DESENVOLVIMENTO WEB BACK END

AULA 1

Prof. Rafael Veiga de Moraes

CONVERSA INICIAL

Antes de começar a codificação do sistema, o desenvolvedor deve entender as necessidades de seus clientes para que, a partir disso, possa prover uma solução robusta que atenda às suas demandas. Para isso, ele deve realizar a análise de requisitos do projeto. Os requisitos são divididos em dois grupos: os requisitos funcionais (RF) e os requisitos não funcionais (RNF).

Os *requisitos funcionais* especificam todas as regras de negócio do projeto, fornecendo uma visão para o desenvolvedor de todos os recursos que devem ser implementados no sistema. Já os *requisitos não funcionais* especificam como os requisitos funcionais serão implementados, definindo, assim, a infraestrutura do projeto. Para ficar mais claro esses conceitos, vamos elencar alguns requisitos, tendo como base um sistema de gerenciamento de contas a pagar e a receber.

Requisitos funcionais

- Manter cadastro de usuário.
- Manter cadastro de clientes.
- Manter cadastro de fornecedores.
- Manter cadastro de contas a pagar.
- Manter cadastro de contas a receber.
- Manter cadastro de forma de pagamento.
- Emissão de Nota Fiscal eletrônica (NFe).
- Gerar relatório de contas a pagar e a receber pela data de emissão.
- Gerar relatório de contas a pagar e a receber pela data da baixa.

Requisitos não funcionais

- O sistema deve ser desenvolvido em linguagem Java.
- O tempo máximo para processamento de uma requisição não pode exceder 10s.
- O sistema gerenciador de banco de dados utilizado pela aplicação será o MySql.
- A aplicação deve ser compatível com os navegadores: Safari, Chrome, Firefox e Edge.

- O sistema deverá utilizar o serviço web (WS – W*eb Service*) dos correios para consulta dos endereços pelo CEP.

Anteriormente, citamos apenas alguns requisitos de um sistema de contas a pagar e a receber. Em um projeto comercial, iremos nos deparar com centenas ou até milhares de requisitos a serem implementados. Portanto, desenvolver uma aplicação *Web* é uma tarefa complexa que irá demandar um considerável tempo de desenvolvimento. Se implementar os requisitos funcionais exige um grande esforço, imagine se o desenvolvedor tivesse que se preocupar com a implementação dos requisitos não funcionais, como persistência em banco de dados, interpretação das requisições HTTP, gerenciamento de sessão, gerenciamento de *threads,* serviços *web* e enviar notificações por *e-mail.*

Certamente, a complexidade do projeto seria ainda maior e talvez até o tornasse inviável. Diante disso, o desenvolvedor deve buscar ferramentas que minimizem a complexidade do projeto para que ele foque apenas na codificação das regras de negócio da aplicação, aumentando a sua produtividade e reduzindo o custo de desenvolvimento do projeto.

Para isso, iremos abordar o *Spring Framework*, uma das ferramentas mais utilizadas para o desenvolvimento de aplicações Java Web. Antes de abordar este *framework* em específico, precisamos compreender alguns conceitos de arquitetura de sistemas e como funciona uma aplicação Java dentro da plataforma Java EE (*Enterprise Edition*).

TEMA 1 – ARQUITETURA DE SISTEMAS

Para compreender a arquitetura de uma aplicação Java EE, iremos abordar nesse tópico alguns conceitos referentes a arquitetura de sistemas de um modo geral. A arquitetura de uma aplicação é dividida em duas categorias: arquitetura física e arquitetura lógica.

1.1. Arquitetura física

Corresponde à infraestrutura necessária para execução da aplicação e pode ser dividida em três camadas distintas: cliente, servidor Java EE e servidor EIS.

1.1.1 Cliente

O cliente (*Client Machine*) corresponde ao dispositivo ou software utilizado para acessar a aplicação. O acesso pode ser realizado através de um dispositivo móvel utilizando um aplicativo ou estação de trabalho por meio de um navegador ou sistema *desktop*.

