referências realiza queries dentro de um contexto transacional. Fora desse contexto, apenas as referências que foram inicializadas quando estavam no contexto transacional são acessíveis, não as instâncias contendo dados.

Se você usar um cliente externo que não mantém sempre aberta a sessão do EntityManager, poderá ter exceções de inicialização lazy quando acessar objetos dentro de coleções. O query, por default, só retornará a coleção com os ponteiros para os objetos, e eles só podem ser recuperados do banco se houver uma sessão aberta.

A forma como o provedor reage diante dessa situação depende de implementação. Ao tentar acessar uma referência não inicializada, o Hibernate provoca uma *LazyInitializationException*. Já o EclipseLink, quando acontece a tentativa de acesso a uma coleção de um objeto desligado,

mesmo com o *EntityManager* fechado, ele abrirá uma nova conexão no banco e preencherá a coleção com dados obtidos do query, evitando o *LazyInitializationException*. Uma exceção Lazy ainda pode ocorrer se o objeto estiver serializado (pode ocorrer durante passivação em EJB, por exemplo). Neste caso é preciso lidar com o problema buscando alternativas para carregar os dados da coleção previamente.

É possível configurar esse comportamento configurando a recuperação como *eager* (ansiosa), em vez de *lazy* (preguiçosa). Isto pode ser feito de duas maneiras: através de mapeamento (default para todas as situações) e instruções no query (configurado a cada consulta).

O modo de recuperação *eager* pode ser configurado como default através de mapeamento, declarando nas associações o atributo *fetch=FetchType* da forma

```
@OneToOne(mappedBy="pedido", fetch=FetchType.EAGER)
private Set<Item> itens = new HashSet<Item>();
```

Configurar todas as operações para usar eager loading é uma má idéia pois isso forçará o sistema a recuperar toda a árvore de objetos quando baixar uma coleção. O ideal é analisar a aplicação e descobrir o que realmente precisa ser usado, e buscar apenas esses dados. Por exemplo, se há 1000 objetos em uma coleção talvez só seja necessário buscar 10, e depois mais 10 em outra transação, em vez dos 1000 de uma vez. Portanto, *eager loading* idealmente deve ser configurado apenas no query. Mapeamentos devem configurar relacionamentos como *lazy* por default.

Em JPQL a palavra-chave *fetch* é usada para sinalizar eager loading (em queries Criteria existe o método *fetch()* da interface *From*).

Para atualizar objetos envolvidos em relacionamentos através de persistência transitiva, declare-os com *CascadeType.MERGE*. Desta forma, quando um objeto for atualizado, o restante da árvore de relacionamentos será atualizada em cascata.

3.5 Mapeamento de Herança

JPA não interfere no modelo de domínio da aplicação. Herança em Java, no domínio do da linguagem, continua funcionando como se espera. Entidades também suportam herança de classes, associações e queries polimórficos. Podem ser abstratas ou concretas, e podem ser subclasses ou superclasses de classes que não são entidades.

Entidades abstratas, assim como qualquer classe abstrata, não podem ser instanciadas. Mas pode-se fazer queries em classes abstratas. Em um query envolvendo classes abstratas, todas as subclasses concretas da entidade abstrata serão envolvidas.

Herança é o descasamento mais visível entre os mundos relacional e orientado a objetos. O mundo OO possui relacionamento “é um” e “tem um”, enquanto que o mundo relacional apenas possui relacionamento “tem um”. Há três estratégias para lidar com esse problema (sugeridas por Scott Ambler, 2002). Essas estratégias são adotadas na arquitetura do JPA.

3.5.1 MappedSuperclass

Entidades podem ter subclasses que não são entidades. Se a superclasse possuir a anotação *@MappedSuperclass*, seus atributos serão herdados e farão parte do estado persistente das subclasses. Uma *@MappedSuperclass* também pode incluir anotações de persistência nos seus atributos, mas não é uma entidade e não pode ser usada com operações de *EntityManager* ou Query. Ela não está mapeada a uma tabela. Apenas as subclasses de entidades concretas são mapeadas a tabelas e podem conter colunas correspondentes aos atributos herdados.

No exemplo abaixo a classe *Quantia* foi anotada como *@MappedSuperclass*, portanto seu atributo valor e o ID serão mapeados a colunas das tabelas mapeadas às entidades *Pagamento* e *Dívida*:

```

@MappedSuperclass
public class Quantia {
    @Id
    protected Long id;
    protected BigDecimal valor; ...
}

@Entity
public class Pagamento extends Quantia { ... }

@Entity
public class Dívida extends Quantia { ... }

```

É possível também herdar de classes comuns que não têm nenhuma participação no JPA. Essas classes poderão ter métodos e atributos que serão herdados mas cujo estado *não será persistido no banco* (não haverá colunas correspondentes aos atributos herdados).

A anotação `javax.persistence.Inheritance` é usada para configurar o mapeamento de herança entre entidades em JPA. Elas implementam as três estratégias citadas que são selecionadas através das constantes da enumeração `InheritanceType`:

```

public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};

```

A estratégia default é `SINGLE_TABLE`, que mapeia todas as classes de uma hierarquia a uma única tabela no banco de dados.

3.5.2 Tabela por classe concreta

A estratégia chamada “*Uma tabela por classe concreta*” é, de acordo com a especificação, opcional, mas é suportada pelos principais provedores JPA usados hoje no mercado. Nela, o modelo relacional ignora herança e polimorfismo (o polimorfismo ocorre implicitamente), mapeando apenas as classes concretas a tabelas. A interface da classe abstrata ainda será usada nos queries, mas é ignorada na implementação que considera apenas a interface que foi herdada em cada classe concreta.

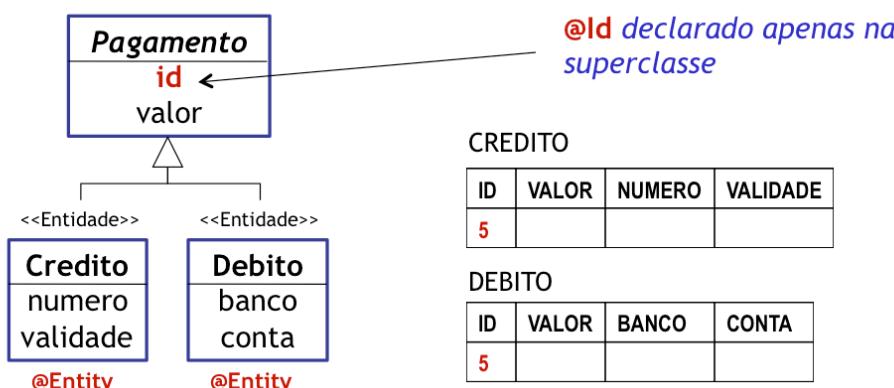
A ilustração abaixo mostra como essa estratégia poderia ser implementada com JPA.

```

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Pagamento {
    @Id @GeneratedValue private long id;
}

@Entity
public class Débito extends Pagamento {...}

```



Ou seja, todas as classes são declaradas como `@Entity`, mas apenas as subclasses são mapeadas a tabelas (não se pode usar `@Table` na classe abstrata). Como o ID é o mesmo para as subclasses, ele não é declarado nas entidades concretas e herda todas as suas definições de mapeamento da superclasse. Usando um gerador sequencial de IDs, o sistema irá usar uma contagem única para todas as subclasses.

Essa estratégia tem como vantagem o mapeamento direto entre classe e tabela normalizada. O gravação é realizada sempre através de uma classe concreta, mas a pesquisa pode ser polimórfica, realizada via classe abstrata e com o resultado que inclui todas as subclasses concretas.

Por exemplo, é possível filtrar os Pagamentos por valor, sem precisar saber se são de Crédito ou Débito:

```
select p from Pagamento p where p.valor > 1000
```

A desvantagem dessa estratégia é a complexidade e ineficiência dos queries gerados. Para realizar a pesquisa, será necessário envolver todas as tabelas mapeadas às subclasses das entidades.

Por exemplo, considere a estrutura o *query polimórfico* (realizado na entidade da superclasse abstrata) mostrados acima. Esse query irá gerar um SQL composto de subqueries concatenados com union para cada subclasse.

A listagem abaixo ilustra um *possível* resultado da geração do SQL para o exemplo acima:

```
select ID, VALOR, NUMERO, VALIDADE, BANCO, CONTA from (
    select ID, VALOR, NUMERO, VALIDADE,
           null as BANCO, null as CONTA from CREDITO
    union
    select ID, VALOR, null as NUMERO, null as VALIDADE,
           BANCO, CONTA from DEBITO
) where VALOR > 1000
```

Este é um SQL *conceptual*. É possível e bastante provável que o provedor de persistência utilizado otimize o SQL gerado e produza outro resultado mais eficiente, mas isso depende da implementação usada. O importante aqui é considerar que o query em uma única entidade (*Pagamento*) filtrado por um atributo declarado nessa entidade (*valor*) irá exigir queries em pelo menos duas tabelas (as subclasses concretas que *herdam* o atributo) que terão quer ser combinados. Portanto, essa solução é ineficiente se a hierarquia for grande e se a aplicação realizar muitos queries polimórficos.

3.5.3 Tabela por hierarquia

A estratégia chamada de “**Uma tabela por hierarquia de classes**” é default em JPA. Ou seja, se nenhuma `@Entity` que participa de uma hierarquia for anotada com `@Inheritance`, esta será a estratégia usada: a hierarquia inteira será mapeada a uma única tabela. Isto permite consultas polimórficas eficientes, já que há uma tabela única, mas as tabelas não serão normalizadas e poderão registros com muitos campos vazios.

Como apenas uma tabela é usada, é necessário que exista uma coluna identificando o tipo. Isto é feito através da anotação `@DiscriminatorColumn`. O nome default é `DTYPE` (essa coluna terá que existir, ter este nome e tipo String caso um `@DiscriminatorColumn` não seja definido explicitamente).

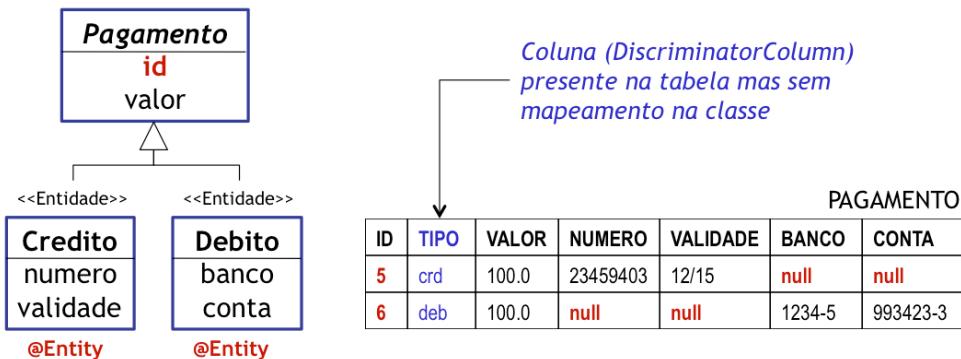
O exemplo abaixo ilustra essa estratégia usando o mesmo modelo de domínio apresentado anteriormente:

```

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TIPO")
public abstract class Pagamento {
    @Id @GeneratedValue private long id;
}

@Entity
@DiscriminatorValue("crd")
public class Debito extends Pagamento {...}

```



Portanto, é necessário que a tabela tenha colunas para todas as propriedades de todas as classes.

A vantagem é que teremos a forma mais eficiente de fazer uma pesquisa polimórfica, já que tudo acontece em uma única tabela. Este é o SQL conceitual para um query polimórfico usando esta estratégia:

```

select ID, VALOR, BANCO, CONTA, NUMERO, VALIDADE
  from PAGAMENTO
 where VALOR > 1000

```

Se for feito um query em classe concreta, o provedor de persistência poderia gerar o seguinte código SQL:

```

select ID, VALOR, BANCO, CONTA, NUMERO, VALIDADE
  from PAGAMENTO
 where TIPO = 'deb' and VALOR > 1000

```

A principal desvantagem dessa estratégia é que as tabelas não são normalizadas. A tabela poderá ter muitas colunas e campos vazios. As colunas de propriedades declaradas em subclasses precisam aceitar valores nulos. Isto pode ser ruim para grandes hierarquias.

3.5.4 Tabela por subclasse

A terceira estratégia, “**Uma tabela por subclasse**” é configurada em JPA usando *InheritanceType.JOINED*. Ela representa os relacionamentos “é um” através de relacionamentos “tem um” (chave estrangeira). Cada subclasse possui a sua própria tabela que contém colunas apenas para os campos não-herdados, e para a chave primária que é chave estrangeira da superclasse. A recuperação de dados é realizada de forma eficiente através de um *join* das tabelas.

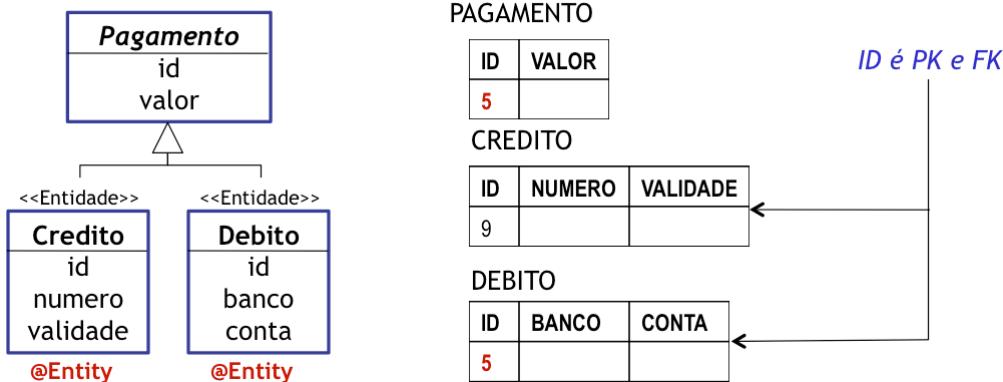
As vantagens são utilizar um modelo relacional normalizado, permite que a evolução não tenha efeitos colaterais e facilita a aplicação de restrições de integridade. Novas classes e tabelas podem ser criadas sem afetar classes e tabelas existentes.

O diagrama abaixo ilustra um mapeamento desse tipo:

```

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Pagamento {
    @Id @GeneratedValue private long id;
}
@Entity
public class Debito extends Pagamento {...}

```



Essa solução é melhor para gerar que para codificar a mão (bem trabalhosa se for feito em um sistema legado). A performance ainda pode não ser a ideal caso as hierarquias sejam muito complexas. Queries geralmente usam *outer join* para pesquisas polimórficas, e *inner join* para queries em classes concretas.

Dante das três opções, a escolha de qual estratégia usar depende do contexto de uso. Seguem algumas sugestões a considerar:

- Se não houver necessidade de queries polimórficos ou associações (e houver suporte por parte do provedor de persistência usado) use *Tabela por Classe Concreta* (`InheritanceType.TABLE_PER_CLASS`).
- Se houver necessidade de queries polimórficos, e hierarquia for simples use *Tabela por Hierarquia* (`InheritanceType.SINGLE_TABLE`), que é default.
- Se houver necessidade de associações polimórficas mas a hierarquia grande (ou não permitir tabelas não normalizadas) use *Tabela por Subclasse* (`InheritanceType.JOINED`).

Rode os exemplos que exploram estratégias de mapeamento de herança, e observe o log do provedor JPA que mostra o SQL gerado em cada situação.

3.6 Outros mapeamentos

3.6.1 Mapeamento de enumerações

Enumerações podem ser persistidas se marcadas com a anotação `@Enumerated`. Por exemplo:

```

public enum Regiao {
    NORTE, NORDESTE, SUL, SUDESTE, CENTRO_OESTE;
}

```

Para usar o enum acima em uma entidade e ter os valores persistentes, deve-se usar:

```

@Entity
public class SalaDeCinema {
    ...
@Enumerated(EnumType.STRING)
private Regiao regiao;

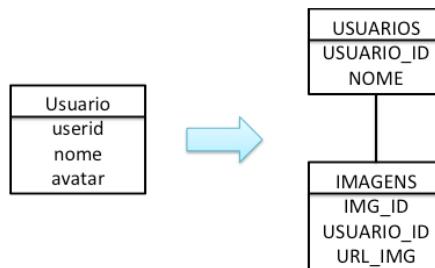
```

O EnumType *default* é *ORDINAL*, que grava um número no registro da tabela. *STRING* é mais seguro porque não é afetado pela *ordem* dos enums na classe (*STRING* grava texto na tabela, que independe da ordenação ou re-ordenação dos elementos do enum). Se forem inseridos outros itens no enum, mudando a ordem das constantes, os registros inseridos antes da mudança ficarão inconsistentes.

3.6.2 Mapeamento de uma instância a duas tabelas

Usa-se *@SecondaryTable* para construir uma entidade que obtém seus dados de duas tabelas.

No exemplo abaixo a entidade *Usuario* está mapeado a uma tabela principal (*USUARIOS*) e uma tabela secundária (*IMAGENS*) de onde obtém o estado do atributo *avatar* da coluna *URL_IMG*.



@Table é usado para mapear a tabela principal (se necessário) e *@SecondaryTable* informa a tabela secundária, indicando a(s) coluna(s) contendo a chave primária usada(s) para realizar a junção.

@PrimaryKeyJoinColumn informa a chave primária da tabela principal, e a chave referenciada na tabela secundária (se forem iguais, apenas o primeiro atributo é necessário). O mapeamento do atributo da segunda tabela usa *@Column* para informar a tabela e a coluna que deve ser mapeada.

```

@Entity
@Table(name="USUARIOS")
@SecondaryTable(name="IMAGENS",
    pkJoinColumns = @PrimaryKeyJoinColumn(name="USUARIO_ID",
        referencedColumnName="IMG_ID"))
public class Usuario implements Serializable {
...
    @Id Column(name="USUARIO_ID")
    private Long userId;

    @Column(table="IMAGENS", name="URL_IMG", nullable=true)
    private String avatar;
...
}
    
```

4 Queries

Uma vez configurados os mapeamentos, a aplicação poderá ser usada para inserir dados, e pesquisas podem ser construídas para localizar e recuperar entidades, com base em seu conteúdo. JPA possui duas estratégias nativas para realizar tais pesquisas:

- JPQL – uma linguagem similar a SQL baseada em comandos de texto, e
- Criteria – uma API usada para construir pesquisas através da hierarquia de objetos.

Em termos de resultado, tanto faz uma API como a outra. Todas possuem vantagens e desvantagens. JPQL é geralmente mais fácil de aprender, mas Criteria permite a criação de queries com maior potencial de reuso e evolução, além de ser recomendada para queries dinâmicas.

4.1 Cláusulas e joins

Queries em objetos têm princípios similares a queries em bancos de dados, mas há algumas diferenças.

Em todas as queries, a cláusula mais importante é a que declara *onde a consulta está sendo executada*. A API Criteria representa essa cláusula por um objeto chamado *Root*, ou raiz, e é representado nas duas formas de consulta pela cláusula *FROM*.

A cláusula SELECT indica *o que* está sendo selecionado. Consultas em JPA permitem selecionar entidades e seus atributos, e resultados de operações realizados sobre eles. É possível também selecionar múltiplos elementos.

A cláusula WHERE é opcional inclui filtros que restringem os resultados da pesquisa. Sem uma cláusula WHERE todos os elementos declarados no FROM serão consultados.

Quando as consultas são realizadas sobre objetos que possuem relacionamentos, os objetos que fazem parte do relacionamento são incluídos na consulta através de uma operação de JOIN, que podem e costumam ser implícitos tanto em JPQL como em Criteria. Operações que navegam no grafo de objetos relacionados usando o operador ponto (ex: *cliente.pedido.item*) escondem joins implícitos.

Joins podem ser usados com qualquer um dos quatro tipos de relacionamento, e com mapeamentos de ElementCollection.

Joins em árvores de objetos são parecidos mas não são iguais a joins no mundo relacional. Há três tipos de joins em JPA :

- Inner Joins (*default* – os joins implícitos são sempre inner joins)
- Left (ou outer) joins
- Fetch joins (que podem ser inner ou left)

Inner Joins consideram apenas os resultados que contém o relacionamento. Esse é o comportamento default. Se dentre os resultados houver elementos que não possuem o relacionamento declarado no JOIN, eles serão filtrados. Por exemplo, em uma consulta sobre objetos Cliente para obter o total de pedidos, um inner join não retorna os clientes que não têm pedidos. Já o left join retorna tudo, mesmo que o campo seja null.

A forma de inicialização dos elementos de uma coleção geralmente é lazy por default, e o comportamento da aplicação depende de como o contexto transacional é usado, e do provedor de persistência usado. Fetch Joins são usados para garantir que a coleção usada no relacionamento seja inicializada antes do uso. É uma alternativa melhor que declarar esse tipo de comportamento como default no mapeamento.

Exemplos de configuração de joins serão demonstrados nas seções a seguir usando JPQL e Criteria.

4.2 JPQL

JPA Query Language (JPQL) é uma linguagem de recuperação de dados similar a outras linguagens de query de objetos (HQL, EJB-QL, etc.) que a precederam. Parece com SQL, mas é diferente: opera sobre *objetos* e não tabelas, e é mais simples e eficiente para uso em aplicações orientadas a objetos.

Consultas em JPQL são objetos da classe *Query/TypedQuery* e podem ser construídas através de métodos de EntityManager. As instruções do query podem ser passadas diretamente para o método *createQuery()* como parâmetro do tipo String.

```

@PersistenceContext
EntityManager em;
...
TypedQuery query = em.createQuery("SELECT p FROM Produto p", Produto.class);

```

O query pode também ser declarado previamente em anotações `@NamedQuery` em cada entidade:

```

@Entity
@NamedQueries({
    @NamedQuery(name="selectAllProdutos", query="SELECT p FROM Produto p")
})
public class Produto implements Serializable { ... }

```

associado a um identificador de referência, que é passado para `createNamedQuery()`:

```
TypedQuery query = em.createNamedQuery("selectAllProdutos", Produto.class);
```

Uma consulta pode ser parametrizada. Os parâmetros são declarados em JPQL usando identificadores numéricos ou nomes precedidos por “:”:

```
TypedQuery query =
    em.createQuery("SELECT p FROM Produto p WHERE p.preco > :maximo", Produto.class);
```

Se houver parâmetros, eles podem ser preenchidos através de um ou mais métodos `setParameter()` antes de executar o query.

```
query.setParameter("máximo", 1000);
```

Os métodos de execução retornam os resultados. Se o resultado for uma coleção de entidades, pode-se usar `getResultSet()`:

```
List<Produto> resultado = query.getResultSet();
```

Se retornar apenas um item (por exemplo, se o produto for selecionado por ID ou um campo único), pode-se usar `getSingleResult()`:

```
Produto produto = query.getSingleResult();
```

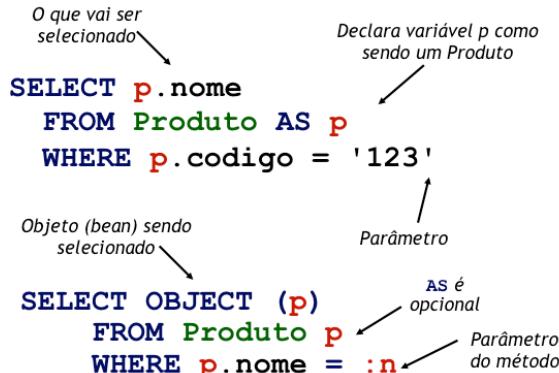
Existem outros métodos, inclusive métodos que não fazem pesquisa mas são usados para remoção ou atualização, como `executeUpdate()`. Existem também queries mais complexos que recebem muitos parâmetros e que devolvem outros dados, coleções de valores, atributos, e não apenas entidades.

4.2.1 Sintaxe essencial do JPQL

A principal cláusula de JPQL é SELECT. Também há suporte a UPDATE e DELETE, embora sejam menos usados. A sintaxe de uma expressão SELECT é:

```
cláusula_select cláusula_from
[cláusula_where] [cláusula_group_by] [cláusula_having] [cláusula_order_by]
```

As cláusulas entre colchetes são opcionais. O trecho abaixo ilustra a sintaxe elementar da cláusula SELECT:



A cláusula *FROM* é quem informa *qual o objeto que está sendo pesquisado*, e declara aliases usados no resto do query. Cada alias tem um identificador e um tipo. O *identificador* é qualquer palavra não-reservada e o *tipo* é o nome da entidade identificada como `@Entity`. A palavra-chave AS conecta o tipo ao identificador, mas é opcional e raramente é usada:

```
FROM Produto AS p
```

A cláusula FROM também pode selecionar múltiplos objetos e atributo e incluir vários conectores de JOIN (inner join, left outer join).

A cláusula *SELECT* informa *o que se deseja obter com a pesquisa*. Pode retornar *entidades*, *atributos* dos objetos, resultados de expressões, ou múltiplos elementos envolvendo atributos, entidades e resultados de expressões. Pode ser seguida de DISTINCT para eliminar valores duplicados nos resultados. SELECT utiliza o *alias* declarado na cláusula FROM:

```
SELECT p FROM Produto p
SELECT DISTINCT p FROM Produto p
SELECT p.codigo FROM Produto p
SELECT DISTINCT p.codigo FROM Produto p
```

A cláusula *WHERE* é opcional e restringe os resultados da pesquisa com base em uma ou mais expressões condicionais concatenadas. As expressões podem usar: *literais* (strings, booleanos ou números), *identificadores* (declarados no FROM), *operadores*, *funções* e *parâmetros* identificados por um número sequencial (?1, ?2, etc.) ou um identificador (:nome, :item).

Os literais usados nas pesquisas podem ser:

- Strings, representados entre apóstrofes: 'nome'
- Números, que têm mesmas regras de literais Java long e double
- Booleanos, representados por TRUE e FALSE (case-insensitive)

Os operadores do JPQL incluem:

- Expressões matemáticas +, -, *, /
- Expressões de comparação =, >, >=, <, <=, <>
- Operadores lógicos NOT, AND, OR
- Outros operadores: BETWEEN, NOT BETWEEN, IN, NOT IN, LIKE, NOT LIKE, NULL, NOT NULL, IS EMPTY, IS NOT EMPTY, MEMBER, NOT MEMBER

O operador LIKE possui operadores adicionais usados para indicar a parte de um string que está sendo pesquisada:

- _ representa um único caractere
- % representa uma seqüência de zero ou mais caracteres
- \ caractere de escape (necessário para usar _ ou %) literalmente

Funções podem aparecer em várias cláusulas. Algumas das funções agregadas do JPQL incluem:

- Manipulação de strings (usados em WHERE): CONCAT, SUBSTRING, TRIM, LOWER, UPPER
- LENGTH, LOCATE
- Funções aritméticas (usados em WHERE): ABS, SQRT, MOD, SIZE
- Data e hora (usados em WHERE): CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP
- Funções de agregação (usados em SELECT): COUNT, MAX, MIN, AVG, SUM

Cada versão nova do JPA têm introduzido novos comandos, funções, operadores. Para uma abordagem mais completa, consulte a documentação oficial do JPA. Apresentaremos o JPQL através de exemplos que ilustram várias situações típicas. Rode os exemplos fornecidos com este tutorial para verificar os resultados.

4.2.2 Exemplos de queries simples JPQL

Entendendo-se a estrutura básica do JPQL, não é muito difícil compreender os queries. Seguem alguns exemplos de queries usando JPQL:

- 1) Encontre todos os produtos que são chips e cuja margem de lucro é positiva

```
SELECT p
  FROM Produto p
 WHERE p.descricao = 'chip' AND (p.preco - p.custo > 0)
```

- 2) Encontre todos os produtos cujo preço é pelo menos 1000 e no máximo 2000

```
SELECT p
  FROM Produto p
 WHERE p.preco BETWEEN 1000 AND 2000
```

- 3) Encontre todos os produtos cujo fabricante é Sun ou Intel

```
SELECT p
  FROM Produto p
 WHERE p.fabricante IN ('Intel', 'Sun')
```

- 4) Encontre todos os produtos com IDs que começam com 12 e terminam em 3

```
SELECT p
  FROM Produto p
 WHERE p.id LIKE '12%3'
```

- 5) Encontre todos os produtos que têm descrições null

```
SELECT p
  FROM Produto p
 WHERE p.descricao IS NULL
```

- 6) Encontre todos os pedidos que não têm itens (coleção)

```
SELECT pedido
  FROM Pedido pedido
 WHERE pedido.itens IS EMPTY
```

- 7) Retorne os pedidos que contêm um determinado item (passado como parâmetro)

```
SELECT pedido
  FROM Pedido pedido
 WHERE :item IS MEMBER pedido.itens
```

- 8) Encontre produtos com preços entre 1000 e 2000 ou que tenham código 1001

```
SELECT p
  FROM Produto p
 WHERE p.preco BETWEEN 1000 AND 2000 OR codigo = 1001
```

4.2.3 Exemplos de queries usando relacionamentos

Pesquisas envolvendo relacionamentos produzem joins (que podem ser implícitos). Os dois queries a seguir produzem o mesmo resultado. O primeiro possui um inner join *implícito*:

- 9) Selecione todos os clientes com pedidos que tenham total maior que 1000:

```
SELECT c
  FROM Cliente c, IN(c_pedidos) p
 WHERE p.total > 1000
```

O mesmo query poderia ser escrito desta forma, expondo o join:

```
SELECT c
  FROM Cliente c
 INNER JOIN c_pedidos AS p
 WHERE p.total > 1000
```

Os três queries a seguir também obtém o mesmo resultado. O primeiro contém três joins implícitos (cada ponto é um join).

- 10) Selecione todos os lances onde o item é de categoria que começa com “Celular” e que tenha obtido lance acima de 1000:

```
SELECT lance
  FROM Lance lance
 WHERE lance.item.categoria.nome LIKE 'Celular%'
   AND lance.lanceObtido.total > 1000
```

Um dos joins é exposto neste segundo query:

```
SELECT lance
  FROM Lance lance
  JOIN lance.item item
 WHERE item.categoria.nome LIKE 'Celular%'
   AND item.lanceObtido.total > 1000
```

Finalmente, todos os joins são expostos neste query:

```
SELECT lance
  FROM Lance AS lance
  JOIN lance.item AS item
  JOIN item.categoria AS cat
  JOIN item.lanceObtido AS lanceVencedor
 WHERE cat.nome LIKE 'Celular%'
   AND lanceVencedor.total > 1000
```

4.2.4 Exemplos de queries usando funções, group by e having

- 11) Encontre a média do total de todos os pedidos:

```
SELECT AVG(pedido.total) FROM Pedido pedido
```

- 12) Obtenha a soma dos preços de todos os produtos dos pedidos feitos no bairro de Botafogo:

```
SELECT SUM(item.produto.preco)
  FROM Pedido pedido
  JOIN pedido.itens item
  JOIN pedido.cliente cliente
 WHERE cliente.bairro = 'Botafogo' AND cliente.cidade = 'Rio de Janeiro'
```

- 13) Obtenha a contagem de clientes agrupadas por bairro:

```
SELECT cliente.bairro, COUNT(cliente)
  FROM Cliente cliente GROUP BY cliente.bairro
```

- 14) Obtenha o valor médio dos pedidos, agrupados por pontos, para os clientes que têm entre 1000 e 2000 pontos:

```
SELECT c.pontos, AVG(pedido.total)
  FROM Pedido pedido
  JOIN pedido.cliente c
 GROUP BY c.pontos
 HAVING c.pontos BETWEEN 1000 AND 2000
```

4.2.5 Exemplos de subqueries

É possível usar os resultados de um query como parâmetro de outro através de subqueries. O query abaixo usa como restrição o resultado de um query que será testado com EXISTS.

- 15) Obtenha os empregados que são casados com outros empregados:

```
SELECT DISTINCT emp
  FROM Empregado emp
 WHERE EXISTS (SELECT conjuge
                  FROM Empregado conjuge
                 WHERE conjuge = emp.conjuge)
```

ALL e ANY são usados com subqueries. ALL retorna true se todos os valores retornados forem true, e ANY só retorna true o resultado da query for vazio ou se todos os valores retornados forem false.

16) Retorne apenas os produtos cujo preço seja maior que o valor incluído em todos os orçamentos:

```
SELECT produto
  FROM Produto p
 WHERE p.preco > ALL (SELECT o.item.preco
                           FROM Orcamento o
                           WHERE o.item.codigo = p.codigo)
```

4.2.6 Queries que retornam múltiplos valores

Queries que retornam múltiplos valores têm os resultados armazenados em arrays de objetos, onde cada índice do array corresponde respectivamente ao item selecionado, na ordem em que é expresso em JPQL. Considere por exemplo a seguinte entidade:

```
@Entity
public class Filme {
    @Id private Long id;
    private String imdb;
    private String titulo;
    @ManyToMany
    private Diretor diretor;
    ...
}
```

A query a seguir retorna três valores: um Long, um String e um Diretor, que é uma entidade, respectivamente.

```
SELECT f.id, f.titulo, d
  FROM Filme f join f.diretores d
 WHERE d.nome LIKE '%Allen'
```

A execução desse query irá retornar cada elemento como um *array* do tipo Object[] onde cada valor será armazenado em um índice. Se houver mais de um resultado, o tipo retornado será List<Object[]>. Por exemplo, para obter os dados da query acima pode-se usar:

```
List<Object[]> resultado = (List<Object[]>)query.getResultAsList();
for(Object obj : resultado) {
    Long id      = (Long)obj[0];
    String titulo = (String)obj[1];
    Diretor diretor = (Diretor)obj[2];
    ...
}
```

Muitas vezes, queries que retornam múltiplos valores são pesquisas com o objetivo de gerar relatórios, ou para coletar dados para uma interface. Depois de extrair os dados do array, provavelmente eles serão enviados para um objeto para preencher alguma View. Considere, por exemplo, esta classe, que poderia ser usada para armazenar os dados da query:

```
package com.acme.filmes;
public class DataTransferObject {
    private Long id;
    private String titulo;
    private Diretor diretor;
    public DataTransferObject(Long id, String titulo, Diretor diretor) {
        this.id = id;
        this.titulo = titulo;
        this.diretor = diretor;
    }
    ...
}
```

Poderíamos chama-la dentro do loop:

```
List<Object[]> resultado = (List<Object[]>)query.getResultAsList();
for(Object obj : resultado) {
    Long id      = (Long)obj[0];
    String titulo = (String)obj[1];
    Diretor diretor = (Diretor)obj[2];
    DataTransferObject dto = new DataTransferObject(id, titulo, director);
    // envia dto para algum lugar
}
```

Uma alternativa a `Object[]` é declarar o tipo do `CriteriaQuery` com a interface `javax.persistence.Tuple`. Os resultados ainda precisam ser extraídos individualmente, mas são então retornados em um objeto que permite recuperar os elementos como uma `List` (`tupla.get(0)`, `tupla.get(1)`, etc.)

Existe, porém, uma sintaxe de SELECT que elimina a necessidade de escrever todo esse código, e permite chamar o construtor diretamente dentro do query:

```
SELECT new com.acme.filmes.DataTransferObject(f.id, f.titulo, d)
  FROM Filme f join f.diretores d
 WHERE d.nome LIKE '%Allen'
```

A especificação do JPA 2.1, porém, requer que o construtor use o nome qualificado da classe. Usando esta sintaxe, o query retornará uma `List<DataTransferObject>` em vez de uma `List<Object[]>`, que poderá ser enviada diretamente para o componente que irá usá-la:

```
@Named
public class ManagedBean {
    public List<DataTransferObject> getDataToPopulateComponent() {
        ...
        return query.getResultAsList();
    }
    ...
}
```

4.2.7 Named Queries

Queries podem ser declarados em anotações e recuperadas pelo nome. Isto, em geral é uma boa prática porque mantém todos os queries juntos, e pode facilitar a manutenção deles. Por outro lado, os mantém distante do código que cria os queries e preenche os parâmetros, que pode dificultar o uso.

Para declarar queries desta forma, use a anotação `@NamedQueries` como mostrado no exemplo abaixo:

```
@Entity
@NamedQueries({
    @NamedQuery(name="produtoMaisBarato",
                query="SELECT x FROM Produto x WHERE x.preco > ?1"),
    @NamedQuery(name="produtoPorNome",
                query="SELECT x FROM Produto x WHERE x.nome = :nomeParam")
})
public class Produto { ... }
```

Para usar, chame o query pelo nome quando usar o EntityManager através do método `createNamedQuery()`, e preencha seus parâmetros se houver:

```
EntityManager em = ...
Query q1 = em.createNamedQuery("produtoMaisBarato");
q1.setParameter(1, 10.0);
List<Produto> resultado1 = q1.getResultList();

Query q2 = em.createNamedQuery("produtoPorNome");
q2.setParameter("nomeParam", "10.0");
List<Produto> resultado2 = q2.getResultList();
```

Named queries não são sempre recomendados, mas queries parametrizados sim. Deve-se evitar a construção de queries através da concatenação de strings por vários motivos. Queries parametrizados não apenas tornam o código mais legível e evitam erros, como melhoram a performance e evitam a necessidade de converter tipos.

4.3 Criteria

Criteria é uma API do JPA que permite a construção de queries dinâmicos usando objetos. Os queries são expressos usando construções orientadas a objeto.

Uma das vantagens da API Criteria é a possibilidade de descobrir erros de sintaxe em queries em tempo de compilação. Por exemplo, considere o query JPQL:

```
select p from Produto where p.preco < 50.0
```

É muito fácil esquecer uma letra e é muito difícil achar o erro. O que está errado no query acima?

Apenas uma letra. Este é o query correto:

```
select p from Produto p where p.preco < 50.0
```

Portanto, se o programador esquecer de digitar o *alias* “p” após *Produto*, a sintaxe do query estará incorreta. A menos que tenha alguma ferramenta especial para validar o JPQL em tempo de desenvolvimento, o programador só descobrirá o erro após o deploy, pois erros da string do JPQL não são descobertos em tempo de compilação.

Por outro lado, queries em Criteria têm muitas linhas e são complexos. É preciso conhecer bem a API antes de obter os benefícios dela. E muitas linhas de código, APIs gigantes, também são uma fonte de bugs. O código necessário para construir um query Criteria que retorne os mesmos resultados que o JPQL mostrado acima seria:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class); // tipo do select
Root<Produto> raiz = query.from(Produto.class); // tipo do from
Predicate condicao = cb.lessThan(raiz.get("preco"), 50.0); // predicado
query.where(condicao); // adiciona a cláusula where
query.select(raiz); // adiciona a cláusula select (diz o que vai ser selecionado)
```

Depois o query pode ser manipulado da forma como era manipulado em JPQL, passar parâmetros se houver, executar e extraír resultados.

Apesar de burocráticos e longos, queries Criteria são ideais para a criação de consultas dinâmicas, evitando a arriscada concatenação de strings costuma acontecer em consultas dinâmicas expressas em JPQL.

Queries de Criteria são um *grafo de objetos*. A raiz do grafo é encapsulado no *CriteriaQuery*. Um query mínimo requer o uso de três classes de *javax.persistence.criteria*: *CriteriaBuilder*, que encapsula vários métodos para construir o query, *CriteriaQuery*, que representa a consulta, e *Root* que é raiz da consulta e representa os objetos selecionados pela cláusula *FROM*.

4.3.1 Sintaxe essencial de Criteria

O primeiro passo é criar um *CriteriaQuery* através da classe *CriteriaBuilder*:

```
CriteriaBuilder builder = entityManagerFactory.getCriteriaBuilder();
```

E em seguida obter o objeto que irá encapsular o query:

```
CriteriaQuery<Produto> query = builder.createQuery( Produto.class );
```

O objeto raiz do grafo é declarado através da interface *Root*, que constrói a cláusula “from” do query. Isto é equivalente a fazer “*from Produto p*” em JPQL:

```
Root<Produto> p = query.from ( Produto.class );
```

Finalmente constrói-se a cláusula “select” do query usando o método select():

```
query.select(p);
```

O query está pronto. Neste momento ele é equivalente ao string JPQL. Para executá-lo é preciso passar o objeto query como parâmetro de um TypedQuery, para em seguida obter os resultados:

```
TypedQuery<Produto> query = em.createQuery(query);
List<Produto> resultado = query.getResultList();
```

Consultas envolvendo múltiplos objetos podem ser feitas com from(), join() ou fetch() e métodos similares. O query JPQL:

```
SELECT p1, p2 FROM Produto p1, Produto p2
```

Pode ser escrito usando Criteria da seguinte forma:

```
CriteriaQuery<Tuple> criteria = cb.createQuery(Tuple.class);
Root<Produto> p1 = criteria.from(Produto.class);
Root<Produto> p2 = criteria.from(Produto.class);
criteria.multiselect(p1, p2);
```

Um join como na query abaixo:

```
SELECT item, produto.nome FROM Item item LEFT OUTER JOIN item.produto produto
```

Pode ser escrito em Criteria da seguinte forma:

```
CriteriaQuery<Tuple> criteria = cb.createQuery(Tuple.class);
Root<Item> item = criteria.from(Item.class);
Join<Item> produto = item.join("produto", JoinType.LEFT);
criteria.multiselect(item, produto.get("nome"));
```

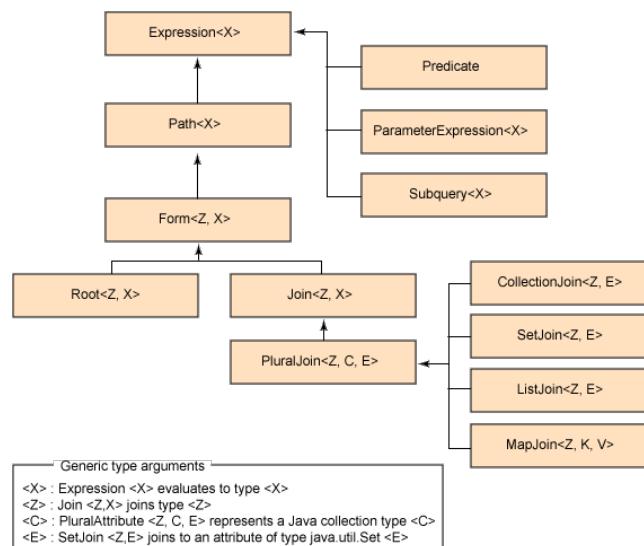
A interface Join (que é retornada pelo método join()) é subinterface da interface From, portanto pode ser usada em métodos que esperam um From. Um join usando fetch como este em JPQL:

```
SELECT item FROM Item item JOIN FETCH item.produto
```

pode ser expresso em Criteria usando a seguinte sintaxe:

```
CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
Root<Item> item = criteria.from(Item.class);
Fetch<Item, Produto> produto = item.fetch("produto");
criteria.select(item);
```

A API Criteria possui muitas interfaces, classes e métodos. O diagrama a seguir ilustra a hierarquia das expressões de query do Criteria, que são usadas para construir as cláusulas da consulta. Um query típico usa a maioria dessas interfaces:



Fonte: <http://www.ibm.com/developerworks/library/j-typesafejpa/>

Algumas dessas interfaces são:

- **Expression** – é uma interface que representa uma expressão (é superinterface para uma grande hierarquia de expressões). Expressões são construídas recebendo outras expressões como parâmetro. A maioria é retornada por métodos de CriteriaQuery e CriteriaBuilder (ex: sum(), diff(), equal())
- **Predicate** – é um tipo especial de Expression que representa a conjunção (ou disjunção) das restrições de um query e representa um valor booleano. A cláusula where recebe um Predicate. Predicados complexos são composições de outros predicados. Grande parte é criado por métodos de CriteriaBuilder (ex: equal(), and(), or(), between())
- **Path** – Representa um caminho de navegação em um grafo de objetos, por exemplo: *objeto.referencia.atributo*. O método get() de Root permite acesso a atributos de objetos através de expressões de path.
- **Join** – Representa o join para uma @Entity, objeto @Embeddable ou @Basic.

Algumas outras classes e interfaces importantes da API Criteria são:

- **Selection** – superinterface que representa qualquer item que é retornado em um resultado de query, como uma expressão (Expression), subquery (SubQuery), predicho (Predicate), caminho (Path), condicionais (Case), etc.
- **SubQuery** – Representa um subquery.

A hierarquia que representa um query em Criteria possui duas classes: CriteriaQuery (query principal) e Subquery. Elas têm uma superclasse em comum chamada de AbstractQuery. Os métodos da classe AbstractQuery representam *cláusulas* e outras partes de um query e valem para queries principais e subqueries. Os principais métodos são:

- Herdados de AbstractQuery (valem para queries principais e subqueries): *distinct(), from(), groupBy(), having(), subQuery(), where()*
- Definidos em CriteriaQuery (valem apenas para queries principais): *multiselect(), select()*
- Definidos em SubQuery (valem apenas para subqueries): *correlate(), getParent()*

A classe CriteriaBuilder possui métodos para a construção de todas as expressões usadas nas cláusulas dos queries e subqueries. As expressões são encapsuladas em instâncias de Expression ou subclasses e podem ser criadas simplesmente chamando os factory methods de CriteriaBuilder. As expressões envolvem desde a simples criação de literais até condicionais, expressões booleanas, de comparação, etc.

Exemplo:

```
Expression<Integer> i1 = builder.literal(123);
Expression<Integer> i2 = builder.sum(3,4);
Predicate p1 = builder.and(true, false);
Predicate p2 = builder.not(p1);
```

Normalmente os métodos são chamados diretamente na construção de condições (usadas em cláusulas where(), por exemplo). Os métodos englobam todas as expressões disponíveis em JPQL e têm nomes auto-explicativos.

A documentação oficial cobre de forma abrangente e com muitos exemplos o uso de queries Criteria. Foge ao escopo deste curso introdutório explorar esses recursos em detalhes, mas, como fizemos com JPQL, apresentaremos vários exemplos que envolvem situações típicas mais comuns. Tente executar o código dos exemplos fornecidos e construir seus próprios queries.

4.3.2 Exemplos de queries simples usando Criteria

Os exemplos abaixo mostram queries em Criteria API que produzem os mesmos resultados que os que foram mostrados anteriormente usando JPQL. Compare as duas formas:

- 1) Encontre todos os produtos que são chips e cuja margem de lucro é positiva

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Predicate igual = cb.equal(root.get("descricao"), "chip");
Expression<Double> subtracao = cb.diff(root.get("preco"), root.get("custo"));
Predicate maiorQue = cb.greaterThan(subtracao, 0.0);
Predicate clausulaWhere = cb.and(igual, maiorQue);

query.where(clausulaWhere);
query.select(root);

TypedQuery<Produto> q = em.createQuery(query);
// ...
```

- 2) Encontre todos os produtos cujo preço é pelo menos 1000 e no máximo 2000

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Predicate between = cb.between(root.get("preco"), 1000.0, 2000.0);
query.where(between);
query.select(root);
```

- 3) Encontre todos os produtos cujo fabricante é Sun ou Intel

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
query.where(root.get("fabricante").in("Intel", "Sun"));
query.select(root);
```

- 4) Encontre todos os produtos com IDs que começam com 12 e terminam em 3

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
query.where(cb.like(root.get("id"), "12%3"));
query.select(root);
```

- 5) Encontre todos os produtos que têm descrições null

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
query.where(cb.isNull(root.get("descricao")));
query.select(root);
```

- 6) Encontre todos os pedidos que não têm itens (coleção)

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pedido> query = cb.createQuery(Pedido.class);
Root<Pedido> root = query.from(Pedido.class);
query.where(cb.isEmpty(root.get("itens")));
query.select(root);
```

- 7) Retorne os pedidos que contêm um determinado item (passado como parâmetro)

```
Item item = new Item(); // item a ser testado
// ...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pedido> query = cb.createQuery(Pedido.class);
Root<Pedido> root = query.from(Pedido.class);
Predicate isMember = cb.isMember(item, root.get("itens"));
query.where(isMember);
query.select(root);
```

- 8) Encontre produtos com preços entre 1000 e 2000 ou que tenham código 1001

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Predicate between = cb.between(root.get("preco"), 1000.0, 2000.0);
Predicate igual = cb.equal(root.get("codigo"), 1001);

query.where(cb.and(between, igual));
query.select(root);
```

4.3.3 Exemplos de queries usando relacionamentos

Pesquisas envolvendo relacionamentos produzem joins (que podem ser implícitos). Os dois queries a seguir produzem o mesmo resultado. O primeiro possui um inner join *implícito*:

- 9) Selecione todos os clientes com pedidos que tenham total maior que 1000:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Cliente> query = cb.createQuery(Cliente.class);
Root<Cliente> root = query.from(Cliente.class);
Path<Pedido> pedido = root.get("pedidos");

query.where(cb.greaterThan(pedido.get("total"), 1000.0));
query.select(root);
```

Mesmo query usando um inner join explícito:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Cliente> query = cb.createQuery(Cliente.class);
Root<Cliente> root = query.from(Cliente.class);

Join<Cliente, Pedido> pedido = root.join("pedidos");

query.where(cb.greaterThan(pedido.get("total"), 1000.0));
query.select(root);
```

- 10) Selecione todos os lances onde o item é de categoria que começa com “Celular” e que tenha obtido lance acima de 1000:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Lance> query = cb.createQuery(Lance.class);
Root<Lance> root = query.from(Lance.class);

query.where(cb.and(
    cb.like(root.get("item").get("categoria").get("nome"), "Celular%"),
    cb.greaterThan(root.get("item").get("lanceObtido").get("total"), 1000.0))
));

query.select(root);
```

Mesmo query usando vários inner joins explícitos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Lance> query = cb.createQuery(Lance.class);
Root<Lance> root = query.from(Lance.class);

Join<Lance, Item> item = root.join("item");
Join<Item, Categoria> cat = item.join("categoria");
Join<Item, Lance> vencedor = item.join("lanceObtido");

query.where(cb.and(cb.like(cat.get("nome"), "Celular%"),
    cb.greaterThan(vencedor.get("total"), 1000.0)));
query.select(root);
```

4.3.4 Exemplos de queries usando funções, group by e having

11) Encontre a média do total de todos os pedidos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Double> query = cb.createQuery(Double.class); // avg retorna Double
Root<Pedido> root = query.from(Pedido.class); // from Ingresso i
Expression<Double> media = cb.avg(root.get("total"));
query.select(media);
```

12) Obtenha a soma dos preços de todos os produtos dos pedidos feitos no bairro de Botafogo:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<BigDecimal> query = cb.createQuery(BigDecimal.class);
Root<Pedido> root = query.from(Pedido.class);

Join<Pedido, Item> item      = root.join("itens");
Join<Pedido, Cliente> cliente = item.join("cliente");

Predicate where = cb.and(
    cb.equal(cliente.get("bairro"), "Botafogo"),
    cb.equal(cliente.get("cidade"), "Rio de Janeiro")
);
query.where(where);
Expression<BigDecimal> total = cb.sum(item.get("produto").get("preco"));
query.select(total);
```

13) Obtenha a contagem de clientes agrupadas por bairro:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Cliente> root = query.from(Cliente.class);
query.multiselect(root.get("bairro"), cb.count(root));
query.groupBy(root.get("bairro"));

TypedQuery<Object[]> q = em.createQuery(query);
...
```

14) Obtenha o valor médio dos pedidos, agrupados por pontos, para os clientes que têm entre 1000 e 2000 pontos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Pedido> root = query.from(Pedido.class);
query.multiselect(root.get("pontos"), cb.avg(root.get("total")));
Join<Pedido, Cliente> cliente = root.join("cliente");
query.groupBy(cliente.get("pontos"));
query.having(cb.between(cliente.get("pontos"), 1000.0, 2000.0));
```

4.3.5 Exemplos com subqueries

15) Obtenha os empregados que são casados com outros empregados:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Empregado> query = cb.createQuery(Empregado.class);
Root<Empregado> root = query.from(Empregado.class);

Subquery<Empregado> subquery = query.subquery(Empregado.class);
Root<Empregado> conjugue = subquery.from(Empregado.class);
subquery.where(cb.equal(conjuge.get("conjuge "), root.get("conjuge ")));
subquery.select(conjuge);

query.where(cb.exists(subquery));
query.select(root).distinct(true);
```

16) Retorne apenas os produtos cujo preço seja maior que o valor incluído em todos os orçamentos:

```

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Subquery<Empregado> subquery = query.subquery(Empregado.class);
Root<Orcamento> orcamento = subquery.from(Orcamento.class);
subquery.where(cb.equal(orcamento.get("item").get("codigo"), root.get("codigo")));
subquery.select(orcamento.get("item").get("preco"));

query.where(cb.greaterThan(root.get("preco"), cb.all(subquery)));
query.select(root);

```

4.3.6 Queries que retornam múltiplos valores

O query usando select com construtor mostrado na seção anterior com JPQL pode ser construído com Criteria da forma abaixo:

```

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Lugares> query = cb.createQuery(DataTransferObject.class);
Root<Filme> root = query.from(Filme.class);
Join<Filme, Diretor> diretor = root.join("etapas");
query.where(cb.like(root.get("nome"), "%Allen"));
query.select(cb.construct(Lugares.class,
    etapa.get("origem").get("nome"),
    etapa.get("destino").get("nome"))
);

```

4.3.7 Typesafe Criteria com static metamodel

É bem mais fácil encontrar erros em queries Criteria, comparados a JPQL, porque o compilador ajuda na tarefa e detecta queries incorretos. Mas eles ainda podem acontecer já que a leitura dos campos das entidades feita através de um get() recebe um String. Por exemplo, a linha abaixo para ler o campo “preco” não contém erros de compilação:

```
Predicate condicao = qb.lt(raiz.get("preco"), 50.0);
```

Mas se o string estiver errado, o query está incorreto. Portanto, existem erros de sintaxe em queries do Criteria que também não serão capturados em tempo de compilação.

Mas há uma solução: o *typesafe query*.

```

CriteriaBuilder qb = em.getCriteriaBuilder();
CriteriaQuery<Produto> cq = qb.createQuery(Produto.class);
Root<Produto> raiz = cq.from(Produto.class)
Predicate condicao = qb.lt(raiz.get(Produto_.preco), 50.0);
cq.where(condicao);
TypedQuery<Person> query = em.createQuery(cq);

```

Observe que o nome do atributo agora é informado através de código Java, usando um atributo estático e público da classe *Produto_*. Essa classe é chamada de *metamodelo estático (static metamodel)*. Usando os atributos através dela em vez de usar strings torna os queries Criteria typesafe.

Elas precisam ser geradas. Todos os IDEs que suportam JPA 2 têm ferramentas para isto. Por exemplo, em um projeto Maven com provedor EclipseLink, pode-se incluir a geração automática em uma das fases do POM.xml através de um plugin que requer esta dependência:

```

<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
  <version>2.5.0</version>
</dependency>

```

E a configuração abaixo. Com isso os metamodelos serão gerados durante a fase de geração de código do build (o plugin usa as classes declaradas no persistence.xml):

```

<plugin>
    <groupId>org.bsc.maven</groupId>
    <artifactId>maven-processor-plugin</artifactId>
    <version>2.2.4</version>
    <executions>
        <execution>
            <id>eclipselink-jpa-metamodel</id>
            <goals>
                <goal>process</goal>
            </goals>
            <phase>generate-sources</phase>
            <configuration>
                <processors>
                    <processor>
                        org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor
                    </processor>
                </processors>
                <outputDirectory>
                    ${project.build.directory}/generated-sources/meta-model
                </outputDirectory>
            </configuration>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>
            <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
            <version>2.5.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</plugin>

```

5 Tuning em JPA

Várias operações JPA precisam de um contexto transacional para execução, e todas utilizam um ou mais níveis de cache. Um método transacional pode ser insuficiente para proteger a integridade de um processo e estratégias de locks otimistas ou pessimistas poderão ser necessárias.

Nesta seção serão discutidas algumas configurações que podem ser realizadas em aplicações JPA e que envolvem transações, locks, caches e operações em lote.

5.1 Transações

Transações em JPA podem ser distribuídas (*JTA*) ou configuradas como um recurso local *RESOURCE_LOCAL*. Apenas transações JTA podem ser gerenciadas pelo container e usadas para configurar suporte transacional a métodos de forma transparente e declarativa. Transações JTA também podem ser controladas programaticamente através da API da classe *UserTransaction*, que pode ser injetada como resource em componentes Java EE.

5.1.1 Resource-local javax.persistence.EntityTransaction

Usada em ambientes Java SE ou onde não há suporte a transações distribuídas, o método *getTransaction()* de *EntityManager* pode ser usado para obter o contexto transacional obrigatório para operações de persistência. As transações são delimitadas pela chamada dos métodos *begin()* e *commit()/rollback()*:

```

public class AlunoDAO {
    private EntityManagerFactory emf;

    AlunoDAO() {
        emf = Persistence.createEntityManagerFactory("escola-PU");
    }

    public void addAluno(Aluno aluno) {
        EntityManager em = emf.getEntityManager();

        try {
            em.getTransaction().begin();
            em.persist(aluno);
            em.getTransaction().commit();
        } catch(Exception e) {
            em.rollback();
        } finally {
            em.close();
        }
    }
}

```

O *persistence.xml* deve indicar *RESOURCE_LOCAL* como transaction-type:

```
<persistence-unit name="tutorial-jpa" transaction-type="RESOURCE_LOCAL">
```

5.1.2 JTA javax.transaction.UserTransaction

Disponível em servidores Java EE. Pode ser obtida e injetada como um *@Resource* em WebServlets, EJBs, e outros componentes que rodam no container Java EE. Em ambientes Java EE com CDI também pode ser injetada com *@Inject*.

```

public class AlunoDAOBean {
    @PersistenceContext(unitName="escolar-PU")
    private EntityManager;

    @Resource
    UserTransaction ut;

    public void addAluno(Aluno aluno) {
        try {
            ut.begin();
            em.persist(aluno);
            ut.commit();
        } catch(Exception e) {
            ut.rollback();
        } finally {
            em.close();
        }
    }
}

```

Em EJB transações gerenciadas pelo container (CMT) é o *comportamento default* e os métodos de um SessionBean são automaticamente incluídos em um contexto transacional.

```

@Stateless
public class AlunoSessionBean {

    @PersistenceContext
    EntityManager em;

    public void addAluno(Aluno aluno) {
        em.persist(aluno);
    }
}

```

Em CDI o mesmo comportamento pode ser obtido nos métodos declarados como `@Transactional`.

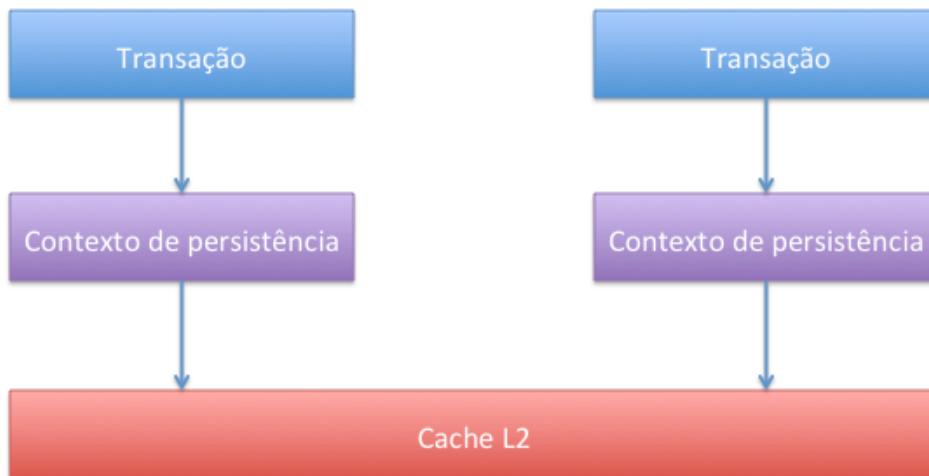
```
@Model
public class AlunoSessionBean {

    @Inject
    EntityManager em;

    @Transactional
    public void addAluno(Aluno aluno) {
        em.persist(aluno);
    }
}
```

5.2 Cache

O JPA possui dois níveis de cache. O primeiro é o *Contexto de Persistência*, controlado pelo `EntityManager`. O segundo nível de cache (L2) é um mecanismo compartilhado. Cache é um recurso que deve ser usado com cuidado, pois há grande risco de resultar em inconsistência de dados que podem inclusive introduzir bugs. Mas o custo-benefício com os ganhos de performance poderá compensar o esforço extra de gerenciar o cache.



5.2.1 Cache de primeiro nível (L1, EntityManager)

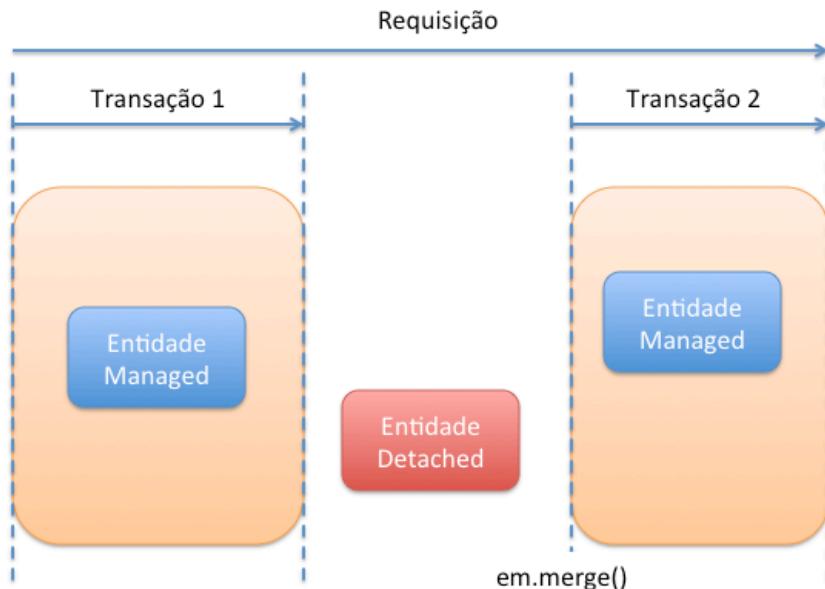
O `EntityManager` normalmente está vinculado a uma transação, que dura no mínimo o tempo de uma operação. Durante essa operação o objeto é sincronizado com o banco e copiado para o contexto de persistência.

Nos ambientes que possuem transações gerenciadas pelo container, a duração do contexto de persistência é a mesma do contexto transacional, que geralmente é aplicado em métodos que podem conter várias operações do `EntityManager`. Dentro do contexto transacional, o contexto de persistência mantém o estado dos objetos e evita que eles tenham que ser recuperados do banco.

O padrão recomendado para ambientes com transações controladas pela aplicação é também abrir o contexto transacional logo após a obtenção da sessão do `EntityManager`, e fechar o `EntityManager` logo após cometer ou fazer rollback da transação.

O `EntityManager` garante uma única instância por contexto de persistência. Mas pode haver vários contextos (ex: transações de outros usuários). Neste caso ainda é preciso usar estratégias de locking para garantir a consistência de dados.

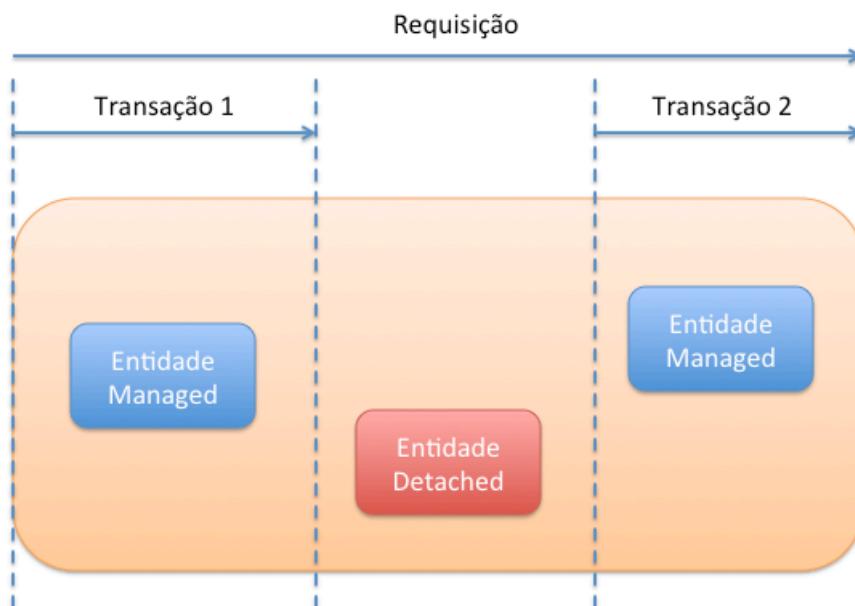
Quando uma transação e o contexto de persistência são fechados, as entidades entram no estado detached (desligadas) e precisam de alguma política de locking para garantir sua integridade, caso precisem ser religadas na próxima transação. A imagem abaixo ilustra essa situação.



O contexto de persistência pode ser estendido para manter-se aberto além do escopo de uma transação. Isto permitirá que a aplicação tenha acesso aos objetos carregados no contexto de persistência, que poderão ser usados para leitura ou exibição em uma View. Em transações controladas pela aplicação, o EntityManager é aberto no início da requisição, e só é fechado (com `close()`) no final. O objeto poderá, nesse intervalo, passar por várias transações e usando os dados do cache do contexto de persistência.

Em ambientes gerenciados pelo container, o EntityManager pode ser injetado com a opção `PersistenceContextType.EXTENDED` que manterá o contexto de persistência aberto entre transações.

```
@PersistenceContext(PersistenceContextType.EXTENDED)
private EntityManager entityManager;
```



Por um lado existe a vantagem de não precisar fazer uma nova chamada ao banco para ter acesso a uma entidade. Por outro, se a operação se estende por muito tempo há o risco do objeto ficar obsoleto, há uma chance maior dele ser alterado em outra transação (e necessitar de *locking*) e do uso possivelmente desnecessário de recursos.

5.2.2 Cache de segundo nível (L2)

O cache de segundo nível (L2) é compartilhado entre contextos de persistência. É preciso habilitá-lo explicitamente. Pode-se habilitar o cache para todas as entidades introduzindo a seguinte linha no persistence.xml:

```
<persistence-unit ...>
    <shared-cache-mode>ALL</shared-cache-mode>
</persistence-unit>
```

ou através de uma propriedade:

```
<persistence-unit ...>
...
<properties>
    ...
        <property name="javax.persistence.sharedCache.mode" value="ALL"/>
    </properties>
</persistence-unit>
```

Habilitar o cache para todas as entidades pode não ser uma boa idéia. A estratégia oposta é desabilitar para todas as classes *exceto* aquelas anotadas como `@Cacheable`. Neste caso, em vez de ALL, use `ENABLE_SELECTIVE`. Que desabilita o cache para todas as entidades, a menos que elas tenham a anotação `@Cacheable`.

```
@Cacheable
@Entity
public class Produto { ... }
```

Se a maioria dos objetos deve participar do cache, pode-se usar a opção `DISABLE_SELECTIVE`. Neste caso, todas as entidades vão para o cache L2 a menos que tenham declarado `@Cacheable(false)`.

Se o serviço estiver habilitado, objetos que não forem encontrados no contexto de persistência serão buscados no cache L2.

Para que um objeto tenha acesso ao cache é preciso utilizar a API da classe Cache. Pode-se obter uma instância de Cache através de

```
Cache cache = emf.getCache();
```

Os principais métodos dessa classe são:

- `contains(classe, entidade)` – que retorna true se o objeto estiver no cache.
- `evict(classe, objeto)` – remove o objeto do cache
- `evictAll()` – esvazia o cache

O trecho de código abaixo mostra como verificar se um objeto está no cache, e remove-lo do cache se for o caso.

```
boolean isCached = cache.contains(Produto.class, Long.valueOf(123));
if (isCached) {
    cache.evict(Produto.class, Long.valueOf(123));
}
```

Pode-se também invalidar todas as entidades de uma classe:

```
cache.evict(MyEntity.class);
```

Ou ainda esvaziar o cache inteiro:

```
cache.evictAll();
```

Os objetos cacheáveis são automaticamente adicionados ao cache durante o commit().

O cache também pode ser configurado para um EntityManager específico, ou mesmo para um query específico usando a propriedade `javax.persistence.cache.retrieveMode`.

```
Query query = em.createQuery("Select p from Produto p");
query.setHint("javax.persistence.cache.retrieveMode", CacheStoreMode.BYPASS);
```

A propriedade `retrieveMode` informa um modo de gravação para o query (`CacheStoreMode`), que pode ter os valores:

- *USE* (usa o objeto do cache)
- *BYPASS* (ignora o objeto do cache)
- *REFRESH* (atualiza o objeto do cache)

Usar o cache de segundo nível tem vantagens e desvantagens. Entre as vantagens está evitar acessar o banco para entidades que já foram carregadas. Isto é recomendado para entidades que são acessadas com frequência e nunca ou raramente são modificadas. Entidades que são boas candidatas ao cache L2 são aquelas que são lidas com frequência, modificadas raramente e que não sejam inválidas se desatualizadas.

Como desvantagens estão o consumo de memória maior e objetos que podem ficar obsoletos (caso sejam atualizados no banco). Obviamente esse cache não deve ser usado para objetos que serão atualizados com frequência.

5.3 Locks

Locks travam um objeto para evitar alterações. Há duas estratégias:

- Lock otimista: não impede o acesso ao objeto, mas incrementa um número de versão das alterações. Se o número mudar antes do commit, rejeita as alterações. Esta é a estratégia default.
- Lock pessimista: impede acesso a um objeto, geralmente por um determinado tempo e por um determinado modo (leitura, gravação)

5.3.1 Locks otimistas

Uma trava otimista é configurado através de mapeamento e aplicado automaticamente no commit de uma transação.

Para usar um campo de versão é preciso definir um atributo numérico e anotá-lo com `@Version`. Pode ser especificada em um atributo numérico inteiro (Long/long, Integer/int, Short/short) ou Timestamp:

```
@Entity
public class Produto {
    @Version
    long version;
}
```

O provedor de persistência incrementa automaticamente esse campo a cada commit realizado com sucesso (não deve ser manipulado pela aplicação). Se outra transação tenta alterar a entidade e a versão foi alterada desde a última leitura ocorrerá uma `OptimisticLockException`.

Em alguns provedores JPA o lock otimista é automático. Em outros é necessário configurar uma coluna extra nas tabelas (`@Version`) explicitamente.

5.3.2 Locks pessimistas

Se uma colisão precisar ser revelada antes do commit da transação é preciso usar uma trava pessimista. Travas pessimistas obtém exclusividade de acesso a uma entidade durante o período

que uma aplicação estiver usando-a. Apenas uma transação poderá tentar atualizar a entidade por vez. Essa estratégia limita o acesso concorrente aos dados.

Travas pessimistas são criadas e usadas através do EntityManager. Para travar um objeto, use o método `lock()` escolhendo o tipo com `LockModeType`. Há vários tipos. Os principais são `PESSIMISTIC_READ` (compartilhada) e `PESSIMISTIC_WRITE` (exclusiva):

```
em.lock(produto, LockModeType.PESSIMISTIC_WRITE);
```

O método `lock()` requer uma transação ativa (será lançada uma exceção se ela não estiver presente). Poderá haver também uma exceção (`LockTimeoutException`) se a trava não puder ser fornecida (ex: se for uma trava compartilhada e outro usuário tiver uma trava exclusiva, por exemplo).

Definir um timeout é importante para, entre outras coisas, evitar deadlock. A configuração default pode ser feita em `persistence.xml`

```
<properties>
    <property name="javax.persistence.lock.timeout" value="1000"/>
</properties>
```

Mas também pode ser especificada programaticamente via EntityManager ou até mesmo para um query específico. As travas são sempre liberadas ao final da transação.

As travas podem ser usadas durante a recuperação de objetos (métodos `find()`, `refresh()` e `queries`). É possível também sobrepor o timeout default.

```
Map<String, Object> props = new HashMap();
props.put("javax.persistence.lock.timeout", 2000);
Produto produto = em.find(Produto.class, 1, LockModeType.PESSIMISTIC_WRITE, props);
em.refresh(produto, LockModeType.PESSIMISTIC_WRITE, props);
```

O uso de travas pessimistas é raro e tem impacto na escalabilidade. Deve ser usada como último recurso e não como primeira opção para garantir a integridade de um objeto.

5.4 Operações em lote

Operações em lote (batch) são soluções para um problema conhecido como dos “N+1 queries” comuns em aplicações ORM que gera queries desnecessários e tem grande impacto na performance, principalmente em consultas grandes. Analisando o SQL gerado, encontra-se, para queries simples, vários queries extras que não são usados. Exemplo:

```
SELECT E.* FROM EMPLOYEE E WHERE E.STATUS = 'Part-time'
... N selects to ADDRESS
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID = 123
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID = 456
...
... N selects to PHONE
SELECT P.* FROM PHONE P WHERE P.OWNER_ID = 789
SELECT P.* FROM PHONE P WHERE P.OWNER_ID = 135
...
```

Uma das formas de resolver esse problema é através de join fetching (`@JoinFetch`). Outra é usando batch fetching que pode ser configurado com a anotação `@BatchFetch`.

```
@BatchFetch(type=BatchFetchType.EXISTS)
```

As opções são EXISTS, IN e JOIN. O ganho de performance é notável em qualquer uma das opções (costuma ser um pouco pior para IN, mas isso depende do tamanho e tipo de dados envolvidos).

5: Enterprise JavaBeans (EJB)

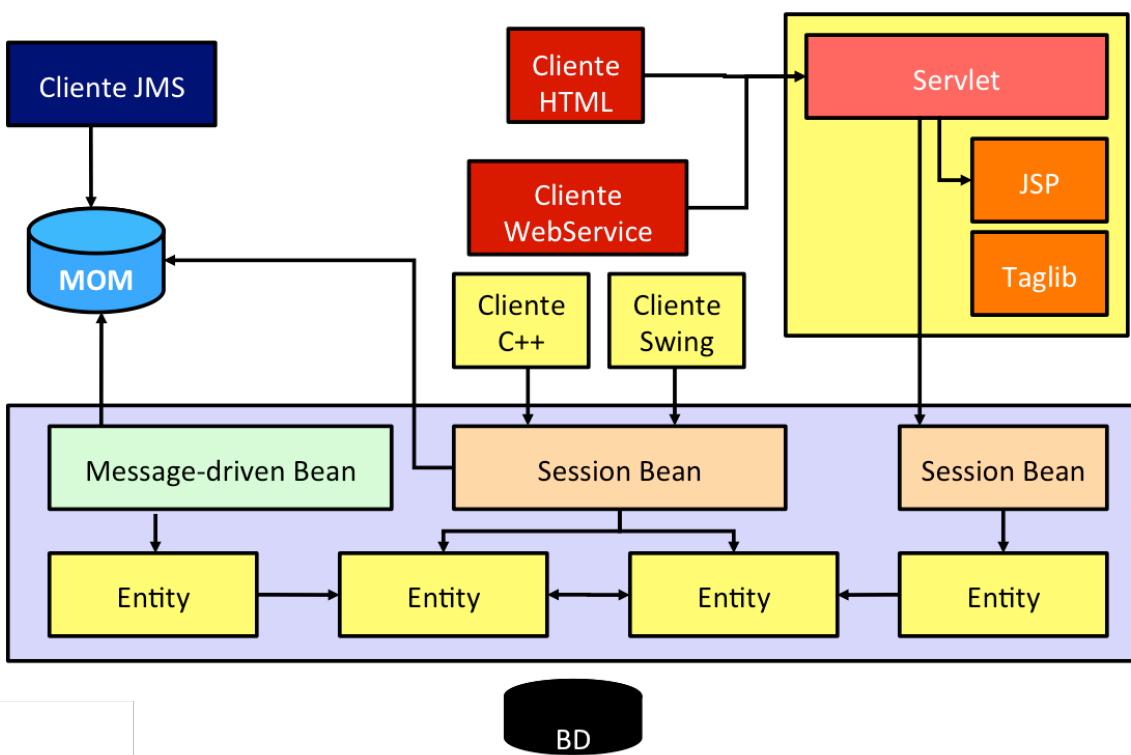
1 Enterprise JavaBeans	2
2 Session Beans	3
2.1 Tipos de session beans	3
2.1.1 Stateful Session Beans	3
2.1.2 Stateless Session Beans	4
2.1.3 Singleton Session Beans	5
2.2 Interfaces e acesso	5
2.2.1 Interfaces para clientes locais	5
2.2.2 Interfaces para clientes remotos	6
3 Message Driven Beans (MDB)	7
3.1 Produtor JMS	8
4 Ciclo de vida	8
4.1 Stateful Session Bean	9
4.2 Stateless Session Bean	9
4.3 Singleton Session Bean	9
4.4 Message-driven Bean	10
4.5 Callbacks	10
5 Acesso via JNDI	11
6 Concorrência, chamadas assíncronas e agendamento	12
6.1 Acesso concorrente em singletons	12
6.1.1 Concorrência gerenciada pelo bean	13
6.1.2 Concorrência gerenciada pelo container	13
6.2 Chamadas assíncronas	14
6.3 Timers	14
6.4 Utilitários de concorrência do Java EE 7	15

7 Transações	16
7.1 Transações gerenciadas pelo bean (BMT)	16
7.1.1 Propagação de transações	16
7.1.2 UserTransaction	17
7.2 Transações gerenciadas pelo container (CMT)	18
7.2.1 Atributos (políticas de propagação)	18
7.2.2 Destino de uma transação em CMT	20
7.3 Transações em Message-driven beans	21
7.4 Sincronização de estado em Stateful Session Beans	21
7.4.1 Interface SessionSynchronization	21
8 Clientes EJB	21

1 Enterprise JavaBeans

Enterprise JavaBeans são componentes que encapsulam a lógica de negócios de uma aplicação. Implementam serviços síncronos e assíncronos, podem exportar uma interface para clientes locais, mas também para clientes remotos usando protocolos de objetos remotos (IIOP) ou Web Services (SOAP). Instâncias de EJBs têm o ciclo de vida controlado em tempo de execução pelo container no qual ele está instalado, que também disponibiliza o acesso a serviços de segurança, transações, agendamento, concorrência, sincronização de estado, distribuição e outros de forma transparente.

O desenho abaixo situa os Enterprise JavaBeans (session beans e message-driven beans) dentre os outros componentes de uma aplicação Java EE



Enterprise JavaBeans são *JavaBeans*, ou POJOs – um objeto Java qualquer contendo um construtor default (sem argumentos), atributos encapsulados (*private*) e acessíveis através de métodos get/set.

Existem dois tipos de EJBs:

- Session beans
- Message-driven beans

2 Session Beans

Session beans são componentes de negócios que exportam uma interface de serviços. Eles representam uma sessão de comunicação com um cliente (que não necessariamente corresponde a uma sessão Web).

Um session bean deve ser criado quando houver interesse em fornecer uma *interface de serviços*. Esta interface pode ser local (acessível dentro da mesma aplicação) ou remota (acessível de outras aplicações ou mesmo de outras máquinas). É possível ainda exportar uma interface para clientes SOAP. A interface é definida por uma coleção de métodos.

Existem diversas formas de declarar uma interface para um session bean:

- *Interface local* usando *no-interface view*: o bean é acessível localmente, apenas para clientes da mesma aplicação. A interface de serviços consiste de todos os métodos do bean (inclusive os herdados de suas superclasses).
- *Interface local* declarada usando *@Local*: o bean implementa uma ou mais interfaces Java que contém todos os métodos de sua interface de serviços que podem ser acessadas localmente por clientes da mesma aplicação.
- *Interface remota* (para clientes IIOP) declarada usando *@Remote*: o bean implementa uma ou mais interfaces que contém todos os métodos de sua interface de serviços que acessíveis por clientes locais e clientes em outras aplicações, servidores ou máquinas.
- *Interface remota* (para clientes SOAP) declarada usando *@WebService*: o bean implementa uma interface Java que declara os métodos exportados ou marca-os com *@WebMethod* no próprio bean.

2.1 Tipos de session beans

Existem três tipos de session beans:

- *Stateful Session Beans* – usado para comunicação que precisa preservar o estado do cliente entre chamadas de métodos do bean.
- *Stateless Session Beans* – usado para comunicação que não precisa manter o estado do cliente.
- *Singleton Session Beans* – usado para comunicação que precisa compartilhar o estado entre diversos clientes.

2.1.1 Stateful Session Beans

Stateful session beans modelam diálogos que consistem de uma ou mais requisições, onde certas requisições podem depender do estado de requisições anteriores. Esse estado é guardado nas variáveis de instância do bean.

A forma mais simples de criar um stateful session bean é usando o estilo *no-interface view*, anotando a classe com *@Stateful*:

```
@Stateful
public class CestaDeCompras { ... }
```

Isto irá exportar uma interface local que consiste de todos os métodos de instância da classe (inclusive os que foram herdados de superclasses). Qualquer outro componente da aplicação que injetar o bean poderá chamar qualquer um desses métodos.

Como em stateful session beans o estado do cliente é mantido por varias chamadas, poderá chegar o momento em que o cliente decida que o diálogo foi finalizado. Ele poderá então desejar remover o bean e encerrar sua sessão. Para isto é necessário anotar um método da interface com `@Remove`:

```
@Remove public void remover() {}
```

O método não precisa conter nenhuma instrução. Se o bean também participa de um escopo CDI, ele só poderá ser removido pelo cliente se o seu escopo for `@Dependent` (default). Caso ele tenha outro escopo, ele deve deixar que o container remova o bean quando o escopo terminar.

O container removerá o bean depois que o método anotado com `@Remove` terminar.

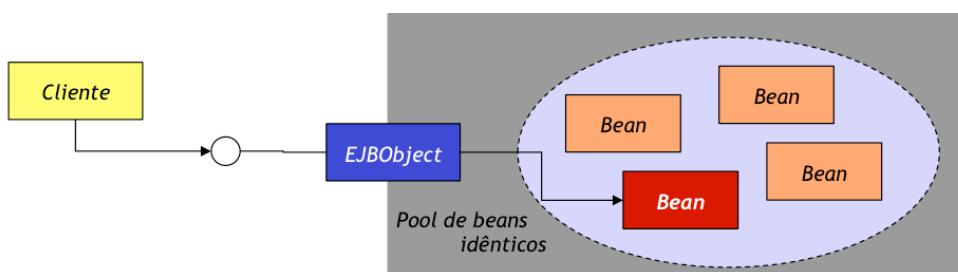
Stateful session beans mantém estado para cada cliente, portanto o container não pode fazer o mesmo tipo de pooling que faz com stateless session beans. Os beans não são considerados equivalentes. Se há mais clientes que beans no pool, mas grande parte permanece inativa por certos períodos, o container usa um mecanismo de ativação e passivação para otimizar o uso de recursos. Ele armazena o estado (não transiente) de beans menos acessados liberando a instância para reuso por um novo cliente. Se o cliente antigo faz uma nova chamada no bean passivo, ele é automaticamente ativado. Essa estratégia aumenta a capacidade de atender clientes.

Por exemplo, se num pool de 5 beans, todos estão ativos e em comunicação com seus clientes, aparecer um sexto cliente, o estado do bean que foi usado há mais tempo é serializado e gravado em meio persistente. Se houver um método anotado com `@PrePassivate` ele é chamado antes da passivação.

Se esse cliente está continuando um diálogo iniciado anteriormente, o estado anterior do bean que ele usou é recuperado. Se houver um método anotado como `@PostActivate` ele é chamado após a ativação.

2.1.2 Stateless Session Beans

Stateless session beans modelam diálogos que consistem de apenas uma requisição. Como não mantém estado do diálogo, todas as instâncias da mesma classe são *equivalentes e indistinguíveis*. Qualquer instância disponível de um session bean pode servir a qualquer cliente. Session Beans podem ser guardados em pool, reutilizados e passados de um cliente para outro em cada chamada.



A forma mais simples de criar um stateless session bean é usando o estilo *no-interface view*, anotando a classe com `@Stateless`:

```
@Stateless
public class VitrineVirtual { ... }
```

2.1.3 Singleton Session Beans

Singleton session beans foram projetados para compartilhar informação entre beans. São instanciados uma única vez por aplicação, e têm escopo de aplicação. Por terem estado compartilhado, precisam lidar com questões relacionadas ao acesso concorrente. Só pode haver uma instância do bean por aplicação (mas se uma aplicação for distribuída por várias máquinas virtuais, haverá uma instância em cada JVM).

A forma mais simples de criar um stateful session bean é usando o estilo *no-interface view*, anotando a classe com `@Singleton`:

```
@Singleton
public class ServicosGlobais { ... }
```

É comum instruir o container a carregar o bean durante a inicialização da aplicação anotando-o com `@Startup`:

```
@Startup
@Singleton
public class ServicosGlobais { ... }
```

Às vezes um singleton depende da inicialização prévia de outros singletons. Isto pode ser configurado usando `@DependsOn`:

```
@DependsOn("ServicosGlobais")
@Singleton
public class EstatisticasDeAcesso { ... }
```

O acesso concorrente a um Singleton session bean é controlado pelo container (default) e todos os seus métodos tem (por default) acesso exclusivo. Isto significa que apenas um cliente de cada vez pode ter acesso ao método. Esse comportamento pode ser alterado para maior eficiência (ex: permitir que métodos que não alteram o estado sejam acessados simultaneamente.)

2.2 Interfaces e acesso

2.2.1 Interfaces para clientes locais

A forma mais simples de exportar a interface de um SessioBean é usar a configuração default (*no-interface view*), que considera *todos* os métodos do bean (inclusive os herdados) como parte da interface.

A forma mais simples (e recomendada) de obter acesso a este bean em uma classe cliente da mesma aplicação é usando injeção de dependências (ou CDI):

```
@EJB
BibliotecaBean bean;
```

A estratégia no-interface view expõe *todos os métodos*, o que pode não ser desejável se a classe contém métodos que não devem ser expostos. Pode-se ter mais controle criando uma interface Java anotada com `@Local` e contendo apenas os métodos que deverão ser exportados.

```
@Local
public interface Biblioteca {
    void emprestar(Livro livro);
    void devolver(Livro livro);
}
```

O bean só precisa implementar a interface:

```
@Stateless
public class BibliotecaBean implements Biblioteca { ... }
```

Os clientes de um bean `@Local` devem pertencer à mesma aplicação e podem usar CDI ou injeção de dependências (com `@EJB`) para injetar a *interface* (não o bean):

```

@SessionScoped
public class Administrador {
    @EJB Biblioteca biblioteca;

    public void recebidos(List<Livro>) {
        for(Livro m : livros) {
            biblioteca.devolver(m);
        }
    }
    ...
}

```

Ou fazer o lookup JNDI (também pela *interface*):

```

@SessionScoped
public class Administrador {
    Biblioteca biblioteca;

    @PostConstruct
    public void init() {
        Context ctx = new InicialContext();
        biblioteca = (Biblioteca)ctx.lookup("java:app/Biblioteca");
    }
    ...
}

```

Se a interface usada não puder ser anotada como `@Local`, ainda é possível declará-la como tal se ela for implementada pelo bean e passada como parâmetro de uma anotação `@Local` na classe do bean:

```

@Local(Biblioteca.class)
public class BibliotecaBean implements Biblioteca {}

```

2.2.2 Interfaces para clientes remotos

Clientes remotos precisam ser declarados através de uma interface de serviços. Se a interface de serviços for anotada com `@Remote`, o bean só precisa implementá-la.

```

@Remote
public interface BibliotecaRemote {
    void consultar(Livro livro);
    void reservar(Livro livro);
}

@Stateless
public class BibliotecaBean implements BibliotecaRemote { ... }

```

Se a interface *não* for anotada com `@Remote`, o bean ainda pode usá-la passando seu nome como argumento para `@Remote`:

```

@Remote(BibliotecaRemote.class)
public class BibliotecaBean implements BibliotecaRemote, OutraInterface {}

```

Dentro de um container EJB é possível obter a referência para um bean remoto via injeção de dependências (chamando pela interface, sempre):

```

@EJB BibliotecaRemote biblioteca;

```

Dependendo da configuração e do servidor, às vezes é possível usar injeção de dependências em containers localizados em máquinas diferentes, mas nesses casos o mais garantido é usar JNDI:

```

Context ctx = new InicialContext();
BibliotecaRemote biblioteca =
    (BibliotecaRemote)ctx.lookup("java:global/libraryapp/Biblioteca");

```

Em clientes RMI/IOP que não estão usando um container ainda é possível acesso a um EJB via acesso ao JNDI global proprietário do servidor. Em alguns servidores (ex: WebSphere 7)

o proxy obtido via JNDI é um objeto CORBA que precisa ser convertido. Nesse caso, é recomendado usar o método `PortableRemoteObject.narrow()` para converter o proxy do objeto:

```
java.lang.Object stub = ctx.lookup("java:app/BibliotecaRemote ");
BibliotecaRemote biblioteca = (BibliotecaRemote)
    javax.rmi.PortableRemoteObject.narrow(stub, BibliotecaRemote.class);
```

Chamadas remotas tem uma natureza diferente de chamadas locais. Os parâmetros dos métodos e os tipos de retorno são sempre passados por valor (são proxies de rede), portanto não é possível fazer operações que alteram propriedades de um bean remoto por referência (é necessário alterar o bean localmente e depois sincronizar).

Proxies de beans remotos e seus parâmetros precisam ser transferidos pela rede. É geralmente mais eficiente transferir objetos maiores em poucas chamadas, portanto interfaces remotas devem usar o padrão Data-Transfer Objects (DTOs) e transferir objetos de baixa granularidade, para reduzir o número de chamadas.

3 Message Driven Beans (MDB)

Message-driven beans (MDB) são processadores assíncronos. São usados para modelar eventos, notificações e outras tarefas assíncronas. Um MDB não possui uma interface e não pode ser chamado diretamente por um cliente. Mas um MDB pode agir como cliente e chamar outros beans, iniciar contextos transacionais e injetar e usar serviços disponíveis à aplicação.

MDBs são listeners de mensageria. São geralmente usados para processar mensagens JMS e implementam a interface `MessageListener`. São anotados com `@MessageDriven` informando o nome da fila à qual o bean está registrado como listener. A menos que seja configurado com um filtro de mensagens, todas as mensagens recebidas na fila serão recebidas pelo método `onMessage()` do bean, que poderá processá-las.

```
@MessageDriven(mappedName="pagamentos")
public class ProcessadorCartaoDeCredito implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            // processa a mensagem
        } catch (JMSException ex) { ... }
    }
}
```

O atributo `mappedName` é uma forma *portável* de informar o nome da fila, mas de acordo com a especificação é de *implementação opcional*. Funciona na implementação de referência (Glassfish) mas pode não funcionar em outros servidores. Nesses casos a fila precisa ser informada usando propriedades de ativação (que podem variar entre fabricantes).

A propriedade `activationConfig` de `@MessageListener` recebe uma lista de propriedades de configuração em anotações `@ActivationConfigProperty`. No WildFly/JBoss e WebSphere é obrigatório usar a propriedade `destination`, que informa a fila ao qual o bean deve ser registrado:

```
@MessageDriven(
    activationConfig={
        @ActivationConfigProperty(propertyName="destination",
                               propertyValue="pagamentos")
    }
)
public class ProcessadorCartaoDeCredito implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            // processa a mensagem
        } catch (JMSException ex) { ... }
    }
}
```

Existem outras propriedades que podem ser configuradas. Uma delas é *messageSelector* que permite filtrar as mensagens que serão recebidas pelo bean. O filtro atua sobre os cabeçalhos e propriedades da mensagem. Cabeçalhos são gerados automaticamente pelo sistema, e incluem dados como data de validade, id da mensagem, etc. Propriedades são definidas pelo remetente da mensagem, que pode gravar uma propriedade em uma mensagem JMS usando:

```
mensagem.setIntProperty("quantidade", 10);
```

O MDB pode, por exemplo, ser configurado para não receber mensagens que não tem essa propriedade ou que tem com valor menor, usando:

```
@MessageDriven(
    activationConfig={ ...,
        @ActivationConfigProperty(
            propertyName="messageSelector",
            propertyValue="quantidade is not null and quantidade >= 10")}
)
public class ProcessadorCartaoDeCredito implements MessageListener {...}
```

3.1 Produtor JMS

Produtores de mensagens são clientes que enviam mensagens a um destino. Um produtor JMS pode ser qualquer componente Java EE ou até mesmo aplicações cliente standalone que tenham como obter acesso a conexões e destinos no servidor. Se estiver dentro do container, o produtor pode injetar a conexão e destino (fila) para onde enviará mensagens:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

De clientes externos, ele pode obter proxies para esses objetos através de JNDI:

```
Context ctx = new InitialContext();
ConnectionFactory factory = (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");
Queue queue = (Queue) ctx.lookup("jms/Queue");
```

MDBs também podem injetar Session Beans usando @EJB, outros serviços usando @Resource, e qualquer objeto ou serviço injetável (usando @Inject) via CDI.

Obtidos os objetos, mensagens podem ser enviadas usando os métodos do JMS:

```
JMSContext ctx = connectionFactory.createContext();
context.createProducer().send(queue, "Texto contido no corpo da mensagem");
```

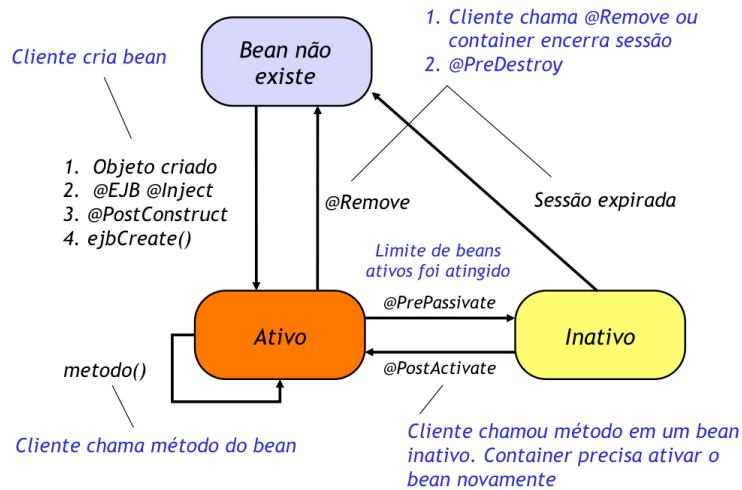
Usando CDI é possível injetar o contexto JMS em vez de obter um ConnectionFactory:

```
@Inject @JMSConnectionFactory("jms/ConnectionFactory")
private JMSContext ctx2;
```

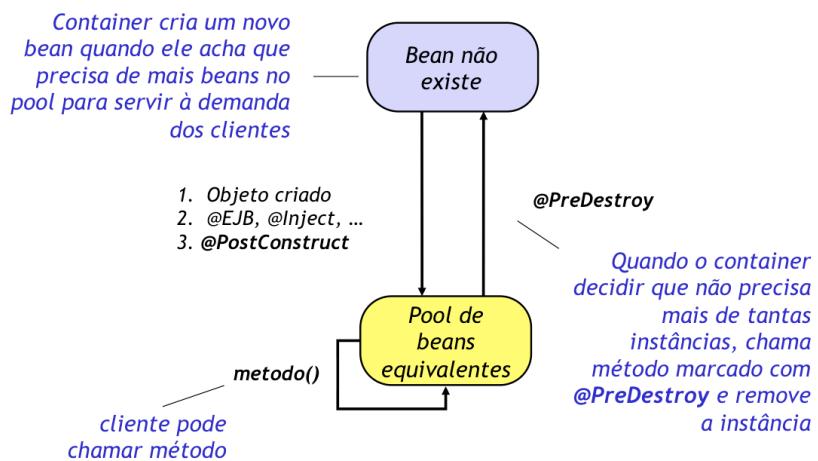
4 Ciclo de vida

Os diagramas abaixo ilustram os ciclos de vida de cada tipo de Enterprise JavaBean.

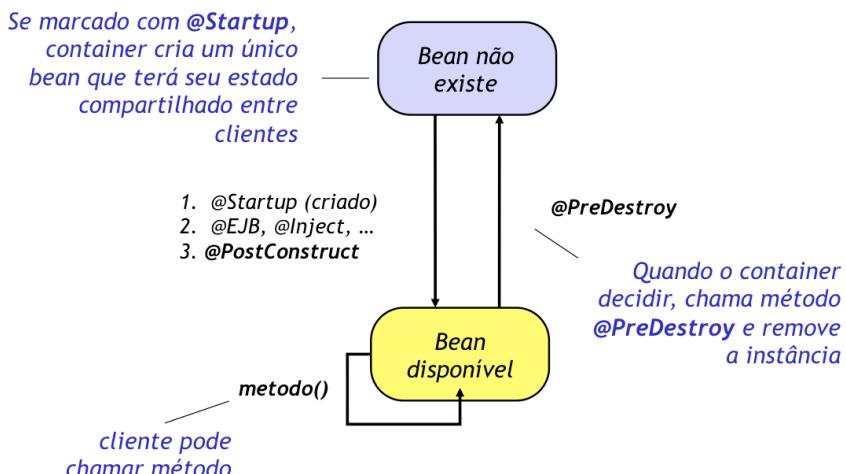
4.1 Stateful Session Bean



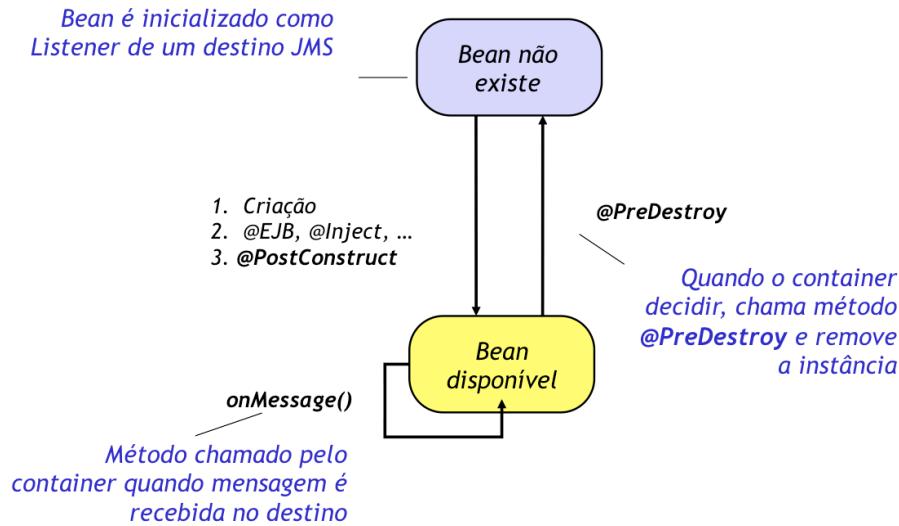
4.2 Stateless Session Bean



4.3 Singleton Session Bean



4.4 Message-driven Bean



4.5 Callbacks

Existem cinco anotações EJB para marcar métodos de callback, que são chamados automaticamente em eventos do ciclo de vida de um bean. Quatro delas são usadas em session beans:

@PostConstruct é usada para anotar um método que deve ser chamado após a execução do construtor default do bean e injeção das dependências, e antes que qualquer método da interface do bean seja chamado:

```
@Stateless
public class Biblioteca {

    @Inject ServicoIsbn servico;

    @PostConstruct
    private void inicializar() {
        servico.login();
    }
    ...
}
```

@PreDestroy é usada para anotar um método que deve ser chamado antes do bean ser removido. O bean pode ser removido explicitamente pelo cliente (ex: através de um método anotado com **@Remove** em stateful session beans) ou pelo container (a qualquer momento para stateless beans, e na finalização da aplicação para singletons).