1.1.2 Servidor

O servidor é a máquina responsável por prover os serviços à aplicação e realizar a comunicação com outros servidores. A seguir, listamos algumas das responsabilidades de um servidor:

- persistência em banco de dados;
- interpretação das requisições HTTP;
- gerenciamento de sessão;
- gerenciamento de *threads*;
- gerenciamento de carga;
- serviços *web*; e
- enviar notificações por *e-mail.*

Para que o servidor possa prover os serviços, é necessário que nele seja instalado um software que implemente esses serviços. Dentro da plataforma Java EE, temos dois tipos de softwares que realizam tal tarefa: servidor de aplicação e *servlet container*.

1.2 Sistemas de Informação Corporativos

Os Sistemas de Informação Corporativos (EIS – *Enterprise Information System*) é um servidor externo à aplicação que fornece algum tipo de serviço para a aplicação, sendo, na maioria das vezes, composto apenas pelo servidor de banco de dados (*Database Server*).

1.3 Arquitetura lógica

Uma das principais preocupações do desenvolvedor é com relação à manutenção do código do projeto, pois à medida que a aplicação vai crescendo, novas classes e serviços vão sendo incorporados ao sistema. Para diminuir a complexidade e manter o código do projeto bem estruturado, flexível e de fácil compreensão, é comum dividir a arquitetura da aplicação em diferentes partes lógicas denominadas *camadas*.

1.3.1 Modelo de uma camada

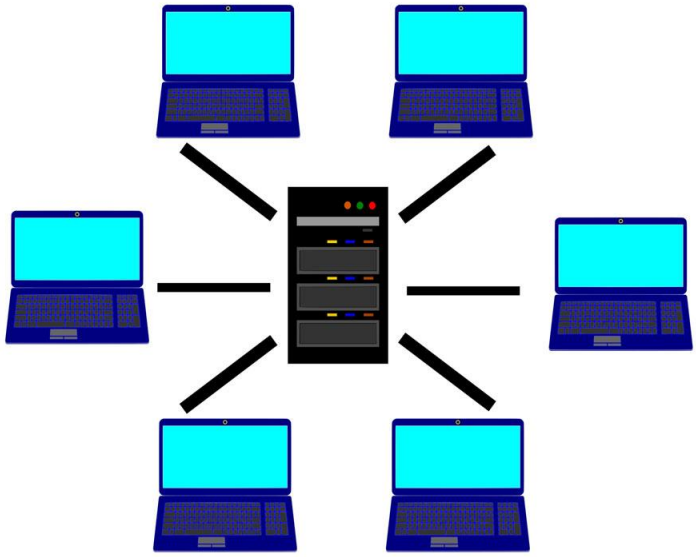
Também conhecido como aplicações monolíticas ou centralizadas, surge nos anos 1960 com o uso de banco de dados ainda bem rudimentar, perdurando até meados dos anos 1980. É caracterizado por todo o processamento da aplicação ser realizado de forma centralizada em computadores de grande porte (*mainframes*), nos quais estão alocados todos os recursos do sistema (interface, banco de dados e a implementação das regras de negócio). Os terminais de interação com os *mainframes* são conhecidos como terminais *burros* ou *mudos*, pois eles não armazenam e tampouco processam qualquer tipo de informação.

1.3.2 Modelo de duas camadas

Amplamente conhecido como modelo cliente-servidor, essa arquitetura predomina ao longo da década de 1980, devido às diversas transformações e inovações que ocorreram nesse período. Entre elas, destacam-se:

- surgimento das interfaces gráficas;
- a popularização dos computadores pessoais (PC – *Personal Computer*);
- a utilização de redes locais (LAN – *Local Area Network*); e
- a adoção de um SGBD (Sistema Gerenciador de Banco de Dados) para gerenciamento de dados da aplicação.

Figura 1 – Modelo cliente-servidor



Crédito: Snehalata/Shutterstock.

Uma aplicação cliente-servidor é dividida em duas camadas distintas.

- **Camada do cliente**: composta pelas estações de trabalho nas quais a aplicação está instalada, cuja finalidade é prover a interação entre o usuário e o software.
- **Camada do servidor**: representada por uma máquina dedicada a atender as requisições efetuadas pelo usuário, cuja finalidade é prover os dados para a aplicação. Nela está instalado o SGBD.

A implementação das regras de negócio da aplicação pode estar ou na camada do cliente ou na camada do servidor, porém é preferível implementá-las na camada do cliente a fim de evitar a sobrecarga de processamento no servidor. Essa solução é conhecida como *cliente gordo* (*fat-client*).

1.3.3 Modelo multicamadas

A arquitetura de modelo multicamadas se consolida com a popularização da internet a partir da metade da década de 1990. Nela, as estações de trabalho se comunicam com o servidor de banco de dados através de uma camada intermediária, diferentemente do que ocorre no modelo cliente-servidor. Essa evolução arquitetônica tem como objetivo distribuir o processamento da aplicação a fim de evitar a sobrecarrega de processamento sobre uma única

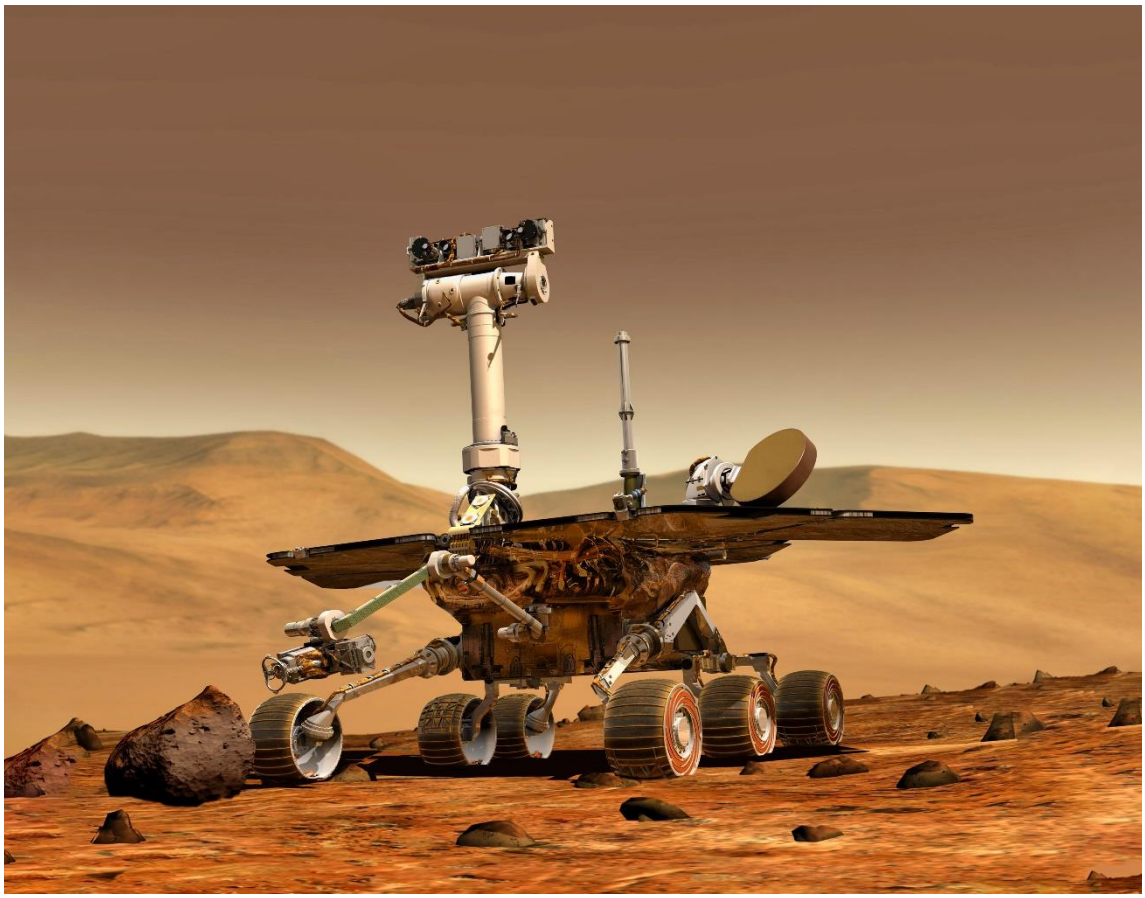
camada. Para uma aplicação ser considerada como multicamadas, é necessário que ela apresente ao menos três camadas específicas.