```
@Stateless
public class Biblioteca {
    ...
    @PreDestroy
    private void finalizar() {
        servico.logout();
    }
    ...
}
```

@PrePassivate é usada apenas em stateful session beans e chamado antes que o bean entre no estado passivo.

```

@Stateful
public class CestaDeCompras implements Serializable {
    private transient String codigoAcesso;
    @Inject private Cliente cliente;

    @PrePassivate
    private void passivar(InvocationContext ctx) {
        cliente.liberarCodigoAcesso();
    }
    ...
}

```

@PostActivate é usado apenas em stateful session beans e chamado depois que o bean é reativado do seu estado passivo.

```

@Stateful
public class CestaDeCompras implements Serializable {
    private transient String codigoAcesso;
    @Inject private Cliente cliente;

    @PostActivate
    private void ativar(InvocationContext ctx) {
        codigoAcesso = cliente.obterNovoCodigoAcesso();
    }
    ...
}

```

@PostConstruct e **@PreDestroy** também podem ser usados em *MessageDrivenBeans*. A quinta anotação, **@AroundConstruct**, é rara e usada apenas em interceptadores (tratados em outro capítulo). Métodos marcados com **@PrePassivate** ou **@PostActivate** em beans que não são stateful session beans são ignorados.

5 Acesso via JNDI

Componentes geralmente são injetados pelos clientes que os acessam de dentro de uma aplicação. Mas, se for necessário, eles também podem ser carregados diretamente através de JNDI. O acesso JNDI é necessário para acesso a componentes remotos entre aplicações. Servidores diferentes podem estipular formas diferentes para acessar serviços via JNDI, mas a especificação Java EE define três formas padrão de acesso JNDI que podem ser usadas independente do servidor: *global*, no contexto de uma *aplicação* (se o bean estiver em modulo empacotado em um EAR), e no contexto de um *módulo*.

Um nome JNDI global tem a forma mínima:

```
java:global/aplicacao/bean
```

para beans armazenados em EJB JARs ou WARs. A aplicação é (por default) o nome do arquivo sem a extensão (aplicacao.war ou aplicacao.jar)

Se o WAR ou EJB JAR estiver dentro de um arquivo EAR, ele é considerado um módulo da aplicação que é representada pelo EAR. Nesse caso, é preciso incluir o EAR no caminho global:

```
java:global/aplicacao/modulo/bean
```

Onde o nome da aplicação (geralmente) é o arquivo EAR sem a extensão (ex: aplicacao.ear) e os nomes dos módulos, se não forem o nome do arquivo (default), estão definidos no /META-INF/application.xml do EAR.

Se o bean tiver mais de uma interface e houver necessidade de identificar especificamente uma delas, o nome (qualificado) da interface pode ser incluído depois do nome do bean, separado por uma exclamação:

```
java:global/aplicacao/modulo/bean!pacote.Interface
```

Se o acesso ao bean ocorrer entre módulos instaladas na mesma aplicação (EAR), pode-se usar o acesso no escopo do container com prefixo java:app:

```
java:app/modulo/bean
```

Se a comunicação ocorrer no contexto do módulo, ou se a aplicação não estiver empacotada em um EAR, pode-se usar qualquer uma das duas formas:

```
java:app/bean
java:module/bean
```

Por exemplo, se um bean @Remote chamado LivroBean é empacotado em um WAR biblioteca.war. Ele pode ser chamado localmente (ex: por um managed bean ou servlet no mesmo WAR) usando:

```
public class LivroClientServlet extends HttpServlet {
    public void init() {
        try {
            Context ctx = new InitialContext(); // inicialização do JNDI
            LivroBean bean = (LivroBean)ctx.lookup("java:app/LivroBean");
            bean.inicializar();
        ...
    }
}
```

E por um cliente remoto usando:

```
public class LivroRemoteTest extends TestCase {
    @Test
    public void testLookup() {
        Context ctx = new InitialContext();
        LivroBean bean = (LivroBean)ctx.lookup("java:global/biblioteca/LivroBean");
        ...
    }
}
```

Mas para o primeiro caso, que executa dentro do container, talvez seja mais prático simplesmente injetá-lo usando @EJB (ou @Inject, se configurado com CDI):

```
public class LivroClientServlet extends HttpServlet {
    @EJB LivroBean bean;
    public void init() {
        try {
            bean.inicializar();
        ...
    }
}
```

6 Concorrência, chamadas assíncronas e agendamento

A especificação EJB explicitamente proíbe beans de criar threads (além de não recomendar que EJBs não abram sockets ou streams de I/O). Embora a proibição esteja apenas na especificação (e não seja verificada pelo container), é uma má idéia criar threads porque eles podem interferir com o controle do ciclo de vida dos componentes realizado pelo container. Também não é recomendado o uso de travas e sincronização, que podem interferir no controle de transações e causar deadlocks.

Mas existem várias aplicações de threads, como notificações e agendamento que podem ser realizadas em EJB usando recursos como MDB, timers e métodos assíncronos. A sincronização de dados compartilhados é automática em singletons, mas existem várias opções de configuração.

6.1 Acesso concorrente em singletons

Por default, todo singleton bean tem controle de acesso concorrente, que impede acesso simultâneo a seus métodos. Esse acesso pode ser configurado através da anotação

`@ConcurrencyManagement` a nível de classe e com travas de leitura/gravação em cada método usando a anotação `@Lock`.

6.1.1 Concorrência gerenciada pelo bean

Por default, a gerência de concorrência é feita pelo container. Mas anotando-se a classe do Singleton com:

```
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
```

os métodos do singleton não serão mais thread-safe e o controle de concorrência ficará a cargo do bean, que pode usar modificadores da linguagem Java como `synchronized` ou `volatile`, ou travas do pacote `java.util.concurrent.locks`, como `Condition`, `Lock` e `ReadWriteLock`.

É uma solução flexível, porém mais complexa e deve ser usada raramente.

6.1.2 Concorrência gerenciada pelo container

Singleton beans anotados com

```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
```

têm a concorrência gerenciada pelo container. Essa anotação não é necessária porque este é o default.

O container permite configurar o acesso concorrente de cada método com a anotação `@Lock`, que recebe como parâmetro um dos valores do enum `LockType`: `READ` ou `WRITE`. A opção `READ` permite acesso simultâneo, e com `WRITE` o acesso é exclusivo. Por default, todos os métodos de um Singleton se comportam como se estivessem anotados com `@Lock(LockType.WRITE)`.

Impedir o acesso simultâneo a todos os métodos pode introduzir um gargalo de performance em um singleton que seja acessado com muito mais frequência para ler (e não alterar) seu estado. Portanto, é uma boa prática anotar métodos que não alteram o estado do singleton com `@Lock(LockType.READ)`:

```
@Singleton
public class ExampleSingletonBean {

    private String state;

    @Lock(LockType.READ)
    public String getState() {
        return state;
    }

    @Lock(LockType.WRITE)
    public void setState(String newState) {
        state = newState;
    }
}
```

Se um método é marcado com `WRITE`, apenas um cliente poderá acessá-lo de cada vez. Os outros precisarão esperar o método terminar. Pode-se estabelecer um timeout para isto usando a anotação `@AccessTimeout` (na classe, para todos os métodos, ou em cada método individualmente) informando quantos milissegundos o cliente pode esperar. Quando o timeout acabar, o container lança uma exceção (`ConcurrentAccessTimeoutException`):

```
@Singleton
@AccessTimeout(value=60000)
public class MySingletonBean {
    ...
}
```

6.2 Chamadas assíncronas

Uma das maneiras de executar uma operação assíncrona usando session beans é implementá-lo como cliente JMS. O método deve criar uma mensagem, empacotar os dados necessários para que a operação assíncrona seja executada, e enviar a mensagem para uma fila JMS mapeada por um MDB. Ao receber a mensagem, o MDB desempacota os parâmetros e chama a operação.

Embora esta possa ser uma solução eficiente, ela requer a criação de bastante código extra, além da necessidade de empacotar e desempacotar dados em uma mensagem. Uma outra forma de realizar operações assíncronas é usando métodos assíncronos.

Métodos assíncronos podem ser usados em qualquer session bean e precisam ou retornar void ou Future<V>. A primeira solução já é uma substituição adequada para a solução com MDB, que também não retorna valor. A segunda utiliza um objeto Future, do pacote *java.util.concurrent*, que permite monitorar chamadas assíncronas. É preciso também anotar o método com @Asynchronous (que também pode ser usado na classe, se todos os métodos da classe forem assíncronos).

O exemplo abaixo ilustra um bean contendo um método assíncrono:

```
@Stateless
public class Tradutor {
    @Asynchronous
    public Future<String> traduzir(String texto) {
        String textotraduzido;
        // ... realiza a tradução (demorada)
        return new AsyncResult<String>(textotraduzido.toString());
    }
}
```

O cliente pode chamar o método e receber de volta um objeto Future. O resultado pode ser extraído quando estiver pronto usando o método get(). O método isDone() pode ser usado para monitorar o estado do objeto e descobrir quando o resultado pode ser extraído:

```
@SessionScoped @Named
public class ManagedBean {
    @EJB
    private Tradutor tradutor;

    private Future<String> future;

    public void solicitarTraducao(String texto) {
        future = tradutor.traduzir(texto);
    }
    public String receberTraducao() {
        while (!future.isDone()){
            Thread.sleep(1000);
            // fazer outras coisas
        }
        return (String)future.get();
    }
}
```

6.3 Timers

Timers são serviços de agendamento que podem ser configurados em stateless e singleton session beans. Permitem que métodos sejam chamados periodicamente, ou uma única vez em data e hora marcada, ou depois de um intervalo especificado. Timers automáticos são configurados anotando métodos com @Schedule (ou @Schedules para múltiplos timers), que

recebe como parâmetros as regras de agendamento. Os parâmetros são *hour*, *minute*, *second*, *dayOfWeek*, *dayOfMonth*, *timezone*, *year*, que podem receber expressões (para timers periódicos) ou valores fixos (para timers que executam apenas uma vez):

Exemplo

```
@Stateless
public class MyTimer {
    @Schedule(hour="*", minute="*", second="*/10")
    public void printTime() { ... }
    ...
}
```

A seguir alguns exemplos de expressões:

- hour="7,19,23", dayOfWeek="1-5" – executa as 7, 19 e 23 horas, de segunda a sexta
- minute="30", hour="5" - executa uma vez às 5 horas e 30 minutos
- hour="*", minute="*", second="*/10" - toda hora e minuto a cada 10 segundos

Timers também podem ser criados programaticamente usando os métodos de *TimerService* em beans que implementam a interface *TimedObject* e seu método *ejbTimeout(Timer)*. Neste caso o timer geralmente é configurado na inicialização do bean:

```
@Singleton
@Startup
public class MyTimer implements TimedObject {

    @Resource TimerService timerService;

    @PostConstruct
    public void initTimer() {
        if (timerService.getTimers() != null) {
            for (Timer timer : timerService.getTimers()) {
                timer.cancel();
            }
        }
        timerService.createCalendarTimer(
            new ScheduleExpression().hour("*").minute("*").second("*/10"),
            new TimerConfig("myTimer", true)
        );
    }

    @Override
    public void ejbTimeout(Timer timer) {
        //código a executar quando o timer disparar
    }
}
```

Além do *createCalendarTimer()*, que cria um Timer para agendamento periódico, o *TimerService* também permite criar timers que disparam uma única vez com *createSingleActionTimer()*.

Se o bean não implementa a interface *TimedObject*, ele também pode definir um timeout usando um método anotado com *@Timeout*. O método precisa retornar void, e não precisa receber parâmetros (se receber deve ser um objeto Timer):

```
@Timeout
public void timeout(Timer timer) {
    //código a executar quando timer disparar
}
```

6.4 Utilitários de concorrência do Java EE 7

Existem vários utilitários de concorrência que devem ser usados caso haja a necessidade de trabalhar com threads dentro de aplicações Java EE. Elas estão declaradas no pacote

`javax.enterprise.concurrent` e são similares a classes do pacote Java SE `java.util.concurrent`, porém gerenciadas pelo container e adequadas ao uso em EJBs e servlets. As mais importantes são:

- `ManagedExecutorService`
- `ManagedScheduledExecutorService`
- `ManagedThreadFactory`
- `ContextFactory`

7 Transações

Transações representam operações atômicas, indivisíveis. Um mecanismo de transações visa garantir que um procedimento ou termine com sucesso ou que todas as suas etapas sejam completamente desfeitas.



A especificação EJB não suporta transações aninhadas. Se uma transação começa quando já existe um contexto transacional, ela pode:

- Continuar o contexto transacional existente
- Interromper o contexto transacional existente
- Iniciar um novo contexto transacional

O controle de transações em EJB resume-se a *demarcação de transações*, ou seja, determinar quando ela será *iniciada* e quando será *concluída* (ou desfeita).

Estão disponíveis duas maneiras de demarcar transações:

- Explícita, ou programática (Bean-Managed Transactions – BMT)
- Implícita, ou declarativa (Container-Managed Transactions- CMT) - default

7.1 Transações gerenciadas pelo bean (BMT)

Transações gerenciadas pelo bean (BMT) envolvem o controle de transações através do uso direto de APIs para demarcação de transações em Java. Isto inclui métodos de controle transacional de `java.sql.Connection` (JDBC), `javax.jms.Session` (JMS), `EntityTransaction` em JPA e `UserTransaction` em JTA. Esses métodos são proibidos em CMT.

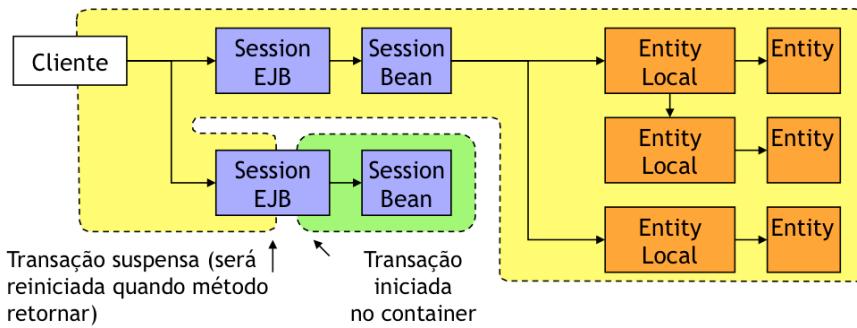
Para permitir que um bean utilize qualquer uma dessas APIs, é necessário configurá-lo usando: `@TransactionManagement`:

```

@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Fachada { ... }
  
```

7.1.1 Propagação de transações

Transações terminam no mesmo lugar onde começaram. O contexto da transação será propagado para todos os métodos chamados (se eles não iniciarem nova transação). Se métodos chamados iniciarem nova transação, a transação principal será suspensa até que o método termine.



O bean abaixo usa transações do JDBC, portanto precisa ser declarado como BMT:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Bean {

    @Resource DataSource ds;

    public void processarPedido (String id, int quantidade) {
        Connection con = ds.getConnection();
        try {
            con.setAutoCommit(false);
            atualizarEstoque(id, quantidade);
            con.commit();
        } catch (Exception e) {
            try {
                con.rollback();
            } catch (SQLException sqx) {}
        }
    }
}
```

7.1.2 UserTransaction

Containers JavaEE implementam a interface `javax.transaction.UserTransaction`, que pode ser usada em qualquer componente EJB, servlets e clientes standalone para demarcar transações em um container. As transações são distribuídas e usam a técnica two-phase commit. `UserTransaction` é usada transparentemente em transações CMT, mas também pode ser usada explicitamente em transações BMT.

Os principais métodos de `UserTransaction` usados para demarcar transações via BMT são:

- `begin()`: marca o início
- `commit()`: marca o término
- `rollback()`: condene a transação

O exemplo abaixo usa JTA para controlar uma transação:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Bean {

    @Resource EJBContext ctx;
    private double saldo;

    public void sacar(double quantia) {
        UserTransaction ut = ctx.getUserTransaction();
        try {
            double temp = saldo;
            ut.begin();
            saldo -= quantia;
            atualizar(saldo); // afetado pela transação
        } catch (Exception e) {
            ut.rollback();
        }
    }
}
```

```
        ut.commit();
    } catch (Exception ex) {
        try {
            ut.rollback();
            saldo = temp;
        } catch (SystemException e2) {}
    }
}
```

Clientes remotos podem obter uma referência para *UserTransaction* através de JNDI. O container deve disponibilizar o JTA na localidade `java:comp/UserTransaction`.

7.2 Transações gerenciadas pelo container (CMT)

O uso de Container-Managed Transactions (CMT) garante um controle de transações totalmente gerenciado pelo container. É mais simples de usar que BMT, e recomendado e default (mas pode ser especificado explicitamente via anotações ou configuração XML):

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER)  
public class Fachada {...}
```

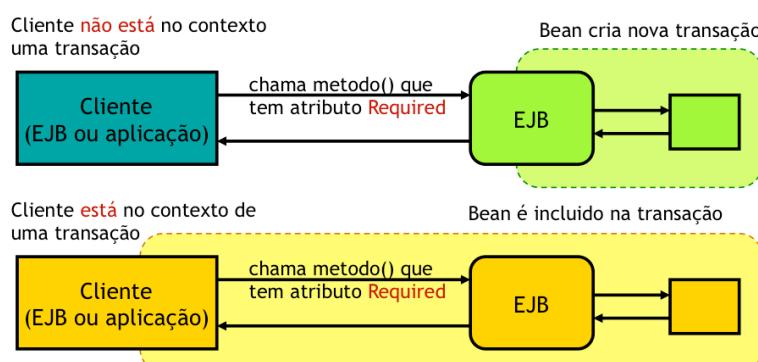
Em CMT não é possível demarcar blocos avulsos de código. A granularidade mínima é o método. As transações são gerenciadas por default de acordo com a política REQUIRED, que garante que o método estará sempre dentro de um contexto transacional (continua contexto existente, ou cria um novo). CMT não permite o uso de nenhuma API transacional dentro do código (isto inclui métodos da API de transações de `java.sql`, `javax.jms`, `EntityTransaction` e `UserTransaction`).

7.2.1 Atributos (políticas de propagação)

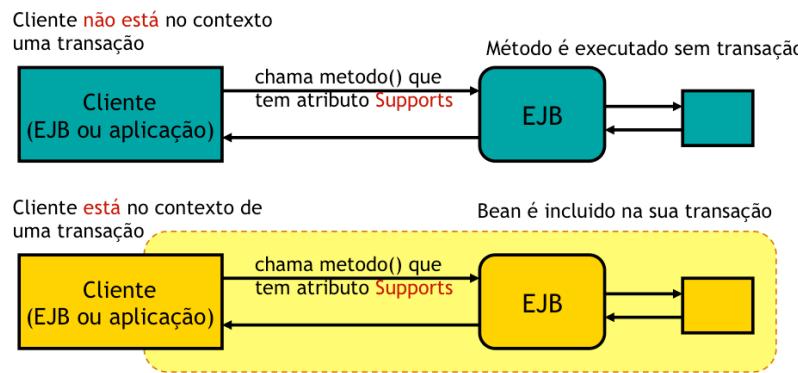
As políticas transacionais determinam como cada método irá reagir quando for chamado por um cliente dentro de um contexto transacional existente ou não. Pode-se definir uma política default para toda a classe, ou especificar individualmente em cada método com `@TransactionAttribute` e um dos tipos definidos no enum `TransactionAttributeType`:

- MANDATORY
 - REQUIRED
 - REQUIRES_NEW
 - SUPPORTS
 - NOT_SUPPORTED
 - NEVER

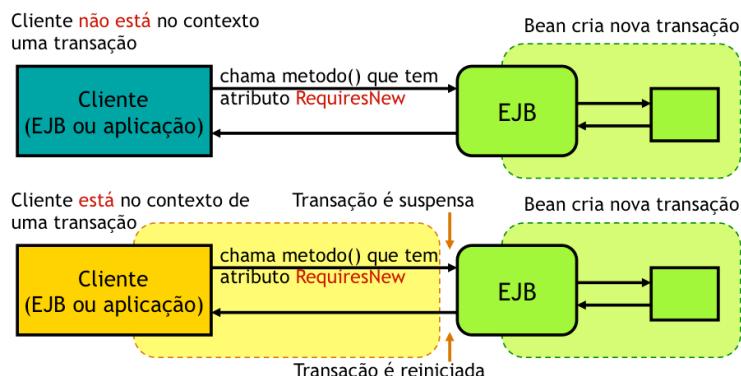
TransactionAttributeType.REQUIRED - Indica que o método precisa ser chamado dentro do escopo de uma transação. Se não existe transação, uma transação nova é criada e dura até que o método termine (é propagada para toda a cadeia de métodos chamados). Se já existe uma transação iniciada pelo cliente, o bean é incluído no seu escopo durante a chamada do método.



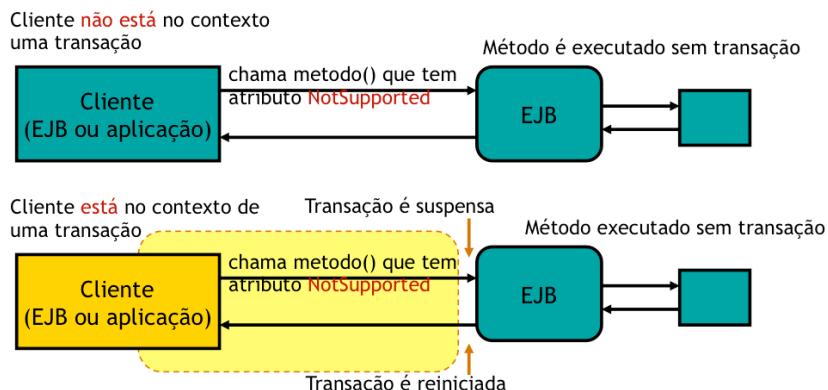
TransactionAttributeType.SUPPORTS - Indica que o método suporta transações, ou seja, ele será incluído no escopo da transação iniciada pelo cliente se ela existir. Se ele for chamado fora do escopo de uma transação ele realizará suas tarefas sem criar transações e poderá chamar objetos suportam ou não transações.



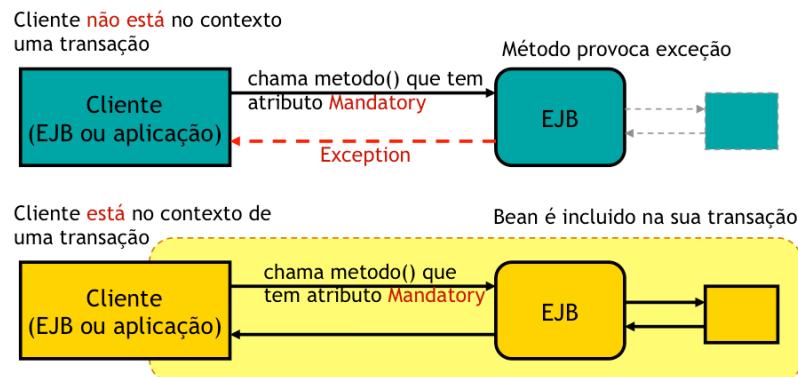
TransactionAttributeType.REQUIRES_NEW - Indica que uma nova transação, iniciada no escopo do bean, sempre será criada, estando ou não o cliente no escopo de uma transação. Se o cliente já tiver iniciado um contexto transacional, este será suspenso até que a nova transação iniciada no método termine (ao final do método).



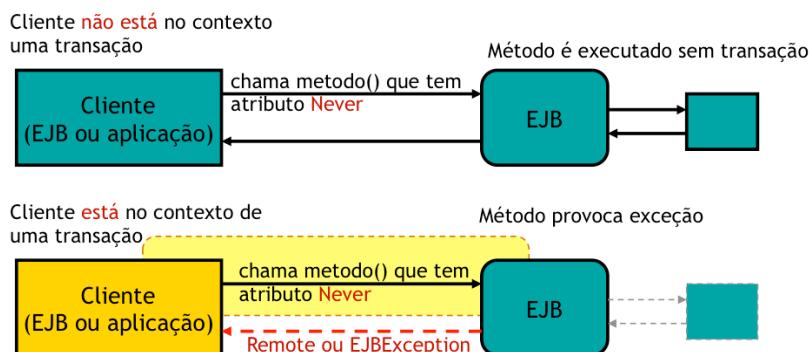
TransactionAttributeType.NOT_SUPPORTED - Indica que o método não suporta transações. Se o método for chamado pelo cliente no escopo de uma transação, ela será suspensa enquanto durar a chamada do método e não será propagada. A transação do cliente será retomada quando o método terminar.



TransactionAttributeType.MANDATORY - Indica que a presença de um contexto transacional iniciado pelo cliente é obrigatória. Chamado fora do contexto transacional, causará javax.transaction.TransactionRequiredException.



TransactionAttributeType.NEVER - Indica que o método nunca pode ser chamado no escopo de uma transação. Se o cliente que chama o método iniciar ou propagar o contexto de uma transação, o bean causará uma RemoteException se for um bean remoto, ou EJBException se for local.



A declaração de `@TransactionAttribute` no nível da classe não afeta os métodos de callback do ciclo de vida (anotados com `@PostConstruct`, etc.) que têm um contexto transacional não especificado e dependente do tipo do bean.

7.2.2 Destino de uma transação em CMT

Apenas exceções do sistema (*Runtime, Remote, EJBException*) provocam rollback automático. O container não assume que outras exceções devam causar rollback. Capturar a exceção e chamar `rollback()` explicitamente não é uma alternativa legal pois o método não é permitido em CMT. Portanto, para que uma exceção dispare automaticamente um rollback há duas alternativas:

- Anotar a exceção com `@ApplicationException`
- Capturar a exceção e condenar a transação CMT usando o método `setRollbackOnly()` de um `EJBContext`. O status da transação está disponível via `getRollbackOnly()`:

```
try {
    return new ClientePK(clienteDAO.create(clienteDTO));
} catch (UsuarioJaExisteException e) {
    if (!ctx.getRollbackOnly())
        ctx.setRollbackOnly(); // doom this transaction
    throw e;
}
```

7.3 Transações em Message-driven beans

Como um cliente não chama um MDB diretamente, não é possível propagar um contexto transacional em um MDB, mas MDBs podem iniciar novos contextos transacionais. O escopo da transação deve iniciar e terminar dentro do método *onMessage()*.

É mais simples usar CMT, que considera a entrega da mensagem como parte da transação. Havendo rollback, o container poderá reenviar a mensagem. Se for usado BMT isto precisa ser feito explicitamente, lançando uma *EJBException* para evitar o *acknowledgement* e forçar o reenvio.

Para message-driven beans, apenas *NOT_SUPPORTED* e *REQUIRED* podem ser usados.

7.4 Sincronização de estado em Stateful Session Beans

Entities (JPA) mantêm seu estado no banco de dados. Quaisquer alterações em suas variáveis de instância, durante uma operação, serão revertidas em um rollback. Já o estado de Stateful Session Beans é mantido em variáveis de instância e não no banco de dados. O container não tem como recuperar o estado de suas variáveis em caso de rollback. É preciso guardar o estado anterior do bean para reverter caso a transação falhe.

Se o bean inicia a transação (BMT), podemos guardar o estado antes do begin e recuperar após o rollback. E se o container iniciar a transação (CMT)?

7.4.1 Interface SessionSynchronization

A interface *SessionSynchronization* pode ser implementada por Stateful Session Beans configurados com CMT para capturar eventos lançados nos pontos de demarcação. Ela consiste de três métodos:

- void **afterBegin()**: Chamado logo após o início da transação, deve guardar o estado do bean para recuperação em caso de falha.
- void **beforeCompletion()**: Chamado antes do commit() ou rollback(). Geralmente vazio, mas pode ser usado pelo bean para abortar a transação se desejar (usando setRollbackOnly())
- void **afterCompletion(boolean state)**: Chamado depois do commit() ou rollback(). Se a transação terminou com sucesso, state é true; caso contrário, é false deve-se restaurar o estado do bean aos valores guardados em afterBegin()

8 Clientes EJB

Clientes EJB que rodam fora do container só podem acessar beans *@Remote* ou *@WebService*. Beans *@Remote* requerem proxy IIOP, obtido via lookup JNDI. Com a API *Embeddable EJB*, pode-se realizar a conexão sem depender de configurações proprietárias. Uma vez configurada, *EJBContainer.createEJBContainer()* obtém um contexto JNDI global para localizar o bean:

```
public void testEJB() throws NamingException {
    EJBContainer ejbC =
        EJBContainer.createEJBContainer(
            ResourceManager.getInitialEnvironment(new Properties()));
    Context ctx = ejbC.getContext();
    MyBean bean = (MyBean) ctx.lookup ("java:global/classes/org/sample/MyBean");
    // fazer alguma coisa com o bean
    ejbC.close();
}
```

O container deve ser fechado após o uso.

Para configurar, é preciso ter no classpath os JARs requeridos pelo container, que pode incluir JARs específicos do servidor usado. É necessário especificar as seguintes propriedades em um arquivo jndi.properties localizado no classpath:

- javax.ejb.embeddable.initial
- javax.ejb.embeddable.modules
- javax.ejb.embeddable.appName

O valor das propriedades é dependente do fabricante e do ambiente do servidor. O fabricante também poderá requerer propriedades adicionais.

6: Web Services SOAP

1	SOAP Web Services	1
2	Web Services SOAP em Java com JAX-WS	2
2.1	Componente Web	2
2.1.1	WSDL gerado	3
2.2	Anotações e Contexto	4
2.2.1	@OneWay	4
2.2.2	@WebParam e @WebResult	4
2.3	Stateless remote SOAP bean	5
3	Clientes SOAP	5
3.1	Compilação de WSDL	6
3.1.1	Cliente SOAP usando classes geradas	6
3.2	Tipos de clientes	7
3.3	Cliente de WebService rodando em container	8
4	Metadados de uma mensagem SOAP	9
4.1	WebServiceContext	9
4.2	MessageContext	10
5	JAX-WS Handlers	10
5.1	SOAP Handler	11
5.2	Logical Handler	11
5.3	Como usar handlers	12

1 SOAP Web Services

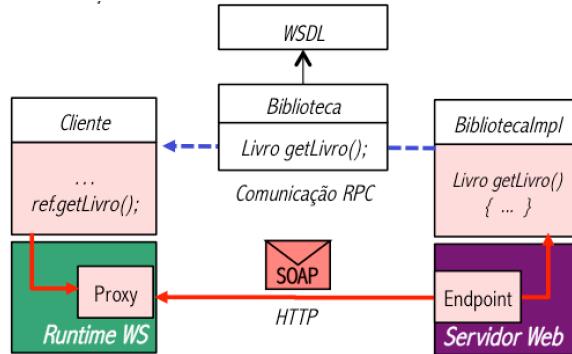
Web Services é uma arquitetura de objetos distribuídos que geralmente usa como camada de transporte o protocolo HTTP. SOAP Web Services representa serviços que utilizam protocolos em XML para praticamente todos os serviços, inclusive transporte (SOAP), registro (UDDI) e descrição de interfaces comuns (WSDL).

A comunicação depende de uma interface comum compartilhada e implementada entre cliente e servidor. O objeto remoto implementa interface comum através de um proxy. Essa interface é

exportada em WSDL, que é independente de linguagem. Qualquer cliente de qualquer linguagem pode usar um WSDL para gerar código de acesso ao serviço que exportou a interface.

A principal alternativa a SOAP para implementar Web Services é usar a arquitetura REST, que usa a infraestrutura do HTTP para oferecer uma interface de serviços remotos.

O diagrama abaixo ilustra a arquitetura de Web Services SOAP.



2 Web Services SOAP em Java com JAX-WS

JAX-WS é uma API do Java EE que permite a criação de serviços e clientes SOAP, de forma transparente, escondendo todos os detalhes da comunicação. Para criar um serviço SOAP em Java EE há duas alternativas:

1. Através da criação de um **Componente Web** – requer a criação de uma interface de terminal de serviços (*SEI – Service Endpoint Interface*) configurada com anotações, uso de ferramentas para gerar código, e empacotamento das classes compiladas em um WAR para implantação em um container Web.
2. Através de um **Session Bean** (EJB) – É preciso criar um *Stateless Session Bean* com ou sem uma interface para exportar (que será o SEI) configurado com anotações, empacotar como EJB e implantar o EJB-JAR ou EAR em um container EJB.

2.1 Componente Web

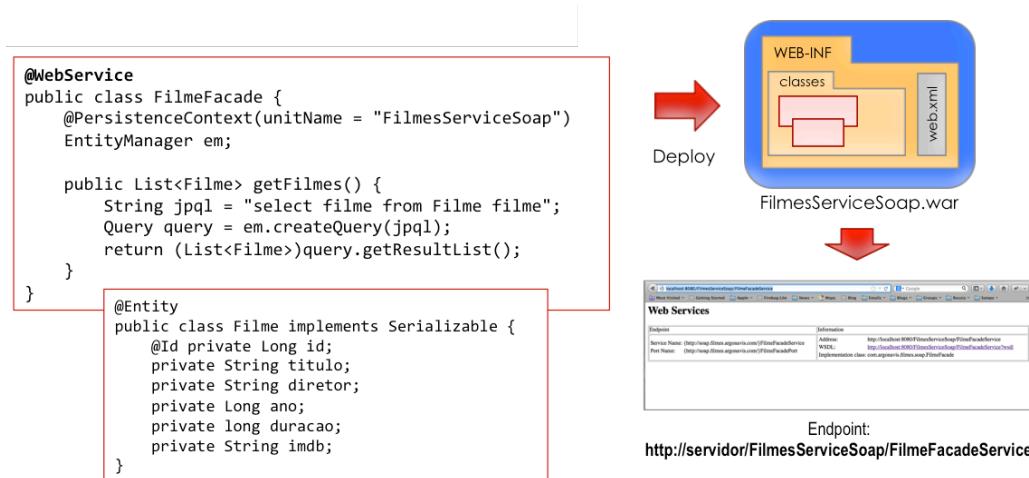
O serviço é implementado em um JavaBean comum anotado com `@WebService` (pacote `javax.jws`) que representa um *Service Endpoint Interface (SEI)*. Todos os métodos públicos de um SEI são automaticamente incluídos na interface do serviço (Java EE 7).

```
@WebService
public class FilmeFacade {
    @PersistenceContext(unitName = "FilmesServiceSoap")
    EntityManager em;

    public List<Filme> getFilmes() {
        String jpql = "select filme from Filme filme";
        Query query = em.createQuery(jpql);
        return (List<Filme>)query.getResultList();
    }
}
```

Usando a anotação `@WebMethod` em um método, os métodos não anotados serão excluídos da SEI e terão que receber uma anotação `@WebMethod` se devem ser incluídos.

O bean deve ser empacotado em um WAR (seguindo a estrutura comum do WAR, dentro de `WEB-INF/classes`), e depois instalado em um container Web.



2.1.1 WSDL gerado

A partir do deployment, um WSDL será gerado e disponibilizado para os clientes através da URL <http://servidor/nome-do-war/NomeDaClasseService?wsdl>. O WSDL gerado para o exemplo acima está listado abaixo.

```

<definitions targetNamespace="http://soap.filmes.agonavis.com/" name="FilmeFacadeService">
    <types>
        <xss:schema version="1.0" targetNamespace="http://soap.filmes.agonavis.com/">
            <xss:element name="getFilmes" type="tns:getFilmes"/>
            <xss:element name="getFilmesResponse" type="tns:getFilmesResponse"/>
            <xss:complexType name="filme">
                <xss:sequence>
                    <xss:element name="ano" type="xs:long" minOccurs="0"/>
                    <xss:element name="diretor" type="xs:string" minOccurs="0"/>
                    <xss:element name="duracao" type="xs:long"/>
                    <xss:element name="id" type="xs:long" minOccurs="0"/>
                    <xss:element name="imdb" type="xs:string" minOccurs="0"/>
                    <xss:element name="titulo" type="xs:string" minOccurs="0"/>
                </xss:sequence>
            </xss:complexType>
            <xss:complexType name="getFilmes"> <xss:sequence/> </xss:complexType>
            <xss:complexType name="getFilmesResponse">
                <xss:sequence>
                    <xss:element name="return" type="tns:filme" minOccurs="0" maxOccurs="unbounded"/>
                </xss:sequence>
            </xss:complexType>
        </xss:schema>
    </types>
    <message name="getFilmes">
        <part name="parameters" element="tns:getFilmes"/>
    </message>
    <message name="getFilmesResponse">
        <part name="parameters" element="tns:getFilmesResponse"/>
    </message>
    <portType name="FilmeFacade">
        <operation name="getFilmes">...</operation>
    </portType>
    <binding name="FilmeFacadePortBinding" type="tns:FilmeFacade">... </binding>
    <service name="FilmeFacadeService">
        <port name="FilmeFacadePort" binding="tns:FilmeFacadePortBinding">

```

```

<soap:address
    location="http://localhost:8080/FilmesServiceSoap/FilmeFacadeService"/>
</port>
</service>
</definitions>
```

2.2 Anotações e Contexto

Além de `@WebService` (única anotação obrigatória), várias outras anotações, dos pacotes `javax.jws.*` e `javax.jws.soap.*`, podem ser usadas para configurar detalhes do serviço.

Anotações aplicadas na classe:

- `@SOAPBinding` – especifica mapeamento SOAP;
- `@BindingType` – especifica tipo de mapeamento;
- `@HandlerChain` – associa o Web Service a uma cadeia de handlers;

Anotações usadas em métodos:

- `@WebMethod` – configura métodos da interface SEI, inclui e exclui;
- `@OneWay` – declara método uma operação sem retorno (só mensagem de ida);

Anotações para parâmetros de um método:

- `@WebParam` – configura nomes dos parâmetros;

Anotações para valores de retorno de um método

- `@WebResult` – configura nome e comportamento;

2.2.1 `@OneWay`

Métodos anotados com `@OneWay` têm apenas mensagem de requisição (sem resposta). Esta anotação pode ser usada em métodos que retornam void. Exemplos:

```

@WebMethod @OneWay
public void enviarAvisoDesligamento() {
    ...
}

@WebMethod @OneWay
public void ping() {
    ...
}
```

2.2.2 `@WebParam` e `@WebResult`

Permitem configurar o WSDL que será gerado e o mapeamento entre o SEI e o SOAP.

`@WebResult` serve para configurar o elemento XML de retorno. No exemplo abaixo, a resposta estará dentro de um elemento XML `<filme></filme>`. O default é `<return></return>`.

```

@WebMethod @WebResult(name="filme")
public Filme getFilme(String imdbCode) {
    return getFilmeObject(imdbCode);
}
```

`@WebParam` permite configurar nomes dos parâmetros. No exemplo abaixo, o parâmetro da operação `getFilme()` no SOAP e WSDL é `imdb`. Seria `imdbCode` (o nome da variável local) se o `@WebParam` não estivesse presente:

```

@WebMethod
public Filme getFilme(@WebParam(name="imdb") String imdbCode) {
    return getFilmeObject(imdbCode);
}
```

2.3 Stateless remote SOAP bean

Usar EJB é a forma mais simples de criar e implantar um serviço. Session Beans podem ter sua interface exportada como um Web Service SOAP. O resultado é idêntico à do Web Service via componente Web, mas a configuração é mais simples e permite acesso aos serviços básicos do EJB, mais o acesso remoto via porta HTTP.

Pode-se criar um session bean remoto que exporta uma interface SEI criando uma interface anotada com @WebService

```
@WebService
interface LoteriaWeb {
    int[] numerosDaSorte();
}
```

E depois implementando-a em um bean @Stateless:

```
@Stateless
public class LoteriaWebBean implements LoteriaWeb {
    @Override
    public int[] numerosDaSorte() {
        int[] numeros = new int[6];
        for(int i = 0; i < numeros.length; i++) {
            int numero = (int) Math.ceil(Math.random() * 60);
            numeros[i] = numero;
        }
        return numeros;
    }
}
```

Se a interface não puder ser alterada (não puder receber a anotação), ela ainda pode ser configurada no próprio bean usando @WebService com o atributo endpointInterface:

```
@Stateless
@WebService(endpointInterface="nome.da.Interface")
```

Com o deploy, o servidor gera as classes TIE do servidor, WSDL, e estabelece um endpoint. Por exemplo, o bean anterior pode ser configurado da seguinte forma:

```
@Stateless @WebService(endpointInterface="LoteriaWeb")
public class LoteriaWebBean implements LoteriaWeb {
    @Override
    public int[] numerosDaSorte() {
        int[] numeros = new int[6];
        for(int i = 0; i < numeros.length; i++) {
            int numero = (int) Math.ceil(Math.random() * 60);
            numeros[i] = numero;
        }
        return numeros;
    }
}
```

3 Clientes SOAP

Clientes SOAP podem ser criados de várias formas, em Java ou mesmo em outras linguagens. A maneira mais simples consiste em usar ferramentas para gerar código estaticamente compilando o WSDL, ou usar um container do fabricante. Outras estratégias permitem gerar stubs, proxies e classes dinamicamente, ou ainda usar *reflection* para chamar a interface dinamicamente.

O exemplo abaixo mostra um cliente (estático) em Java rodando como aplicação standalone:

```

public class WSClient {
    public static void main(String[] args) throws Exception {

        LoteriaWebService service = new LoteriaWebService();
        LoteriaWeb port = service.getLoteriaWebPort();
        List<String> numeros = port.numerosDaSorte();

        System.out.println("Numeros a jogar na SENA:");
        for(String numero : numeros) {
            System.out.println(numero);
        }
    }
}

```

As classes em destaque acima (*LoteriaWebService* e *LoteriaWeb*) são classes que foram geradas a partir do WSDL por ferramentas.

3.1 Compilação de WSDL

Ferramentas de compilação do WSDL existem em vários IDEs e fazem parte do Java Development Kit. No Java SDK existe a ferramenta *wsimport* (Java). O CXF (JBoss/WildFly) usa uma ferramenta semelhante chamada *wsconsume*. Ambas geram artefatos Java necessários para clientes através da compilação de WSDL.

Exemplo de geração de código com *wsimport* (Java SE SDK)

```

wsimport -keep -s gensrc -d genbin
          -p com.argonavis.filmes.client.soap.generated
          http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl

```

Exemplo de geração de código com *wsconsume* (WildFly / JBoss)

```

wsconsume.sh -k -s gensrc -o genbin
          -p com.argonavis.filmes.client.soap.generated
          http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl

```

Abaixo está uma lista das classes geradas para os exemplos mostrados anteriormente (filmes). Estas classes são as que o cliente precisará usar para utilizar o serviço remote. Elas devem ser incluídas no classpath do cliente:

- *Filme.class*
- *FilmeFacade.class*
- *FilmeFacadeService.class*
- *GetFilmes.class*
- *GetFilmesResponse.class*
- *ObjectFactory.class*
- *package-info.class*

As ferramentas de linha de comando podem ser executadas durante a construção da aplicação em plugins Maven.

3.1.1 Cliente SOAP usando classes geradas

O exemplo abaixo mostra um cliente SOAP que usa as classes geradas:

```

public class FilmeClient {

    public static void main(String[] args) {
        FilmeFacadeService service = new FilmeFacadeService();
        FilmeFacade proxy = service.getFilmeFacadePort();

        listarFilmes(proxy.getFilmes());
    }
}

```

```

public static void listarFilmes(List<Filme> filmes) {
    for(Filme f : filmes) {
        System.out.println(f.getImdb() + ": " + f.getTitulo()
                           + "(" + f.getAno() + ")");
        System.out.println("          " + f.getDiretor());
        System.out.println("          " + f.getDuracao() + " minutos\n");
    }
}
}

```

Executando a classe acima, o proxy conecta-se ao servidor e obtém os objetos remotos:

```

$ java -jar FilmeClient.jar
tt0081505: The Shining(1980)
Stanley Kubrick
144 minutos
tt1937390: Nymphomaniac(2013)
Lars von Trier
330 minutos
tt0069293: Solyaris(1972)
Andrei Tarkovsky
167 minutos
tt1445520: Hearat Shulayim(2011)
Joseph Cedar

```

3.2 Tipos de clientes

Cientes podem ser mais dinâmicos e menos acoplados, depender menos de implementações e de interfaces, e configurar-se em tempo de execução. Há duas estratégias:

- *Proxy dinâmico*: tem cópia local da interface do serviço, mas gera código em tempo de execução através do WSDL remoto
- *Cliente totalmente dinâmico*: não depende de interface, WSDL ou quaisquer artefatos gerados para enviar requisições e obter resposta, mas é necessário trabalhar no nível mais baixo das mensagens XML (SOAP)

Cliente com *proxy dinâmico*:

```

URL wsdl = new URL("http://servidor/app/AppInterfaceService?wsdl");
QName nomeServ = new QName("http://app.ns/", "AppInterfaceService");
Service service = Service.create(wsdl, nomeServ);
AppInterface proxy = service.getPort(AppInterface.class);

```

Cliente 100% dinâmico (Dispatch client – trecho)

```

Dispatch<Source> dispatch =
    service.createDispatch(portName, Source.class, Service.Mode.PAYLOAD);
String reqPayload =
    "<ans1:getFilmes xmlns:ans1=\"http://soap.filmes.agonavis.com/\">\n"
    + "</ans1:getFilmes>";
Source resPayload = dispatch.invoke(new StreamSource(new StringReader(reqPayload)));

DOMResult domTree = new DOMResult();
TransformerFactory.newInstance().newTransformer().transform(resPayload, domTree);
Document document = (Document)domTree.getNode();
Element root = document.getDocumentElement();
Element filmeElement =
    (Element)root.getElementsByTagName("return").item(0); // <return>...</return>
String tituloDoFilme =
    filmeElement.getElementsByTagName("titulo").item(0).getFirstChild()
        .getTextContent(); // <titulo>CONTEUDO</titulo>...

```

A classe abaixo ilustra uma implementação de cliente para os exemplos de WebServices listados anteriormente, usando a técnica de cliente dinâmico (Proxy dinâmico):

```

public class FilmeDynamicClient {
    public static void main(String[] args) throws MalformedURLException {
        URL wsdlLocation =
            new URL("http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl");
        QName serviceName =
            new QName("http://soap.filmes.agonavis.com/", "FilmeFacadeService");
        Service service = Service.create(wsdlLocation, serviceName);
        FilmeFacade proxy = service.getPort(FilmeFacade.class);
        listarFilmes(proxy.getFilmes());
    }

    public static void listarFilmes(List<Filme> filmes) {
        for(Filme f : filmes) {
            System.out.println(f.getImdb() + ": " + f.getTitulo()
                + "(" + f.getAno() + ")");
            System.out.println("          " + f.getDiretor());
            System.out.println("          " + f.getDuracao() + " minutos\n");
        }
    }
}

```

3.3 Cliente de WebService rodando em container

Cientes localizados em um container Java EE (ex: servlet ou managed bean) podem injetar o serviço através da anotação @WebServiceRef:

```

@Named("filmesBean")
public class FilmesManagedBean {
    @WebServiceRef(wsdlLocation=
        "http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl")
    private FilmeFacadeService service;
    private List<Filme> filmes;

    @PostConstruct
    public void init() {
        FilmeFacade proxy = service.getFilmeFacadePort();
        this.filmes = proxy.getFilmes();
    }
    ...
}

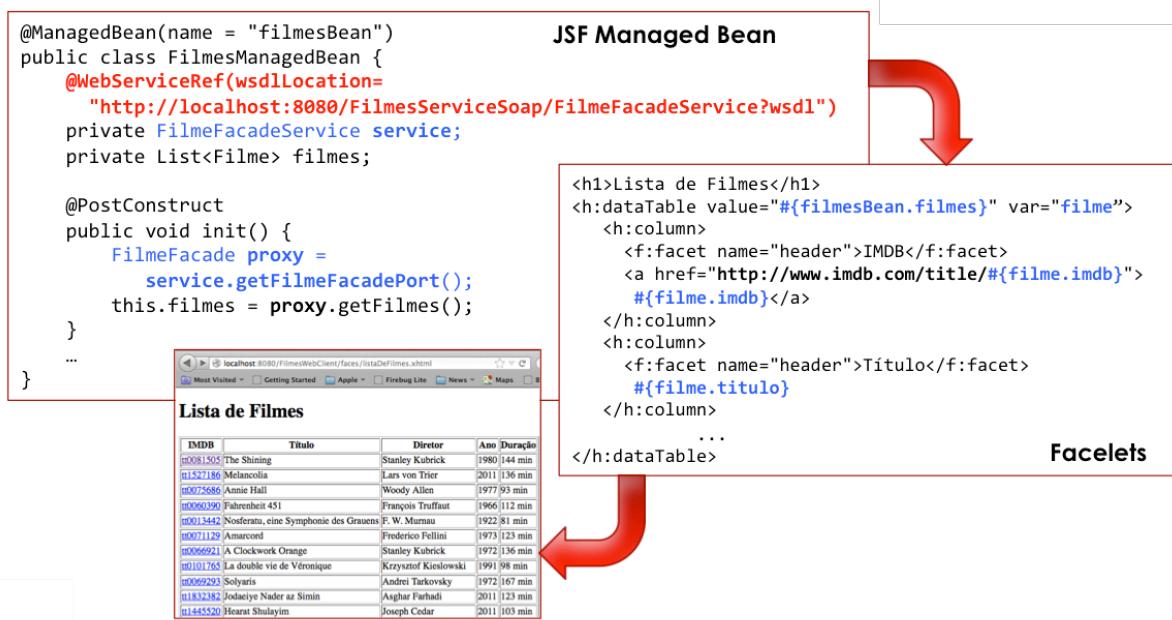
```

O JSF abaixo usa o bean acima:

```

<h1>Lista de Filmes</h1>
<h: dataTable value="#{filmesBean.filmes}" var="filme">
    <h: column>
        <f: facet name="header">IMDB</f: facet>
        <a href="http://www.imdb.com/title/#'{filme.imdb}">
            #{filme.imdb}</a>
    </h: column>
    <h: column>
        <f: facet name="header">Título</f: facet>
        #{filme.titulo}
    </h: column>
    ...
</h: dataTable>

```



4 Metadados de uma mensagem SOAP

4.1 WebServiceContext

WebServiceContext dá acesso a um objeto de contexto que permite acesso a informações sobre a mensagem e autenticação/autorização, se houver. Os métodos são:

- *MessageContext getMessageContext()*: retorna objeto *MessageContext* que permite acesso a metadados da mensagem (cabeçalhos, porta, serviço, info do servlet, etc.)
- *Principal getUserPrincipal()*: permite acesso ao *javax.security.Principal* do usuário autenticado. Principal contém informações de autenticação.
- *boolean isUserInRole(String role)*: retorna *true* se usuário autenticado faz parte de um grupo de autorizações (role).

Roles de autorização precisam ser configurados em web.xml para o contexto do Web Service (Web Resource Collection). Veja detalhes na documentação sobre segurança de aplicações Web.

Exemplo de uso:

```

@Resource
private WebServiceContext ctx;

@WebMethod()
public String metodoSeguro(String msg) {
    String userid = ctx.getUserPrincipal().getName();
    if (userid.equals("czar")) {
        ...
    } else if (ctx.isUserInRole("admin")) {
        ...
    }
}

```

4.2 MessageContext

MessageContext é um dos objetos obtidos de um *WebServiceContext*. É um dos objetos que permite acesso a metadados de uma mensagem (ex: cabeçalhos SOAP, se mensagem é inbound ou outbound, dados do WSDL, servidor, etc.)

O acesso às propriedades ocorre através do método *get()*, passando-se uma constante correspondente à propriedade desejada. As propriedades são armazenadas em um *Map*. Algumas propriedades incluem:

- *MESSAGE_OUTBOUND_PROPERTY* (boolean)
- *INBOUND_MESSAGE_ATTACHMENTS* (Map)
- *HTTP_REQUEST_METHOD* (String)
- *WSDL_OPERATION* (Qname)

Exemplo de uso (para obter os cabeçalhos HTTP da mensagem):

```
@Resource
WebServiceContext wsctx;

@WebMethod
public void metodo() {
    MessageContext ctx = wsContext.getMessageContext();
    Map headers = (Map)ctx.get(MessageContext.HTTP_REQUEST_HEADERS);
    ...
}
```

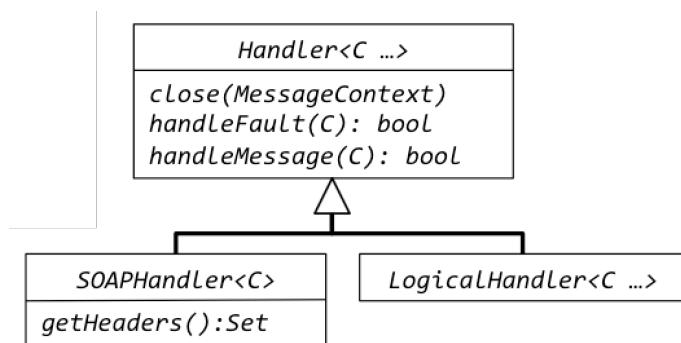
5 JAX-WS Handlers

Handlers são interceptadores (filtros) de mensagens. Há dois tipos: protocol (*MESSAGE*) e logical (*PAYOUTLOAD*):



Eles podem ser usados para fazer alterações na mensagem antes que ela seja processada no servidor ou cliente. Geralmente são configurados em cascata.

Para criar um handler deve-se implementar *SOAPHandler* ou *LogicalHandler*:



Depois é necessário criar um XML para configurar a corrente:

```
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"
                         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <javaee:handler-chain>
        <javaee:handler>
            <javaee:handler-class>pacote.ClasseDoHandler</javaee:handler-class>
        </javaee:handler>
    </javaee:handler-chain>
</javaee:handler-chains>
```

Para usar anota-se a classe do Web Service com `@HandlerChain`, passando como parâmetro o arquivo XML de configuração:

```
@HandlerChain(file="handlers.xml")
```

5.1 SOAP Handler

Implementa `javax.xml.ws.handler.soap.SOAPHandler`. É um interceptador para a *mensagem inteira* (permite acesso a cabeçalhos da mensagem):

```
public class MyProtocolHandler implements SOAPHandler {
    public Set getHeaders() { return null; }

    public boolean handleMessage(SOAPMessageContext ctx) {
        SOAPMessage message = ctx.getMessage();
        // fazer alguma coisa com a mensagem
        return true;
    }

    public boolean handleFault(SOAPMessageContext ctx) {
        String operacao = (String)
            ctx.get(MessageContext.WSDL_OPERATION);
        // logar nome da operacao que causou erro
        return true;
    }
    public void close(MessageContext messageContext) {}
}
```

5.2 Logical Handler

Implementa a interface `javax.xml.ws.handler.LogicalHandler`. Dá acesso apenas ao payload (corpo) da mensagem:

```
public class MyLogicalHandler implements LogicalHandler {
    public boolean handleMessage(LogicalMessageContext ctx) {
        LogicalMessage msg = context.getMessage();
        Source payload = msg.getPayload(); // XML Source
        Transformer transformer =
            TransformerFactory.newInstance().newTransformer();
        Result result = new StreamResult(System.out);
        transformer.transform(payload, result); // Imprime XML na saída
    }

    public boolean handleFault(LogicalMessageContext ctx) {
        String operacao = (String)
            ctx.get(MessageContext.WSDL_OPERATION);
        // logar nome da operacao que causou erro
        return true;
    }

    public void close(MessageContext context) {}
}
```

5.3 Como usar handlers

Arquivo de configuração (handlers-chains.xml)

```
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"
                         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <javaee:handler-chain>
        <javaee:handler>
            <javaee:handler-class>com.acme.MyProtocolHandler</javaee:handler-class>
        </javaee:handler>
        <javaee:handler>
            <javaee:handler-class>com.acme.MyLogicalHandler</javaee:handler-class>
        </javaee:handler>
    </javaee:handler-chain>
</javaee:handler-chains>
```

Uso no servidor (no SEI)

```
@WebService
@HandlerChain(file="handler-chains.xml")
public class FilmesFacade { ... }
```

Uso no cliente (edite classe gerada que implementa o Service)

```
@HandlerChain(file="handler-chains.xml")
public class FilmesFacadeService extends Service { ... }
```

7: Java Server Faces (JSF)

1	Introdução	4
1.1	Como construir aplicações JSF	5
1.2	Exemplos de aplicações JSF	6
1.2.1	Página web mínima	8
1.2.2	Como executar	8
1.2.3	Com managed bean	9
1.2.4	Identificadores de navegação e métodos de ação HTTP	10
1.2.5	Uso de resources	10
2	Linguagem de expressões (EL)	11
2.1	Method & value expressions	12
2.2	Como acessar as propriedades de um JavaBean	13
2.3	Sintaxe da EL	13
2.4	Funções JSTL	14
3	Arquitetura e ciclo de vida	16
3.1	Principais APIs	16
3.2	Componentes	16
3.2.1	Modelo de componentes UI	17
3.2.2	Interfaces de comportamento	18
3.2.3	Modelo de eventos	19
3.2.4	Modelo de renderização	20
3.2.5	Modelo de conversão	20
3.2.6	Modelo de validação	20
3.2.7	Modelo de navegação	21
3.3	Tipos de requisições JSF	21
3.4	Ciclo de vida do JSF	22
3.4.1	Monitoração do ciclo de vida	24
4	Facelets e componentes HTML	25
4.1	O que são Facelets?	25
4.2	Componentes e tags que geram HTML	27
4.3	Atributos	28
4.3.1	Propriedades binding e rendered	28
4.4	Estrutura de uma página	29
4.4.1	Bloco <h:form>	29
4.4.2	NamingContainer, atributo name e client ids	30

4.4.3	Componentes para entrada e saída de texto	31
4.4.4	Ações e navegação	32
4.4.5	Botões e links que não disparam eventos de action	32
4.4.6	Gráficos e imagens	32
4.4.7	Seleção simples e múltipla	33
4.4.8	Layout e tabelas	35
4.5	Core tags	36
4.5.1	Tags para tratamento de eventos	36
4.5.2	Tags para conversão de dados	37
4.5.3	Tags para validação	37
4.5.4	Outros tags	37
4.6	Core tags do JSTL	38
4.7	Tags de templating	39
4.7.1	Repetição com ui:repeat	39
5	Managed beans	40
5.1	Mapeamento de propriedades em um managed bean	41
5.1.1	Binding de componentes	41
5.1.2	Mapeamento de valores	42
5.2	Comportamento de um managed bean	43
5.2.1	Métodos de ação	43
5.2.2	Métodos de processamento de eventos	44
5.2.3	Métodos para realizar validação	44
5.3	Escopos	45
5.3.1	Escopos em managed beans	45
5.3.2	Escopos fundamentais: requisição, sessão e contexto	46
5.3.3	Escopos de sessão: view, conversation e flow	47
5.3.4	Outros escopos	47
6	Conversores	48
6.1	Como usar um conversor	48
6.2	Conversão de datas	49
6.3	Conversão numérica e de moeda	49
6.4	Conversores customizados	50
7	Listeners	51
7.1	Listeners de eventos disparados por componentes (UI)	51
7.1.1	Eventos que pulam etapas	52
7.2	Eventos do sistema e ciclo de vida	53
7.2.1	PhaseListener	53
7.2.2	<f:event>	53
8	Validadores	54
8.1	Tipos e processo de validação	54
8.2	Exibição de mensagens	55
8.3	Validação implícita	55
8.4	Validação explícita	56
8.4.1	Validação manual com método validator ou action	56
8.4.2	Validação automática	57

8.5 Validators customizados	57
8.6 Bean Validation	58
9 Templates	58
9.1 Por que usar templates?	58
9.2 Templates em JSF	59
9.3 Como construir um template	60
9.4 Como usar templates	62
9.5 Contratos	63
10 Componentes	64
10.1 Quando usar um componente customizado	64
10.2 Criação de componentes em JSF	64
10.3 Componentes compostos	65
10.3.1 Nome do tag e implementação	65
10.3.2 Atributos	67
10.3.3 Componente mapeado a classe UINamingContainer	69
11 Ajax	69
11.1 Por que usar Ajax?	70
11.2 Características do Ajax no JSF	71
11.3 Atributos de <f:ajax>	72
11.4 Ajax e eventos de ação/navegação	73
11.5 Monitoração de eventos Ajax	74
11.5.1 Eventos Ajax tratados em JavaScript	74
11.5.2 Eventos Ajax tratados em Java	74
11.5.3 Erros de processamento tratados em JavaScript	74
11.6 Limitações do Ajax	75
12 Primefaces	75
12.1 Configuração e instalação	77
12.1.1 Teste	78
12.2 Temas (themes, skins)	78
12.2.1 Configuração de CSS	79
12.3 Alguns componentes Primefaces	79
12.3.1 <p:spinner>	79
12.3.2 <p:calendar>	80
12.3.3 <p:rating>	81
12.3.4 <p:autoComplete>	82
12.3.5 <p:input-mask>	82
12.3.6 <p:colorPicker>	83
12.3.7 <p:captcha>	83
12.3.8 <p:password>	83
12.3.9 <p:editor>	84
12.3.10 Accordion	84
12.3.11 Tab view	85
12.3.12 <p:panelGrid>	85
12.3.13 Outros componentes	86

13 Mecanismos de extensão	87
13.1 Passthrough	87
13.2 Integração com Bootstrap	88
14 Referências	88
14.1 Especificações e documentação oficial	88
14.2 Tutoriais e artigos	88
14.3 Livros	89
14.4 Produtos	89

1 Introdução

Java Server Faces é um framework de interface do usuário (UI) para aplicações Web em Java. Possui uma arquitetura baseada em uma árvore de componentes cujo estado e comportamento são mapeados a tags, e sincronizados em tempo de execução. Implementa uma arquitetura Web-MVC (Model-View-Controller) ao permitir a separação das responsabilidades da apresentação gráfica (View), processamento de informações (Controller) e estado dos componentes (Model). Essa separação de responsabilidades, um dos fundamentos da arquitetura Java EE, permite a criação de GUIs em HTML puro, delegando tarefas de controle envolvendo ciclos de vida, estado e eventos para objetos Java. As camadas são interligadas com baixo acoplamento através de uma Linguagem de Expressões. A separação de responsabilidades facilita testes, desenvolvimento, manutenção, evolução. Também é uma tecnologia baseada em padrões e independente de ferramentas. É também um framework extensível, que pode ser ampliado e melhorado através de frameworks integrados, desenvolvidos por terceiros.

Alguns benefícios do JSF 2.2 incluem

- Transparência no gerenciamento do estado e escopo das requisições síncronas e assíncronas
- Controle declarativo e condicional de navegação e suporte a processamento multi-página de formulários
- Suporte nativo e extensível a *validação*, *eventos* e *conversão* automática de tipos entre as camadas da aplicação
- Encapsulamento de diferenças entre browsers
- Plataforma extensível (através de bibliotecas de componentes criadas por terceiros)

O JSF esconde detalhes de baixo nível da plataforma HTTP, retirando do desenvolvedor a necessidade de lidar com eles, mas permite que sejam monitorados e configurados se necessário.

Assim como os outros padrões Java EE, a especificação JSF (através da arquitetura MVC) promove a separação de responsabilidades permitindo a divisão de tarefas durante o desenvolvimento entre diferentes perfis de desenvolvedores. A especificação define os seguintes perfis:

- **Autores de página:** *Programadores Web* que poderão construir páginas em HTML usando tags, facelets, bibliotecas de terceiros. **Responsabilidades:** construir views em XHTML, imagens, CSS, etc., interligar views com componentes usando a linguagem declarativa EL, e declarar namespaces para usar bibliotecas de tags e extensões.

- **Autores de componentes:** Programadores Java SE que poderão construir os componentes que serão mapeados a tags, ou que irão suportar páginas e aplicações. *Responsabilidades:* criar conversores, validadores, managed beans, event handlers e eventualmente escrever componentes novos.
- **Desenvolvedores de aplicação:** Programadores Java EE que irão utilizar o JSF como interface para serviços e aplicações. *Responsabilidades:* configurar a integração de aplicações JSF com dados e serviços (componentes EJBs, JPA, MDB, SOAP, REST), através de CDI e JNDI.
- **Fornecedores de ferramentas e implementadores JSF:** Usam a especificação para construir ferramentas e containers/servidores que irão suportar aplicações JSF.

As principais *desvantagens* do JSF incluem: a complexidade e flexibilidade de sua arquitetura e a grande quantidade de componentes, tags e atributos, que impõe uma longa curva de aprendizado; e o controle rígido que detém sobre a estrutura das views, baseada em geração de código HTML, JavaScript e CSS, que dificulta o uso independente dessas tecnologias em aplicações baseadas em HTML5 e frameworks populares como Bootstrap e JQuery. É possível a integração com essas tecnologias, mas não de forma transparente.

1.1 Como construir aplicações JSF

O desenvolvimento JSF envolve a criação dos componentes de uma arquitetura MVC: *Views*, representados principalmente por documentos XHTML (compostos de tags especiais chamados de *facelets*), e o uso e criação de *Controllers* e *Models*, representados principalmente por componentes Java (*managed beans*, conversores, modelos de componentes, handlers de eventos, etc.)

O *managed bean* é um papel exercido por uma classe Java comum (POJO, ou *JavaBean*) que serve para *mapear* o estado dos componentes de um formulário (ou os modelos dos próprios componentes) e para definir operações que podem ser realizadas, de forma sincronizada, sobre os dados e componentes. Managed beans são simplesmente POJOs que são gerenciados pelo runtime do JSF. Geralmente eles são disponibilizados para injeção via EL (Expression Language) através de anotações CDI ou EJB.

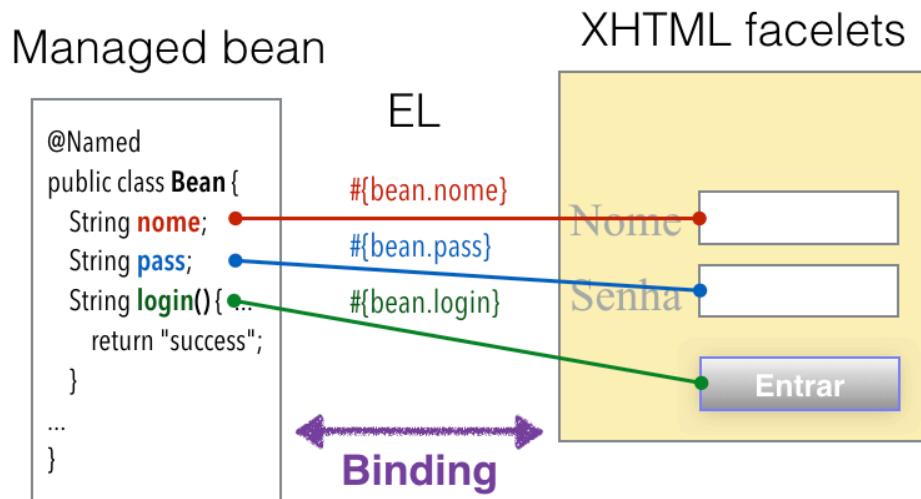
Exemplo de um POJO exercendo o papel de managed bean:

```
@Named
@RequestScoped
public class Bean {
    private String name;
    private String pass;
    public String getName() { return name; }
    public String getPass() { return pass; }
    public String login() {
        if(service.check(name, pass)) {
            return "success";
        }
        return "fail";
    }
}
```

O design das interfaces é realizado usando tags especiais que são mapeados a componentes de bibliotecas do JSF. Os tags são *templates* para tags HTML que serão gerados pelo JSF e preenchidos pelo *managed bean*. Para indicar em que parte da View e como serão usados ou transformados os dados recebidos pelo bean, o autor de uma página JSF usa o *Expression Language (EL)* que, através de uma sintaxe declarativa, permite acesso a componentes do

contexto Web, managed beans, coleções e mapas, além de permitir operações aritméticas, booleanas e de strings.

O desenho abaixo ilustra como um managed bean interage com uma View do JSF:



Os managed beans contém *propriedades*, que são mapeadas a dados de formulários, e *métodos* que implementam ações, handlers de eventos, operações de conversão e validação e que podem ser chamados por eventos gerados na View. O managed bean também cuida da navegação. Típicamente um managed bean está associado a uma View, que normalmente é composto de um único documento XHTML (mas também que pode ser representado por múltiplas páginas embutidas de um template ou componente, ou ainda ser fragmento de uma página maior).

A configuração de aplicações JSF é realizada através de anotações aplicados em classes Java, ou através dos arquivos *faces-config.xml* e *web.xml*.

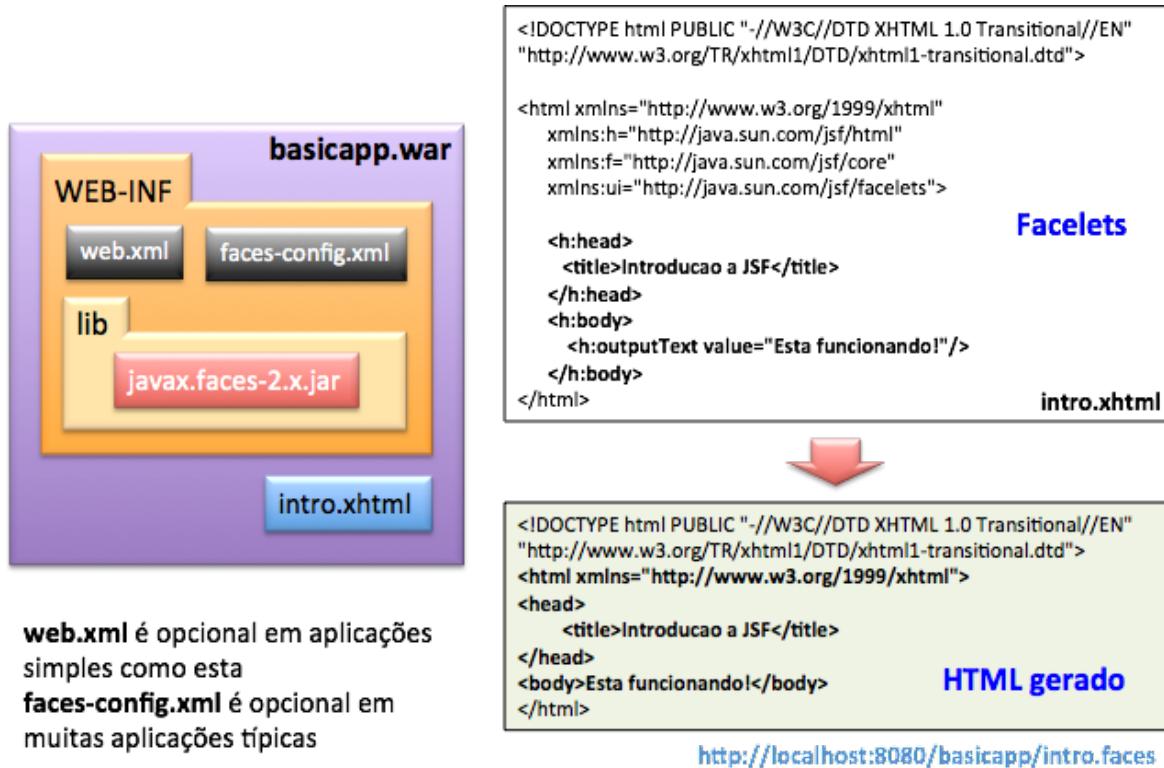
1.2 Exemplos de aplicações JSF

Uma aplicação JSF *mínima* consiste de um arquivo XHTML usando facelets (tags JSF) empacotado em uma aplicação Web (arquivo WAR). Se o servidor usado não tiver uma implementação nativa do runtime JSF, ela pode ser incluída da distribuição do componente (na pasta WEB-INF/lib). Uma opção é a implementação de referência (codinome Mojarra) que é distribuída como um único JAR.

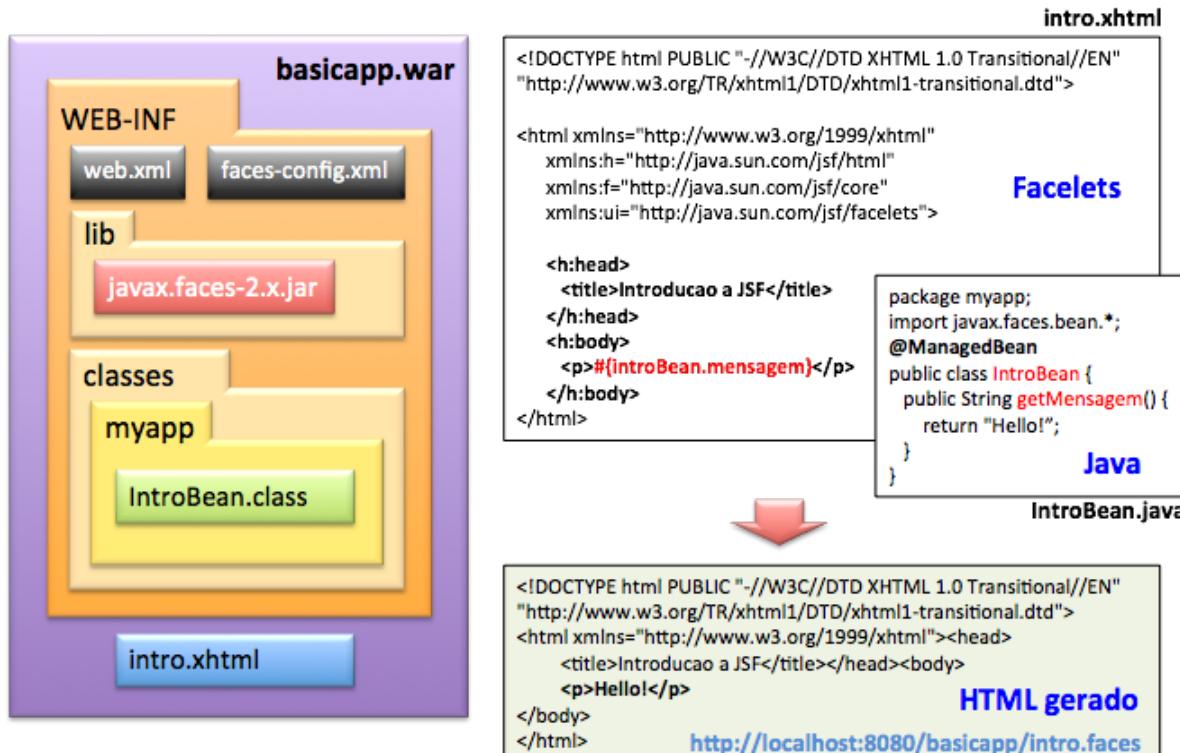
O arquivo *faces-config.xml* é opcional em JSF 2.x, e se presente, pode estar vazio.

Se o *web.xml* não for incluído, o mapeamento da URL de acesso é determinada pela implementação usada. O *Mojarra*, por exemplo, mapeia automaticamente arquivos terminados em *.faces ou *.jsf para processamento pelo runtime JSF. Para escolher outro mapeamento é preciso criar um *web.xml* e definir um *servlet-mapping* próprio.

A figura a seguir ilustra uma aplicação JSF mínima. Apesar dos arquivos *web.xml* e *faces-config.xml* serem opcionais, é uma boa prática tê-los presentes pois não é incomum serem necessários, mesmo que se decida usar os defaults. Há configurações que não são possíveis apenas através de anotações (ex: navegação condicional), e registro de listeners e filtros ainda requerem o *web.xml*.



A aplicação mostrada é mínima e serve para testar um ambiente quanto ao suporte JSF. Uma aplicação típica contém pelo menos um managed bean, como ilustrado na figura a seguir.



A ilustração mostra um bean anotado com `@ManagedBean` em vez de `@Named`. `@ManagedBean` é uma anotação válida, mas será deprecada no próximo lançamento do Java EE (junto com as outras anotações do mesmo pacote) em favor de `@Named`, que é a anotação do CDI. Neste tutorial usaremos apenas as anotações CDI.

1.2.1 Página web mínima

Um *web.xml* com a configuração default já é fornecido no JAR da implementação de referência (Mojarra 2.x), portanto não é preciso criar um *web.xml* a menos que se deseje configurar o ambiente adicionando listeners e outros recursos. Se criado, o *web.xml* irá substituir o fornecido pelo Mojarra e deve ter a configuração mínima abaixo:

```
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
</web-app>
```

O exemplo acima mantém o mapeamento *.faces* padrão do Mojarra, mas ele pode ser alterado. Um mapeamento comum é usar */faces/.xhtml* ou simplesmente **.xhtml*.

O *faces-config.xml* é usado para configurar mapeamentos de componentes, navegação, conversores, validadores, managed beans e outros recursos do JSF. Tudo isto pode ser feito via comportamento default e anotações, mas às vezes é mais prático, legível e eficiente fazer via XML (quando se usa ferramentas gráficas que geram XML, por exemplo).

A configuração *faces-config.xml* tem precedência e sobrepõe a configuração via anotações. Se usada, deve ter a seguinte configuração mínima

```
<?xml version="1.0"?>
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
</faces-config>
```

Ambos devem ser colocados na raiz da pasta WEB-INF.

Para usar JSF 2.0, a versão do *web.xml* deve ser 2.5 ou superior e do *faces-config* 2.0 ou superior. JSF 2.1 requer *web.xml* 3.0 e JSF 2.2 usa *web.xml* 3.1.

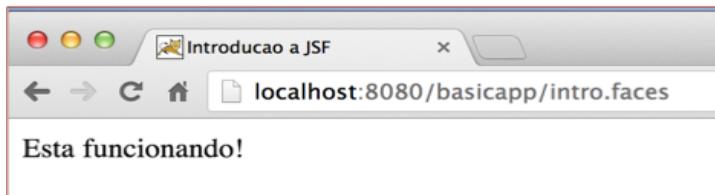
1.2.2 Como executar

Para executar uma aplicação JSF, ela precisa ser implantada (deploy) em um servidor de aplicações ou servidor Web que contenha um servlet container. No exemplo abaixo usamos o servidor de referência para containers Web do Java EE, o Tomcat, mas as regras valem para qualquer servidor de aplicações.

O deploy no Tomcat consiste em transferir o WAR para a *\$TOMCAT_HOME/webapps/* ou usar a interface de administração para fazer upload do WAR e iniciá-lo. Neste exemplo transferimos o arquivo *basicapp.war* para *webapps*. Uma vez implantada a aplicação, é preciso acessar através da URL *servidor/porta/contexto/página*. No caso do Tomcat, o nome do contexto é o nome do WAR (*basicapp*). Portanto, a URL completa para executar a aplicação é:

```
http://localhost:8080/basicapp/intro.faces
```

Se não houver erros, o resultado deve ser similar ao ilustrado abaixo:



A instalação usando outro servidor ou IDE deve produzir o mesmo resultado.

1.2.3 Com managed bean

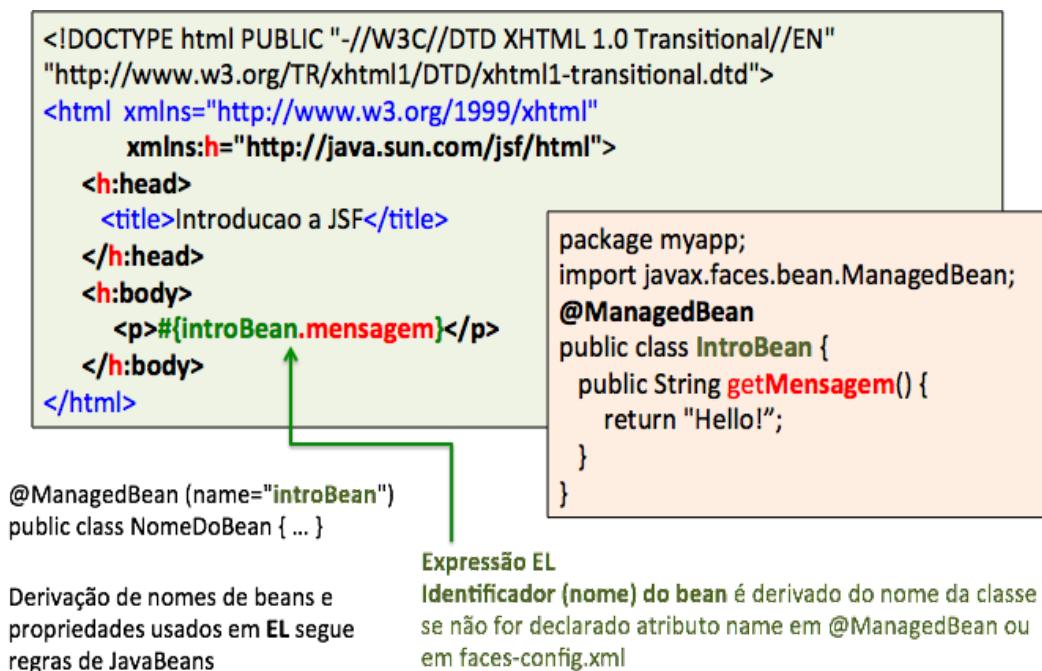
O managed bean obedece regras de formação de JavaBeans. Ao anotá-lo com `@Named` (ou `@ManagedBean`), é criado um *identificador* para o bean que pode ser usado em expressões EL, que por default é derivado do nome da classe. *Propriedades de leitura* são derivadas dos métodos *getter* (iniciados em “get” ou “is”). *Propriedades de gravação* são derivadas de métodos *setter* (iniciados em “set”). Nomes de beans e propriedades são derivadas removendo o *get/set/is* e convertendo o formato da primeira letra para minúscula. A conversão só ocorre se a segunda letra também for maiúscula. Por exemplo:

- `getNome` → `nome`
- `NomeDoBean` → `nomeDoBean`
- `getID` → `ID`
- `RESTBean` → `RESTBean`

A comunicação entre o componente e página é realizada através da expression language (EL), usada na página através dos marcadores `#{}...{}` ou `${}...{}`. A EL permite ler ou gravar propriedades do bean, transformar dados, executar expressões sobre os dados, usar resultados de operações e executar métodos no bean. Expressões EL geralmente são usadas em atributos de tags que as suportam, mas expressões de leitura de propriedades podem ser usadas diretamente na página.

- `#{{identificadorDoBean.propriedade}}`
- `#{{identificadorDoBean.metodo}}`
- `#{{identificadorDoBean.prop1.prop2.prop3}}`
- `#{{identificadorDoBean.colecao[5].value + 20}}`

A ilustração abaixo mostra o mapeamento entre um método de ação HTTP e um botão na View através de uma expressão EL:



1.2.4 Identificadores de navegação e métodos de ação HTTP

Alguns tags recebem o valor de retorno de métodos de ação HTTP (*action events*), que retornam strings que funcionam como identificadores com regras de navegação (indicam qual será a próxima View a ser mostrada). Exemplos típicos são tags usados para submissão de formulários (botões), tags de links e expressões que causam action events.

Os identificadores retornados podem ser mapeados a nomes de páginas no *faces-config.xml*. Na ausência de mapeamentos explícitos, o string retornado é interpretado por default como nome de uma página sem a extensão e contexto do mapeamento padrão, dentro do contexto da aplicação. Por exemplo:

- Se o mapeamento padrão do *FacesServlet* no *web.xml* for */faces/*.xhtml*, “pagina” refere-se à URL */faces/pagina.xhtml* dentro do contexto da aplicação.
- Se o mapeamento padrão for **.faces*, “pagina” refere-se a *pagina.faces*
- Se o mapeamento padrão for **.xhtml*, “pagina” refere-se a *pagina.xhtml*

No exemplo abaixo, em uma aplicação *jsf-intro.war* rodando em *localhost:8080* com mapeamento **.faces*, o método de ação *throwCoin()* retorna aleatoriamente ou a URL *http://localhost:8080/jsf-intro/coroafaces* ou *http://localhost:8080/jsf-intro/carafaces*:

```
@Named
@RequestScoped
public class CoinBean implements Serializable {
    public String throwCoin() {
        int coin = (int)(Math.random()*2);
        if(coin == 0) {
            return "coroa";
        } else {
            return "cara";
        }
    }
}
```

1.2.5 Uso de resources

Muitas aplicações contém *resources*, como arquivos de imagem, CSS, etc. que são parte essencial da aplicação. Resources podem ser colocados na pasta raiz e carregados normalmente do HTML através de URIs relativas se forem resources do HTML, ou na pasta classes se forem resources do Java. Mas o JSF recomenda que elas sejam colocadas em pastas especiais para que sejam carregadas pelo runtime JSF. As pastas são (relativas ao contexto):

- */META-INF/resources* – para resources que serão carregadas por arquivos Java (ex: arquivos **.properties* como resource bundles).
- */resources* – para resources carregadas pelas páginas Web.

Resources não são processados durante a requisição. Se referenciados em tags HTML, devem ser carregados informando o caminho completo.

Considere a seguinte árvore de resources Web:

```
WEB-INF
resources/
  css/
    arquivo.css
  imagens/
    icone.jpg
```

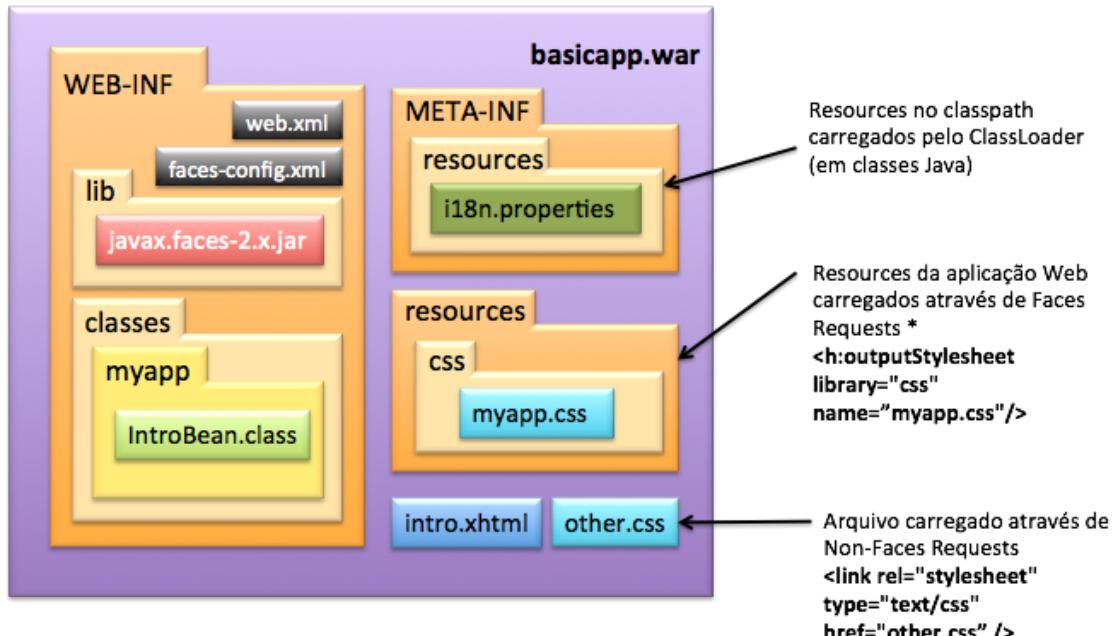
Sem usar JSF, os resources podem ser carregados da seguinte forma, em *caminho relativo ao contexto da aplicação*:

```
<link rel="stylesheet" href="resources/css/arquivo.css"/>

```

Se carregados via tags JSF, que é a forma recomendada, os resources devem ser referenciados usando o atributo “name” e caminho *relativo a essa pasta*.

```
<h:outputStylesheet name="css/arquivo.css"/>
<h:graphicImage name="imagens/icone.jpg"/>
```



* Esta é a forma recomendada de carregar CSS

JSF 2.2 ainda permite outras duas formas. Uma delas é usar as pastas de contexto como uma library (que atualmente é a melhor prática):

```
<h:outputStylesheet library="css" name="arquivo.css"/>
<h:graphicImage library="imagens" name="icone.jpg"/>
```

Pode-se também usar a seguinte sintaxe (que é compatível com URLs em CSS):

```
<h:outputStylesheet value="#{resource['css:arquivo.css']}"/>
<h:graphicImage value="#{resource['imagens:icone.jpg']}"/>
```

Exemplo de uso em CSS:

```
div {
    background-image: url( #{resource['imagens:icone.jpg']} );
```

Veja e explore os exemplos simples de aplicações JSF usando resources e beans no projeto *jsf-intro*.

2 Linguagem de expressões (EL)

Expression Language (EL) é uma linguagem simples que permite a comunicação entre as camadas de apresentação (view) e aplicação (beans). Fornece objetos implícitos, comandos, operadores para construir expressões para gerar valores, localizar componentes e chamar métodos, transformar e converter valores, definir e usar variáveis e constantes, gravar e recuperar dados, executar operações aritméticas, booleanas, de strings, etc.

Expressões podem ser usadas dentro de atributos e às vezes diretamente na página (quando geram dados de saída).

Há duas sintaxes possíveis e que podem produzir resultados diferentes em JSF. Elas determinam quando as expressões serão processadas.

A sintaxe:

```
#{ ... expressão ... }
```

é usada para expressões que precisam ser executadas *imediatamente* (assim que a página é processada pela primeira vez). Esta é a forma normalmente usada em JSP. Funciona em JSF apenas em expressões *somente-leitura*.

O ciclo de vida do JSF envolve etapas distintas para leitura de propriedades, validação, conversão, gravação e geração da View (e será explorado em uma seção mais adiante.) Expressões \${} não são afetadas por esse ciclo de vida.

A sintaxe.

```
#{ ... expressão ... }
```

é usada para expressões que podem ter a execução adiada para uma *fase posterior* da geração da página. Esta é a forma mais usada e *recomendada* em aplicações onde a geração da resposta passa por um ciclo de vida de múltiplas fases como JSF. Expressões #{} podem ser usadas em expressões de leitura e gravação. A leitura e gravação normalmente ocorrerão em fases diferentes do ciclo de vida do JSF.

No campo abaixo:

```
<h:inputText id="name" value="#{produto.nome}" />
```

O valor de #{produto.nome} é *lido* em uma fase inicial do ciclo de vida. Mas uma fase posterior poderá preenchê-lo com um valor obtido em um request após a validação e antes de ser enviado para atualizar o campo mapeado no bean.

2.1 Method & value expressions

Existem dois tipos de expressões EL:

- *Method expressions* - expressões que chamam métodos em um bean
- *Value expressions* - expressões que referenciam valores.

E existem dois tipos de *Value expressions*:

- *Rvalue* - somente leitura
- *Lvalue* - leitura e gravação

As expressões \${} são sempre *rvalue* mas expressões #{} podem ser *lvalue* ou *rvalue*.

Expressões que referenciam valores podem acessar componentes JavaBean, coleções e arrays, enums, objetos implícitos, além de propriedades ou atributos declarados no contexto de outros JavaBeans, coleções, arrays, enums e objetos.

A primeira variável de uma *value expression* é (geralmente) o nome de um bean. O bean deve ter sido previamente *registrado* e armazenado em um contexto (*Map*) com escopo limitado (ex: request, sessão, aplicação, página). Normalmente isto é feito de forma declarativa através de CDI.

A expressão:

```
#{{produto}}
```

irá procurar um componente registrado como “produto” nos escopos *page*, *request*, *session* e *application*.

Como foi mostrado na seção anterior, o registro pode ter sido feito através de tags no *faces-config.xml*, via anotação `@Named("produto")` (ou `@ManagedBean(name="produto")`) em uma classe com qualquer nome, ou ainda com a anotação `@Named` (ou `@ManagedBean`) em uma

classe de nome Produto (ou ainda programaticamente através de inserção direta nos mapas do contexto do JavaServer Faces). O escopo default nos registros declarativos é *request*.

2.2 Como acessar as propriedades de um JavaBean

Normalmente beans são acessados por causa de suas propriedades. Elas podem ser acessadas via bean através do operador “.”(ponto) ou através de “[...]” (colchetes). Normalmente, como convenção, o ponto “.” é usado para acessar propriedades enquanto que colchetes “[...]” são normalmente usados para acessar coleções, mapas e arrays, ou ainda quando é necessário referenciar uma propriedade usando um literal String. As duas formas abaixo são equivalentes:

```
#{{produto.nome}}
#{{produto["nome"]}}
```

Para acessar item de um array ou lista, é preciso usar o *índice* (ou número que contenha valor que seja traduzido em inteiro), por exemplo:

```
#{{item.itens[5]}}
```

Um *Map* pode ser acessado através de sua chave:

```
#{{banco.contas["3928-X"]}}
```

As propriedades, itens de coleções e mapas podem ser *rvalues* ou *lvalues*. A forma como serão tratadas (leitura ou gravação) depende de onde são usadas. A configuração sobre os tipos de expressões que podem ser recebidos em cada atributo é feita na definição do tag. Os tags das bibliotecas padrão informam os tipos aceitos na sua documentação.

Expressões *rvalue* podem ser usadas em atributos configurados para *receber* valores e diretamente no texto da página. Expressões *lvalue* podem ser usadas em atributos configurados para *receber e enviar* valores.

Expressões usando *valores literais* sem referência a um bean também podem ser usadas para construir expressões *rvalue*:

- #{{false}}
- #{{123}}
- #{{produto.preco + 12.5}}
- #{{"Texto"}}

2.3 Sintaxe da EL

EL possui 16 palavras reservadas:

- and, or, not
- eq, ne, lt, gt, le, ge
- true, false
- null
- instanceof,
- empty
- div, mod

Os literais do JSF são

- true e false (boolean)
- números (inteiros e ponto flutuante)
- strings entre aspas ou apóstrofes (escapes \", \' e \\)
- null

Escapes também são necessários quando uma expressão literal usada em atributos precisa imprimir as sequencias \${ ou #\$. Use

- \\${ ou \#\${
- \${'\${'}' } ou #{'#\${'}' }

Atributos que recebem expressões também podem receber expressões *literais* fixas que são expressas em texto sem os #\${}:

```
<h:inputText value="texto" />
```

Expressões compostas são concatenadas e executadas da *esquerda para a direita* (convertendo, em cada passo, o resultado em string). O String final é convertido no tipo esperado pelo atributo:

```
<h:inputText value="${produto.codigo or produto.nome}" />
```

Expressões de método permitem a chamada de métodos públicos de um bean (que geralmente retornam um valor). É possível chamar métodos com ou sem parâmetros.

A sintaxe para chamar métodos deve sempre ser #\${} uma vez que podem ser executados em fases diferentes do ciclo de vida.

Métodos são chamados através de atributos configurados para executar ações ou receber os valores retornados. Podem ser chamadas a validadores, métodos de ação (navegação), listeners de eventos, etc. O exemplo abaixo ilustra métodos que serão chamados em fases diferentes:

```
<h:form>
    <h:inputText id="codigo"
        value="#{device.codigo}"
        validator="#{device.validarCodigo}"/> <!-- fase validação -->
    <h:commandButton id="sincronizar"
        action="#{device.sincronizar}" /> <!-- fase aplicação -->
</h:form>
```

Métodos de ação HTTP (navegação) são geralmente chamados *sem* parâmetro, mas qualquer método pode ser parametrizado. Eles devem passar a lista de parâmetros (separados por vírgula) entre parênteses como normalmente são chamados em Java:

```
<h:inputText value="#{produtoBean.porCodigo('100')}">
```

Vários operadores podem ser usados para formar expressões aritméticas e booleanas. São todas do tipo *rvalue*.

- Aritméticos (+, -, *, /, %, mod, div)
- Lógicos (and, or, not, &&, ||, !)
- Relacionais (=, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le)
- empty (testa se é null ou vazio)
- A ? B : C (condicional).