- **Camada de apresentação**: a camada de apresentação também é conhecida como Interface Gráfica com o Usuário (GUI – *Graphical User Interface*). Nela ocorre a interação entre o usuário e o software por meio de uma interface gráfica, que é acessada por meio de uma estação de trabalho. A camada de apresentação se comunica apenas com a camada de domínio da aplicação, na qual serão processadas todas as ações efetuadas pelo usuário através da interface do software.
- **Camada de domínio**: na camada de domínio da aplicação, estão implementadas todas as regras de negócio da aplicação. Ela se comunica tanto com a camada de apresentação, efetuando o processamento das requisições efetuadas pelo usuário, quanto com a camada de fonte de dados, realizando a comunicação com o banco de dados e/ou outros sistemas. A comunicação com as camadas de apresentação e de fonte de dados ocorre por meio do servidor Java EE.
- **Camada de fonte de dados**: a camada de fonte de dados é responsável pela troca de informações com outros sistemas, sendo, na maioria das vezes, responsável apenas pela comunicação da aplicação com o banco de dados. Nessa camada, encontramos os sistemas de informação corporativos (EIS – *Enterprise Information System*) que possuem a capacidade de trocar informações com outros sistemas, característica conhecida como *interoperabilidade*.

TEMA 2 – PLATAFORMA JAVA EE

Atualmente, a linguagem Java é a plataforma de desenvolvimento mais utilizada no mundo e conta com mais de 9 milhões de desenvolvedores ao redor do globo. As aplicações desenvolvidas em Java estão presentes no nosso dia a dia, podemos encontrá-las executando em diversos dispositivos, como *smartphones*, *tablets*, *notebooks*, eletrodomésticos, televisores, além de jogos, *websites* e até mesmo em Marte.

Figura 2 – Robô *Spirit* controlado remotamente por uma aplicação Java



Crédito: Best-Backgrounds/Shutterstock.

A linguagem Java conta com quatro plataformas distintas, na qual cada plataforma provém um conjunto de especificações para o desenvolvimento de aplicações com propósitos bem definidos. Entre elas, iremos abordar a plataforma Java EE (*Enterprise Edition*), voltada para o desenvolvimento de aplicações corporativas de grande porte.

Conforme a documentação da Oracle, a plataforma Java EE fornece um poderoso conjunto de APIs a fim de reduzir o tempo e a complexidade de desenvolvimento do projeto, além de otimizar o desempenho da aplicação. Utilizando as especificações dessa plataforma, é possível desenvolver aplicações distribuídas, escaláveis, portáteis, transacionais que aproveitam a velocidade, a segurança e a confiabilidade da tecnologia do lado do servidor.

2.1 Arquitetura

A arquitetura de uma aplicação Java EE é baseada no modelo distribuído em multicamadas, composta por três camadas físicas e quatro camadas lógicas.

A **arquitetura física** é composta pelas seguintes camadas.

- Máquina do cliente (*Cliente Machine*): dispositivo pelo qual o usuário interage com a aplicação.
- Servidor Java EE (*Java EE Server*): máquina responsável por prover os serviços para a aplicação.
- Servidor de banco de dados (*Database Server*): máquina responsável pelo armazenamento de dados da aplicação.

A **arquitetura lógica** é composta pelas seguintes camadas.