Se combinados os operadores seguem regras de precedência similares às de Java que podem ser alteradas com parênteses.

Veja no projeto *jsf-el* vários exemplos de expressões EL e explore os resultados.

2.4 Funções JSTL

O JSTL (JSP Standard Tag Library) contém uma coleção de tags e funções reutilizáveis que podem ser usadas em JSF. Geralmente operações sobre arrays e strings devem ser feitas em Java nos managed beans e terem apenas seus resultados transferidos para a View. Mas há ocasiões em que é justificável usar operações na View, para, por exemplo, verificar se um string contém

um substring, contar o número de caracteres ou itens de um array, ou ainda fazer transformações mínimas. Para isto, pode-se usar as funções JSTL.

Para usar essas funções é preciso declarar o namespace das funções JSTL e definir um prefixo:

```
<html ...>
    xmlns:fn="http://java.sun.com/jsp/jstl/functions">
```

Depois as funções podem ser usadas dentro de expressões EL:

```
Número de filmes: #{fn:length(cinemateca.filmes)}
```

A tabela abaixo relaciona as funções disponíveis:

<code>fn:contains(s1, s2)</code>	Retorna true se string s1 contém o substring s2.
<code>fn:containsIgnoreCase(s1, s2)</code>	Mesmo que <code>fn:contains()</code> , com ignorando formato maiúscula/minúscula.
<code>fn:endsWith(s1, s2)</code>	Retorna true se string s1 termina com s2.
<code>fn:escapeXml(s)</code>	Converte caracteres especiais do XML em escapes <, > etc.
<code>fn:indexOf(s1, s2)</code>	Retorna posição da ocorrência de s2 dentro de s1, ou -1 se não ocorrer.
<code>fn:join(a[], delim)</code>	Concatena os elementos do array com o delimitador e retorna um string.
<code>fn:length(obj)</code>	Retorna número de elementos de uma coleção ou número de caracteres de um string.
<code>fn:replace(s1, s2, s3)</code>	Retorna um string no qual todas as ocorrências de s2 ocorridas dentro de s1 são trocadas por s3.
<code>fn:split(s, delim)</code>	Divide o string s pelo delimitador e devolve um array
<code>fn:startsWith(s1, s2)</code>	Retorna true se o string s1 começar com s2.
<code>fn:substring(s1, inicio, fim)</code>	Retorna o substring entre as posições inicio (inclusive) até antes da posição fim.
<code>fn:substringAfter(s1, s2)</code>	Retorna o substring depois da ocorrência de s2, no string s1.
<code>fn:substringBefore(s1, s2)</code>	Retorna o substring antes da ocorrência de s2, no string s1.
<code>fn:toLowerCase(s)</code>	Retorna o string em caixa baixa.
<code>fn:toUpperCase(s)</code>	Retorna o string em caixa alta.
<code>fn:trim(s)</code>	Remove os espaços em branco (tabulações, espaços, quebras de linha) antes e depois do string.

3 Arquitetura e ciclo de vida

O JavaServer faces possui uma arquitetura complexa, que envolve APIs Java, bibliotecas de componentes em Java, bibliotecas de tags em XHTML, diversos modelos abstratos e um mecanismo de tempo de execução baseado em um ciclo de vida com fases e responsabilidades bem definidas.

3.1 Principais APIs

A maior parte das APIs Java usadas para construir aplicações JSF 2.2 estão no pacote `javax.faces` e subpacotes nas distribuições do Java EE desde a versão 6.0. A seguir uma lista de alguns desses sub-pacotes e uma breve descrição das classes e interfaces ou funcionalidades mais importantes.

- **javax.faces.component:** contém uma hierarquia de classes baseada em `UIComponent`, que compõe o modelo de componentes abstrato do JSF, representando componentes gráficos genéricos. O subpacote `javax.faces.component.html` baseia-se nesse modelo abstrato e define um modelo de componentes para renderização de HTML.
- **javax.faces.application:** coleção de classes utilitárias que ajudam a integrar objetos relacionados à logica de negócios de uma aplicação ao runtime do JSF (`FacesMessage`, `Resource`, `Application`).
- **javax.faces.model:** API que contém classes que representam modelos para estruturas de dados usados em JSF, como arrays, tabelas, listas, resultados e itens de menu.
- **javax.faces.context:** API para acesso ao estado da requisição (via `FacesContext`); permite acesso a vários contextos e objetos criados durante a execução do JSF. As classes deste pacote permitem que beans e classes Java tenham acesso a componentes Faces, componentes HTML, runtime e contexto do EL, objetos de escopo, etc.
- **javax.faces.convert:** API para conversores de dados (interface `Converter`). Contém interfaces para a construção de conversores e várias implementações nativas.
- **javax.faces.event:** API para eventos (`FacesEvent`, `FacesListener`). Contém interfaces para a construção de listeners e classes de eventos.
- **javax.faces.render:** API para renderização gráfica (`RenderKit`). Permite a construção de componentes customizados e extensões ao kit de renderização HTML que é default.
- **javax.faces.validator:** API de validação. Contém interfaces para a criação de validadores e diversas implementações nativas.
- **javax.faces.flow** – API do Faces Flow, que define um escopo para tarefas que envolvem várias páginas, com um ponto de entrada e um ponto de saída.
- **javax.faces.bean:** Contém as anotações para managed beans (`@ManagedBean`, `@SessionScoped`, etc.) que devem ser *deprecadas* em um futuro próximo, portanto, não devem mais ser usadas. O mesmo comportamento pode ser obtido usando CDI.

3.2 Componentes

O Java Server Faces define diversos componentes que são abstrações de objetos que podem ter seus dados renderizados graficamente. São classes que representam componentes de interface gráfica (UI) e estendem a classe abstrata `UIComponent`. Cada classe de `UIComponent` representa um tipo específico de componente e descreve seu estado e comportamento.

Também compõem a arquitetura do JSF quatro outros modelos abstratos: renderização, conversão, eventos e validação.

3.2.1 Modelo de componentes UI

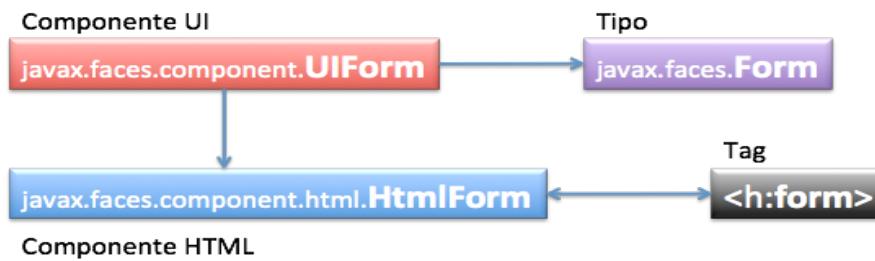
O pacote `javax.faces.component` contém a hierarquia fundamental dos componentes UI que descreve a sua estrutura e comportamento independente da interface. Em geral, programadores de aplicações JSF não instanciam componentes diretamente, mas os utilizam indiretamente através de tags. As classes, porém, são importantes para construir componentes programaticamente e para realizar *binding*, mapeando objetos Java a facelets para poder alterar suas propriedades via código Java.

As principais classes de `javax.faces.component` estão relacionadas na tabela abaixo.

<code>UIComponentBase</code>	Superclasse concreta de todos os componentes UI (implementa todos os métodos abstratos de <code>UIComponent</code>). Define estado e comportamento default para todos os componentes.
<code>UIData</code>	Mapeamento de dados a uma coleção de dados representados por um DataModel (em HTML, representa uma tabela)
<code>UIColumn</code>	Coluna de um componente <code>UIData</code> (representa uma coluna de uma tabela HTML)
<code>UICommand</code>	Controle que dispara eventos de ação (em HTML, representa um botão, um item de menu ou um link)
<code>UIForm</code>	Formulário de entrada de dados. Componentes contidos no formulário representam os campos de entrada de dados enviados quando o campo é submetido. Representa um elemento <form> em HTML
<code>UIGraphic</code>	Representa uma imagem.
<code>UIOutput</code>	Representa dados de saída (somente leitura) em uma View.
<code>UIInput</code>	Representa dados de entrada. É subclasse de <code>UIOutput</code> . Em HTML tipicamente representa elementos <input> e <textarea>.
<code>UIMessage, UIMessages</code>	<code>UIMessage</code> representa mensagens (geralmente de erro) associadas a um componente <code>UIComponent</code> . <code>UIMessages</code> obtém mensagens do contexto inteiro.
<code>UIOutcomeTarget</code>	Em HTML, representa um link como link (a href) ou botão
<code>UIPanel</code>	Componente para controlar o layout de componentes filho (qualquer <code>UIComponent</code>)
<code>UIParameter</code>	Representa parâmetros para um elemento pai.
<code>UISelectBoolean</code>	Subclasse de <code>UIInput</code> . Permite seleção de valor booleano em um controle. Em HTML representa um checkbox ou radio button individual.
<code>UISelectItem; UISelectItems</code>	<code>UISelectItem</code> representa um ítem individual dentro de um conjunto de itens; <code>UISelectItems</code> representa o conjunto de itens como uma coleção. Em HTML representam coleções contendo opções <option> de um <select>, ou os checkboxes/radio buttons individuais pertencentes a um mesmo grupo.
<code>UISelectMany</code>	Permite que um usuário selecione vários itens de um grupo. Em HTML representa um elemento <select> com opção de seleção múltipla, ou uma coleção de checkboxes de um mesmo grupo.

UISelectOne	Permite selecionar um item de um grupo. Em HTML representa um elemento <select> ou uma coleção de radio buttons de um mesmo grupo.
UIViewParameter	Representa um parâmetro de um request HTTP. Possui métodos para decodificar e extrair nome e valor.
UIViewRoot	Representa a raiz da árvore de componentes, através do qual pode-se navegar e obter todos os elementos-filho. Em DOM/HTML representa o elemento document/<body>.

Os componentes *javax.faces.component* são associados a elementos HTML através de um kit de renderização HTML, que combina cada *tipo* com uma implementação de *UIComponent* no pacote *javax.faces.component.html*, mapeando os componentes a tags XHTML. Este mapeamento é mostrado ilustrado para o tipo *Form*, abaixo:

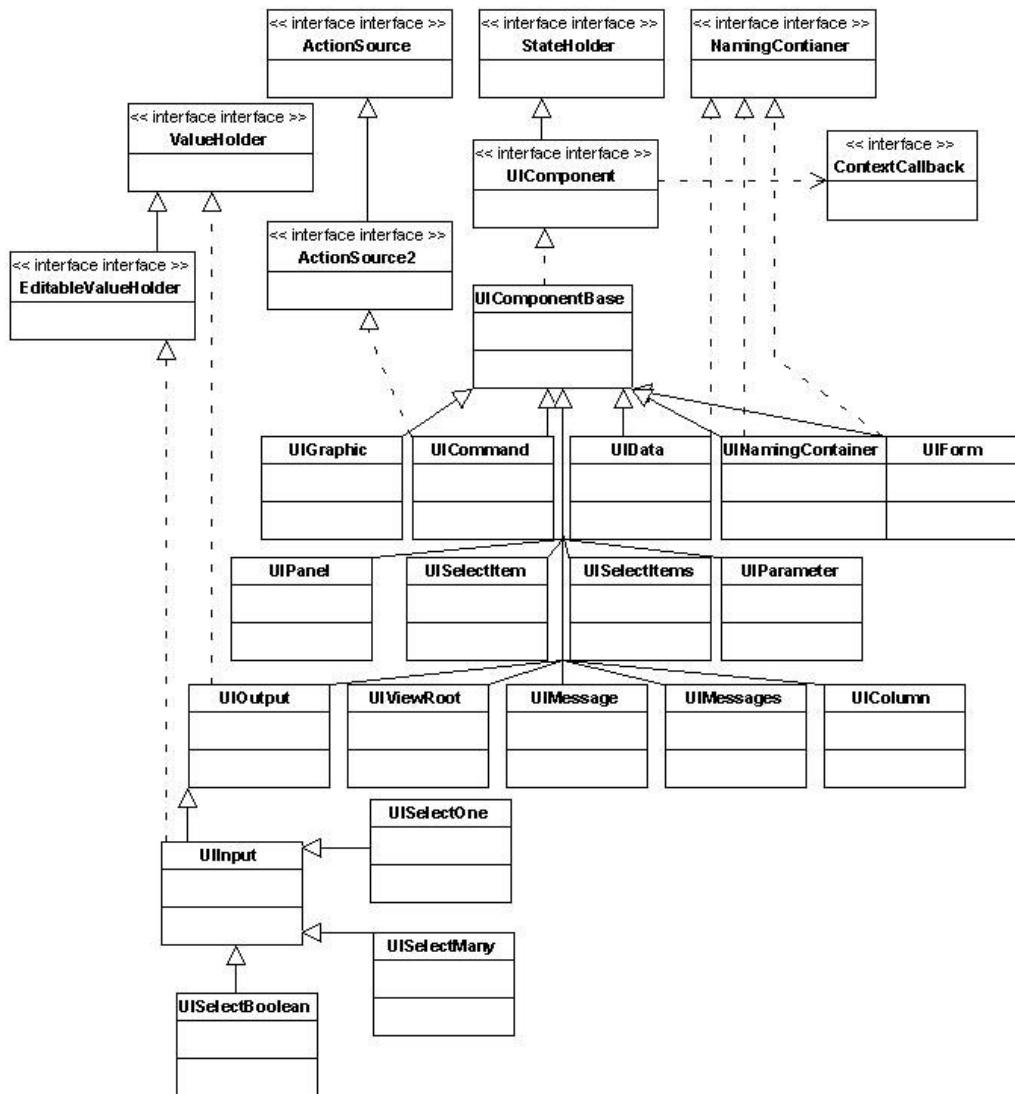


3.2.2 Interfaces de comportamento

Além das classes de componentes, o pacote *javax.faces.component* contém uma coleção de *interfaces de comportamento* que estão associados aos componentes e que mapeiam diferentes tipos de valores e ações. A tabela abaixo lista algumas dessas interfaces.

ActionSource2	Permite que o componente possa disparar evento de ação HTTP (evento <i>action</i>). Links e botões (<i>UICommand</i>) implementam esta interface.
ValueHolder	Permite que o componente possa manter um valor local e acessar dados na camada do modelo de dados através de uma expressão de valor. A maior parte dos elementos de texto e de formulário (<i>UIOutput</i>) implementam esta interface.
EditableValueHolder	Estende ValueHolder e especifica recursos adicionais para componentes editáveis. Implementado por elementos de entrada de dados (<i>UIInput</i>).
StateHolder	Indica que o componente possui estado que precisa ser preservado entre requisições. Implementado por todos os componentes <i>UIComponent</i> .
NamingContainer	Obriga ID único a componente que implementa essa interface. Todos os componentes (<i>UIComponent</i>) implementam esta interface e definem um <i>Client ID</i> , que é formado pelo <i>ID</i> do elemento pai, seguido por um separador (JSF usa ":" por default) e o <i>ID</i> local do componente.

A ilustração abaixo (extraída da especificação JSF) ilustra a hierarquia completa dos componentes UI do JSF e sua relação com as interfaces de comportamento.



3.2.3 Modelo de eventos

Há seis tipos de eventos suportados por componentes JSF 2.2:

1. **Ação HTTP** (`javax.faces.event.ActionEvent`) gerados por `UIComponent`
2. **Mudança de valor** (`javax.faces.event.ValueChangeEvent`) gerados por `UIComponent`
3. Alteração no **modelo de dados** (`javax.faces.model.DataModelEvent`) por `UIData`
4. **Fase do ciclo de vida** (`javax.faces.event.PhaseEvent`) gerados por `FacesContext`
5. Comportamento **Ajax** (`javax.faces.event.AjaxBehaviorEvent`) gerados por Ajax
6. **Sistema** (Subclasses de `javax.faces.event.SystemEvent`) gerados por outros objetos.

Além desses eventos, aplicações JSF também podem capturar outros eventos lançados em aplicações Java, como por exemplo `java.beans.PropertyChangeEvent` para detectar mudanças em propriedades de beans.

Todos os eventos possuem interfaces listener que podem ser implementados para capturá-los. Em seguida devem ser registrados. Os eventos mais importantes são os eventos gerados por `UIComponent`: `ActionEvent` e `ValueChangeEvent`, que são disparados por todos os componentes. Eles podem ser registrados através de facelets.

Há duas formas de registrar eventos de `UIComponent`:

- Pode-se *implementar uma classe* do event listener correspondente e registrá-lo no componente (aninhando um `<f:valueChangeListener>` ou `<f:actionListener>` dentro do tag), ou
- Pode-se *criar um método* em um managed bean vinculá-lo através de uma expressão EL em atributo compatível com expressões de método (ex: atributo `listener`, `actionListener`, `valueChangeListener`).

Maiores detalhes sobre registro e captura de eventos de UIComponent serão explorados em outras seções deste tutorial.

3.2.4 Modelo de renderização

Um *Render Kit* define de que forma as classes de componentes serão mapeadas a tags para um determinado tipo de cliente. Atualmente o único Render Kit incluído no JSF serve para renderização HTML.

Mas mesmo em HTML há benefícios no modelo e reuso. Renderizações de um mesmo componente podem produzir aparências diferentes. Por exemplo a classe `UISelectOne` pode ser renderizada como um grupo de opções (rádio button) `<input type="radio">`, um combo, lista ou menu pull-down (`<select><option/>...</select>`). Já um `UICommand` pode ser renderizado como botão ou link(`<a href>`, `<input type="submit">`).

O modelo de renderização é usado nas fases de restauração da view (1) para construir uma árvore de componentes a partir de um documento XHTML com raiz em `UIRootView`, e na fase de renderização da resposta (6) onde o `UIRootView`, depois de ser atualizado é usado para gerar o HTML da página da resposta.

Em versões anteriores a JSF 2.0 usava-se o render kit para construir componentes reutilizáveis. Desde JSF 2.0 há uma API declarativa muito mais simples para criar bibliotecas de tags e componentes, portanto neste tutorial não exploraremos a API de renderização.

3.2.5 Modelo de conversão

Quando um componente é mapeado a um objeto, a aplicação passa a ter *duas visões dos dados*. A visão do modelo (*Model View*) onde os dados são representados por *tipos* (int, long, objetos), e a visão de apresentação (*Presentation View*), onde os dados são apresentados em formato legível (ex: String).

Por exemplo, um objeto `Produto` pode ser um objeto com *preço*, *nome*, *descrição*, *código*, *fornecedor*, etc. (*Model*) mas aparecer em uma combo representado apenas pelo seu nome ou código (*Presentation*). Quando um usuário selecionar o produto ou código, a seleção precisará ser convertida de volta em um objeto `Produto`. O modelo de conversão do JSF pode cuidar dessa conversão transparentemente.

A conversão para tipos primitivos e tipos básicos do Java (como `BigDecimal`) é automática. Para converter outros tipos é preciso implementar e registrar um Converter. Tipos comuns, como Date, Number, moeda, etc. têm conversores nativos e internacionalizados que podem ser reusados, e a API permite criar conversores customizados através da implementação de uma interface.

A conversão de dados ocorre em duas fases do ciclo de vida do JSF. Na fase de processamento de validação (3) ocorre a conversão dos dados para produzir uma representação em forma de objeto, que possa ser incluída na árvore que será processada. Na fase de renderização da resposta (6) o conversor é usado para converter o objeto em uma representação String.

3.2.6 Modelo de validação

A validação dos dados em uma fase específica (fase 3 - processamento de validação) *antes* do modelo de dados ser atualizado. A validação pode ter vários níveis de detalhamento. Pode ser

apenas uma validação de campo obrigatório (configurada simplesmente incluindo um atributo `required="true"` no componente), ou um algoritmo mais complexo que envolve várias etapas.

Pode-se usar validadores existentes, para ou criar validadores customizados. Os existentes geralmente suportam testes de limites (em números, listas), comprimentos (strings), valor nulo e expressões regulares (que permitem uma validação muito detalhada). É possível também usar Bean Validation, que fornece validação declarativa (nativa em CDI).

Validadores customizados, para tarefas mais complexas (ex: validação com XML Schema) podem ser criados ou através da implementação da interface `Validator`, ou através da criação de um método de validação no próprio bean, que será identificado como validador ao ser chamado em um atributo.

3.2.7 Modelo de navegação

A navegação consiste de regras usadas para escolher a próxima pagina a ser mostrada (`outcome`). Pode ser implícita – quando regras de navegação não são configuradas e o sistema usa `defaults`, ou explícita – quando regras são declaradas em `faces-config.xml`.

O `NavigationHandler` é responsável por localizar a próxima view a ser exibida com base nas regras declaradas. Se nenhuma regra combinar com o resultado, a mesma View é exibida. Geralmente um processamento terminará com a chamada de um método de ação HTTP (no envio do formulário) que devolve um string. O string devolvido produz um identificador implícitos de navegação, que será usado para localizar a URL da próxima View a ser exibida. Retornar `null` é a forma padrão de provocar a recarga da mesma página.

Saber a próxima View a ser exibida é necessário para renderizar uma resposta de uma nova View, portanto o processamento da ação é realizada na fase de execução da aplicação (5).

3.3 Tipos de requisições JSF

Uma aplicação JSF distingue as requisições HTTP que são direcionadas a aplicações JSF e respostas produzidas por aplicações JSF. Essas requisições e respostas são chamadas de *Faces Requests* e *Faces Responses*. Elas são interceptadas pelo *Faces Runtime* e tratadas de forma diferenciada. O Faces Runtime processa dois tipos de Faces Request/Response:

- *Faces Resource Request/Response* (para transferir imagens, CSS, etc.)
- *Faces Request/Response* (para processar uma página JSF)

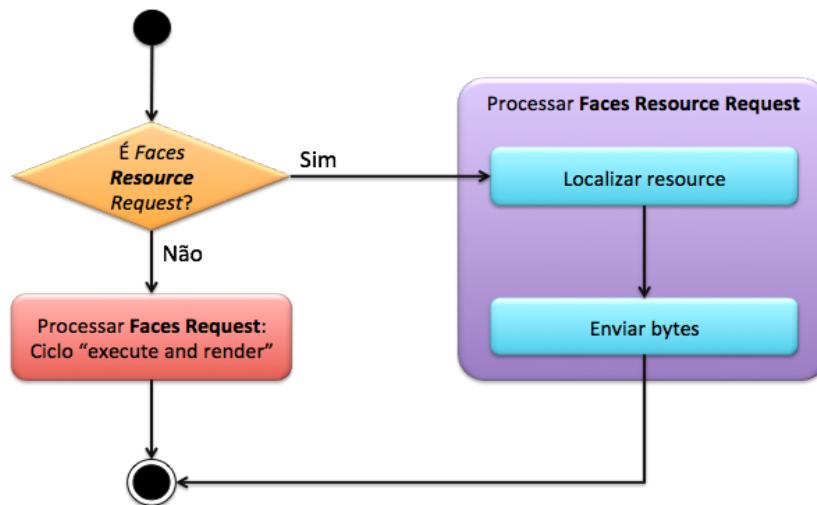
Uma requisição HTTP envolvendo JSF pode incluir, portanto, *seis* diferentes tipos de requisições:

- *Faces Response (FRs)*: Resposta criada pela execução da fase Render Response
- *Faces Request (FRq)*: Requisição iniciada a partir de uma FRs prévia
- *Faces Resource Request (FRRq)*: um FRq para um resource (imagem, CSS, etc.)
- *Non-Faces Request (NFRq)*: Requisição não iniciada a partir de FRs prévia
- *Non-Faces Response (NFRs)*: Resposta que não passou pelo Faces Runtime, e
- *Faces Resource Response (FRRs)*: um NFRs para um resource iniciado por um FRRq

Cenários *relevantes* para JSF são as respostas iniciadas por um FRq e a requisição inicial (que é um NFRq). São, portanto, três os cenários relevantes:

- NFRq gerando FRs (requisição inicial)
- FRq gerando FRs (requisição que causa ciclo execute & render)
- FRq (FRRq) gerando NFRs (FRRs)

Apenas um dos Faces request participa do ciclo de vida que é responsável pela renderização gráfica da página. A ilustração abaixo mostra como o Faces Runtime distingue os dois.



3.4 Ciclo de vida do JSF

O ciclo de vida (chamado de *Execute & Render*) processa todos os Faces Requests e realiza o tratamento de conversão, eventos, renderização, etc. em fases distintas da requisição. Existem *seis* fases. Os nomes abaixo em maiúsculas são as constantes *default* da classe *PhaseId*:

1. **RESTORE_VIEW** (restaurar a view) - na primeira requisição não faz nada, nas requisições seguintes, recupera o *UIViewRoot* com o estado da View armazenado no *FacesContext*.
2. **APPLY_REQUEST_VALUES** (aplicar valores da requisição) - copia os valores da requisição de cada componente para um local temporário (*submittedValue*).
3. **PROCESS_VALIDATIONS** (processar validações) - processa validações e conversões, e se obtiver sucesso, copia os valores do local temporário para o componente (*value*).
4. **UPDATE_MODEL_VALUES** (atualizar valores do modelo) - copia os valores validados e convertidos para o modelo (*managed beans*)
5. **INVOKER_APPLICATION** (chamar aplicação) - executa os métodos de ação HTTP e escolhe a próxima View.
6. **RENDER_RESPONSE** (renderizar a resposta) - constrói um *UIViewRoot* com os componentes atualizados e gera o HTML da nova View.

Suponha que a View seja o XHTML seguinte:

```

<h:form id="formulario">
    <h:panelGrid columns="3">
        <h:outputText value="Mensagem" />
        <h:inputText value="#{mensagemBean.mensagem}" id="msg" required="true">
            <f:converter converterId="mensagemConverter" />
            <f:validator validatorId="mensagemValidator" />
            <f:valueChangeListener type="br...MensagemValueChangeListener"/>
        </h:inputText>
        <h:message for="msg" style="color: red;" />
    </h:panelGrid>
    <h:commandButton action="#{mensagemBean.processar}" value="Processar" />
    <h:commandButton action="index" value="Cancelar" immediate="true"/>
</h:form>
  
```

Este é o managed bean usado:

```

@Named
@SessionScoped
public class MensagemBean implements Serializable {

    private Mensagem mensagem;

    public Mensagem getMensagem() {
        return mensagem;
    }

    public void setMensagem(Mensagem mensagem) {
        this.mensagem = mensagem;
    }

    public String processarMensagem() {
        // processar a mensagem
        return "resultado";
    }
}

```

Na primeira requisição, para mostrar a view, haverá apenas duas fases, e nada acontece de especial na primeira fase, já que não há view ainda para ser restaurada. As fases são:

1. RESTORE_VIEW
6. RENDER_RESPONSE: gera uma árvore de componentes UIViewRoot, que contém um UIForm contendo um UIInput (campo de entrada) e UICommand (botão).

Com a página mostrada no browser, digitamos um texto qualquer (ex: “teste”) no campo de entrada e apertamos o botão. Desta vez as fases serão:

1. RESTORE_VIEW: os dados do formulário são usados para reconstruir a *UIViewRoot*. Os valores do *UIInput* (que está mapeado ao campo de entrada que contém o texto “teste”) ainda não recebeu nada. O valor armazenado no objeto Mensagem no bean é *null*.
2. APPLY_REQUEST_VALUES: O texto digitado é usado para atualizar o valor do método *setSubmittedValue()* do *UIInput* da árvore de componentes. O valor armazenado no objeto Mensagem no bean ainda é *null*.
3. PROCESS_VALIDATIONS: O texto passa pelo conversor (método *getAsObject*) e pelo validador. Como o texto digitado não causa erro de validação, ele é transferido para o *setValue()* do *UIInput*; como o valor novo (“teste”) é diferente do anterior (*null*), um evento *ValueChange* é disparado. O valor armazenado no objeto Mensagem no bean ainda é *null*.
4. UPDATE_MODEL_VALUES: os valores válidos e convertidos da árvore de componentes são finalmente usados para atualizar os atributos do managed bean (O valor armazenado no objeto Mensagem no bean finalmente recebe o valor que estava em *UIInput.getValue()*).
5. INVOKE_APPLICATION: Agora que o bean está atualizado, seus dados podem ser usados por outras aplicações. O método *processarMensagem()* configurado como *action* do formulário é executado, e a View de resposta é selecionada.
6. RENDER_RESPONSE: Os dados a serem exibidos são obtidos do bean (*getMensagem()*), convertidos em string (chama *getAsString()* no conversor) e usados para montar a árvore de componentes *UIViewRoot* que irá gerar o HTML da resposta.

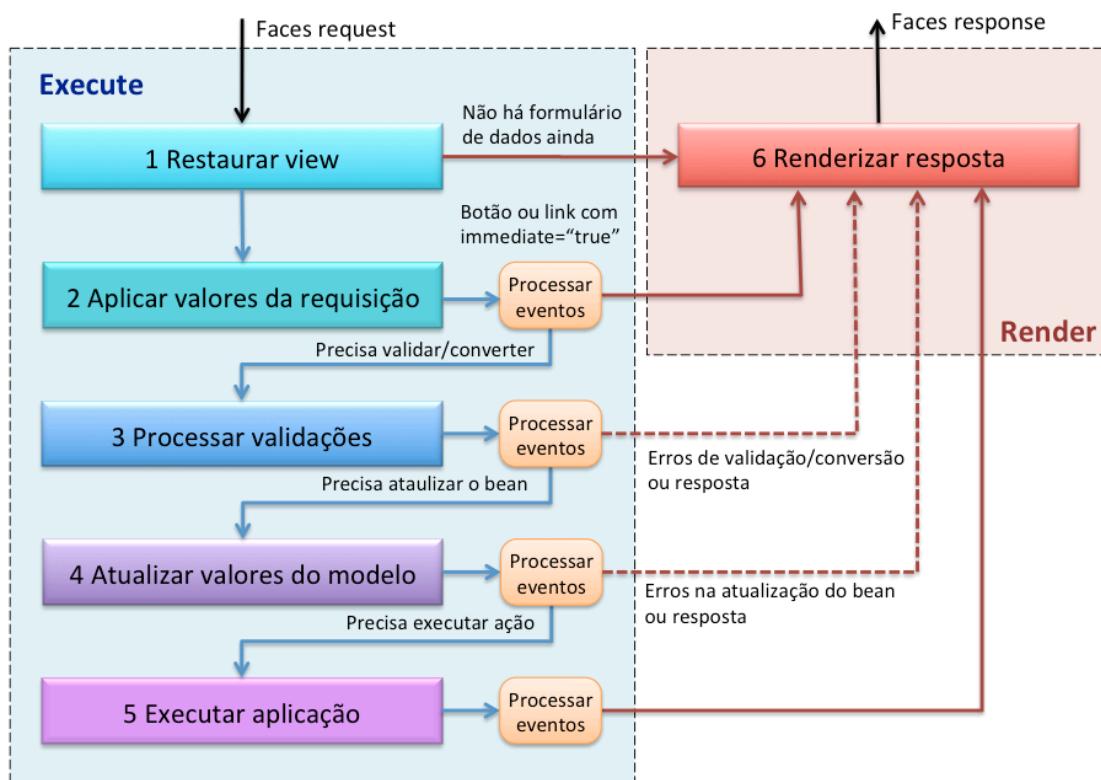
Se houver erros de validação ou de conversão (na etapa 3), *as etapas 4 e 5 serão puladas*, e o controle pula para a etapa 6, onde a resposta será renderizada, possivelmente com dados de mensagem (FacesMessage) de erro incluídas na etapa 3 que serão exibidas no lugar do componente <h:message>. Portanto as etapas executadas serão:

1. RESTORE_VIEW
2. APPLY_REQUEST_VALUES

3. PROCESS_VALIDATIONS: Neste ponto, ocorre um erro de conversão ou validação. Uma exceção é lançada e FacesMessage são criadas. O controle pula para a última etapa. `UIInput.getValue()` continua `null`, e `UIInput.getSubmittedValue()` continua com o valor enviado.
4. RENDER_RESPONSE: O formulário é exibido com uma FacesMessage contendo a mensagem de erro.

Expressões EL são executadas *imediatamente* (na etapa 2) se forem do tipo \${}. Se forem do tipo #{} a execução ocorre em várias outras fases. O uso do atributo `immediate="true"` pode fazer o JSF pular etapas (se for usado em um botão ou link) ou priorizar o processamento (se for usado em um componente que guarda valor). No exemplo mostrado, o botão Cancelar tem o atributo `immediate="true"`. Apertá-lo irá fazer o controle pular as etapas 3, 4 e 5.

A imagem abaixo ilustra o ciclo de vida completo do JSF (sem levar em conta uso do atributo `immediate` em componentes de entrada de dados), mostrando como o fluxo da requisição e da resposta passa por todas as seis fases.



3.4.1 Monitoração do ciclo de vida

A depuração do ciclo de vida de uma aplicação JSF pode ser feita com um `PhaseListener`, que captura os eventos do Faces Runtime e permite executar código antes e depois de cada fase. Rode os exemplos do projeto `jsf-architecture` que monitora os eventos de fase. Um exemplo de `PhaseListener` é mostrado no capítulo sobre listeners.

4 Facelets e componentes HTML

4.1 O que são Facelets?

JSF fornece uma arquitetura que mapeia tags, componentes e renderizadores de interface do usuário. Os tags usados em aplicações JSF compõem uma linguagem de declaração de páginas (*View Declaration Language*) através da qual se pode montar árvores de componentes declarativamente. Essa árvore é posteriormente renderizada e transformada em uma tela ou página Web.

Esta linguagem é chamada de *Facelets*. Trata-se de uma aplicação do XML similar ao XHTML que serve de template para gerar páginas XHTML dinamicamente, e que permite combinar tags nativos do XHTML com tags de outras bibliotecas, prefixados, agrupados em coleções identificadas por um *namespace*. Alguns tags representam componentes gráficos em que são renderizados em HTML, outros representam transformações que geram código dinamicamente (loops, condicionais) e outras encapsulam código JavaScript e cabeçalhos HTTP. É uma linguagem extensível. Pode-se ainda criar facelets para renderizar outras coisas como JavaScript, SVG, XML, etc.

O código abaixo, similar aos exemplos mostrados e executados na seção anterior, é um documento XHTML usando facelets que geram elementos HTML. Observe os diferentes prefixos usados e os atributos contendo expressões EL:

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

<h:head></h:head>
<body>
<h:form>
    <h1>Exemplos</h1>
    <h:panelGrid columns="2" columnClasses="pre-col1,col2,col3">
        <h:selectOneListbox id="sol" value="#{bean.filme}">
            <f:selectItems value="#{cinemateca.filmes}" var="filme"
                           itemLabel="#{filme.titulo}" itemValue="#{filme.imdb}" />
        </h:selectOneListbox>
        <h:message for="sol" />
    </h:panelGrid>
    <h:commandButton value="Enviar" action="#{bean.processar}" />
</h:form>
</body>
</html>
```

Este é o código-fonte do HTML final gerado no browser:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="j_idt2"></head>
<body>
<form id="j_idt4" name="j_idt4"
      method="post"
      action="/jsf-facelets/facelets-example.faces"
      enctype="application/x-www-form-urlencoded">

    <input type="hidden" name="j_idt4" value="j_idt4" />
    <h1>Exemplos</h1><table>
    <tbody>
    <tr>
        <td class="pre-col1"><select id="j_idt4:sol" name="j_idt4:sol" size="6">
```

```

<option value="tt0062622">2001: A Space Odyssey</option>
<option value="tt0013442">Nosferatu, eine Symphonie des Grauens</option>
<option value="tt1937390">Nymphomaniac</option>
<option value="tt1527186">Melancolia</option>
<option value="tt0113083">La Flor de mi Secreto</option>
<option value="tt0101765">La double vie de Véronique</option>
</select></td>
<td class="col2"></td>
</tr>
</tbody></table>
<input type="submit" name="j_idt4:j_idt13" value="Enviar" />
<input type="hidden" name="javax.faces.ViewState"
       id="j_id1:javax.faces.ViewState:0"
       value="2302106755987392480:3479051282761317321" autocomplete="off" />
</form>
</body>

```

Através de facelets é possível executar expressões (na linguagem *EL - Expression Language*) para gerar strings, números e outros valores, transferir dados entre componentes e a página, realizar mapeamento (*binding*) entre tags, páginas e componentes, construir e reusar componentes. As principais bibliotecas de tags nativamente suportadas e seus namespaces com os prefixos recomendados estão listadas abaixo:

- Templating: *xmlns:ui="http://java.sun.com/jsf/facelets"*
Exemplos: ui:component, ui:insert, ui:repeat
- HTML: *xmlns:h="http://java.sun.com/jsf/html"*
Exemplos: h:head, h:body, h:outputText, h:inputText
- Core library JSTL1.2: *xmlns:c="http://java.sun.com/jsp/jstl/core"*
Exemplos: c:forEach, c:catch
- Core library do JSF: *xmlns:f="http://xmlns.jcp.org/jsf/core"*
Exemplos: f:actionListener, f:attribute, f:ajax
- Funções JSTL1.2: *xmlns:fn="http://java.sun.com/jsp/jstl/functions"*.
Exemplos: fn:toUpperCase, fn:toLowerCase
- Passthrough (suporte a atributos não suportados e HTML5):
xmlns:a="http://xmlns.jcp.org/jsf/passthrough"

O namespace *default* (target namespace da página XML que afeta todos os elementos que não têm prefixo) deve ser declarado da forma *xmlns="http://www.w3.org/1999/xhtml"*. Todos os namespaces devem ser declarados no elemento raiz *<html>*.

Outros namespaces podem ser declarados para suporte a extensões de bibliotecas de componentes que não fazem parte do JSF, como componentes customizados ou bibliotecas de terceiros como PrimeFaces.

Só é necessário declarar os namespaces dos tags que forem usados. Declará-los desnecessariamente não afeta a performance (apenas acrescenta alguns caracteres a mais na página).

Tags do HTML, SVG e XML *podem* ser usados em páginas JSF. Eles apenas não participam do ciclo de vida e são copiados diretamente para a saída como texto. Não é necessário usar templating para tudo. Pode-se usar *<body>* em vez de *<h:body>*, ** em vez de *<h:graphicImage>* e até mesmo *<form>* ou *<input>* em uma página se o templating do JSF for indesejado, mas é boa prática usar esses tags nas páginas que serão processadas pelo runtime do JSF. Declarar *<h:head>*, mesmo que vazio é obrigatório em páginas onde há geração de JavaScript ou CSS, como páginas que usam Ajax, já que a geração de JavaScript procura o *<h:head>* para inserir código.

4.2 Componentes e tags que geram HTML

As páginas HTML são os principais componentes da camada de apresentação de aplicações Web. Uma pagina de uma aplicação Web típica de aplicações JSF inclui:

- Um conjunto de declarações de namespace no elemento raiz (`<html>`)
- Elementos `<h:head>` e `<h:body>`
- Um elemento `<h:form>` que contém componentes de entrada de dados

No mínimo é preciso declarar o namespace raiz do XHTML

```
<html xmlns="http://www.w3.org/1999/xhtml" >
```

e a biblioteca HTML padrão (prefixo “h”).

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

A maior parte das páginas também irá precisar do JSF Core Library (normalmente usada com prefixo “f”):

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

A tabela abaixo relaciona os tags da biblioteca HTML e os tags ou efeitos gerados na página HTML final.

Tag JSF	O que gera em HTML
<code>h:form</code>	<code><form></code>
<code>h:commandButton</code>	<code><input type="submit reset image"></code>
<code>h:button</code>	<code><input type="button"></code>
<code>h:commandLink</code>	<code></code>
<code>h:outputLink</code>	<code></code>
<code>h:link</code>	<code></code>
<code>h:dataTable</code>	<code><table></code>
<code>h:column</code>	Uma coluna em uma tabela HTML
<code>h:graphicImage</code>	<code></code>
<code>h:inputFile</code>	<code><input type="file"></code>
<code>h:inputHidden</code>	<code><input type="hidden"></code>
<code>h:inputSecret</code>	<code><input type="password"></code>
<code>h:inputText</code>	<code><input type="text"></code>
<code>h:inputTextarea</code>	<code><textarea></code>
<code>h:message e h:messages</code>	Um ou mais <code></code> se estilos forem usados
<code>h:outputFormat</code>	Texto (ou <code>Texto</code> se houver ID)
<code>h:outputLabel</code>	<code><label></code>
<code>h:outputText</code>	Texto (ou <code>Texto</code> se houver ID)

<code>h:panelGrid</code>	<code><table><tr><td>...</td></tr></table></code>
<code>h:panelGroup</code>	<code><div></code>
<code>h:selectBooleanCheckbox</code>	<code><input type="checkbox"></code>
<code>h:selectManyCheckbox</code>	<code><input type="checkbox"></code>
<code>h:selectManyListbox</code>	<code><select></code>
<code>h:selectManyMenu</code>	<code><select></code>
<code>h:selectOneListbox</code>	<code><select></code>
<code>h:selectOneMenu</code>	<code><select></code>
<code>h:selectOneRadio</code>	<code><input type="checkbox"></code>

Os nomes e funções dos atributos nos tags de facelets geralmente possuem muitas semelhanças com os tags HTML que eles geram. Muitas vezes têm o mesmo nome e funcionam de forma idêntica. Às vezes é possível descobrir o tag HTML associado pela semelhança do nome, que pode ser igual ou parecido (ex: `<h:form>` e `<form>`). Mas os componentes às vezes têm funções completamente diferentes e os atributos usados em JSF muitas vezes não existem em HTML.

4.3 Atributos

Cada tag tem atributos próprios, e vários componentes compartilham atributos comuns, descritos brevemente a seguir (para maiores detalhes consulte a documentação).

- **id** – identificação do componente (usado por vários outros atributos JSF); o JSF poderá gerar um ID diferente no HTML resultante.
- **immediate** – se *true*, eventos, validação e conversão devem ocorrer tão logo parâmetros de requisição sejam aplicados (dependendo do componente e do resultado da validação, isto poderá fazer com que etapas do ciclo de vida sejam puladas). Veja detalhes na seção sobre arquitetura e ciclo de vida.
- **rendered** – renderizado; se *true*, componente é exibido na tela; se *false*, ele é removido da árvore de componentes e não é exibido (similar a CSS `display`).
- **style** – contém regras de estilo CSS (mesmo comportamento que `style` em HTML)
- **styleClass** – define uma classe CSS (mesmo comportamento que `class` em HTML)
- **value** – define o valor do componente (o tipo de dados depende do componente e da forma como é usado)
- **binding** – mapeia uma instância a uma propriedade de bean

4.3.1 Propriedades binding e rendered

Os tags são mapeados a componentes que são instanciados como objetos Java. As classes dos componentes podem ser declaradas como membros de um managed bean para que se possa estabelecer mapeamentos (**binding**) entre suas propriedades e o tag. Por exemplo, um bean pode declarar um atributo `UIOutput`:

```
@Named("mybean")
public class MyBean {
    private UIOutput output;
    ...
}
```

Ele pode ser mapeado a um elemento da página usando o atributo *binding* e uma expressão em EL que associa o atributo declarado na classe do bean um tag <h:outputText>:

```
<h:outputText binding="#{mybean.output}" />
```

Agora, o bean poderá definir o texto que será impresso na página através da propriedade value do componente:

```
public setOutput(UIOutput output) {
    this.output = output;
    this.output.setValue("Texto inserido via binding");
}
```

Nem sempre esse tipo de binding é necessário ou desejável, sendo mais comum e eficiente mapear apenas os valores de entrada e saída (*value*) utilizados, dos componentes/tags, a propriedades declaradas no bean. No exemplo acima, para apenas alterar o valor do componente, pode-se declarar um campo do tipo String no bean:

```
@Named("mybean")
public class MyBean {
    private String outputValue;
    ...
}
```

e mapear apenas o atributo *value* (e não o componente *UIOutput* inteiro):

```
<h:outputText value="#{mybean.outputValue}" />
```

Mas o *binding* é mais poderoso. Através dele pode-se mudar quaisquer propriedades do componente dinamicamente. Por exemplo, pode-se controlar quando exibir ou não um componente variando o valor da sua propriedade *rendered*, se ele estiver mapeado ao bean. Por exemplo, o bean poderia ter um método para controlar a exibição do texto:

```
public mostrarTexto(boolean mostrar) {
    this.output.setRendered(mostrar);
}
```

É possível também ter acesso aos valores mapeados em outras fases do ciclo de vida, antes deles serem validados, convertidos e transferidos para o bean, através dos métodos *getValue()* e *getSubmittedValue()* de *UIInput*.

4.4 Estrutura de uma página

O código abaixo mostra um template básico XHTML mínimo para uma página JSF que gera HTML:

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

    </h:body>
</html>
```

4.4.1 Bloco <h:form>

Grande parte das páginas JSF são formulários HTML. Um bloco <h:form> deve ser usado sempre que houver necessidade de entrada de dados, mas também pode incluir links (*commandLink*) e botões (*commandButton*) e outros elementos HTML e componentes que permite-se usar fora de um bloco <h:form>. Uma página pode ter vários <h:form> mas apenas um será enviado em cada requisição.

A menos que a página tenha apenas um pequeno formulário simples, não é considerado uma boa prática colocar todos os elementos dentro de um único `<h:form>` pai. Uma boa prática é agrupar em um cada `<h:form>` apenas os elementos necessários para cumprir uma determinada responsabilidade de entrada de dados.

Aninhar blocos `<h:form>` é ilegal em HTML e JSF. É preciso tomar cuidado principalmente quando se usa templates e fragmentos de facelets que são inseridos dinamicamente nas páginas.

4.4.2 NamingContainer, atributo name e client ids

Um `<h:form>` possui vários atributos, mas todos são opcionais. É uma boa prática definir um ID explícitamente, já que o `<h:form>` pertence a um grupo de componentes que é derivado de *NamingContainer*. Componentes desse tipo redefinem o ID dos componentes filho. Portanto, o ID *local* do formulário será prefixado nos ids gerados para todos os seus componentes, por default. Uma sintaxe mínima recomendada é, portanto:

```
<h:form id="f1">
  ...
</h:form>
```

Todos os elementos de entrada de dados em HTML possuem um atributo *name*, que contém o nome do campo que será enviado através de parâmetros HTTP. Nos componentes UI do JSF não existe atributo *name*, e se for usado ele será *ignorado* e removido do HTML final. Deve-se sempre usar *id*, e no HTML final um atributo *name* com o mesmo conteúdo será *gerado*. Se um *id* não for informado, o JSF irá automaticamente *gerar* um ID e um nome para o componente. Os atributos *id* e *name* gerados são chamados de *Client ID*, e têm um formato especial recursivo:

```
client-id-do-elemento-pai:id-local-do-componente
```

Por exemplo, se o formulário estiver na raiz da página, seu ID (local) for “*f1*”, e ele contiver um componente declarado com ID local de “*comp5*” em JSF:

```
<h:form id="f1">
  ...
    <h:inputText id="comp5" .../>
  ...

```

No HTML gerado, o *client ID* desse componente será gerado pelo JSF e usado nos atributos *id* e *name*. O HTML resultante é:

```
<input type="text" id="f1:comp5" name="f1:comp5" >
```

Se houver muitos componentes aninhados, o cliente-id pode ter vários componentes separados por “:”. Por exemplo, se o *form* estiver dentro de outro *NamingContainer* com ID local *c1*, o client-id do *inputText* mudará para:

```
c1:f1:comp5
```

E assim por diante.

Em facelets XHTML, entre componentes localizados dentro de um mesmo *form* (ou qualquer *NamingContainer*), pode-se usar o ID ou o Client ID para referenciá-los (ex: *f1:comp* ou *comp*), mas se o componente referenciado estiver em outro escopo, é preciso usar o client ID (ex: *f2:comp*).

Se um elemento não estiver contido em um *NamingContainer*, ele tanto pode ser localizado por seu ID local (ex: *form1*), como por seu client ID (*:form1*). Um elemento com client ID começando com “:” não pertence a nenhum *NamingContainer*.

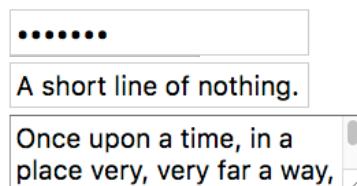
É importante saber desse comportamento porque isto afeta a integração de aplicações JSF com outras aplicações lado-servidor (que usem nomes dos parâmetros da requisição HTTP), bibliotecas JavaScript e até mesmo folhas de estilo CSS que accessem elementos pelo seu ID.

Existem estratégias para impedir esse comportamento, mas todas elas afetam o funcionamento do JSF de alguma maneira. Por exemplo, é possível desligar esse comportamento em um `<h:form>` usando o atributo `prependId="false"`, porém isto impedirá que código JavaScript gerado pelo JSF funcione (`<f:ajax>`, por exemplo, não irá funcionar).

4.4.3 Componentes para entrada e saída de texto

Dentro de um formulário podem ser usados vários diferentes componentes de entrada de texto. Existem quatro tipos. Geram elementos `<input>` ou `<textarea>` em HTML:

- `<h:inputHidden>` - campo oculto `<input type="hidden">`
- `<h:inputSecret>` - campo de senha `<input type="password">`
- `<h:inputText>` - campo de entrada de texto `<input type="text">`
- `<h:inputTextArea>` - campo multilinha - `<textarea>`



Atributos comuns a todos os input tags são: `converter`, `converterMessage`, `label`, `required`, `requiredMessage`, `validator`, `validatorMessage`, `valueChangeListener`.

A maioria são auto-explicativos e referem-se a configurações conversores, validadores e listeners, e suas mensagens de erro. Serão detalhados mais adiante.

Existem também vários componentes usados para a geração de texto *somente leitura*, que podem ser mapeados a propriedades de um managed bean:

- `<h:outputText>` - Gera uma linha de texto (sem ID não gera nenhum elemento HTML; com ID inclui o texto gerado em um ``)
- `<h:outputFormat>` - Um `<h:outputText>` que gera uma linha de texto que pode ser formatada e parametrizada.