- Camada do cliente (*Client Tier*): camada de apresentação da aplicação.
- Camada *Web* (*Tier Web*): camada responsável pelo processamento das requisições e das páginas *Web*.
- Camada de negócio (*Business Tier*): camada na qual estão implementadas as regras de negócio.
- Camada de sistemas de informação corporativos (*EIS Tier*): camada de fonte de dados da aplicação.

Além disso, uma aplicação Java EE é composta por componentes, unidade de software funcional independente que são executados dentro de uma aplicação Java EE. Entre eles, destacam-se:

- cliente de aplicativo (*Application client*) e miniaplicativo (*Applet*), componentes executados na camada do cliente;
- *servlets*, JavaServer *Faces* (JSF) e JavaServer *Pages* (JSP), componentes executados na camada *Web* do servidor Java EE; e
- enterprise JavaBeans (EJB), componente executado na camada de negócio do servidor Java EE.

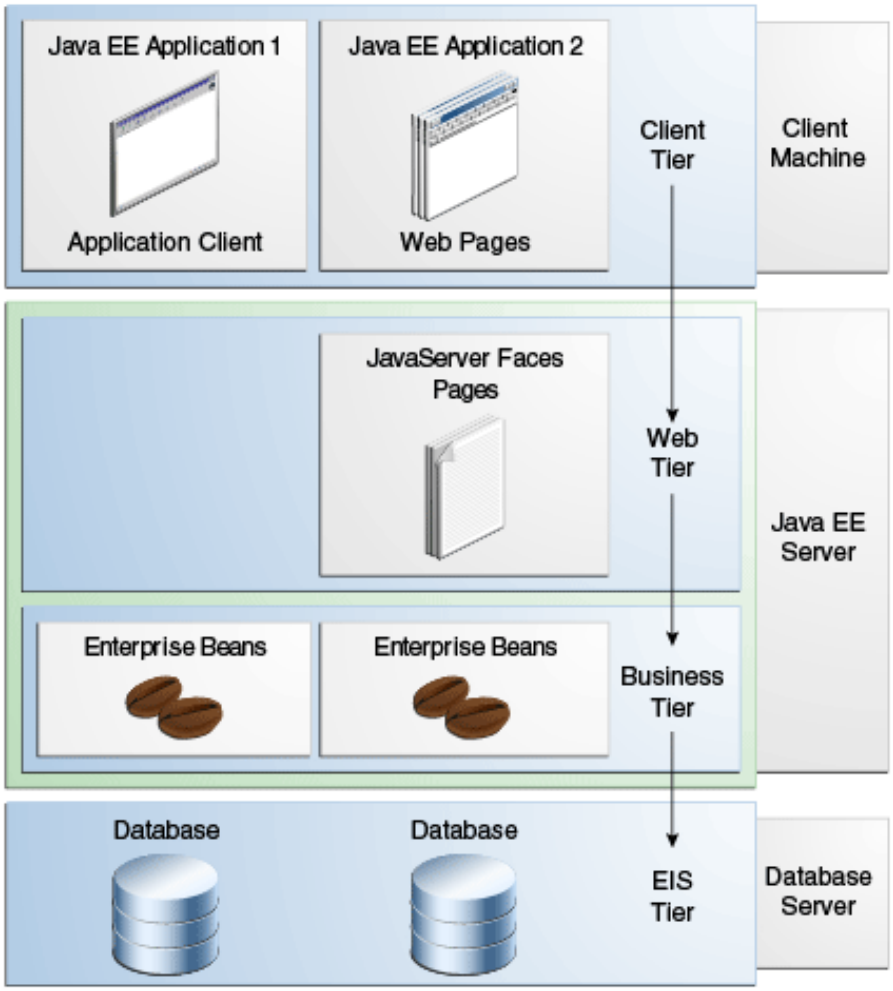
O ciclo de vida de cada componente é gerenciado por um *container*. No ambiente de uma aplicação Java EE, deparamo-nos com os seguintes *containers*:

- *Applet Container* – gerencia o ciclo de vidas dos *applets;*
- *Application Client Container* – gerencia o ciclo de vida dos componentes do cliente de aplicativo;
- *Web Container* – gerencia o ciclo de vida dos componentes *Web;* e

- EJB *Container* – gerencia o ciclo de vida dos componentes de negócio.

A disposição das camadas e dos componentes de uma aplicação Java EE podem ser visualizados na figura a seguir, extraída da documentação oficial da Oracle.

Figura 3 – Arquitetura Java EE



Fonte: Oracle, [S.d.].

Nos próximos temas, serão abordados mais detalhadamente os elementos que compõem o ambiente de uma aplicação Java EE.

TEMA 3 – AMBIENTE DA PLATAFORMA JAVA EE

Conforme mencionado anteriormente, a arquitetura da aplicação Java EE é distribuída em multicamadas, devendo conter ao menos três camadas lógicas: apresentação, domínio e fonte de dados. A seguir, serão abordados os elementos que compõem cada uma dessas camadas.

3.1 Camada de apresentação

A camada de apresentação da aplicação corresponde à camada do cliente (*Tier Client*), responsável por prover a interface gráfica da aplicação. A interface pode ser acessada por meio de uma página *Web* (*Web Page*) ou cliente de aplicativo (*Application Client*).

3.1.1 Cliente *Web*

O cliente *Web* é composto por dois elementos.

- **Páginas *Web* dinâmicas (*Dynamic Web Pages*)**: contém a interface gráfica da aplicação fornecida pelos componentes de camada *Web* do servidor Java EE.
- **Navegador *Web* (*Web Browser*)**: responsável pelo processamento das páginas recebidas do servidor Java EE.

O cliente *Web* também é chamado de *cliente magro* (*thin client*), pois, geralmente, não efetuam o processamento das regras de negócio da aplicação, delegando essa tarefa ao servidor da aplicação.

3.1.1.1 Cliente de aplicativo

Um cliente de aplicativo (*Application Client*) é executado em uma máquina cliente que fornece ao usuário uma interface para que este possa realizar as suas tarefas, geralmente desenvolvida a partir da API *Swing* ou API *Abstract Window Toolkit* (AWT). Os clientes de aplicativo acessam diretamente os EJBs presentes na camada de negócios da aplicação, porém, se necessário, também podem estabelecer uma conexão HTTP com uma *servlet* que esteja sendo executada na camada *Web*. É importante destacar que os clientes de aplicativo desenvolvidos em outras linguagens de programação também podem interagir com uma aplicação Java EE.

3.1.1.2 *Applet*

Criado para adicionar interatividade às páginas *Web* que antes eram totalmente estáticas, o *applet* (miniaplicativo) é um programa desenvolvido para executar uma tarefa específica dentro do contexto de outra aplicação. Como

exemplo de *applet,* podemos citar os *plugins*, programas que são instalados em um navegador com o objetivo de adicionar funcionalidades e recursos das páginas *Web*.

3.1.1.3 Camada de domínio

É responsável por prover os serviços necessários para o processamento de dados da aplicação. Esses serviços são disponibilizados pelo servidor Java EE (*Java EE Server*) através de um conjunto de componentes e APIs que contém a implementação das especificações da plataforma Java EE. Há dois tipos de servidores que implementam essas especificações dentro da plataforma Java EE.

- **Servidor de aplicação (*Application Server*)**: software que contém a implementação de todas as especificações da plataforma Java EE, sendo indicado para o desenvolvimento de aplicações de grande porte. Os servidores de aplicação destacam-se por fornecerem suporte para a criação de componentes *Web* e EBJ. Entre eles, destacam-se:
  - Apache Geronimo;
  - *GlassFish;*
  - JBoss;
  - JOracle *WebLogic;*
  - IBM *WebSphere;*
  - SAP *NetWeaver; e*
  - SAP *Web Application.*
- ***Servlet container***: possui algumas limitações quando comparado ao servidor de aplicações, pois não fornece suporte para todas as especificações da plataforma Java EE, apenas para componentes *Web*, sendo uma ótima alternativa para o desenvolvimento de aplicações de pequeno e médio porte. Os *servlets containers* mais utilizados do mercado são:
  - Apache Tomcat; e
  - Jetty.