```
<h:outputFormat value="Hello, {0}!">
    <f:param value="#{hello.name}" />
</h:outputFormat>
```

- `<h:outputLink>` - gera um `<a href>` para outra página sem causar um action event no clique (ex: para links externos ou que geram non-faces request)
- `<h:outputLabel>` - exibe texto apenas leitura e deve ser associado a um componente usando o atributo `for` – gera um `<label>`.

Exemplos:

```
<h:inputTextarea id="description" value="#{bean.description}" />

<h:inputText id="input" value="#{bean.input}">
    <f:validateLength minimum="5" />
</h:inputText>

<h:outputText value="#{listings.outputFormatCode}" escape="true" />

<h:outputFormat value="O número da sorte de hoje é {1} e o de ontem foi {0}.">
    <f:param value="#{bean.sorte}" />
    <f:param value="#{bean.jogar()}" />
</h:outputFormat>
```

4.4.4 Ações e navegação

Componentes de comando realizam uma ação quando ativados. Os dois elementos abaixo são mapeados a *UICommand*:

- **<h:commandButton>** - gera um *<input type="submit">*
- **<h:commandLink>** - gera um *<a href>*

Podem usar os seguintes atributos (além dos comuns a todos os elementos)

- **action** – executa um método de ação HTTP que retorna um string ou contém um string. O string é usado para identificar a próxima página a acessar.
- **actionListener** – expressão que aponta para o método que processa um evento de clique (*ActionEvent*).

```
<h:commandLink value="Página principal" action="#{bean.goHome()}" />

<h:commandButton action="index" value="Página principal">
    <f:actionListener type="a.b.c.ExitListener" />
</h:commandLink>
```

4.4.5 Botões e links que não disparam eventos de action

Nem todo botão é um *UICommand*. Alguns links e botões são *UIInput*. *<h:link>* e *<h:outputLink>* também geram *<a href>* só que bem mais simples e que não estão automaticamente associados a uma ação HTTP. *<h:button>* gera um botão que não faz o submit do formulário.

O código abaixo mostra como usá-los para redirecionar para uma página index.faces. Observe o uso diferente do atributo *value*:

```
<h:link value="Página principal" outcome="index" />

<h:outputLink value="index.faces">Página principal</h:outputLink>

<h:button value="Página principal" outcome="index" />
```

Na verdade, *<h:outputLink>* é mais antigo. Em *<h:link>* e *<h:button>* o atributo *outcome* funciona de forma similar ao atributo *action* de *commandLink/commandButton*. Esses links e botões podem ser usados fora de *<h:form>*.

Todos os links também aceitam parâmetros passados como elementos-filho *<f:param>*. O exemplo abaixo ilustra um link com parâmetros. Se a opção *includeViewParams* for *true*, os parâmetros passados da requisição da URL atual serão incluídos no link. O exemplo abaixo mostra como passar para outra página os parâmetros de uma requisição:

```
<h:link outcome="segunda" value="Clique para continuar"
    includeViewParams="true">
    <f:param name="nome" value="#{bean.nome}" />
</h:link>
```

4.4.6 Gráficos e imagens

Para gerar um ** dentro de um Faces Request usa-se *<h:graphicImage>*. A imagem pode ser referenciada usando os atributos

- **url** – caminho absoluto (relativo ao contexto) para a imagem.

```
<h:graphicImage url="resource/imagens/bg.jpg" />
```

- **library e name** – *library* é o nome de uma pasta dentro de resources (library) e *name* é o nome do arquivo dentro dessa pasta.

```
<h:graphicImage library="imagens" name="bg.jpg" />
```

- **value** – uma expressão EL usando objeto implícito *resource* e chave *library:name*.

```
<h:graphicImage value="#{resource['imagens:bg.jpg']}"/>
```

Esta sintaxe pode ser usada em código CSS:

```
h1 {background: url(#{resource['imagens:bg.jpg']}) repeat-x; }
```

4.4.7 Seleção simples e múltipla

Há quatro elementos para selecionar *um* item de uma lista:

- **<h:selectOneMenu>** - gera <select> e <option> em menu drop-down
- **<h:selectBooleanCheckbox>** - gera <checkbox>
- **<h:selectOneListBox>** - gera <select> e <option> em lista
- **<h:selectOneRadio>** - gera <radio>

Há três componentes para selecionar múltiplos itens de uma lista:

- **<h:selectManyCheckbox>** – que gera <checkbox>
- **<h:selectManyMenu>** – que gera <select> e <option>
- **<h:selectManyListbox>** – que gera <select> e <option>

O **value** do componente guarda o elemento selecionado apenas *quando o formulário for enviado* (fase 5: chamada da aplicação). A lista é povoada por elementos **<f:selectItem>** ou **<f:selectItems>**. Este último recebe no seu atributo *value* uma *coleção* de objetos, que define as opções da lista que pode ser fornecida pelo bean.

Exemplos de cada componente estão ilustrados abaixo com o código usado para criá-los. O bean usado no exemplo fornece, na propriedade *filmes*, um array de objetos do tipo *Filme*, que possui a seguinte estrutura:

```
public class Filme implements Serializable {
    private String imdb;
    private String titulo;
    private String diretor;
    private int ano;
    private int duracao; ...
}
```

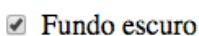
1) **<h:selectOneMenu>**



```
<h:selectOneMenu value="#{bean.filme}">
    <f:selectItems value="#{cinemateca.filmes}"
        var="filme"
        itemLabel="#{filme.titulo}"
        itemValue="#{filme.imdb}" />
</h:selectOneMenu>
```

coleção ou array de itens a exibir variável para acessar itens individuais texto que será exibido valor que será enviado

2) **<h:selectBooleanCheckbox>**



```
<h:selectBooleanCheckbox value="false"/> Fundo escuro
```

3) <h:selectOneListbox>

2001: A Space Odyssey
Nosferatu, eine Symphonie des Grauens
Nymphomaniac
Melancolia
La Flor de mi Secreto
La double vie de Véronique

```
<h:selectOneListbox id="sol" value="#{bean.filme}">
    <f:selectItems value="#{cinemateca.filmes}"
        var="filme"
        itemLabel="#{filme.titulo}"
        itemValue="#{filme.imdb}" />
</h:selectOneListbox>
```

4) <h:selectOneRadio>

- 2001: A Space Odyssey
- Nosferatu, eine Symphonie des Grauens
- Nymphomaniac
- Melancolia
- La Flor de mi Secreto
- La double vie de Véronique

```
<h:selectOneRadio id="sor" value="#{bean.filme}" layout="pageDirection">
    <f:selectItems value="#{cinemateca.filmes}"
        var="filme"
        itemLabel="#{filme.titulo}"
        itemValue="#{filme.imdb}" />
</h:selectOneRadio>
```

5) <h:selectManyCheckbox>

- manha tarde noite

```
<h:selectManyCheckbox value="#{bean.turnos}">
    <f:selectItem itemValue="1" itemLabel="manha" />
    <f:selectItem itemValue="2" itemLabel="tarde" />
    <f:selectItem itemValue="3" itemLabel="noite" />
</h:selectManyCheckbox>
```

6) <h:selectManyListbox>

2001: A Space Odyssey
Nosferatu, eine Symphonie des Grauens
Nymphomaniac
Melancolia
La Flor de mi Secreto
La double vie de Véronique

```
<h:selectManyListbox value="#{bean.selecaoFilmes}">
    <f:selectItems value="#{cinemateca.filmes}"
        var="filme"
        itemLabel="#{filme.titulo}"
        itemValue="#{filme.imdb}" />
</h:selectManyListbox>
```

7) <h:selectManyMenu>

manha
tarde
noite

```
<h:selectManyMenu value="#{bean.turnos}">
    <f:selectItem itemValue="1" itemLabel="manha" />
```

```
<f:selectItem itemValue="2" itemLabel="tarde" />
<f:selectItem itemValue="3" itemLabel="noite" />
</h:selectManyMenu>
```

4.4.8 Layout e tabelas

O elementos **<h:panelGrid>** e **<h:panelGroup>** são usados para realizar layout simples em forma de tabela. **<h:panelGrid>** organiza uma tabela considerando cada bloco de texto ou bloco HTML como uma célula e encaixando no número de colunas declarado **<h:panelGrid columns="número de colunas">**.

Se houver necessidade de tratar vários blocos de texto ou HTML como uma única célula, pode-se usar **<h:panelGroup>** para agrupar vários componentes em uma única célula.

1.1	1.2	1.3.a 1.3.b 1.3.c
2.1	2.2	2.3

```
<h:panelGrid columns="3">
    <h:inputText value="1.1" />
    <h:inputText value="1.2" />
    <h:panelGroup>
        <h:panelGrid>
            <h:inputText value="1.3.a" />
            <h:inputText value="1.3.b" />
            <h:inputText value="1.3.c" />
        </h:panelGrid>
    </h:panelGroup>

    <h:inputText value="2.1" />
    <h:inputText value="2.2" />
    <h:inputText value="2.3" />
</h:panelGrid>
```

Para apresentar uma coleção de itens na forma de uma tabela de dados pode-se usar **<h:dataTable>** que realiza *binding* com uma coleção de objetos (*array*, *List*, *ResultSet*, *DataModel*) e apresenta os dados como uma tabela HTML. A coleção deve ser passada ao atributo *value*, e os elementos individuais podem ser acessados através de uma variável declarada em *var*. Dentro da tabela os dados são agrupados em colunas com **<h:column>**

```
<h:dataTable value="#{cinemateca.filmes}" var="filme" rows="3" first="2">
    <h:column>#{filme.titulo}</h:column>
    <h:column>#{filme.diretor}</h:column>
    <h:column>#{filme.ano}</h:column>
    ...
</h:dataTable>
```

Para incluir cabeçalhos e legendas use **<f:facet>**. O componente **<h:dataTable>** reconhece as facetas *header*, *footer* e *caption* (cabeçalho, rodapé e legenda). Outros atributos de *h:dataTable* incluem:

- *border*, *bgcolor*, *cellpadding*, etc. (todos os atributos do elemento HTML *<table>*)
- **styleClass**, **captionClass**, **headerClass**, **footerClass**, **rowClasses**, **columnClasses** – classes CSS para diferentes partes da tabela
- **first** - Primeiro item da coleção a mostrar na tabela (para usar em paginação)
- **rows** - Número de linhas da coleção a exibir na tabela (para usar em paginação)

Exemplo (usando o mesmo bean dos exemplos anteriores):

Título	Diretor	Ano
2001: A Space Odyssey	Stanley Kubrick	1968
Nosferatu, eine Symphonie des Grauens	F. W. Murnau	1922
Nymphomaniac	Lars von Trier	2013
Melancolia	Lars von Trier	2011
La Flor de mi Secreto	Pedro Almodovar	1995
La double vie de Véronique	Krzysztof Kieslowski	1991

```
<h:dataTable value="#{cinemateca.filmes}" var="filme" rows="3" first="2">
  <h:column>
    <f:facet name="header">Título</f:facet>
    #{filme.titulo}
  </h:column>
  <h:column>
    <f:facet name="header">Diretor</f:facet>
    #{filme.diretor}
  </h:column>
  <h:column>
    <f:facet name="header">Ano</f:facet>
    #{filme.ano}
  </h:column>
</h:dataTable>
```

4.5 Core tags

Os core tags do JSF interagem com os tags de componentes gráficos acrescentando comportamentos diversos como tratamento de eventos, conversores, validadores, comunicação assíncrona, metadados e configuração. Alguns foram usados na seção anterior, para incluir itens em uma lista ou menu. Outros serão abordados em seções próprias. As tabelas abaixo listam os core tags mais importantes.

Todos são usados como *elementos-filho* de outros tags e afetam o contexto no qual foram inseridos (geralmente o elemento pai e outros filhos).

4.5.1 Tags para tratamento de eventos

É uma das formas de associar um handler de eventos ao componente pai (vários componentes permitem também fazer isto através de atributos). Todos possuem um atributo *type* que recebe uma expressão EL vinculando o tag a um método que irá processar o evento quando ele ocorrer.

f:actionListener	Adiciona um action listener, que reage a eventos de ação (ex: cliques)
f:valueChangeListener	Adiciona um value-change listener, que reage a eventos de mudança de valor
f:phaseListener	Adiciona um PhaseListener que monitora as fases do ciclo de vida do JSF
f:setPropertyActionListener	Registra um action listener que define uma propriedade no managed bean quando formulário é enviado
f:event	Registra um ComponentSystemEventListener em um componente

Estes tags serão explorados em maior detalhe no capítulo sobre Listeners.

4.5.2 Tags para conversão de dados

É uma das formas de associar um conversor de dados ao componente pai (vários componentes também permitem associar conversores através de atributos).

<code>f:converter</code>	Registra um conversor ao componente no qual este tag é incluído. O conversor deve ter sido declarado via metadados (anotações ou faces-config.xml) é identificado através do atributo <code>converterId</code> .
<code>f:convertDateTime</code>	Registra um conversor pra formatos de data e hora ao componente pai. Atributos são usados para configurar o formato.
<code>f:convertNumber</code>	Registra um conversor de número ao componente pai. Atributos são usados para configurar o formato.

Estes tags serão explorados em maiores detalhes no capítulo sobre Converters.

4.5.3 Tags para validação

É uma das formas de associar um validador ao componente pai (vários componentes também permitem associar validadores através de atributos, ou via *Bean Validation*).

<code>f:validateDoubleRange</code>	Adiciona um DoubleRangeValidator
<code>f:validateLength</code>	Adiciona um Length Validator
<code>f:validateLongRange</code>	Adiciona um LongRangeValidator
<code>f:validator</code>	Adiciona um validador customizado
<code>f:validateRegEx</code>	Adiciona um validador de expressão regular
<code>f:validateBean</code>	Delega a validação para um BeanValidator
<code>f:validateRequired</code>	Requer validação em um componente

Esses tags serão explorados em maiores detalhes no capítulo sobre Validators.

4.5.4 Outros tags

Alguns desses tags já foram usados na construção de componentes como listas, tabelas, links. Outros são usados para tratamento de parâmetros em requests HTTP, carregar bundles de mensagens e propriedades de configuração, configuração de requisições Ajax, etc. A seguir um resumo de cada um:

<code>f:facet</code>	Adiciona um componente aninhado que tem algum relacionamento com o componente pai (ex: em tabelas e usado para definir os cabeçalhos das colunas)
<code>f:param</code>	Substitui parâmetros e adiciona pares nome-valor a uma URL. (ex: usado em links para passar parâmetros para a URI gerada)
<code>f:selectItem</code>	Representa um item em uma lista (ex: usado em componentes de menu)

<code>f:selectItems</code>	Representa um conjunto de itens (ex: usado em componentes de menu)
<code>f:view</code>	Representa uma View. O tag, em JSF 2.2 é <i>opcional</i> (mas é inserida na árvore automaticamente por default). Pode ser incluída na página (em volta de <code><h:head></code> e <code><body></code>) para configurar defaults, como locale, encoding e tipo MIME.
<code>f:viewParam</code>	Permite o processamento de parâmetros da requisição HTTP GET (usado dentro de um bloco <code><f:metadata></code>)
<code>f:viewAction</code>	Permite associar uma operação diretamente a parâmetros recebidos em HTTP GET (geralmente usado para executar uma operação logo após a carga da página).
<code>f:metadata</code>	Registra uma faceta em um componente pai (frequentemente usado em processamento de parâmetros HTTP (<code><f:viewParam></code>) onde o componente pai é <code><h:head></code>)
<code>f:attribute</code>	Adiciona atributos a um componente (ex: pode ser usado para passar dados para listeners, conversores, etc.)
<code>f:loadBundle</code>	Carrega um resource bundle (properties) que é acessível como um Map
<code>f:ajax</code>	Associa uma ação Ajax com um componente ou grupo de componentes. Este elemento sera explorado em um capítulo específico.

4.6 Core tags do JSTL

Os core tags do JSTL são incluídos através do namespace `http://xmlns.jcp.org/jsp/jstl/core`.

É importante observar que eles apenas são processados quando a árvore de componentes está sendo montada (fase 1), em contraste com os facelets, que são processados na fase 6.. Nesta fase, dados que estão na página ainda não foram transferidos para os seus respectivos componentes e nem para o bean. Se uma expressão condicional depender desses valores, o resultado poderá ser diferente daquele esperado.

O resultado é mais fácil de prever se estes tags forem usados apenas junto com outros tags JSTL, e que apenas *leiam* dados. Em geral, há boas alternativas JSF. Há uma ótima discussão sobre diferenças entre tags JSTL e JSF em <http://stackoverflow.com/questions/3342984/jstl-in-jsf2-facelets-makes-sense>.

<code>c:if</code>	Tag condicional. Equivale a um if, mas não suporta um bloco complementar (default/else). Para situações desse tipo use <code><c:choose></code> . Para exibir/ocultar componentes em JSF é preferível usar expressões EL condicionais como valor para atributo rendered.
<code>c:choose</code> <code>c:when</code> <code>c:otherwise</code>	Tag condicional. Equivale a if-then-else ou switch. <code><choose></code> deve conter um ou mais blocos <code><when test=""></code> (equivalente a if-else/case) e opcionalmente um <code><otherwise></code> (equivalente a else/default).
<code>c:forEach</code>	Tag de repetição que ocorre na primeira fase do JSF (pode ser usado em JSF para dados estáticos já carregados antes). Alternativas JSF são <code><ui:repeat></code> , ou mesmo <code><h:dataTable></code> se o objetivo da repetição for gerar uma tabela.

c:set	Define uma propriedade em um escopo com base no valor dos atributos. Em JSF pode ser usado com valores estáticos que irão ser usados apenas na página (default). No mundo JSF o ideal é usar <ui:param>.
c:catch	Captura qualquer objeto Throwable (exceção) que ocorrer dentro do <c:catch>.

4.7 Tags de templating

A maior parte destes tags são usados principalmente na construção e uso de templates e componentes reutilizáveis. Este assunto será tratado em um capítulo a parte.

ui:component ui:fragment	Usados na construção de componentes. <ui:component> ignora todo o HTML fora do elemento.
ui:composition ui:decorate	Representam uma composição de elementos. Podem ser usados para construir templates reusáveis. <ui:composition> ignora todo o HTML fora do elemento.
ui:debug	Se incluído em uma página, a combinação Ctrl-Shift-D (default – pode ser alterada com o atributo hotkey) faz com que informações sobre a árvore de componentes seja mostrada em uma janela pop-up.
ui:define	Define parte de uma página que irá ser inserido em um ponto de inserção de um template.
ui:include	Inclui conteúdo de outra página.
ui:insert	Usado em templates para definir área de inserção.
ui:param	Usado em elementos <ui:include> para passar parâmetros para um template e para setar variáveis em um escopo. Pode também ser usado em um tag raiz (<f:view>) para definir uma constante para toda a view.
ui:repeat	Permite a repetição do seu conteúdo.
ui:remove	Remove o conteúdo da árvore de componentes. Pode ser usado para “comentar” código JSF, eliminando-o da árvore de componentes para que não seja processado nem gere HTML.

4.7.1 Repetição com ui:repeat

Embora a maior parte dos tags desta coleção em componentes e templates, <ui:repeat> é o principal tag usado para repetição em JSF. O ideal seria usar um componente, já que repetição é uma tarefa que deveria estar dentro de um componente reusável, mas como não há um componente similar a <h:dataTable> em JSF para outras estruturas HTML que se repetem, como listas , <ui:repeat> é uma alternativa muito melhor que <c:forEach>.

O trecho de código abaixo, usando <c:forEach> e <c:if>:

```
<ui:forEach items="#{[1,2,3,4,5,6,7,8]}" var="numero">
    <c:if test="#{numero mod 2 ne 0}">
        <li><h:outputText value="#{numero}" /></li>
    </c:if>
</ui:forEach>
```

Irá gerar os mesmos dados que o código abaixo, usando *ui:repeat* e *rendered*:

```
<ui:repeat value="#{[1,2,3,4,5,6,7,8]}" var="numero">
    <ui:fragment rendered="#{numero mod 2 ne 0}">
        <li><h:outputText value="#{numero}" /></li>
    </ui:fragment>
</ui:repeat>
```

E importante destacar a principal diferença entre *h:dataTable/ui:repeat* e *c:forEach*: os primeiros atuam sobre a *árvore de componentes* (objeto que é alterado durante seis fases do JSF), enquanto que o segundo atua sobre a própria página, gerando o HTML:

- Em JSF o tag *<h:outputText>* é reusado para construir uma coleção de 4 itens na memória que será usada para gerar 4 tags ** contendo números diferentes.
- Em JSTL, o *<h:outputText>* é copiado 4 vezes na página, e depois cada cópia gera 1 tag **. O resultado final é o mesmo, 4 tags ** (mas nem sempre é assim!)

Por exemplo, com base no funcionamento “esperado” de cada tag, poderíamos imaginar que não faria diferença usar um *c:if* dentro de um *ui:repeat*:

```
<ui:repeat value="#{[1,2,3,4,5,6,7,8]}" var="numero">
    <c:if test="#{numero mod 2 ne 0}">
        <li><h:outputText value="#{numero}" /></li>
    </c:if>
</ui:repeat>
```

Mas o que ocorre é que neste caso não é impresso nada! E se a condição for invertida, para:

```
#{{numero mod 2 eq 0}}"
```

Ele imprime todos! A condicional if aparentemente não está funcionando como se espera.

O problema acontece porque *<ui:repeat>* age na árvore de componentes, e reusa o *<h:outputText>*. Ele não faz cópias dele na página como em *<c:forEach>*. Assim, o *c:if* é executado apenas uma vez, e testando se *numero* é zero ou não.

5 Managed beans

Uma aplicação típica JSF contém um ou mais beans gerenciados pelo container, ou *managed beans*. Um managed bean pode ser um POJO configurado com anotações CDI ou um Session Bean (ou ainda um componente anotado com *@ManagedBean*, mas esta última forma está caindo em desuso e não é recomendada para projetos novos). Na arquitetura MVC, este bean é um *controller*, mas faz parte da camada de apresentação. Deve estar mais fortemente relacionado com a interface gráfica de comunicação com o usuário que com os componentes do modelo de domínio da aplicação.

Cada managed bean pode ser associado com os componentes usados em uma página através de mapeamento de valores ou *binding* do componente. Um managed bean segue as regras gerais para a construção de Java Beans:

- Construtor sem argumentos
- Estado na forma de *propriedades* (acessíveis via métodos get/set)
- Atributos privativos

Além disso, se o bean for compartilhado em um escopo que envolva passivação, deve ser serializável (declarar *Serializable* e conter apenas atributos de estado serializáveis).

Muitos beans contém propriedades que serão usadas em coleções, para construir tabelas, menus, etc. É muito importante que esses beans sejam serializáveis, para que possam participar de escopos que requerem passivação, e que estejam bem implementados com os métodos

`equals()` e `hashCode()`, sob pena de introduzirem bugs muito difíceis de encontrar nas aplicações em que forem usados devido a objetos duplicados.

A classe a seguir é um managed bean CDI, acessível via EL através do nome *livros*, e que contém uma propriedade *allLivros*:

```
@Named("livros")
public class LivroDAOManagedBean {
    @Inject EntityManager em;
    public List<Livro> getAllLivros() {
        return em.createNamedQuery("selectAll", Livro.class).getResultList();
    }
}
```

O session bean a seguir *também* é um managed bean, acessível via EL através do nome *livroDAOSessionBean*, e com uma propriedade somente-leitura *livros*:

```
@Stateless
public class LivroDAOSessionBean {
    @PersistenceContext
    EntityManager em;

    public List<Livro> getLivros() {
        return (List<Livro>)em.createNamedQuery("selectAll").getResultList();
    }
}
```

5.1 Mapeamento de propriedades em um managed bean

Cada *propriedade* de um managed bean pode ser mapeada a uma

- Instância de um componente (*binding*)
- Valor de um componente (*value*)
- Instância de um conversor (*converter*)
- Instância de um listener (*listener*)
- Instância de um validador (*validator*)

5.1.1 Binding de componentes

O mapeamento direto *instâncias dos componentes (binding)* permite um grande dinamismo da interface, pois é possível controlar seus atributos diretamente. Propriedades mapeadas via *binding* devem ter tipo idêntico ao do componente.

Por outro lado, *binding* é considerada uma prática rara e usar como principal meio de conectar beans a componentes é uma prática não-recomendável. Componentes UI sempre têm escopo limitado à requisição. Se usados em um escopo mais duradouro (View, Session, etc.) eles serão recriados e podem causar erros de duplicação de ID. Em geral, binding só deve ser usado em escopos locais.

Para mapear um componente representado por um facelet a um bean, é preciso descobrir qual a classe *UIComponent* que está mapeado a este facelet, e declarar a classe como propriedade do managed bean. Por exemplo, o `<h:inputText>` da página abaixo declara estar mapeado à propriedade *inputComponent* de mensagemBean::

```
<h:form id="formulario">
    <h:inputText value="#{mensagemBean.mensagem}"
                binding="#{mensagemBean.inputComponent}" />
    ...
</h:form>
```

O facelet `<h:inputText>` é um objeto `UIInput` na árvore de componentes, portanto na classe que implementa `mensagemBean`, a propriedade deve ser declarada com este tipo:

```
@Named
@SessionScoped
public class MensagemBean implements Serializable {

    private Mensagem mensagem;
    private UIInput inputComponent;
    ...
}
```

Assim, dentro do bean é possível ter acesso ao estado do componente, e alterar propriedades como `rendered` (via `setRendered()`) que podem incluir/remover o objeto da renderização da página:

```
public void método() {
    ...
    inputComponent.setRendered(false);
    ...
}
```

5.1.2 Mapeamento de valores

É muito mais comum mapear o valor dos componentes às propriedades do bean. Os valores não se limitam aos tipos básicos do Java. Através do uso de conversores, pode-se associar objetos, beans, arrays, coleções, enums, preservando as informações encapsuladas nos objetos.

Para realizar o mapeamento é preciso descobrir o tipo de dados que está armazenado no componente. O componente deve ser um `ValueHolder`, ou seja, capaz de armazenar valor. Se o valor for um string ou tipo numérico, ou coleção desses tipos, não é necessário implementar conversores (a menos que se deseje formatar os dados).

Componentes `UIInput` e `UIOutput` guardam valores que podem ser representados como *texto*, e fazem a conversão automática de tipos numéricos (inclusive `BigDecimal`), datas e strings em geral. Requerem conversores para outros tipos de objetos, ou para formatar moeda, datas em relação ao locale, etc. Para isto o JSF disponibiliza alguns conversores prontos.

Os mesmos bean e tag mostrado anteriormente tinha um mapeamento de valor para um `UIInput`. As listagens a seguir destacam esse mapeamento:

```
<h:form id="formulario">
    <h:inputText value="#{mensagemBean.mensagem}"
                 binding="#{mensagemBean.inputComponent}" />

    ...
</h:form>

@Named
@SessionScoped
public class MensagemBean implements Serializable {

    private Mensagem mensagem;
    private UIInput inputComponent;
    ...
}
```

O value de um componente `UISelectBoolean` deve ser mapeado a um tipo `boolean` (ou `Boolean`), e elementos que armazenam coleções, como `UIData`, `UISelectMany`, `UISelectItems`, etc., são mapeados a coleções ou arrays.

Componentes de seleção representam a coleção de dados via `<f:selectItem>` ou `<f:selectItems>`, como foi mostrado no capítulo sobre Facelets. Esses itens podem ser mapeados via binding a `UISelectItem/UISelectItems` e via value a coleções. Por exemplo, considere o bean abaixo que contém um array de objetos Filme:

```

@Named("cinemateca")
@ApplicationScoped
public class CinematecaBean {
    private Filme selecionado;
    private Filme[] filmes;
    ...
}

```

A lista de opções de um menu pull-down pode ser construída mapeando o atributo value de `<f:selectItems>` ao array, e o item selecionado, mapeando o value do `<h:selectOneMenu>` ao atributo selecionado.

```

<h:selectOneMenu id="som" value="#{bean.filme}">
    <f:selectItems value="#{cinemateca.filmes}"
        var="filme"
        itemLabel="#{filme.titulo}"
        itemValue="#{filme.imdb}" />
</h:selectOneMenu>

```

Em ambos os casos, um converter será necessário para que o objeto possa ser selecionado, enviado para processamento, e o resultado exibido.

Os atributos de um bean podem ser mapeados a vários componentes ao mesmo tempo, e refletirão as mudanças ocorridas em cada um.

5.2 Comportamento de um managed bean

Um managed bean pode conter métodos., Tipicamente eles lidam com:

- Ação de processamento e navegação (chamado no *action*, do form)
- Tratamento de eventos (listeners)
- Processamento de validação (validators)

Criar esses métodos no próprio managed bean elimina a necessidade de implementar validators e listeners em classes separadas, quando são simples e não há interesse em reusá-los. A vantagem é que terão acesso aos campos privativos do bean e poderão ser implementados mais facilmente e sem quebrar encapsulamento.

5.2.1 Métodos de ação

É o método que é chamado para processar um formulário, e que retorna uma instrução de navegação, informando a próxima View. Um método de ação deve ser público, pode ou não receber parâmetros e deve retornar um objeto identificador de navegação, que é geralmente um String (pode ser um enum também).

Um método de navegação é referenciado pelo atributo *action* dos componentes que suportam ações (CommandUI)

```

public String submit() { ...
    if ((compra.getTotal() > 1000.00)) {
        desconto.setRendered(true);
        return null;
    } else if (compra.getItens() > 10) {
        oferta.setRendered(true);
        return null;
    } else {
        carrinho.clear();
        return ("fatura");
    }
}

```

Retornar `null` represesta a mesma página. É o default em processamento Ajax.

5.2.2 Métodos de processamento de eventos

Os métodos de tratamento de eventos devem ser métodos públicos que retornam `void`. O argumento depende do tipo de evento gerado. Os componentes UI provocam dois tipos de eventos: `ActionEvent`, disparados por botões e links, e `ValueChangeEvent`, disparados por elementos que retém valor.

`ActionEvent` possui `getComponent()` que pode ser usado para identificar o componente que provocou o evento. O método a seguir, implementado em um managed bean, é chamado quando um botão é apertado, e imprime o seu ID local:

```
public void processadorAction(ActionEvent e) {
    this.setButtonId(e.getComponent().getId());
    System.out.println("Bean.processadorAction: " + this.buttonId);
}
```

Para registrar o listener usa-se o atributo `actionListener` do componente:

```
<h:commandButton value="Button 2" id="btn2"
    actionListener="#{bean.processadorAction}" />
```

Componentes que suportam eventos `ValueChangeEvent` não têm atributo `actionListener`, mas `valueChangeListener`, que deve conter uma expressão EL para vinculá-lo ao método apropriado:

```
<h:inputText id="name" valueChangeListener="#{bean.processarValueChange}" />
```

`ValueChangeEvent` possui `getNewValue()` e `getOldValue()` que permite acesso ao valor novo e o valor antigo.

```
public void processarValueChange(ValueChangeEvent event) {
    if (null != event.getNewValue()) {
        FacesContext.getCurrentInstance()
            .getExternalContext()
            .getSessionMap().put("name", event.getNewValue());
    }
}
```

Listeners e eventos serão explorados em maior detalhe em um capítulo próprio.

5.2.3 Métodos para realizar validação

Um método que realiza validação deve receber como parâmetros

- O `FacesContext`
- O `UIComponent` cujos dados devem ser validados
- Os dados a serem validados

```
public void validarNull(FacesContext context, UIComponent comp, Object value) {
    if(value == null || value.toString().length() == 0) {
        throw new ValidatorException(new FacesMessage("Campo obrigatório!"));
    }
}
```

É referenciado usando o atributo `validator` de componentes descendentes de `UIInput`:

```
<h:inputText value="#{transporte.destinatario.nome}" id="nome"
    validator="#{transporte.validarNull}"/>
```

Validação será explorada em maior detalhe em um capítulo próprio.

5.3 Escopos

Escopos representam duração da vida de um bean. São escopos de persistência da plataforma Web. Aplicações Web em Java (WebServlets, JSP) definem quatro escopos, em ordem de durabilidade:

- *Página*: durante a execução do método `service()` de um *Servlet*, um objeto de requisição é criado e repassado a uma ou mais páginas. Objetos que são instanciados quando a página recebe a requisição deixam de existir quando a requisição é repassada a outra página. Este é o escopo mais curto.
- *Requisição*: tem o escopo do método `service()` do *Servlet*, que instancia objetos quando a requisição HTTP é recebida, e os destrói quando a resposta é devolvida ao cliente.
- *Sessão*: como HTTP é um protocolo que não mantém estado, a sessão precisa ser mantida artificialmente usando um mecanismo externo. O mais comum é usar Cookies de sessão, que são criadas na primeira requisição feita para um domínio/caminho por um cliente, e mantido enquanto o cliente continuar enviando requisições para o mesmo domínio/caminho. Objetos associados à sessão são destruídos apenas quando o cliente encerra a conexão com o domínio/caminho, geralmente fechando a aba do browser ou criando um novo Cookie com data de expiração no passado.
- *Aplicação*, ou *Contexto Web*: representa a instância do container ou servidor Web enquanto ele estiver executando. Objetos que têm escopo de aplicação só deixam de existir se o container ou servlet que representa o contexto de execução for destruído (ex: se o servidor for reiniciado).

Em aplicações Ajax, o conceito de página foi redefinido, pois tornou-se possível realizar várias requisições sem carregar uma nova página. Em JSF, que implementa suporte a Ajax, o escopo de página, do JSP (que dura menos que uma requisição), foi substituído pelo escopo de *View* (que pode durar várias requisições). O escopo de página ainda existe, mas não tem a importância em JSF que tinha antes em JSP. Escopo de *View* também não existe em HTTP, e é, na verdade, é um tipo especial de sessão que se encerra quando o usuário deixa uma página. O JSF 2.2 e CDI ainda define mais dois escopos, que permitem controlar melhor a duração de uma sessão: *Conversation*, que permite controlar programaticamente o início e o fim de uma sessão, e *Flow*, que atrela a sessão a um conjunto de páginas agrupadas por regras de navegação.

5.3.1 Escopos em managed beans

O pacote `javax.faces.beans` define várias anotações para definir escopos em beans controlados pelo runtime JSF, mas essa responsabilidade hoje é duplicada por outros pacotes do Java EE, como CDI e EJB. A partir do próximo lançamento de Java EE, o uso de escopos do JSF será depreciado. Portanto, atualmente recomenda-se que para beans que não são EJB, os escopos usados sejam declarados usando contextos CDI, disponíveis no pacote `javax.enterprise.context`.

Os escopos JSF (antigos) são criados guardando referências para os objetos em um *Map*. Os mapas são obtidos do *FacesContext*.

```

FacesContext ctx = FacesContext.getCurrentInstance();
ExternalContext ectx = ctx.getExternalContext();

Map requests = ectx.getRequestMap();
Map sessions = ectx.getSessionMap();
Map appContexts = ectx.getApplicationMap();

```

O mapa do *View* é obtido do *ViewRoot*:

```
Map view = ctx.getViewRoot().getViewMap();
```

Tendo-se o *Map*, pode-se obter a instância de um bean que tenha sido declarado com `@ManagedBean`:

```
Bean bean = (Bean)sessions.get("bean");
String propriedade = bean.getMensagem();
```

Outra forma de localizar um bean é executar o *ELResolver*, que irá procura-lo em todos os escopos que o runtime EL tiver acesso (inclusive escopos CDI e EJB):

```
FacesContext ctx = FacesContext.getCurrentInstance();
ELContext elCtx = ctx.getELContext();
Application app = ctx.getApplication();
Bean bean = (Bean)app.getELResolver().getValue(elCtx, null, "bean");
```

Esse procedimento é ideal quando não se sabe em que escopo está o bean.

Finalmente, se o bean foi anotado com *@Named*, e foi armazenado em um escopo CDI, que é a forma recomendada para JSF 2.2 e Java EE 7, a instância pode ser recuperada através de um *@Inject*:

```
public class MyListener ... {
    @Inject
    Bean bean;
    ...
    public void listenerMethod(...) {
        String propriedade = bean.getMensagem();
        ...
    }
}
```

As seções a seguir apresentam um resumo dos escopos do CDI e seu significado. Para maiores informações, consulte a documentação do CDI.

5.3.2 Escopos fundamentais: requisição, sessão e contexto

Estes três escopos de persistência são abstrações importantes em qualquer aplicação Web de qualquer tecnologia. Em JSF/CDI o escopo mais curto é a *requisição*. A *sessão* tem geralmente a duração definida pelo cliente. O *contexto Web/aplicação* dura o tempo em que o WAR da aplicação está instalado, ativo e executando no container Web. A vida de um managed bean pode ser configurada para ter a duração desses contextos simplesmente anotando a classe do bean com:

- *@javax.enterprise.context.RequestScoped* – escopo de sessão
- *@javax.enterprise.context.SessionScoped* – escopo de sessão
- *@javax.enterprise.context.ApplicationScoped* – escopo da aplicação (contexto Web)

Um bean declarado com *@RequestScoped* neste escopo será instanciado a cada requisição HTTP, e ficará disponível pelo tempo que durar a requisição (até a resposta ser recebida). Sem Ajax, a requisição termina com a recarga da página ou com a renderização de outra página. Usando Ajax, várias requisições podem ocorrer sem que haja mudança do endereço da página.

O bean declarado com *@SessionScoped* será criado na primeira requisição e estará disponível até que a sessão seja invalidada, que geralmente depende do cliente e da duração da sessão, que pode envolver várias requisições.

Uma sessão também pode ser invalidada através de comando enviado pelo servidor. O método *invalidateSession* do *ExternalContext* pode ser usado para este fim.

```
public String logout() {
    ctx.getExternalContext().invalidateSession();
    return "/home.xhtml?faces-redirect=true";
}
```

O redirect é fundamental para que a página não permaneça mostrando dados obsoletos.

É importante observar que o *invalidateSession()* destrói *todas* as sessões, não apenas aquela identificada por *@SessionScoped*, mas também beans configurados como *@ViewScoped*, *@FlowScoped* e *@ConversationScoped*.

Uma vez instanciado, um bean declarado como **@ApplicationScoped** estará disponível durante toda existência da aplicação (enquanto a aplicação Web no container não for reiniciada). Este escopo é ideal para serviços e estados que serão compartilhados por vários usuários e aplicações. Pode haver problemas de concorrência se vários usuários tentam alterar mesmos dados.

Escopos similares a **@ApplicationScoped** podem ser implementados usando singletons (`@javax.ejb.Singleton` ou `@javax.inject.Singleton`)

5.3.3 Escopos de sessão: view, conversation e flow

Estes três escopos estão atrelados à sessão HTTP. São intermediárias entre a duração de uma requisição e a duração de uma sessão. A principal diferença entre elas consiste no controle da sua duração. Como são atraladas à sessão, chamar um método para invalidar a sessão destrói não apenas os beans que são **@SessionScoped**, mas também qualquer um anotado com as sessões abaixo:

- `@javax.faces.bean.ViewScoped` – uma view (ex: página + ajax)
- `@javax.faces.flow.FlowScoped` – um flow (conjunto de páginas relacionadas)
- `@javax.enterprise.context.ConversationScoped` – um diálogo (com início e fim)

Um bean anotado com **@ViewScoped** dura o tempo de uma View. Este escopo é adequado a aplicações Ajax que realizam várias requisições na mesma página. O bean será destruído e removido do escopo quando a página mudar.

@ConversationScoped pode também ser usado para aplicações Ajax, e oferece mais controle pois permite que a aplicação defina quando o escopo começa e quando termina. Para controlar é preciso injetar um `javax.enterprise.context.Conversation` no bean que irá determinar a duração do escopo. Usando a instância injetada, pode-se iniciar o escopo com o método `begin()`, e encerrá-lo com `end()`:

```
@Inject Conversation conv;
...
public void inicioDaConversa() { ...
    conv.begin(); // escopo está ativo
}
...
public void conversaTerminada() {... 
    conv.end();
    return "/home.xhtml?faces-redirect=true";
}
```

O redirecionamento no final é importante, pelos mesmos motivos que a invalidação de uma sessão.

@FlowScoped é usado em componentes que fazem parte de um *Faces Flow*: uma coleção de Views usadas em conjunto, relacionadas entre si por regras de navegação (configuradas em `faces-config.xml`), com um único ponto de entrada e saída. O escopo inicia quando o Flow é iniciado, e encerra automaticamente quando o usuário sai do Flow. Um Flow pode envolver várias páginas.

5.3.4 Outros escopos

Outros dois escopos são usados em beans CDI. O primeiro **@Dependent** não é realmente um escopo, mas um mecanismo que indica que o bean anotado com ele herda o escopo do bean que o utiliza. **@Dependent** é default em CDI e aplicado em beans que não têm escopo declarado.

@TransactionScoped dura o tempo de uma transação, que geralmente é menos que uma requisição, já que aplicações CDI e EJB típicas demarcam transações que duram no máximo o escopo de um método de negócios, e o uso de transações distribuídas em WebServlets nunca

devem ter escopo maior que o método `service()`. Beans anotados com esta anotação, portanto, deixam de existir logo que acontecer o `commit()` da transação.

6 Conversores

Conversores possibilitam que JSF trabalhe com objetos que nem sempre podem ser exibidos em uma página sem algum tipo de conversão. Conversores encapsulam em um objeto as regras de conversão Objeto-String-Objeto, simplificando o desenvolvimento e facilitando o reuso. Grande parte dos objetos usados em aplicações Web consistem de textos, números e datas, que são facilmente convertidos em String. Esses formatos são automaticamente convertidos pelo JSF, que também é capaz de extrair e converter objetos individuais de uma coleção. Em situações em que a conversão não é óbvia, o JSF fornece implementações prontas com atributos que permitem configurar e ajustar a conversão de certas estruturas como datas, locales, moeda, etc. Essas implementações estão disponíveis no pacote `javax.faces.convert` e incluem:

- `BigDecimalConverter`
- `BooleanConverter`
- `DateTimeConverter`
- `EnumConverter`
- `IntegerConverter`, etc.

Cada `converter` tem uma mensagem de erro padrão associada, que será exibida se conversão falhar. Por exemplo, `BigIntegerConverter` contém a mensagem: “*{0} must be a number consisting of one or more digits*”. Essas mensagens, e outros aspectos de configuração, podem ser ajustados de maneira declarativa.

Há três tags que permitem configurar converters:

- `<f:converter>` - usado para registrar um conversor nativo ou customizado.
- `<f:numberConverter>` - para conversão e formatação de números e moeda.
- `<f:dateTimeConverter>` - para conversão e formatação de datas.

6.1 Como usar um conversor

Há várias formas de usar conversores. A mais simples é automática não requer nada além do mapeamento de valor. Por exemplo, se um managed bean declara o seguinte tipo:

```
Integer idade = 0;
public Integer getIdade(){ return idade;}
public void setIdade(Integer idade) {this.idade = idade;}
```

Mapeado, na página XHTML, da seguinte forma:

```
<h:inputText value="#{bean.idade}" />
```

O texto (string) digitado, contendo o número, será automaticamente convertido para Integer antes de ser gravado no bean. Se o usuário digitar algo que não pode ser convertido em inteiro, será lançado um erro de conversão/validação, que impedirá o envio do formulário. Conversões similares podem ser feitas com qualquer tipo primitivo e tipos básicos como `BigDecimal` e datas.

é incluir um dos facelets de converter dentro do elemento que representa o componente UI que precisa converter dados.

A conversão automática é realizada por um objeto conversor. Ele pode ser indicado explicitamente (embora neste caso não seja necessário) de duas formas. Uma delas é usando o atributo `converter` do componente :

```
<h:inputText value="#{...}" converter="javax.faces.convert.IntegerConverter" />
```

Outra forma é usar o tag `<f:converter>` dentro de um componente e referenciá-lo pelo seu ID (o ID default é o tipo convertido):

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

Embora não sejam necessárias para tipos básicos, essas formas são usadas para incluir converters customizados.

6.2 Conversão de datas

`DateTimeConverter` converte datas em objeto e vice-versa. A conversão default usa apenas a representação `toString()` para a data, que geralmente é insuficiente e dificulta o processamento. Incluindo e configurando o facelet `<f:convertDateTime>` em um componente que recebe ou exibe datas, oferece muito mais controle. O componente encapsula vários formatos, baseados nos disponíveis na classe utilitária `DateFormat`.

```
<h:outputText value="#{bean.aniversario}" />
  <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

O tag acima gera: **Saturday, October 5, 2016**

Pode-se também usar um padrão

```
<f:convertDateTime pattern="EEEEEEE, MMM dd, yyyy" />
```

ou aplicar um Locale

```
<f:convertDateTime dateStyle="full" locale="pt" timeStyle="long" type="both" />
```

O exemplo acima produz:

Quarta-feira, 5 de outubro de 2016 07:15:04 GMT-04

A tabela abaixo lista alguns atributos de `f:convertDateTime`:

<code>dateStyle</code>	Baseado em <code>java.text.DateFormat</code> . Pode ser <code>default, short, medium, long, full</code> .
<code>locale</code>	Default é <code>FacesContext.getLocale()</code> . Pode ser um <code>Locale</code> ou <code>String</code> ("pt", "pt-BR", "en", "fr", "ru", etc.)
<code>pattern</code>	Formatação baseado em padrão <code>java.text.DateFormat</code> . Este atributo anula o efeito de <code>dateStyle</code> , <code>timeStyle</code> e <code>type</code> .
<code>timeStyle</code>	Baseado em <code>java.text.DateFormat</code> . Pode ser <code>default, short, medium, long, full</code> .
<code>type</code>	Pode ser <code>date, time</code> ou <code>both</code> (default é <code>date</code>)

6.3 Conversão numérica e de moeda

`NumberConverter` permite converter números, moedas, aplicar locale, formatar, etc. Os recursos também são similares aos disponíveis nas classes do pacote `java.text`. Considere o fragmento a seguir onde `total` é do tipo `BigDecimal`:

```
<h:outputText value="#{pedido.total}" />
  <f:convertNumber currencyCode="BRL" pattern="#,###.##"
    type="currency" locale="pt"/>
</h:outputText>
```

Quando o *outputText* for renderizar o valor *320.0*, recebido em formato *BigDecimal*, ele será convertido na string “R\$320,00”. Usando em um campo de entrada, o recebimento do texto “R\$1.230,55” causará a conversão para *BigDecimal.valueOf(1235.55)* (automaticamente trocando a vírgula pelo ponto, uma vez que o Locale escolhido usa outra representação).

Alguns dos atributos de *f:convertNumber* estão listados abaixo:

<code>currencyCode</code>	Código ISO 4217 para moeda: USD, BRL, EUR, RUB, CNY, JPY, etc.
<code>currencySymbol</code>	“\$”, “¥”, “£”, “R\$”, “₩”, “€”
<code>locale</code>	Default é <i>FacesContext.getLocale()</i> . Pode ser um <i>Locale</i> ou <i>String</i> (“pt”, “pt-BR”, “en”, “fr”, “ru”, etc.)
<code>pattern</code>	Formato (baseado em <i>java.text.DecimalFormat</i>)
<code>type</code>	<i>number</i> (default), <i>currency</i> ou <i>percent</i> .

6.4 Conversores customizados

Pode-se criar conversores para outros tipos use a interface *javax.faces.convert.Converter*, e implemente os métodos *getAsObject()* e *getAsString()*, que servem para converter o objeto em uma representação unívoca, e vice-versa. O conversor é registrado com um ID, que é fornecido como argumento para a anotação *@FacesConverter*, usada na classe (também é possível registrar o conversor em *faces-config.xml*). Para usar, deve-se referenciar o ID em *<f:converter>*.