O domínio da aplicação está dívida em duas camadas com propósitos diferentes: camada *Web* e camada de negócio.

3.1.1.4 Camada *Web*

A camada *Web* é responsável por efetuar o processamento das requisições HTTP e das páginas Web. Nessa camada, encontram-se os seguintes componentes:

- ***Servlets*** – responsável por tratar as requisições HTTP solicitadas pelos clientes.
- **JavaServer *Faces* (JSF)** – tecnologia utilizada para desenvolvimento de páginas *Web* dinâmicas por meio de componentes pré-fabricados.
- **JavaServer *Pages* (JSP)** – tecnologia utilizada para o desenvolvimento das páginas Web dinâmicas por meio de *taglibs* ou *scriptlets*.

3.1.1.5 Camada de negócio

A camada de negócio é responsável pela implementação das regras de negócio da aplicação. Nessa camada, encontra-se o seguinte componente:

- **Enterprise JavaBean (EJB)** – tecnologia que permite o desenvolvimento rápido e simplificado de aplicações distribuídas, transacionais, seguras e portáteis.

3.1.2 Camada de sistemas de informação corporativos

Camada de sistemas de informação corporativos corresponde a camada de fonte de dados, responsável pela interação com servidores externos a aplicação, sendo, na maioria das vezes, composta apenas pelo servidor de banco de dados.

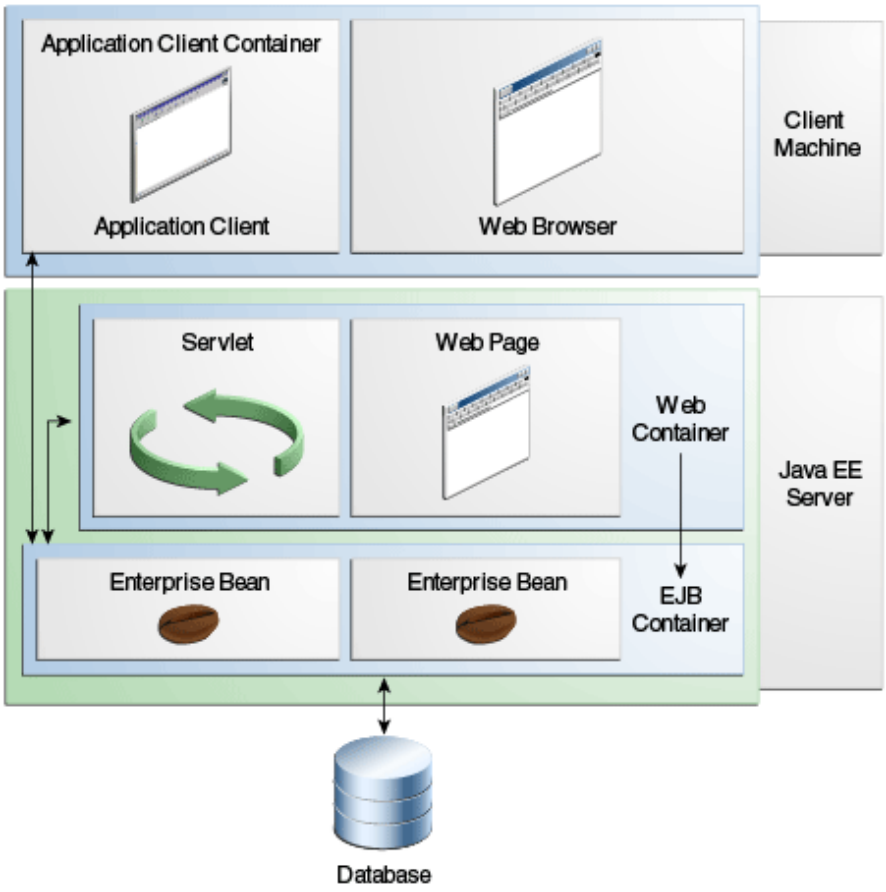
3.2 Containers

Os *containers* da plataforma Java EE são responsáveis por manter a comunicação entre componentes, APIs e demais serviços da aplicação, sendo eles gerenciados pelo software do fornecedor de servidor Java EE, que disponibilizam um container para cada tipo de componente. Os *containers* são divididos em quatro grupos.