O conversor abaixo traduz no método *getAsString()* um objeto Filme em uma representação String (seu código IMDB, que é único). No método *getAsObject()* ele usa o código IMDB para realizar um query no banco e recuperar o objeto:

```
@FacesConverter("filmeConverter")
public class FilmeConverter implements Converter {

    @Inject
    private FilmDAO filmDatabase;

    @Override
    public Object getAsObject(FacesContext ctx, UIComponent comp, String imdb) {
        if(imdb == null || imdb == "") return null;
        return filmDatabase.findByIMDB(imdb);
    }

    @Override
    public String getAsString(FacesContext ctx, UIComponent comp, Object objFilme) {
        if(objFilme == null) return "";
        return ((Filme)objFilme).getImdb();
    }
}
```

O conversor está sendo usado dentro de um componente *<h:selectOneMenu>* para que seja possível exibir filmes e gravar o valor selecionado como objeto:

```
<h:selectOneMenu id="menu" value="#{cinemateca.filme}">
    <f:selectItem itemLabel="Selecione" noSelectionOption="true" />
    <f:selectItems value="#{cinemateca.filmes}"
        var="filme"
        itemLabel="#{filme.titulo}"
        itemValue="#{filme}"/>
    <f:converter converterId="filmeConverter" />
</h:selectOneMenu>
```

Este outro componente lê o objeto e imprime sua representação String (que é apenas o IMDB do filme). Sem o Converter, ele usaria o `toString()` do objeto:

```
<h:outputText id="imdb" value="#{cinemateca.filme}">
    <f:converter converterId="filmeConverter"/>
</h:outputText>
```

7 Listeners

Listeners são usados para tratar e processar *eventos*. Um listener precisa registrar-se com uma fonte produtora de *notificações*. Quando um evento ocorre, o listener é notificado e recebe um objeto contendo informações sobre o evento, e tem a oportunidade de executar e realizar qualquer processamento disparado pelo evento. Para criar um listener de eventos disparados por componentes JSF é preciso criar e registrar uma classe ou um método.

Vários tipos de eventos ocorrem durante a execução de uma aplicação JSF: eventos disparados por componentes da interface do usuário, eventos de Ajax, eventos de ação HTTP, eventos do sistema e eventos de modelo de dados. Desses, os mais importantes são:

- *Action events* – lidam com envio de formulários. São lançados *após* preenchimento do bean (fase 5) ou após validação (fase 3) e retornam strings que identificam regras de navegação.
- *UI events* – apenas afetam a interface do usuário. Geralmente lançados antes do bean ser preenchido (fase 2, 3 ou 5 – depende do tipo e do atributo *immediate*) e ignoram lógica de validação. Esses eventos nunca afetam (diretamente) a navegação.

Os eventos disparados são úteis se são capturados por listeners que devem se cadastrar para receber-los e serão notificados quando (ou se) o evento ocorrer. Os listeners devem implementar interfaces relacionadas ao tipo de evento que se deseja tratar.

7.1 Listeners de eventos disparados por componentes (UI)

Existem duas interfaces:

- *ActionListener*: usado para capturar eventos disparados por botões, links, etc. – enviam automaticamente o formulário ao final do processamento.
- *ValueChangeListener*: para capturar eventos disparados por checkboxes, radio, combos, etc – não enviam o formulário.

Como foi mostrado no capítulo sobre managed beans, pode-se escrever o handler de eventos no próprio managed bean e não precisar implementar uma interface. Isto é suficiente se o evento precisar de apenas um listener, tiver necessidade de ter acesso ao estado interno do bean e não ter potencial de reuso.

Listeners também podem ser registrados através dos tags `<f:actionListener>` ou `<f:valueChangeListener>`. Neste caso é necessário escrever uma classe para implementar a interface correspondente. As interfaces são:

```
public interface ActionListener {
    void processAction(ActionEvent event);
}

public interface ValueChangeListener {
    void processValueChange(ValueChangeEvent event) ;
}
```

Desta forma é possível registrar vários listeners para um objeto. No exemplo abaixo várias implementações de ActionListener são registradas para um componente:

```
<h:commandButton value="Button 1" id="btn1">
    <f:actionListener type="br.com...RegistroDeClique" />
    <f:actionListener type="br.com...NotificarUsuarios" />
    <f:actionListener type="br.com...GravarLog" />
</h:commandButton>
```

Esta é uma das implementações, que usa *ActionEvent.getComponent().getComponent()* para obter dados do componente que causou o evento:

```
public class RegistroDeClique implements ActionListener {
    @Inject Bean bean;
    @Override
    public void processAction(ActionEvent evt) throws AbortProcessingException {
        String componenteId = evt.getComponent().getId();
        bean.addClicked(componenteId);
    }
}
```

Este outro exemplo, incluído em um menu, obtém de *ValueChangeEvent* detectar mudança no estado do componente.:

```
public class RegistroDeAlteracao implements ValueChangeListener {
    @Override
    public void processValueChange(ValueChangeEvent evt)
        throws AbortProcessingException {
        Filme antigo = (Filme)evt.getOldValue();
        Filme novo = (Filme)evt.getNewValue();
        System.out.println("Filme anterior: " + antigo);
        System.out.println("Filme atual: " + novo + "\n");
    }
}
```

Um listener já está sendo chamado no bean através do atributo *valueChangeListener*, portanto para que seja possível incluir um *segundo* listener, o componente usou *<f:valueChangeListener>*:

```
<h:selectOneMenu id="menu" value="#{cinemateca.filme}"
    valueChangeListener="#{bean.processadorValueChange}">
    <f:selectItem itemLabel="Seleciona" noSelectionOption="true" />
    <f:selectItems value="#{cinemateca.filmes}" ... />
    <f:converter converterId="filmeConverter" />
    <f:valueChangeListener type="br.com...RegistroDeAlteracao" />
</h:selectOneMenu>
```

7.1.1 Eventos que pulam etapas

Em listeners que precisam alterar apenas a interface gráfica (e não enviar dados) é necessário incluir o atributo *immediate="true"* para forçar a execução na fase 2 (antes da validação e preenchimento dos beans). Se isto não for feito a expressão só será executada após a validação.

Por exemplo, no formulário abaixo o botão *Cancelar* usa *immediate=true*, para pular diretamente para a fase de renderizar página (6) sem precisar validar os campos do formulário (que não faz sentido para um cancelamento). Mais sobre *immediate="true"* no capítulo sobre arquitetura.

```
<h:form id="transporte">
    Código <h:inputText value="#{transporte.codigo}" id="codigo"/>
    Destinatário <h:inputText value="#{transporte.destinatario.nome}" id="nome"/>
    Cidade <h:inputText value="#{transporte.destinatario.cidade}" id="cidade"/>
    <h:commandButton action="#{transporte.validarEnviar}" value="Enviar pedido" />
    <h:commandButton action="#{transporte.cancelar}" value="Cancelar"
        immediate="true"/>
</h:form>
```

7.2 Eventos do sistema e ciclo de vida

Uma das formas de depurar aplicações JSF é monitorar o que ocorre em cada fase. É também uma ótima maneira de entender melhor o funcionamento do JSF. Isto é possível através de listeners que reagem a eventos do sistema e às fases do JSF.

7.2.1 PhaseListener

Para monitorar as fases de uma aplicação inteira, deve-se implementar um *PhaseListener*. Implemente a interface, os métodos de interesse (antes ou depois de cada fase) e as fases a serem monitoradas em *getPhaseId()*:

```
public class ExamplePhaseListener implements PhaseListener {

    @Override
    public void afterPhase(PhaseEvent evt) {
        String clientId = "form:input1";
        UIViewRoot view = evt.getFacesContext().getViewRoot();
        UIComponent input = view.findComponent(clientId);

        System.out.println("AFTER PhaseID: " + evt.getPhaseId()
                           + input.getValue());
    }

    @Override
    public void beforePhase(PhaseEvent evt) {...}

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE; // qualquer fase
    }
}
```

Depois registre em faces-config.xml:

```
<faces-config ... version="2.2">
    <lifecycle>
        <phase-listener>br.com...ExamplePhaseListener</phase-listener>
    </lifecycle>
</faces-config>
```

Assim todas as fases irão chamar os métodos do *PhaseListener*, em todos os beans e views da aplicação.

Para associar um *PhaseListener* apenas a um componente, use *<f:phaseListener>*:

```
<h:inputText ...>
    <f:phaseListener type="br.com...ExamplePhaseListener" />
</h:inputText>
```

7.2.2 <f:event>

O tag *<f:event>* permite capturar eventos do sistema e redirecionar para métodos que irão processá-los. Cinco tipos de eventos do sistema podem ser capturados:

- *preRenderComponent* – antes do componente ser renderizado
- *preRenderView* – antes da View ser renderizada
- *postAddToView* – antes do componente ser adicionado à View
- *preValidate* – antes da validação
- *postValidate* – após a validação

Esses identificadores são passados ao atributo type, de `<f:event>`. O atributo listener deve indicar um método do managed bean que irá processar o evento. Por exemplo, para capturar o evento `preRenderView`:

```
<f:event type="preRenderView" listener="#{bean.antesDaView}" />
```

E no bean:

```
public void antesDaView(ComponentSystemEvent event) {
    System.out.println("PreRenderView");
}
```

8 Validadores

A validação é o processo de verificar que campos necessários estão presentes em um formulário, e se estão no formato correto. Se dados estiverem incorretos a aplicação deve reapresentar o formulário para que o usuário tenha a oportunidade de corrigir os dados. A aplicação deve identificar o problema e descrevê-lo com uma mensagem de erro, preservando os valores que foram inseridos corretamente.

8.1 Tipos e processo de validação

JSF oferece várias alternativas para facilitar esse trabalho

- Validação **explícita**: envolve o uso de tags, métodos validadores, *Bean Validation* ou outros recursos que indicam como a validação deve ser realizada. A validação explícita pode ser “*manual*” através da implementação dos algoritmos de validação escritos em Java e preenchimento das mensagens de erro, ou *automática* através de tags ou Bean Validation.
- Validação **implícita**: ocorre de forma implícita durante a checagem de campos vazios e conversão de tipos nativos..

A validação implícita verifica *restrições básicas* (ex: se o campo pode ou não ser vazio ou nulo) e *conversão de tipos* (ex: se o tipo digitado tem um formato que pode ser convertido no tipo da propriedade).

A validação explícita requer configuração ou programação, e envolve o uso de tags que definem os constraints, classes que implementam validadores ou métodos com algoritmos de validação.

O fluxo de validação segue uma ordem bem-definida.

1. Validação implícita através da verificação do atributo *required*: se o componente tiver atributo `required="true"` e o campo de dados contiver valor *vazio* ou *null*, uma violação é gerada.
2. Validação implícita através da *conversão de tipos*: se um tipo não puder ser convertido, uma violação é gerada.
3. Validação explícita através da execução de *validadores*: se houver validadores explícitos eles são executados. A validação pode ser cancelada se antes o componente mudar o atributo *immediate* para “*true*” (se o componente tiver *binding* no managed bean, pode fazer isto baseado em condições, como eventos).

8.2 Exibição de mensagens

Em todos os casos, *mensagens* são geradas quando ocorrem erros de validação. Essas mensagens podem ser apresentadas ao usuário através dos tags `<h:messages>` e `<h:message>`.

Mensagens de erro são colecionadas pelo contexto atual do runtime JSF (FacesContext). Dentro de um método (validação manual) a chamada precisa ser explícita, obtendo o contexto e chamando métodos `addMessage()`:

```
FacesContext ctx = FacesContext.getCurrentInstance();
ctx.addMessage(null, "msg para todo o form");
ctx.addMessage("form1:id1", "msg para elemento id1 dentro de form1");
ctx.getMessageList().size() // quantas mensagens há?
```

Para mostrar as mensagens na View existem dois tags:

- `<h:messages>` - mostra em um único lugar varias mensagens.
- `<h:message for="id">` - mostra mensagem individual para um determinado componente.

Um exemplo clássico é usar `<h:panelGrid>` e `<h:panelGroup>` com três colunas, sendo a primeira para o *label*, a segunda para o *componente* de entrada, e a terceira para *mensagens*, que só irão aparecer durante a validação.

```
<h:panelGrid columns="3">
    Nome: <h:inputText id="n"/> <h:message for="n"/>
    Email: <h:inputText id="e"/> <h:message for="e"/>
</h:panelGrid>
```

8.3 Validação implícita

A conversão é automática e causa validação implícita. O sistema tenta converter automaticamente *Integer*, *Double*, etc. ou um tipo que possui um conversor customizado. Se houver erro, a mensagem é armazenada automaticamente. Mas mesmo componentes que não precisam de conversão podem ser validados implicitamente, simplesmente incluindo um atributo `required="true"`.

Em caso de erro, a mensagem armazenada será uma mensagem default, em inglês ou de acordo com o locale/bundle usado. O atributo `requiredMessage` pode ser usado para customizar a mensagem de erro. O tag `<h:message>` deve ser posicionado no local onde deseja-se mostrar a mensagem. A mensagem só é renderizada se houver erro. Normalmente se adiciona um atributo `style` ou `styleClass`, para que seja possível configurar o estilo da mensagem de erro:

```
<h:inputText value="#{pacoteBean.codigo}"
    required="true"
    requiredMessage="É necessário fornecer um código"
    id="codigo"/>
<h:message for="codigo" style="color:red"/>

<h:inputText value="#{pacoteBean.peso}"
    required="true"
    requiredMessage="Peso é obrigatório" id="peso"/>
<h:message for="peso" style="color:red"/>
```

Uma alternativa ao atributo `required` é a inclusão do tag `<f:validateRequired>` dentro do elemento que se deseja validar.

8.4 Validação explícita

8.4.1 Validação manual com método validator ou action

A validação explícita pode ser “manual” implementando as regras que testam a validade dos dados em um método disparado por eventos. Pode ser o próprio método de *action* ou um método específico para validação.

Por exemplo, considere as seguintes restrições para um objeto “Pacote”:

- Código (não pode estar ausente – 3 letras)
- Peso (não pode ser mais que 1000)
- Altura (não pode ser menos que 1 ou mais que 10)
- Largura (não pode ser menos que 1 ou mais que 10)
- Profundidade (não pode ser menos que 1 ou mais que 5)

O método de validação no bean deve testar cada condição e se houver violação acrescentar mensagem (*FacesMessage*) no contexto. Se houver mais que 0 mensagens no contexto, retornar *null* no método de ação (isto exibe o formulário novamente).

Na página deve-se usar *<h:messages>* se *addMessage()* foi criado com primeiro argumento *null* ou usar *<h:message>* para cada componente se este foi identificado com *ID*.

```
<h:form>
<h:messages/>
...
Codigo: <h:inputText value="#{bean.codigo}" />
...
```

No exemplo abaixo a validação está sendo feita em um método de *action* e será executado apenas na fase 5 (e não na fase 3, dedicada à validação).

```
public String enviar() {
    FacesContext context = FacesContext.getCurrentInstance();
    if (getCodigo().length() == 3) { // falta checar se sao letras
        context.addMessage(null,
            new FacesMessage("Codigo deve ter 3 caracteres"));
    }
    if (getPeso() < 1000) {
        context.addMessage(null,
            new FacesMessage("Peso deve ser menos que 1000"));
    }
    if (getAltura() >= 1 && getAltura() <= 10) {
        context.addMessage(null,
            new FacesMessage("Altura deve ser entre 1 e 10."));
    }
    if (getLargura() >= 1 && getLargura() <= 10) {
        context.addMessage(null,
            new FacesMessage("Largura deve ser entre 1 e 10."));
    }
    if (context.getMessageList().size() > 0) {
        return(null);
    } else {
        processarEnvio();
        return("mostrarDados");
    }
}
```

A validação também pode ser feita através de um método *validator* criado no managed bean ou em uma classe separada (usando *<f:validator>*). Essas alternativas são melhores pois respeitam o ciclo de vida natural do JSF (o método é chamado na fase 3).

No facelet de um componente o atributo *validator* é usado para informar o nome do método. Este método tem uma assinatura definida (é a mesma declarada do método da interface *javax.faces.validator.Validator*.) O método deve causar *ValidatorException* com o *FacesMessage* como parâmetro se a validação falhar, e não fazer nada se passar.

```
void validar(FacesContext ctx,
             UIComponent fonte,
             Object dados) throws ValidatorException
```

8.4.2 Validação automática

Existem vários tags de validação que cuidam de validações simples de strings e números, mas também permitem a declaração de validações elaboradas usando expressões regulares, evitando a necessidade de se escrever um método de validação em Java. A validação determinada pelos tags também ocorre na fase de *validação explícita* (após teste *required* e *conversão*). Os tags e seus atributos são:

- *f:validateLength* (minimum, maximum)
- *f:validateDoubleRange* (minimum, maximum)
- *f:validateLongRange* (minimum, maximum)
- *f:validateRegEx* (pattern)

Para usar, esses tags devem ser incluídos dentro dos tags dos componentes a serem validados.:

```
<h:inputText value="#{transporte.destinatario.telefone}" id="telefone">
    <f:validateRegEx pattern="^([+]?[\d]+)?$"/>
</h:inputText>
```

8.5 Validators customizados

O tag *<f:validator>* permite que se associe a um componente qualquer validador que implemente a interface *javax.faces.validator.Validator*. Além de implementar a interface, o validador deve ser registrado com um ID, através da anotação *@FacesValidator*.

Por exemplo:

```
@FacesValidator("CEPValidator")
public class CEPValidator implements Validator {

    private Pattern pattern = Pattern.compile("\\d{5}-\\d{3}");

    @Override
    public void validate(FacesContext ctx, UIComponent comp, Object value)
        throws ValidatorException {
        Matcher matcher = pattern.matcher(value.toString());
        if(!matcher.matches()) {
            FacesMessage msg = new FacesMessage("CEP deve ter o formato NNNNN-NNN.");
            throw new ValidatorException(msg);
        }
    }
}
```

Exemplo de uso:

```
CEP
<h:inputText value="#{transporte.destinatario.endereco.cep}" id="cep">
    <f:validator validatorId="CEPValidator" />
</h:inputText>
```

8.6 Bean Validation

Uma alternativa à validação do JSF é o *Bean Validation*, que pode ser usado em conjunto com os recursos nativos do JSF. Bean Validation pode simplificar a validação “manual” substituindo o algoritmo de validação para casos comuns a uma simples anotação. A API também permite que novas anotações sejam criadas para validadores customizados.

A vantagem de usar *Bean Validation* é que as validações podem ser reusadas e compartilhadas em outras partes da aplicação, e não apenas no JSF, já que é uma API nativa do Java EE 7 e integra com containers Java EE e serviços.

Bean validation é configurado através de anotações nos atributos e métodos do bean. Por exemplo:

```
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
}
```

Em ambientes CDI, Bean Validation está habilitado por default. Em outros ambientes, ele pode ser usado com JSF incluindo a tag `<f:validateBean>` no componente que se deseja validar com a Bean Validation API:

```
<h:inputText value="#{bean.email}">
    <h:validateBean />
</h:inputText>
```

9 Templates

Um *template* pode conter a estrutura básica de várias páginas, permitindo o reuso de estrutura, e garantindo um estilo consistente para uma aplicação ou website. Combinado com recursos como imagens, CSS e JavaScript, permite tratar partes de uma View como componentes configuráveis, evitando duplicação de código e simplificando o desenvolvimento.

9.1 Por que usar templates?

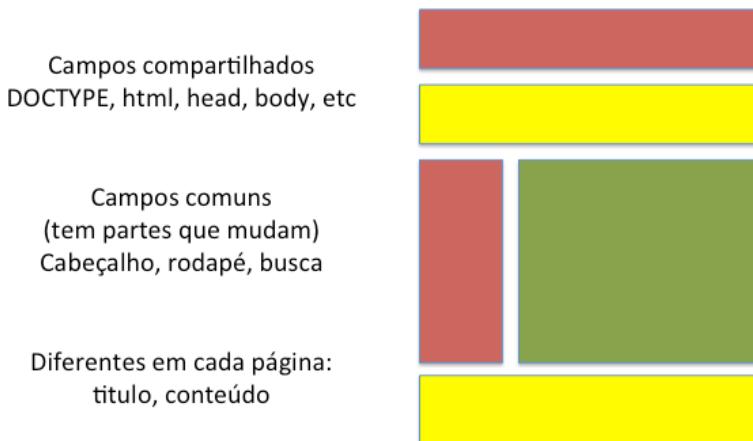
Templates existem desde as primeiras implementações de tecnologias de servidor em HTTP. Alguns exemplos de soluções similares incluem:

- Server-side includes (1994)
- Tiles (Struts)
- `@include` e `<jsp:include>` em HTTP

Templates servem para criar *composições*, que são páginas que reusam uma estrutura similar e alteram sua aparência através de configuração. Use templates para

- Evitar repetir o mesmo código
- Manter consistência de apresentação
- Facilitar a aplicação de estilos
- Facilitar o uso em dispositivos diferentes

Páginas de um site costumam ter áreas quase *estáticas*, que se repetem quase sem alteração. Por exemplo, talvez todas as páginas carreguem as mesmas folhas de estilo e scripts, e tenham uma estrutura de <head>/<body> muito semelhante, um rodapé, um menu superior. Essa estrutura poderia ser definida em um template.



Outras seções talvez tenham pequenas *alterações*, mas preservem semelhanças de estrutura. Por exemplo, um menu contextual, sempre localizado à direita, mas refletindo o contexto de uma seção do site, o estilo de uma seção, que altera fontes e cores, ou mesmo o comportamento diferenciado, de uma seção que mostra um álbum de imagens, omite algumas estruturas e altera outras radicalmente.

Finalmente é provável que haja áreas da página que sejam sempre diferentes, com texto diferentes, ou outras estruturas como formulários, imagens, recursos interativos, etc.

Através de um design cuidadoso do site, buscando identificar estruturas que possam ser reusadas, pode-se identificar as partes estáticas, e que podem ser reusadas *sem alterações*, seções que se repetem *com poucas alterações*, e seções que são *diferentes em cada página*. A partir dessas informações podemos criar um ou mais templates e determinar *pontos de inserção*, onde serão incluídos os fragmentos que irão compor a View final.

O recurso de componentes compostos do JSF (composite components), que é abordado em outra seção, tem semelhanças com templates, mas templates são mais simples. Templates também podem ser usados para criar componentes, mas com menos recursos que composite components.

9.2 Templates em JSF

JSF oferece uma coleção de tags criados com a finalidade de construir templates e componentes reutilizáveis, de forma declarativa, sem a necessidade de programação em Java. Tags usados na construção de templates fazem parte da biblioteca padrão “<http://java.sun.com/jsf/facelets>” e usam normalmente o prefixo *ui*. Alguns tags dessa biblioteca, como foi mostrado anteriormente, não servem apenas para templating mas podem ser usados em outras situações (ex: <*ui:repeat*>).

JSF oferece *duas estratégias* para construção de templates:

- Páginas comuns usando tags <***ui:include src="uri do fragmento***> para inserir partes dinâmicas. Nesta estratégia, os fragmentos que se repetem, como cabeçalhos, menus, rodapés são guardados em fragmentos de XHTML, inseridos em uma página que contém partes dinâmicas através de tags <*ui:include*>. Os próprios fragmentos podem conter outros fragmentos.
- Páginas de template contendo pontos de inserção usando <***ui:insert name="id da inserção***>. Nesta estratégia é usada uma página-cliente contendo uma coleção de

definições `<ui:define>` (fragmentos de XHTML) para cada inserção que deve ser substituída. Inserções que não estão na coleção de definições mantém seu conteúdo original. Quando a página cliente é referenciada, ela carrega seu template e substitui os pontos de inserção com suas definições.

É possível combinar as duas, usando `<ui:include>` em páginas de template e dentro de blocos `<ui:define>` das páginas cliente, quando for interessante incluir pequenos fragmentos.

Os fragmentos, apesar do nome, precisam ser páginas XHTML bem formadas. Precisam ter um elemento raiz. Normalmente usa-se `<ui:fragment>`, ou `<ui:component>` que ignora os tags de fora do componente.

Em JSF 2.2 ainda há uma terceira estratégia chamada de “contracts”, que é baseada na segunda solução e consiste em reorganizar os templates e seus resources dentro de uma estrutura padrão, dentro pasta `resources` e configurada em `faces-config.xml`, que pode ser recuperada através de um identificador (como “temas” ou “skins”)

Os tags de UI relacionados com a construção de templates estão listados na tabela abaixo:

<code>ui:component</code> <code>ui:fragment</code>	Representam <code>UIComponent</code> . Usados na construção de componentes. <code><ui:component></code> é um container genérico que ignora todo o HTML fora do elemento. <code><ui:fragment></code> é idêntico, mas considera o conteúdo externo. E geralmente é usado como um container para usar o atributo <code>rendered</code> .
<code>ui:composition</code> <code>ui:decorate</code>	Representam uma composição ou coleção arbitrária de elementos. <code><ui:composition></code> ignora todo o HTML fora do elemento e é frequentemente usado para construir páginas-cliente para templates de página reutilizáveis (contendo uma coleção de <code><ui:define></code> e <code><ui:param></code>); <code><ui:decorate></code> é idêntico mas pode ser usado para construir componentes-cliente inseridos em um contexto, já que não ignora o HTML fora do elemento.
<code>ui:define</code>	Define parte de uma página que irá ser inserido em um ponto de inserção de um template. Atributo <code>name</code> indica o nome do ponto de inserção a ser substituído.
<code>ui:include</code>	Inclui conteúdo de outra página. Atributo <code>src</code> indica o caminho para o fragmento XHTML, relativo ao contexto.
<code>ui:insert</code>	Usado em templates para definir área de inserção. Atributo <code>name</code> especifica um identificador para o ponto de inserção.
<code>ui:param</code>	Representa um parâmetro usado em diversas situações onde há necessidade de passar parâmetros (ex: elementos <code><ui:include></code> , <code><ui:decorate></code> , <code><ui:composition></code> , <code><f:view></code> , etc.) Este elemento define uma constante que pode ser lida no destino através de uma expressão EL <code>#{}{nome-do-parametro}</code> .

9.3 Como construir um template

Usando a segunda estratégia descrita neste capítulo, uma página de template deve conter toda a estrutura comum de uma página HTML. Deve-se evitar elementos que não podem ser aninhados (ex: `<h:form>`).

Elementos cuja estrutura dependem da estrutura dos fragmentos devem ser usados com cautela, pois podem introduzir bugs e limitar a reusabilidade do template se não forem construídos

corretamente. Exemplos são estruturas de layout como `<h:dataTable>`, `<h:panelGrid>`, `<p:layout>` do PrimeFaces, ou mesmo estruturas `<table>` HTML.

Templates devem ser armazenados em resources, já que são destinados a reuso.

O exemplo abaixo mostra o conteúdo de um *arquivo de template* com a estrutura que será compartilhada por todos os clientes. Este arquivo contém a estrutura básica do HTML (inclusive logotipos e arquivos CSS) como parte estática, e sete pontos de inserção possíveis, (destacados) contendo conteúdo *default* e identificados por blocos `<ui:insert>`.

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ... >

<h:head>
    <h:outputStylesheet library="css" name="layout.css" />
    <title><!-- (1) -->
        <ui:insert name="titulo">PteroCargo</ui:insert>
    </title>
</h:head>

<body>
    <!-- (2) -->
    <ui:insert name="cabecalho">
        <header>
            <h:graphicImage library="images" name="pterosaur2.jpg" height="150" />
            <h1>PteroCargo</h1>
            <h2>Transporte de cargas no tempo e no espaço</h2>
            <!-- (3) - pode ser usado apenas se (2) não for substituído -->
            <ui:insert name="menu-superior">
                <ui:include src="menu.xhtml" />
            </ui:insert>
        </header>
    </ui:insert>
    <!-- (4) -->
    <ui:insert name="conteudo-pagina">
        <div id="conteudo">
            <!-- (5) pode ser usado apenas se (4) não for substituído -->
            <ui:insert name="menu-Lateral">
                <nav class="menu-Lat">
                    <ul class="nav">
                        <li>...</li>
                        <li>...</li>
                        <li>...</li>
                    </ul>
                </nav>
            </ui:insert>
            <!-- (6) pode ser usado apenas se (4) não for substituído -->
            <ui:insert name="conteudo-Lateral">
                <div id="principal">Conteúdo a ser incluído</div>
            </ui:insert>
        </div>
    </ui:insert>

    <!-- (7) -->
    <ui:insert name="rodape">
        <footer>Copyright (c) PteroCargo Ltda.</footer>
    </ui:insert>
</body>
</html>

```

Três pontos de inserção acima estão aninhados, e só poderão ser usados se o ponto que os contém não for substituído quando o template for usado. O exemplo mostrado também usa `<ui:include>` para inserir um menu (fragmento de XHTML) dentro do terceiro ponto de inserção (que o cliente também pode substituir, se quiser).

O arquivo acima será armazenado em `/resources/templates/template.xhtml` para que possa ser referenciado pelas páginas-cliente.

9.4 Como usar templates

O template será usado por um ou mais *arquivos cliente* que irão prover o conteúdo e definir páginas individuais. Esse é o arquivo que define o ponto de acesso Web, e não o template, apesar da estrutura do arquivo ser toda do template.

Os *arquivos cliente* devem usar `<ui:composition>` para especificar o template usado, e conter um ou mais elementos de conteúdo em blocos `<ui:define>`. Tudo o que estiver fora do `<ui:composition>` será ignorado (a estrutura `<head>`, `<body>`, etc. normalmente é determinada pelo arquivo de template.)

Cada bloco `<ui:define>` contém conteúdo que irá substituir, no template, seções correspondentes marcadas com `<ui:insert>`. A ordem dos elementos `<ui:define>` não é relevante, já que eles serão identificados pelo nome e inseridos nos locais definidos pelo template, podendo inclusive ser repetidos. Basicamente, o `<ui:composition>` em uma página cliente é um conjunto de elementos `<ui:define>` contendo conteúdo de inserção no template.

Abaixo um *exemplo de um cliente* que define apenas três pontos de inserção para o template criado na seção anterior:

```

<body>
  <!-- Tudo acima é ignorado -->

  <ui:composition template="resources/templates/template.xhtml">

    <ui:define name="conteudo-pagina"> <!-- Substitui (4) -->
      <div>
        <h3>Lorem</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
        <h3>Ipsum</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
      </div>
    </ui:define>

    <ui:define name="titulo">Página principal</ui:define> <!-- Substitui (1) -->
    <ui:define name="cabecalho"/> <!-- Substitui/elimina (2) -->

  </ui:composition>

  <!-- Tudo abaixo é ignorado -->
</body>

```

O conteúdo default usado em `<ui:insert>` é *preservado* se o arquivo-cliente não o substituir. Elementos que o cliente deseja eliminar, devem ser referenciados em elementos `<ui:define>` vazios. No exemplo acima o *cabecalho* foi eliminado, o *titulo* e *conteudo-pagina* do cliente foram inseridos. Os outros pontos de inserção foram mantidos (exceto os aninhados). Um documento JSF equivalente ao resultado da aplicação do template, destacando os trechos substituídos, está listado abaixo:

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ... >

<h:head>
  <h:outputStylesheet library="css" name="layout.css" />
  <title><!-- (1) -->
    Página principal
  </title>

```

```

</h:head>

<body>
    <!-- (2) -->
    <!-- (4) -->
        <div>
            <h3>Lorem</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            <h3>Ipsum</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
        </div>

    <!-- (7) -->
        <footer>Copyright (c) PteroCargo Ltda.</footer>
</body>
</html>

```

Combinando templates e arquivos cliente é possível obter vários níveis de compartilhamento:

- Conteúdo compartilhado por *todos* os clientes deve ser colocado no *arquivo de template*.
- Conteúdo para um *cliente específico* deve ser colocado no corpo de um *ui:define* no *arquivo-cliente*.
- Conteúdo que é compartilhado por *alguns clientes (fragmento)* deve ser colocado em arquivo separado para ser incluído em um arquivo cliente através de um *ui:include no corpo de um ui:define*.

9.5 Contratos

Contratos são uma forma de organizar templates como *temas*, que podem ser usados e reusados com facilidade, selecionados e configurados através de metadados. Permite a seleção automática de templates para finalidades diferentes (ex: mobile, desktop).

Para configurar, é preciso definir *subpastas* dentro de uma pasta chamada *contracts*, na raiz da aplicação Web. Cada subpasta é o nome de um contrato, e ela deve conter um template e os recursos usados (ex: pastas com CSS, imagens, scripts). Todos os templates de um contrato devem ter o mesmo nome. Por exemplo:

```

WEB-INF/faces-config.xml
contracts
    tema1/
        template.xhtml
        css/
        images/
    tema2/
        template.xhtml
        css/
        images/
index.xhtml
mobile/index.xhtml

```

Os contratos são mapeados ao padrão de URL no *faces-config.xml*, para que possam ser selecionados automaticamente:

```

<faces-config ...>
...
<application>
    <contract-mapping>
        <url-pattern>*</url-pattern>
        <contracts>tema1</contracts>
    </contract-mapping>
    <contract-mapping>

```

```

<url-pattern>/mobile/*</url-pattern>
<contracts>tema2</contracts>
</contract-mapping>
</application>
</faces-config>

```

A seleção pode ser realizada através de ação do usuário ou evento.

10 Componentes

É possível fazer muita coisa com os componentes já existentes no JSF, e, se forem necessários componentes adicionais para aparência e comportamento existem várias bibliotecas disponíveis, como por exemplo o PrimeFaces e OmniFaces, por exemplo. Mesmo assim, pode ser necessário criar um componente customizado para funcionalidades exclusivas da aplicação Web que se está criando. O JSF oferece desde estratégias simples, que não requerem programação, a soluções mais elaboradas de baixo nível que envolvem a implementação de UIComponent.

10.1 Quando usar um componente customizado

Grande parte da API de componentes em JSF existe desde as primeiras versões do JSF. Em JSF 1.x não era incomum criar componentes programando e configurando Render Kits com implementações customizadas de UIComponent. Mas JSF 2.x oferece várias outras alternativas. Em JSF 2.x não é preciso programar um componente novo para:

- Manipular o estado ou funcionalidades de um componente (pode-se usar managed-beans com mapeamento de valor e binding)
- Conversão, validação ou listeners customizados (pode-se criar objetos customizados simplesmente estendendo uma interface, registrando com anotações e inserindo nos componentes via tags padrão)
- Agregar componentes para criar um componente composto (é possível fazer isto usando templates ou criando um composite component com facelets)

Finalmente, se ainda houver necessidade de componentes para suportar recursos como frames, HTML5, mobile, etc., veja se alguma extensão do JSF (como PrimeFaces ou OmniFaces) já não faz o que você quer.

Pode ser justificável investir na criação de um componente UIComponent se houver necessidade de explorar todos os recursos de um componente externo (ex: suporte DOM, eventos e CSS para componentes SVG). Se o objetivo é incluir suporte mínimo a um componente ou widget externo (ex: adaptar um componente JQuery), pode-se usar facelets.

10.2 Criação de componentes em JSF

Há três formas de criar componentes customizados em JSF 2.2.

- A forma mais complexa requer programação em Java, para criar um novo *UIComponent*. Depois pode ser necessário criar uma implementação de *Renderer* para adaptar o componente a um tag, e um *Tag Library Descriptor* (TLD) para especificar o tag e associá-lo com a implementação do UIComponent.
- Uma forma intermediária é construir um tag customizado. Tags customizados não são mapeados a componentes UI. São basicamente geradores de HTML que são processados de forma imediata (na primeira fase do ciclo de vida). É possível criá-los usando apenas

facelets, mas eles também requerem uma configuração em XML (TLD), para declarar regras de uso e atributos.

- A terceira forma (que será explorada nesta seção), consiste na criação de componentes verdadeiros usando apenas facelets sem a necessidade de nenhuma configuração adicional.

Neste tutorial mostraremos como construir componentes usando esta terceira estratégia.

10.3 Componentes compostos

Os tags usados para construir componentes fazem parte do namespace

```
xmlns:composite="http://java.sun.com/jsf/composite"
```

que geralmente é declarado com o prefixo *cc* ou *composite*: Os tags mais importantes desta coleção estão brevemente resumidos abaixo:

<code>cc:interface</code>	Declara a estrutura (contrato) de um componente (conteúdo, atributos, etc) – como o tag vai ser usado
<code>cc:implementation</code>	Define a <i>implementação</i> (deve estar associada a um bloco <code>cc:interface</code>) – o que o tag vai mostrar
<code>cc:actionSource</code>	Permite declarar o componente como alvo de objetos associados (ex: listeners) compatíveis com a implementação de <i>ActionSource2</i>
<code>cc:valueHolder</code> <code>cc:editableValueHolder</code>	Permitem declarar componente como alvo de objetos associados (ex: conversores, validadores) compatíveis com a implementação de <i>ValueHolder</i>
<code>cc:attribute</code>	Declara os atributos do componente
<code>cc:insertChildren</code>	Adiciona elementos-filho no componente

10.3.1 Nome do tag e implementação

A criação de componentes compostos não requer o uso de TLD ou qualquer configuração XML. A meta-informação contida nesses arquivos é geralmente usada para declarar nome da biblioteca, namespace, tag, tipo e restrições de uso para atributos. Componentes compostos fornecem essas informações no próprio tag e na estrutura usada para guardar os arquivos dentro do contexto.

O nome da *taglib* é obtida automaticamente através do nome da *library*: pasta onde o componente é armazenado, dentro da pasta *resources*. E o nome do tag é obtido do nome do arquivo XHTML, armazenado dentro da pasta. Por exemplo, na biblioteca *geocomp* abaixo foram definidos três tags: *coords*, *mercator* e *qgis*:

```
WEB-INF/
resources/
geocomp/
    coords.xhtml
    mercator.xhtml
    qgis.xhtml
    hello.xhtml
```

Cada tag definido no arquivo .xhtml consiste no mínimo de um bloco cc:implementation (se o tag não tiver atributos):

```
<cc:implementation>
    Hello, world!
</cc:implementation>
```

Uma página-cliente que usa o componente declara o namespace padrão que identifica a biblioteca, e associa um prefixo:

```
xmlns:geo="http://java.sun.com/jsf/composite/geocomp"
```

Em seguida, o tag pode ser usado:

```
<geo:hello/>
```

Um tag, portanto, pode ser usado para reusar coleções de componentes menores que formam um componente composto, reusável. Por exemplo, suponha uma página que contenha um componente que consiste de duas caixas de entrada de texto, para coordenadas geográficas:

```
<h:form>
    <h:panelGrid columns="3">
        <h:outputText value="Latitude:" />
        <h:inputText value="#{locationService.location.latitude}" id="lat" />
        <h:message for="lat" style="color: red" />

        <h:outputText value="Longitude:" />
        <h:inputText value="#{locationService.location.longitude}" id="lon" />
        <h:message for="lon" style="color: red" />
    </h:panelGrid>

    <h:commandButton actionListener="#{locationService.register}"
                      id="locBtn" action="map" value="Localizar" />
</h:form>
```

Se a mesma caixa é usada em várias partes do site, com dados idênticos, ela pode ser reusada usando um componente:

```
<html xmlns="http://www.w3.org/1999/xhtml" ...
      xmlns:cc="http://xmlns.jcp.org/jsf/composite">

<cc:interface>
</cc:interface>

<cc:implementation>
    <h:form>
        <h:panelGrid columns="3">
            <h:outputText value="Latitude:" />
            <h:inputText value="#{locationService.location.latitude}" id="lat" />
            <h:message for="lat" style="color: red" />

            <h:outputText value="Longitude:" />
            <h:inputText value="#{locationService.location.longitude}" id="lon" />
            <h:message for="lon" style="color: red" />
        </h:panelGrid>

        <h:commandButton actionListener="#{locationService.register}"
                          id="locBtn" action="map" value="Localizar" />
    </h:form>
</cc:implementation>
</html>
```

E inserida na páginas em que for usada:

```
<body>
    <h1>Coordenadas</h1>
        <geo:coords />
    </body>
```

10.3.2 Atributos

O bloco `<cc:interface>` define o contrato de uso do tag, que consiste dos seus atributos que mapeiam valores e comportamentos associados. Atributos são declarados com `<cc:attribute>`.

```
<cc:interface>
    <cc:attribute name="nomeDoAtributo"/>
</cc:interface>
```

Dentro de `<cc:implementation>` é possível referenciar o valor dos atributos através de EL e da referência `cc`, que dá acesso ao componente que está mapeado ao tag (é uma implementação de `UIContainer` chamada de `UINamingContainer`, e que é fornecida pelo runtime JSF). Abaixo estão algumas das propriedades acessíveis via `UINamingContainer`:

<code>#{cc.attrs}</code>	Mapa de atributos
<code>#{cc.clientId}:</code>	O id do elemento (da forma <code>form:id</code>)
<code>#{cc.parent}:</code>	Permite acesso ao componente pai, e recursivamente a suas propriedades cc (<code>cc.parent.attrs.nomeDoAtributo</code> por exemplo)
<code>#{cc.children}</code>	Acesso aos componentes-filho
<code>#{cc.childCount}</code>	Número de filhos

Portanto, para ter acesso aos atributos declarados, usamos `cc.attrs`:

```
<composite:implementation>
    ...#{cc.attrs.nomeDoAtributo}...
</composite:implementation>
```

O tag agora pode ser usado com atributo:

```
<geo:hello nomeDoAtributo="..."/>
```

Um atributo pode ser obrigatório, assim como pode limitar seu uso a determinadas finalidades ou componentes. Os seguintes atributos podem ser usados em `<cc:attribute>`:

<code>name</code>	Nome do atributo
<code>required</code>	Se <code>true</code> , atributo é obrigatório (<code>false</code> é default)
<code>default</code>	Valor a ser usado como default para atributos não obrigatórios
<code>type</code>	Para atributos que recebem <i>value expressions</i> – o tipo que a expressão deve produzir: <code><cc:attribute ... type="java.util.Date"/></code>
<code>method-signature</code>	Para atributos que recebem <i>method expressions</i> , contém a assinatura do método que será executado: <code><cc:attribute method-signature="java.lang.String action()"/></code> .
<code>targets</code>	Usada em atributos com <code>method-signature</code> para identificar os componentes que devem receber a expressão declarada.

No exemplo abaixo, estendemos o tag `<geo:coords>` com alguns atributos, para que um managed bean e propriedades possam ser determinados pelo usuário da forma:

```
<geo:coords latitude="#{locationService.location.latitude}"
            longitude="#{locationService.location.longitude}"
            actionListener="#{locationService.register}" />
```

Assim ele poderá ser usado com um bean que guarda coordenadas e registra as coordenadas enviadas:

```

@Named("locationService")
@SessionScoped
public class LocationServiceBean implements Serializable {
    @Inject private Location location;
    @Inject @GIS EntityManager em;

    @Transactional
    public void register(ActionEvent evt) {
        em.persist(location);
        System.out.println("Location " + location + " registered!");
    }
    public Location getLocation() {
        return location;
    }
    public void setLocation(Location location) {
        this.location = location;
    }
}

@Entity
public class Location implements Serializable {
    private String latitude = "-23";
    private String longitude = "-46";
    ...
}

```

Para isto, declaramos a seguinte interface, para permitir o registro de um ActionListener:

```

<cc:interface>
    <cc:attribute name="longitude" required="true"/>
    <cc:attribute name="latitude" required="true"/>
    <cc:attribute name="actionListener"
        method-signature="void register(javax.faces.event.ActionEvent)"
        targets="form1:locBtn"/>
</cc:interface>

```

O ID usado em *targets* é usado para referenciar o botão, na *implementation*:

```

<cc:implementation>
    <h:form id="form1">
        <h:panelGrid columns="3">
            <h:outputText value="Latitude:" />
            <h:inputText value="#{cc.attrs.latitude}" id="lat" />
            <h:message for="lat" style="color: red" />

            <h:outputText value="Longitude:" />
            <h:inputText value="#{cc.attrs.longitude}" id="lon" />
            <h:message for="lon" style="color: red" />
        </h:panelGrid>
        <h:commandButton id="locBtn" action="map" value="Localizar" />
    </h:form>
</cc:implementation>

```

Os objetos acessíveis via cc não se limitam apenas aos declarados em `<cc:interface>`. Como o UINamingComponent é um UIComponent, ele também tem acesso a objetos implícitos que estão instanciados neste componente, por exemplo:

```

<composite:implementation>
    ...
    <p>Host da requisição: #{request.remoteHost}</p>
</composite:implementation>

```

Como a intenção ao criar componentes compostos é o reuso, é importante seguir boas práticas da construção de artefatos reutilizáveis, como facilidade de uso da API, nomes auto-

documentáveis, etc. Siga padrões e seja consistente. Por exemplo: crie atributos com nomes padrão (*styleClass*, *required*, etc.) e chame o principal atributo de “*value*”.

O componente mostrado como exemplo contém um `<h:form>`. Como elementos `<h:form>` não podem ser aninhados, usar `<h:form>` em um componente pode limitar o seu uso (este componente, por exemplo, não pode ser usado dentro de um `<h:form>`).

O problema é que o cliente-ID usado no atributo *actionListener* referencia o `<h:form>`. Neste caso a solução é simples. Remover o `<h:form>` e associar o ID diretamente usando :*id*:

```
<cc:interface> ...
    <cc:attribute name="actionListener"
        method-signature="void register(javax.faces.event.ActionEvent)"
        targets=":locBtn"/>
</cc:interface>

<cc:implementation> ...
    <h:commandButton id="locBtn" action="map" value="Localizar" />
</cc:implementation>
```

Agora o componente precisa dserr usado dentro de um `<h:form>`:

```
<h:form id="f1">
    <geo:coords latitude="#{locationService.location.latitude}"
                longitude="#{locationService.location.longitude}"
                actionListener="#{locationService.register}"/>
</h:form>
```

10.3.3 Componente mapeado a classe `UINamingContainer`

Um binding entre o componente composto e uma classe Java oferece a possibilidade de usar recursos da linguagem e permite que componentes compostos tenham acesso a recursos que só seriam possível implementando `UIComponent`.

Para realizar o binding é preciso implementar `UINamingContainer` e dar um nome a ele com uma anotação `FacesComponent`:

```
@FacesComponent("componenteGeolocation")
public class ComponenteGeolocation
    extends UINamingContainer { ... }
```

E informar o componente ao declarar a interface:

```
<cc:interface componentType="componenteGeolocation">
    ...
</cc:interface>
```

11 Ajax

Ajax (Asynchronous JavaScript and XML) é uma tecnologia que utiliza-se de JavaScript para explorar uma requisição HTTP assíncrona (geralmente chamada de `XMLHttpRequest`). A aplicação JavaScript envia um `XMLHttpRequest` para o servidor, espera obter a resposta (geralmente dados em XML ou JSON) e usa esses dados para atualizar o modelo de objetos (DOM) da página, sem que seja necessário carregar uma nova página.

A atualização da página é feita usando recursos dinâmicos de JavaScript e CSS via Document Object Model – DOM, que permite alterar a estrutura da árvore da página. Frameworks como JQuery, JSF e PrimeFaces escondem os detalhes de uma requisição Ajax, que geralmente envolve o envio e recebimento de dados em texto, XML ou JSON em várias etapas, facilitando o seu uso.

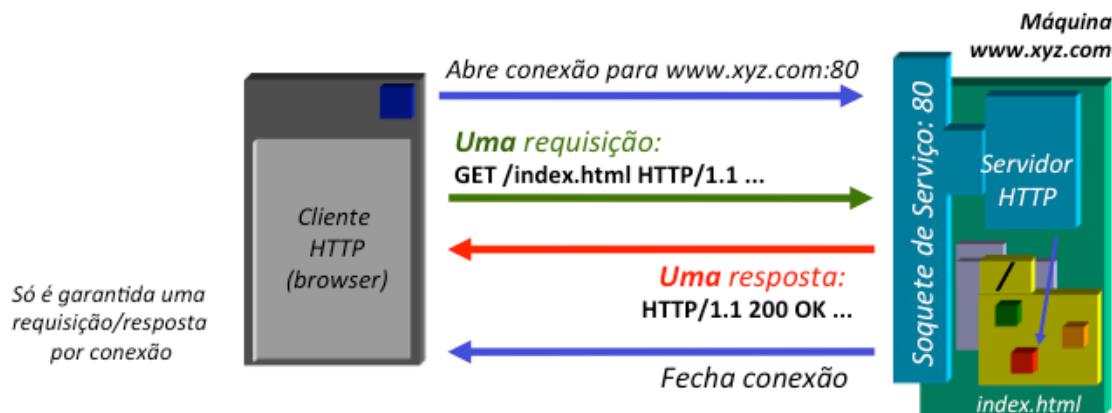
O uso de Ajax permite a construção de páginas Web bastante interativas, e melhora a usabilidade das aplicações Web. Por outro lado, o modelo assíncrono aumenta de forma

significativa a complexidade das aplicações Web, já que requisições e atualizações de dados e páginas podem ocorrer em paralelo.

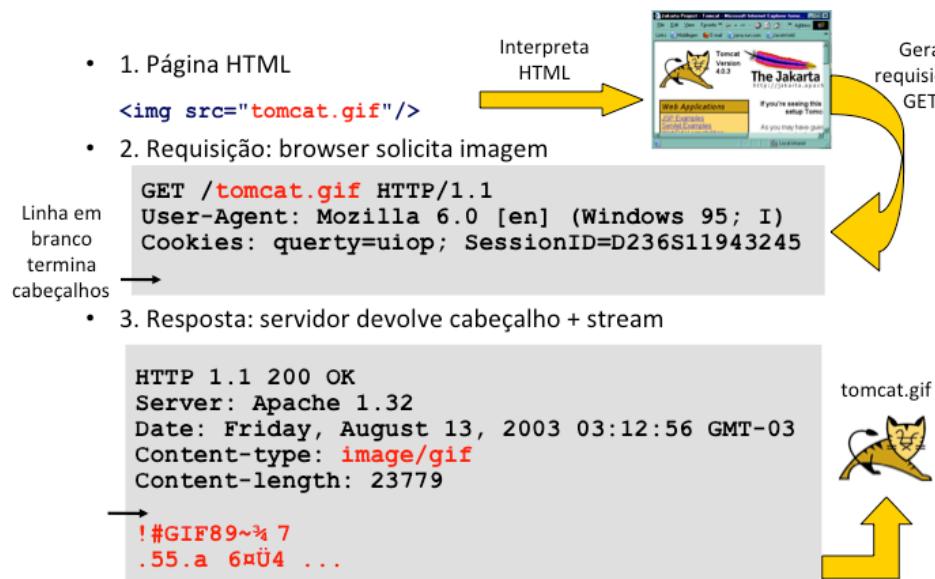
11.1 Por que usar Ajax?

A arquitetura Web é baseada em cliente, protocolo HTTP 1.x e servidor. Tem como principais características um protocolo de transferência de arquivos (HTTP: RFC 2068) que não mantém estado da sessão do cliente, um servidor que representa um sistema de arquivos virtual e responde a comandos representados uniformemente em URIs, e cabeçalhos com meta informação de requisição e resposta.

A ilustração abaixo mostra um exemplo de requisição e resposta HTTP 1.x.

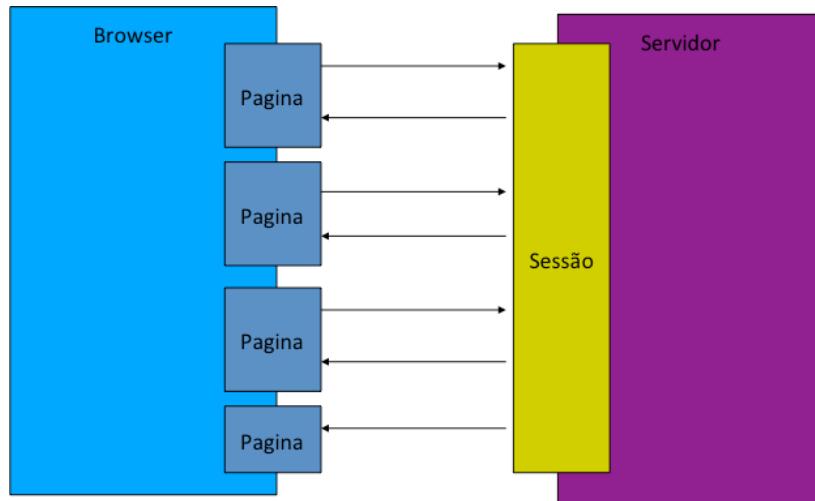


A sintaxe de uma requisição HTTP consiste de uma linha que contém método, URI e versão seguida de linhas de cabeçalho com dados do cliente. A resposta contém versão, código de status e informação de status na primeira linha e cabeçalhos com dados do servidor. A ilustração abaixo mostra detalhes de uma requisição/resposta HTTP:

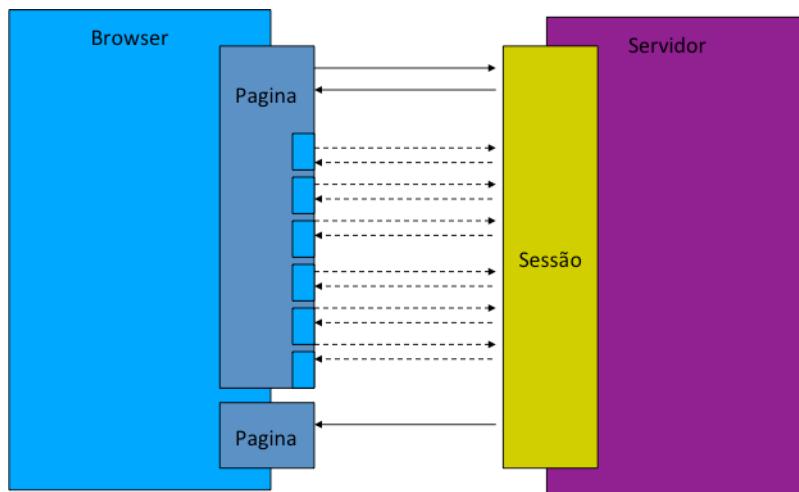


O ciclo de vida de uma aplicação Web convencional associa cada requisição/resposta a uma página. Portanto, *cada vez* que uma requisição é enviada ao servidor, a resposta provoca ou a recarga da mesma página ou o redirecionamento para outra página.

Uma sessão precisa ser mantida artificialmente, através de mecanismos externos ao HTTP, como Cookies, rescrita de URLs ou camadas intermediárias (como SSL). Uma sessão pode preservar o estado entre várias requisições, mas cada requisição devolve uma página.



O ciclo de vida de uma aplicação Ajax (Web 2.0) é diferente. Como as requisições são *assíncronas* e não forciam a recarga da página, é possível receber *fragmentos* de código ou dados (em texto, XML e JSON) para atualizar *parcialmente* a mesma página, alterando o HTML dessa página através de DOM e JavaScript, assim possibilitando a criação de interfaces dinâmicas. Assim várias requisições podem ser feitas sem que a página seja alterada. Ainda assim, o HTTP não preserva o estado da requisição, sendo necessário que a comunicação mantenha uma sessão ativa para que os dados entre as requisições sejam preservados. Com Ajax, é possível manter o estado da sessão da forma usual (Session, usando cookies) e também com escopo menor limitado à própria página (View, usando a página), já que ela pode preservar estado entre requisições.



A programação de Ajax em JavaScript puro requer várias linhas de código, funções de call-back e monitoração de estados da requisição. Atualmente utilizamos frameworks que simplificam o uso do Ajax oferecendo funções para iniciar requisições, e callbacks para lidar com as respostas assíncronas.

11.2 Características do Ajax no JSF

Em JSF 2.2 (Java EE 7) o Ajax é executado por uma biblioteca JavaScript própria (`jsf.ajax.request()`) que contém as funções necessárias para realizar o serviço. Aplicações JSF não

precisam lidar diretamente com JavaScript. As requisições são geradas automaticamente e ativadas por eventos dos componentes, e o resultado causa execução e renderização de outros componentes da página. Toda a configuração é feita através do tag `<f:ajax>`.

Qualquer componente capaz de produzir eventos pode ser configurado para ativar requisições Ajax através da inclusão do tag `<f:ajax/>` como elemento filho:

```
<h:inputText value="#{bean.message}">
  <f:ajax />
</h:inputText>
```

O tag sem atributos define *comportamento default* para:

- O evento que dispara a requisição (é o *evento default do componente*, dependendo do tipo; ex: `ValueChangeEvent` para um `inputText`, `selectOneMenu`, etc., `ActionEvent` para `commandButton`, `commandLink`)
- O componente que é executado para produzir os dados da requisição (default é o form no qual o componente está contido, identificado pelo ID genérico – identificador de agrupamento – `@form`)
- O componente que será atualizado quando os dados de resposta forem recebidos (default é o próprio componente, identificado pelo ID genérico `@this`)

Através de atributos é possível alterar esses defaults.

11.3 Atributos de `<f:ajax>`

A tabela abaixo lista os atributos que podem ser usados no tag `<f:ajax>`:

<code>disabled="true"</code>	Desabilita Ajax (o mesmo que não incluir o tag)
<code>event=</code> "action valueChange click change keyup mouseover focus blur ..."	Evento do DOM a processar. O evento JSF default pode ser <i>action (click)</i> ou <i>valueChange (change)</i> . Depende do componente.
<code>immediate="true"</code>	Processamento imediato (salta fases do ciclo de vida).
<code>listener="#{...}"</code>	Expressão EL mapeada a um método que poderá processar o evento <i>AjaxBehaviorEvent</i> . Este listener é chamado na fase JSF <i>Invoke Application</i> .
<code>execute="id lista de IDs @all @form @none @this #{... EL ...}"</code>	Componentes cujo estado deverá ser enviado para o servidor junto com a requisição – geralmente implementações de <code>UIInput</code> . <code>@all</code> = página inteira, <code>@form</code> = formulário no qual o componente está contido, <code>@none</code> = nenhum componente, <code>@this</code> = componente atual (<i>default</i>). Lista de IDs ou <code>@nome</code> separados por espaço. Expressão EL que gere IDs/@nomes.
<code>render="id lista de IDs @all @form @none @this #{... EL ...}"</code>	Componentes a serem renderizados. Mesmo conteúdo do atributo <code>execute</code> . Default é <code>@this</code> .

<code>onevent="JavaScript"</code>	Função JavaScript que será executada quando ocorrer um evento Ajax. A função recebe um parâmetro que possui uma propriedade <i>status</i> ("begin", "complete", "success") que reflete diferentes fases do processamento Ajax, e <i>source</i> , que contém o elemento HTML que causou o disparo do evento.
<code>onerror="JavaScript"</code>	Função JavaScript executada em caso de erro da requisição Ajax.

Inclua o elemento `<f:ajax>` dentro de um elemento de componente UI. Caso o componente afetado pela resposta não seja o próprio componente (`@this`), é preciso informar o ID do componente que deve ser atualizado através do atributo *render*:

```
<h:commandButton ... action="...">
  <f:ajax render="id1"/>
</h:commandButton>
<h:outputText ... value="#{...}" id="id1"/>
```

O atributo *render* recebe uma lista de IDs de componentes que precisam ser atualizados, separadas por espaço.

Geralmente não é o componente que dispara o evento quem fornece dados para o servidor, mas esse componente geralmente faz parte do mesmo formulário. Esse é o default (`@form`). Caso outros componentes precisem ser executados, os seus IDs devem ser informados através do atributo *execute*:

```
<h:commandButton ... action="...">
  <f:ajax render="id1 id2" execute="id3 id4" />
</h:commandButton>
```

Nos exemplos acima, quando o botão for pressionado (evento *click*) será enviada uma requisição HTTP assíncrona para o servidor, que conterá como parâmetros os dados dos elementos declarados no atributo *execute*. Na resposta, os dados recebidos são atualizados no bean, e os componentes declarados em *render* são renderizados, fazendo com que seus dados exibidos sejam lidos novamente a partir do bean.

O atributo *event* define o evento que irá disparar a ação. O evento default é o evento do componente. Por exemplo o evento *click* (nome do evento em DOM) equivale a *action* (nome do evento em JSF) e é o evento default de botões.

```
<h:commandButton id="submit" value="Submit">
  <f:ajax event="click" render="result" />
</h:commandButton>
<h:outputText id="result"
  value="#{userNumberBean.response}" />
```

Em Ajax, o evento JSF *action* representa o evento de clicar um *h:commandButton* ou *h:commandLink*. Ele é traduzido em DOM/JavaScript como *click*. O evento JSF *valueChange* é disparado em *h:inputText*, *h:inputSecret*, *h:inputTextarea*, *h:selectOneMenu*, etc. Ele é traduzido em DOM/JavaScript como *change*.

11.4 Ajax e eventos de ação/navegação

O atributo *action* de um botão refere-se ao evento que gera uma requisição e resposta HTTP sincrona (ele tem relação com o *action* do `<form>` em HTML e o botão do tipo *submit*), e informa um resultado de navegação que irá mudar a View (página) atual. Em uma requisição Ajax, que não retorna outra página, ele não é relevante.

Os métodos de ação *podem* ser chamados através de requisições Ajax, mas como Ajax é assíncrono o valor de retorno não é usado. Normalmente a assinatura de um método de controle retorna um string que é usado para informar a próxima página da navegação:

```
public String metodoDeAcao() {
    ...
    return "pagina";
}
```

Ajax ignora o valor retornado. Métodos que forem disparados por uma requisição Ajax devem declarar *String* como tipo de retorno, já que são métodos de *action*, mas no final devem retornar *null*.

11.5 Monitoração de eventos Ajax

Eventos Ajax podem ser monitorados através de eventos, que fornecem dados da requisição e resposta HTTP, além de informações de erro. Por ser uma operação assíncrona, a resposta é processada em métodos de *callback* que podem ser configurados através dos atributos *onevent*, *onerror* em JavaScript, e *listener*, em Java.

11.5.1 Eventos Ajax tratados em JavaScript

Usado para obter dados em baixo nível da requisição Ajax, inclua o nome de uma função (*callback*) JavaScript como argumento do atributo *onevent*.

```
<f:ajax render="..." onevent="processar" ... />
```

A função será chamada nos vários estágios da requisição/resposta Ajax. Esta função deve ter um argumento, que será usado para obter dados da requisição Ajax:

```
function processar(dados) { ... }
```

O argumento pode ter as seguintes propriedades.

- *status* (dados.status) – *begin*, *complete* ou *success*
- *source* (dados.source) – o evento DOM que iniciou a requisição
- *responseCode* – dados retornados em código numérico
- *responseText* – dados retornados em texto (ex: JSON)
- *responseXML* – dados retornados em XML

11.5.2 Eventos Ajax tratados em Java

O atributo *listener* permite redirecionar a um método no servidor o tratamento de uma ação do Ajax no cliente. Esse método é chamado na fase *Aplicação* do ciclo de vida (*AjaxBehaviorListener.processAjaxBehavior*)

Em vez de usar *onevent*, use *listener*:

```
<f:ajax event="change" render="total" listener="#{bean.calcularTotal}"/>
```

11.5.3 Erros de processamento tratados em JavaScript

O atributo *onerror* é similar ao *onevent*, mas ele captura apenas erros. Se houver um erro durante o processamento da requisição Ajax, JSF chama a função definida no atributo *onerror* e passa um objeto de dados.

A função deve ter um argumento:

```
function processar(dadosErro) { ... }
```

Propriedades recebidas no argumento *dadosErro* são as mesmas propriedades do atributo *onevent*, mais:

- *description* (`dadosErro.description`)
- *errorName* (`dadosErro.errorName`)
- *errorMessage* (`dadosErro.errorMessage`)

Valores possíveis para o `dadosErro.status`: são *emptyResponse*, *httpError* (null, undefined, < 200, > 300), *malformedXML* ou *serverError*

11.6 Limitações do Ajax

Nem tudo é possível com `<f:ajax>`. Às vezes é necessário forçar atualizações e recargas de páginas para obter os resultados esperados. Algumas limitações podem ser resolvidas com JavaScript, reescrevendo a forma de atualização ou mesmo usando bibliotecas de terceiros como PrimeFaces.

`<h:outputText value="#{bean.prop}" />` ou `#{}{bean.prop}` nem sempre pode ser atualizado com Ajax. Por não ter ID é necessário que esteja dentro de uma árvore de componentes que é renderizada para que seja atualizado.

Acrescentando um ID em `<h:outputText />`, permite que ele seja identificado em um atributo *render*, mas isto também fará com que seja renderizado dentro de um ``, impondo limites onde pode ser usado (não é mais texto cru – não pode ser colocado dentro de um atributo ou usado para gerar JavaScript).

Não é possível usar `<f:ajax render="..." />` para dinamicamente alterar os atributos de elementos HTML, mas é possível fazer isto com funções JavaScript/DOM.

Por fim, Ajax depende de JavaScript, que pode ter comportamento diferente em browsers diferentes (fabricantes, versões e plataformas).

12 Primefaces

PrimeFaces é uma biblioteca de componentes para JSF que possui uma grande coleção de componentes, com suporte integrado a Ajax e websockets (Ajax Push). Vários componentes podem ser usados em substituição aos componentes JSF (também podem ser usados juntos). PrimeFaces integra naturalmente à arquitetura do JSF.

PrimeFaces possui componentes de várias bibliotecas populares. Muitos componentes são wrappers sobre componentes JQuery, YUI, Google e outros. É uma das bibliotecas mais populares para JSF.

As imagens abaixo ilustram diversos componentes do PrimeFaces.

Model	Year	Manufacturer	Color
6120101	2000	Opel	Green
6120102	2000	Mercedes	Black
6120103	2000	BMW	Orange
2400319	2007	Audi	Red
1500404	2009	Opel	Blue
2700613	1984	Ford	Brown
6120873	2007	Volkswagen	Yellow
6120115	1998	Opel	White
4120127	2003	Vaux	Marron

<p:dataTable>

PrimeFaces

> Godfather Part I

The story begins as Don Vito Corleone, the head of a New York Mafia family, oversees his daughter's wedding. His beloved son Michael has just come home from the war, but does not intend to become part of his father's business. Through Michael's life the nature of the family business becomes clear. The business of the family is just like the head of the family, kind and benevolent to those who give respect, but given to ruthless violence whenever anything stands against the good of the family.

> Godfather Part II

> Godfather Part III

> Bookmarks

Abelley - 20
Abiel - 22
Alves - 2
Adriano - 21

★ Bookmark ★ With Icon ★ ★ Bookmark

<p:autoComplete>



<p:commandButton>

<p:button>

moral Nantes

Type the two words:

<p:captcha>

<p:selectBooleanButton>

<p:tabMenu>



PrimeFaces

<p:menuBar>



<p:messages>

Ajax Menutems

- Non-Ajax Menutem
- Delete
- Navigations
- Links
 - PrimeFaces
 - Home
 - Docs
 - Download
 - Support
 - Mobile

<p:panelMenu>

<p:spinner>

Client Side Progressbar

Start

Cancel

<p:progressBar>

Basic:

Feedback:

Feedback (Turkish):

Inline Feedback:

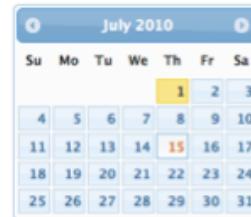


<p:password>

<p:accordionPanel>

<p:confirmDialog>

<p:ajaxStatus>

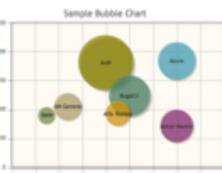


<p:calendar>

<p:colorPicker>



<p:carousel>



<p:pieChart>
<p:lineChart>
<p:areaChart>
<p:barChart>
<p:bubbleChart>
...

I accept terms and conditions: NoSubscribe me to newsletter: Yes

<p:slider>

Basic PanelGrid

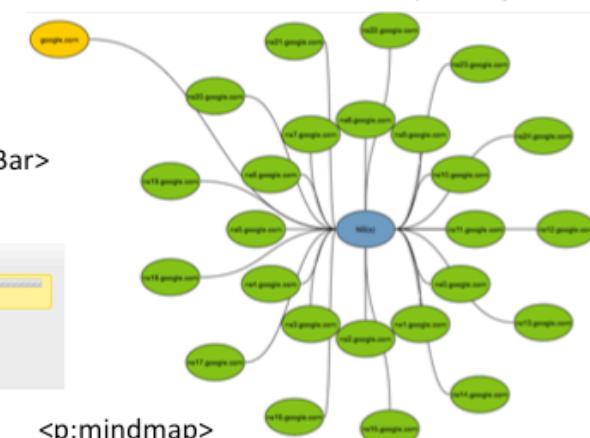
Firstname: *	<input type="text"/>
Surname: *	<input type="text"/>
<input checked="" type="checkbox"/> Save	

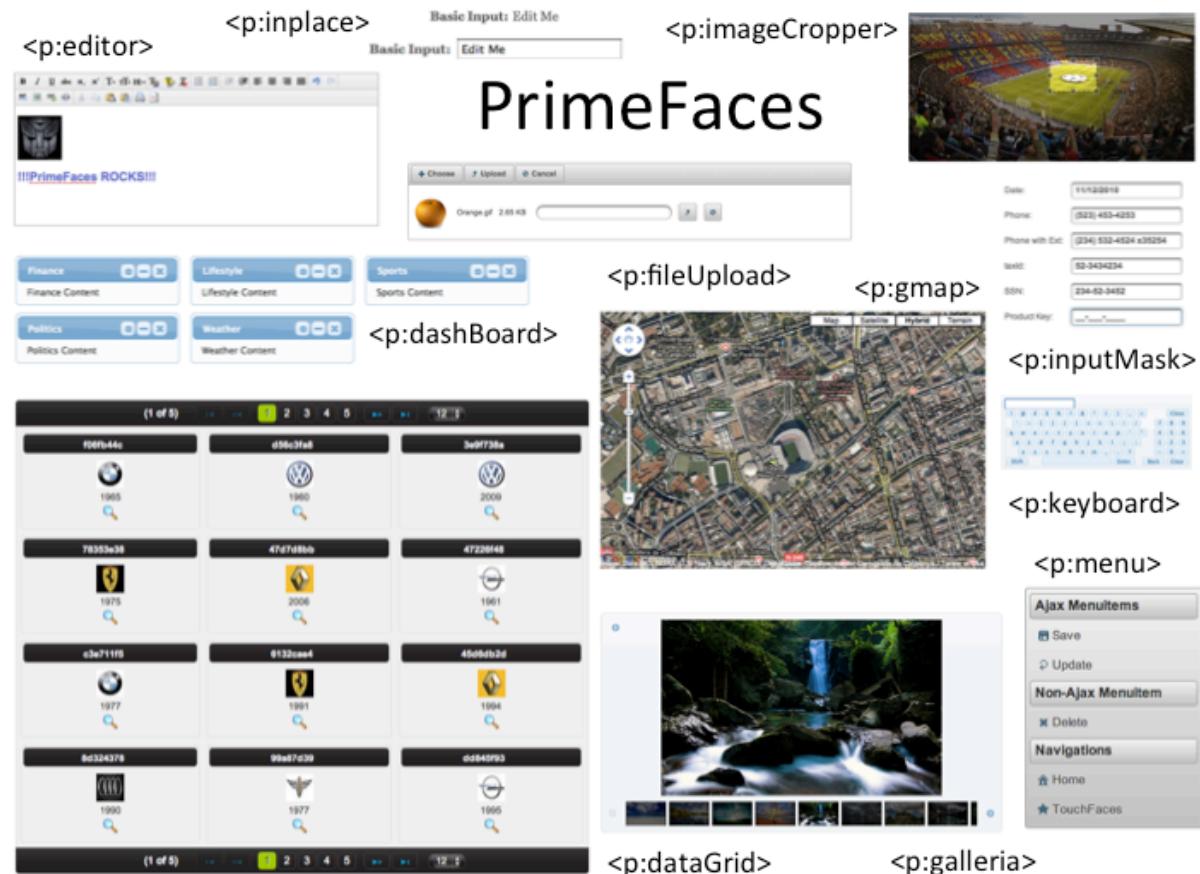
<p:panelGrid>

Basic:
 Callback:
 Ajax Rate:
 Readonly:
 Disabled:

<p:rating>

<p:selectOneListbox>





Muitos componentes do PrimeFaces são variações do JSF padrão, mas têm funcionalidade adaptada ao look & feel e comportamentos do Primefaces, além de poderem ter outros atributos e comportamento. Por exemplo:

- `<p:ajax>` (substitui f:ajax em componentes PrimeFaces)
- `<p:commandButton>` e `<p:commandLink>`
- `<p:dataTable>` e `<p:column>`
- `<p:message>` e `<p:messages>`
- `<p:outputLabel>`
- `<p:panelGrid>`
- `<p:fieldset>`
- `<p:inputText>`, `<p:inputTextArea>`, `<p:password>`, `<p:selectOneMenu>`, etc.

12.1 Configuração e instalação

Baixe o arquivo *primefaces-VERSAO.jar* em www.primefaces.org/downloads.html ou configure a instalação automática no seu projeto usando *Maven*, incluindo a seguinte dependência no *pom.xml*:

```
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>6.0</version>
</dependency>
```

O Primefaces possui um repositório próprio, onde estão as últimas versões (pode ser necessário configurar se for usada uma versão recente):

```
<repository>
    <id>prime-repo</id>
    <name>Prime Repo</name>
    <url>http://repository.primefaces.org</url>
</repository>
```

12.1.1 Teste

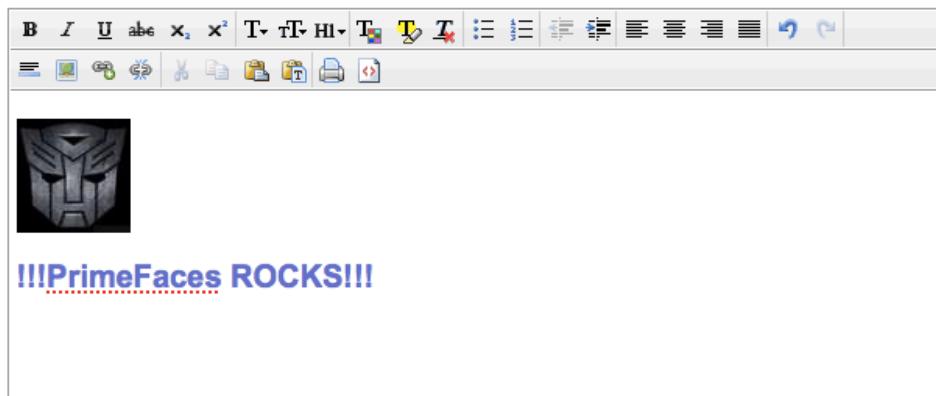
Crie um projeto e uma página XHTML. Inclua o JAR do PrimeFaces como dependência do seu projeto Web (automático se usada a configuração Maven acima: o JAR deve ser distribuído em *WEB-INF/lib*).

Em cada página que usar tags do Primefaces, declare o namespace do primefaces na página de Facelets:

```
xmlns:p="http://primefaces.org/ui"
```

Depois escolha um ou mais componentes da biblioteca (veja documentação)

```
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui">
    <h:head></h:head>
    <h:body>
        <p:editor />
    </h:body>
</html>
```



12.2 Temas (themes, skins)

Temas fornecem um look & feel comum para todos os componentes: fontes, cores, ícones, aparência geral dos componentes. Existem temas gratuitos e pagos. Podem ser baixados de:

- <http://www.primefaces.org/themes.html>
- <http://apps.jsf2.com/primefaces-themes/themes3.jsf>

Para instalar bixe o JAR do tema desejado, distribua-o como dependência do WAR (coloque em *WEB-INF/lib*) e defina um *<context-param>* em *web.xml* com o nome do tema:

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>bluesky</param-value>
</context-param>
```

Os temas podem ser alterados em tempo de execução usando *<p:themeSwitcher>* que é um tipo de *<h:selectOneMenu>*. Os temas devem ser incluidos com *<f:selectItem>* ou *<f:selectItems>*:

Exemplo de uso:

```

<p:themeSwitcher>
    <f:selectItem itemLabel="– Escolha um tema –" itemValue="" />
    <f:selectItems value="#{bean.listaDeTemas}" />
</p:themeSwitcher>

@Named
public class Bean {
    public String[] getListaDeTemas() {
        String[] temas = { "afterdark", "bluesky", "casablanca", "eggplant", "glass-x" };
        return(temas);
    }
}

```

12.2.1 Configuração de CSS

Os temas podem ser ajustados usando CSS. As principais classes estão listadas abaixo (veja mais na documentação – existem classes específicas para cada componente):

```

.ui-widget (afeta todos os componentes)
.ui-widget-header
.ui-widget-content
.ui-corner-all
.ui-state-default
.ui.state-hover
.ui.state-active
.ui-state-disabled
.ui-state-highlight
.ui-icon

```

Primefaces faz alterações dinâmicas em CSS inserindo tags `<link rel="stylesheet">` no final de `<h:head>`. Se você criar uma folha de estilos CSS para alterar classes do PrimeFaces e carregá-la usando `<link>`, ela não irá funcionar. OS CSS do PrimeFaces tem precedência e irá sobrepor os estilos! A alternativa para forçar a carga apos o CSS do Primefaces é carregar sua folha de estilos usando `<h:outputStyleSheet>` (que é a forma recomendada em JSF).

Outras alternativas são usar bloco `<style>` (que tem precedência sobre folhas carregadas externamente). Ainda assim, devido às regras de cascade do CSS, a aplicação do estilo pode não funcionar. Aplique estilos diretamente em elementos pelo ID, use o atributo `style` ou marcar as definições com `!important` para aumentar a precedência da declaração:

```
.ui-widget {font-size:90% !important}
```

Primefaces não é compatível com frameworks responsivos baseados em HTML5, CSS e JavaScript, e o uso combinado com esses frameworks pode causar erros e resultados inesperados devido a conflitos de JavaScript e CSS, dependendo dos recursos usados.

12.3 Alguns componentes Primefaces

A seguir será apresentada uma seleção de componentes do PrimeFaces em exemplos simples. Os exemplos são genéricos e se destinam a ilustrar a finalidade do componente e sua configuração basica. O código é baseado em componentes das versões 3.4 a 5.0, e pode não estar atualizado com a versão mais recente. Para usar, consulte a documentação online (que é muito boa e detalhada) para exemplos utilizáveis, atributos e detalhes de como usar cada componente.

12.3.1 <p:spinner>



Campo de entrada de dados numéricos equivalente a HTML5 `<input type="number">`. Tipos são convertidos automaticamente (doubles, ints, etc).

Atributos: *min*, *max*, *stepFactor*, etc.

Como usar:

```
<p:spinner value="#{bean.numero}" />
```

```
@Named
public class Bean {
    private double numero;
    public double getNumero() {...}
    public void setNumero(double numero) {...}
}
```

Exemplo (*fonte: coreservlets.com*) usando *<p:ajax>* (similar a *<f:ajax>*)

```
<h:form>
    <p:spinner min="32" max="212" value="#{fBean2.f}">
        <p:ajax update="f c"/>
    </p:spinner>
    <h:outputText value="#{fBean2.f}&deg;F" id="f"/> = <h:outputText
    value="#{fBean2.c}&deg;C" id="c"/>
</h:form>
```

12.3.2 <p:calendar>



Campo de entrada para seleção de data (baseada em JQueryUI). É equivalente a HTML5 *<input type="date">*. Várias configurações visuais são possíveis (veja documentação).

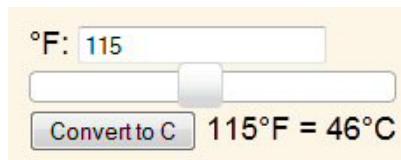
Como usar

```
<p:calendar value="#{bean.data}" />
```

```
@Named
public class Bean {
    private java.util.Date data;
    public java.util.Date getData() {...}
    public void setData(java.util.Date data) {...}
}
```

Alguns atributos de *<p:calendar>*:

- *value* - aponta para propriedade do tipo *Date*
- *pages* - número de meses a exibir de uma vez (default = 1)
- *showOn* (*focus* é default, pode ser *button* ou *both*) - quando o evento de mostrar o calendário será disparado
- *mode* (*popup* – default, ou *inline*) - mostrar o calendário o tempo todo ou apenas quando o usuário clicar
- *pattern="MM/dd/yyyy HH:mm"* - estabelece um padrão para a propriedade (o pattern obedece os padrões da classe *DateFormat*)
- *effect* - mostra calendário com efeito de animação *<p:slider>*



Campo de entrada para faixas de valores inteiros. É equivalente a usar HTML5 `<input type="range">`. Usado em sincronismo com `<h:inputText>`. *Slider* modifica os valores no `inputText` e `inputText` pode modificar posição do *slider*.

Exemplos de uso:

```
<p:slider for="id_do_input" ... />
<h:inputText id="id_do_input" ... />

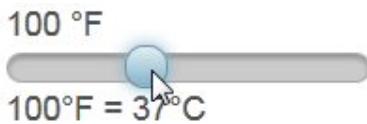
<p:slider for="fonte" display="resultado" ... />
<h:inputHidden id="fonte" ... />
<h:outputText id="resultado" />
```

Exemplo (*fonte: coreservlets.com*) usando ajax para atualizar valores. Facelets:

```
<h:panelGrid width="200">
    <h:panelGroup>
        <h:inputHidden id="fInput3" value="#{fahrBean.f}" />
        <h:outputText id="fDisp3" value="#{fahrBean.f}"/> &deg;F
    </h:panelGroup>
    <p:slider minValue="32" maxValue="212" for="fInput3" display="fDisp3">
        <p:ajax process="fInput3" update="status"/>
    </p:slider>
    <h:outputText value="#{fahrBean.status}" id="status" escape="false"/>
</h:panelGrid>
```

Bean:

```
public class FahrBean {
    ...
    public int getF() { ... }
    public String getStatus() { ... }
}
```



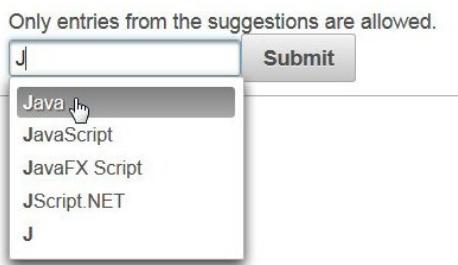
12.3.3 <p:rating>

Basic:	
Callback:	
Ajax Rate:	
Readonly:	
Disabled:	

Geralmente usado para representar um índice de avaliação. Exemplo:

```
<p:rating value="inteiro" />
```

12.3.4 <p:autoComplete>



Campo de texto que mostra um menu com lista reduzida de opções com base nos caracteres digitados.

Alguns atributos:

- *value* – propriedade string do bean que irá guardar o valor selecionado.
- *completeMethod* – método do bean que recebe o string parcial e retorna uma lista de itens que é compatível com esse valor
- *minQueryLength* – número mínimo de caracteres que o value deve ter antes de iniciar o autocomplete.
- *queryDelay* – quantos milisegundos esperar antes de contactar o servidor

Usa ajax para fazer a requisição automaticamente. Pode-se usar <p:ajax> para fazer outras atividades durante os eventos do autoComplete (*itemSelect* ou *itemUnselect*):

```
<p:ajax event="itemSelect" listener="..." />
```

Exemplo de um método (não necessariamente eficiente) que devolve uma lista de opções (para chamar: <p:autoComplete ... completeMethod="#{bean.completar}" .../>)

```
public List<String> completar(String prefixo) {
    List<String> opcoes = new ArrayList<String>();
    for(String opcao: arrayDeOpcoes) {
        if(opcao.toUpperCase().startsWith(prefixo.toUpperCase())) {
            opcoes.add(opcao);
        }
    }
    return(opcoes);
}
```

Exemplo: facelets

```
<p:autoComplete value="#{bean.opcao}"
    completeMethod="#{bean.completar}" />
```

12.3.5 <p:input-mask>



Cria uma máscara para restringir um String de entrada. O formulário mostra como digitar o texto. Não é feita nenhuma conversão e todo o texto é enviado (inclusive a máscara).

Documentação:

<http://digitalbush.com/projects/masked-input-plugin/>

Como usar <p:input-mask>:

- Atributo *value*: o valor armazenado
- Atributo *mask*: texto – qualquer caractere – o texto literal não é editável mas é enviado para o servidor. Caracteres 9, a, * são curingas para número, letra ou ambos, respectivamente.

Ex: "(999) 999-9999" produz a máscara ilustrada acima. Se for digitado 1234567890 será enviado em *value* o texto (123)456-7890.

12.3.6 <p:colorPicker>



Um botão que quando clicado permite escolher uma cor. Valor é string *rrggbb* em hexadecimal e não começa com #. Atributos principais:

- *value*: contém a cor em hexadecimal
- *mode*: *popup* (default) ou *inline* (mostrado na página sem que o usuário aperte o botão)

12.3.7 <p:captcha>



Usa a API do Google (e requer configuração). É necessário registrar em <http://www.google.com/recaptcha/whyrecaptcha> para obter chaves públicas e privadas. Depois as chaves precisam ser registradas como parâmetros no *web.xml*:

```
<context-param>
    <param-name>primefaces.PRIVATE_CAPTCHA_KEY</param-name>
    <param-value>Private key enviada pelo Google</param-value>
</context-param>
<context-param>
    <param-name>primefaces.PUBLIC_CAPTCHA_KEY</param-name>
    <param-value>Public key enviada pelo Google</param-value>
</context-param>
```

Veja documentação em <http://www.google.com/recaptcha/>.

Principais atributos de *<p:captcha />*: *required*, *requiredMessage*, *validatorMessage*, *theme* (*red*, *white*, *blackglass*, *clean*), *language* (default "en"), *validatorMessage* – sobrepõe a mensagem default.

12.3.8 <p:password>

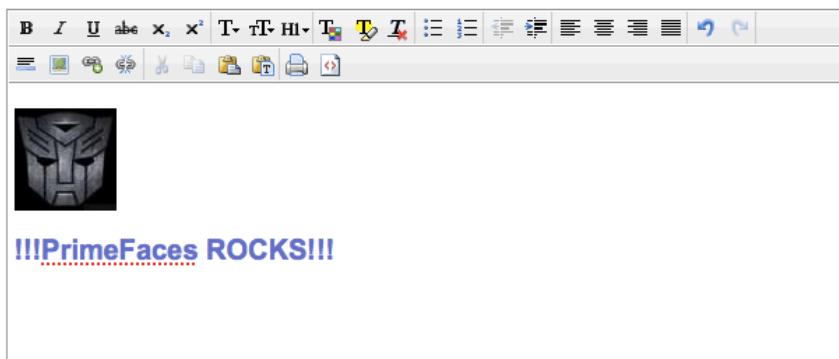


Campo de senha. Versão PrimeFaces para *<p:inputSecret>*. Atributos:

- *value* – propriedade que guarda a senha
- *feedback* (true ou false) - informa se deve haver feedback sobre a qualidade da senha

Pode-se substituir as mensagens default para a qualidade da senha nos atributos: *promptLabel*, *weakLabel*, *goodLabel*, *strongLabel*.

12.3.9 <p:editor>



Editor de textos do *YUI* (usado no Yahoo Mail). O texto digitado pode conter HTML e o componente não protege contra riscos de Cross-Site Scripting (CSS). Usuário precisa tomar as medidas para evitar isso. Veja mais detalhes em www.owasp.org.

Atributos:

- *value* - conteúdo do editor (texto ou EL - opcional)
- *controls* – lista de controles que são exibidos (veja documentação)

12.3.10 Accordion

Wrapper para o painel **accordion** do JQuery UI: tabs horizontais para listar conteúdo e permitir detalhamento ao serem clicados.

Godfather Part I

The story begins as Don Vito Corleone, the head of a New York Mafia family, oversees his daughter's wedding. His beloved son Michael has just come home from the war, but does not intend to become part of his father's business. Through Michael's life the nature of the family business becomes clear. The business of the family is just like the head of the family, kind and benevolent to those who give respect, but given to ruthless violence whenever anything stands against the good of the family.

Godfather Part II

Godfather Part III

Uso básico <*p:accordionPanel*> e <*p:tab*>:

```
<p:accordionPanel>
    <p:tab title="Titulo do primeiro tab">
        Conteúdo JSF
    </p:tab>
    <p:tab title="Titulo do segundo tab">
        Conteúdo JSF
    </p:tab>
    ...
</p:accordionPanel>
```

12.3.11 Tab view

The story begins as Don Vito Corleone, the head of a New York Mafia family, oversees his daughter's wedding. His beloved son Michael has just come home from the war, but does not intend to become part of his father's business. Through Michael's life the nature of the family business becomes clear. The business of the family is just like the head of the family, kind and benevolent to those who give respect, but given to ruthless violence whenever anything stands against the good of the family.

E igual a um JQuery UI Tabbed Panel. Uso básico <p:tabView> e <p:tab>:

```
<p:tabView>
    <p:tab title="Titulo do primeiro tab">
        Conteúdo JSF
    </p:tab>
    <p:tab title="Titulo do segundo tab">
        Conteúdo JSF
    </p:tab>
    ...
</p:tabView>
```

12.3.12 <p:panelGrid>

Versão PrimeFaces do h:panelGrid. Funciona igual ao h:panelGrid mas bordas estão ativadas por default e usa o tema corrente para desenhar a tabela. Em vez de h:panelGroup, usa p:row e p:column. O elemento p:column tem atributos rowspan e colspan (como os que existem em <td>) que são ausentes em h:panelGroup. Por outro lado, não possui os atributos de <table> que estão presentes em h:panelGrid (cellpadding, etc.)

Exemplo de uso básico (fonte: coreservlets.com):

Major Stocks	
coreservlets.com	956.92 (+43.55%)
Prime Technology	887.48 (+37.78%)

```
<p:panelGrid columns="2">
    <f:facet name="header">Minor Stocks</f:facet>
    Google
    <h:outputText value="#{financeBean.google}" />
    Facebook
    <h:outputText value="#{financeBean.facebook}" />
    Oracle
    <h:outputText value="#{financeBean.oracle}" />
</p:panelGrid>
```

Exemplo de row e column:

Mailing List Signup	
Name:	<input type="text"/>
Email:	<input type="text"/>
<input type="button" value="Sign Up"/>	

```
<p:panelGrid>
    <f:facet name="header">
        <p:row>
            <p:column colspan="2">Mailing List Signup</p:column>
        </p:row>
    </f:facet>
    <p:row>
        <p:column>Name:</p:column>
        <p:column><p:inputText/></p:column>
    </p:row>
    <p:row>
        <p:column>Email:</p:column>
        <p:column><p:inputText/></p:column>
    </p:row>
    <p:row>
        <p:column colspan="2" style="text-align: center">
            <p:commandButton value="Sign Up"/>
        </p:column>
    </p:row>
</p:panelGrid>
```

12.3.13 Outros componentes

A seguir uma lista de outros componentes que *não* serão abordados aqui (consulte online detalhes na documentação e exemplos de uso).

1) Componentes que permitem organizar o layout da página:

- p:dashboard
- p:scrollPanel
- p:layout
- p:outputPanel
- p:toolbar
- p:wizard



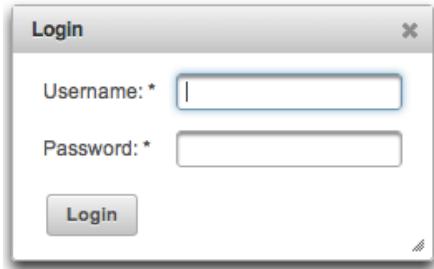
2) Formas alternativas de exibir mensagens

- <p:messages>
- <p:tooltip>
- <p:growl>



3) Janelas, popups, overlays

- <p:dialog>
- <p:confirmDialog>
- <p:overlayPanel>
- <p:lightbox>
- <p:notificationBar>



13 Mecanismos de extensão

13.1 Passthrough

JavaServer Faces gera código XHTML a partir de tags. Esse processo elimina atributos que são gerados automaticamente (ex: name, em formulários) e outros que não fazem parte da especificação). Pode-se usar diretamente quaisquer tags do HTML5 em páginas JSF, mas eles não serão processados como parte de requisições faces nem participação do ciclo de vida. Portanto, o HTML gerado pelos facelets não é compatível com HTML5.

Até a versão 2.1 de JSF era necessário programar com o RenderKit (escrever métodos Java e registrar em faces-config.xml) para lidar com atributos desconhecidos. Com a versão JSF 2.2 foi criado um mecanismo para permitir o uso de quaisquer atributos nos tags gerados. Isto é possível associando-os a um namespace próprio. Isto permite não apenas que o JSF 2.2 suporte HTML5, como também que inclua atributos específicos de outros frameworks, como o Bootstrap ou Foundation. O namespace é:

`http://xmlns.jcp.org/jsf/passthrough`

Para usá-lo ele deve ser associado a um prefixo. Na documentação da Oracle geralmente usa-se “p” que conflita com o uso de Primefaces, então é preciso escolher outro, por exemplo “a”, e declarar o namespace:

```
<html ... xmlns:a="http://xmlns.jcp.org/jsf/passthrough">
```

Os atributos que não fazem parte do tag então podem ser usados desde que prefixados. Por exemplo, para gerar um `<input type="number">` do HTML5 pode-se usar:

```
<h:inputText id="num" value="#{bean.numero}" a:type="number" />
```

Na verdade o JSF 2 não gosta muito dessas combinações. O ideal é escolher entre HTML5 e JSF, Bootstrap e JSF, e evitar combiná-los. *Passthrough* e outros mecanismos são uma espécie de “gambiarra” para viabilizar esse uso, mas é preciso ter cuidado com a maneira como esses componentes serão renderizados no browser, e dos conflitos que eles irão causar. Isto envolve não apenas o conflito de CSS, mas também envolve a forma como os browser renderizam componentes HTML5. Por exemplo, a maior parte dos browsers renderiza `<input type="date">` como um calendário. Embora identificar campos de data com “date” seja recomendado pelo HTML5, haverá problemas se o tag for gerado por um componente de calendário do Primefaces.

13.2 Integração com Bootstrap

Como foi mencionado nas seções anteriores, JSF não funciona muito bem com outros frameworks que interferem no look & feel da interface Web, e combiná-lo com frameworks baseados em HTML5, JavaScript e CSS é um desafio.

Mas, devido à popularidade do HTML5 e de frameworks como JQuery, AngularJS e Bootstrap, têm surgido vários pacotes para integrá-los com JSF. Algumas das soluções mais populares são distribuídos pelo Primefaces:

- *PrimeUI* – uma coleção de componentes HTML5 baseados em JQueryUI
- *PrimeNG* – uma coleção de componentes HTML5 baseados em AngularJS

Para Bootstrap, o Primefaces oferece um tema, que fornece um look & feel *similar* ao BootStrap (mas não é uma integração completa). É possível configurar as folhas de estilo e adaptar algumas funções específicas, removendo conflitos de CSS, mas ainda há conflitos de script em componentes interativos, devido às incompatibilidades das bibliotecas JavaScript.

Existe um framework chamado *BootsFaces*, baseado em JQuery UI e Bootstrap 3 que é compatível com JSF e PrimeFaces e oferece uma integração maior. A desvantagem é um novo vocabulário de tags e atributos para aprender.

Em suma, não existe uma solução que ofereça integração simples e completa, nem existe nenhuma recomendação oficial.

14 Referências

14.1 Especificações e documentação oficial

1. Oracle. Java Server Faces Specification. <https://jcp.org/aboutJava/communityprocess/final/jsr344/index.html>
2. Oracle. The Java EE 6 Tutorial. <http://docs.oracle.com/javaee/6/tutorial/doc/>
3. Oracle. The Java EE 7 Tutorial. <http://docs.oracle.com/javaee/7/tutorial/doc/>
4. Contexts and Dependency Injection Specification. <http://www.cdi-spec.org/>
5. W3C. Ajax. <http://www.w3.org/TR/XMLHttpRequest/>
6. IETF RFC 2616 HTTP: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
7. W3C. XHTML. <http://www.w3.org/MarkUp/> (padrão descontinuado)
8. Manual do PrimeFaces. <http://primefaces.org/documentation.html>

14.2 Tutoriais e artigos

1. Bauke Scholtz. *The BalusC Code*. <http://balusc.omnifaces.org/> e vários artigos sobre JSF do StackOverflow organizados em <https://jsf.zeef.com/bauke.scholtz>.
2. Marty Hall. PrimeFaces Tutorial Series. <http://wwwcoreservlets.com/JSF-Tutorial/primefaces/>
3. M. Kyong. JSF 2.0 Tutorials. <http://www.mkyong.com/tutorials/jsf-2-0-tutorials/>

14.3 Livros

1. David Geary & Cay Horstmann. *Core JavaServer Faces 2.0*. Prentice-Hall, 2010.
2. Ed Burns. *JavaServer Faces 2.0: The Complete Reference*. McGraw-Hill, 2009.
3. Arun Gupta. *Essential Java EE 7*. O'Reilly, 2013.

14.4 Produtos

1. Mojarra <https://javaserverfaces.java.net/2.2/download.html>
2. JBoss Weld <http://weld.cdi-spec.org/>
3. Tomcat <http://tomcat.apache.org/>
4. JBoss <http://www.jboss.org/>
5. Glassfish <https://glassfish.java.net/>
6. PrimeFaces <http://primefaces.org/>