- ***Applet Container***: gerencia o ciclo de vidas dos *applets.*

- ***Application Client Container***: gerencia o ciclo de vida dos componentes do cliente de aplicativo.
- ***Web Container***: gerencia o ciclo de vida dos componentes *Web*.
- **EJB *Container***: gerencia o ciclo de vida dos componentes de negócio.

Figura 4 – Componentes da plataforma Java EE

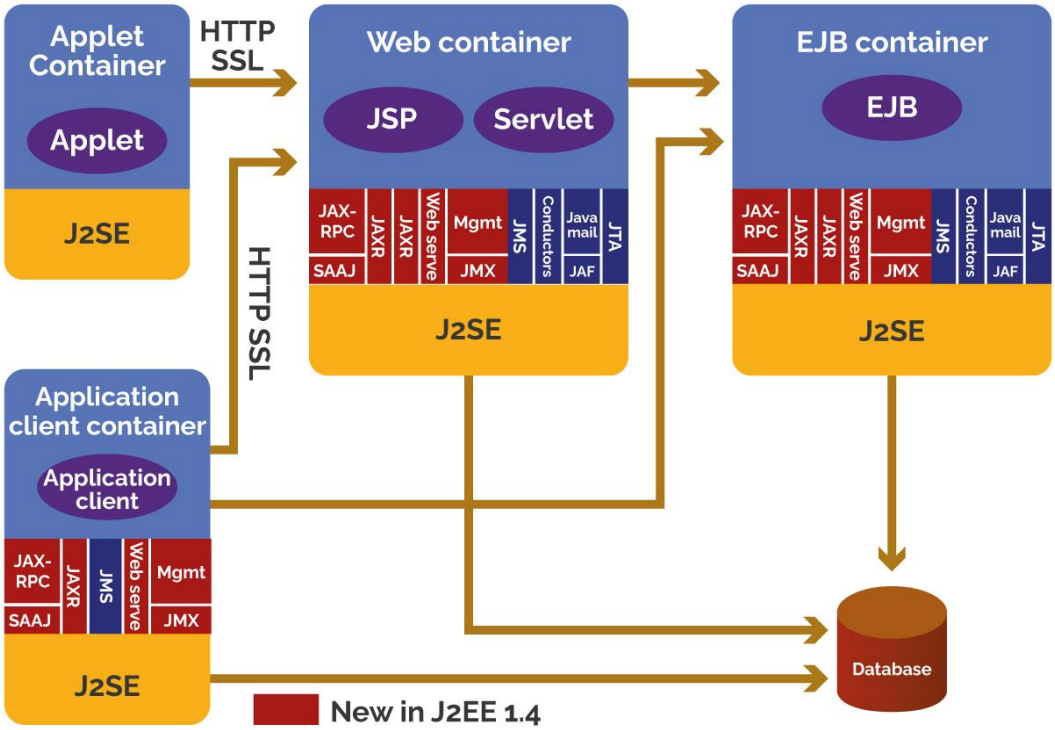


Fonte: Oracle, [S.d.].

Os *containers* fornecem diversos serviços para os componentes. Serão listados alguns deles:

- persistência de dados (JDBC – *Java Database Connection*);
- controle transacional (JTA – *Java Transaction API*);
- segurança (JAAS – *Java Authentication and Authorization Service*);
- envio de notificações por *e-mail* (JavaMail); e
- conectividade (RMI – *Remote Method Invocation*).

Figura 5 – *Containers* da plataforma Java EE



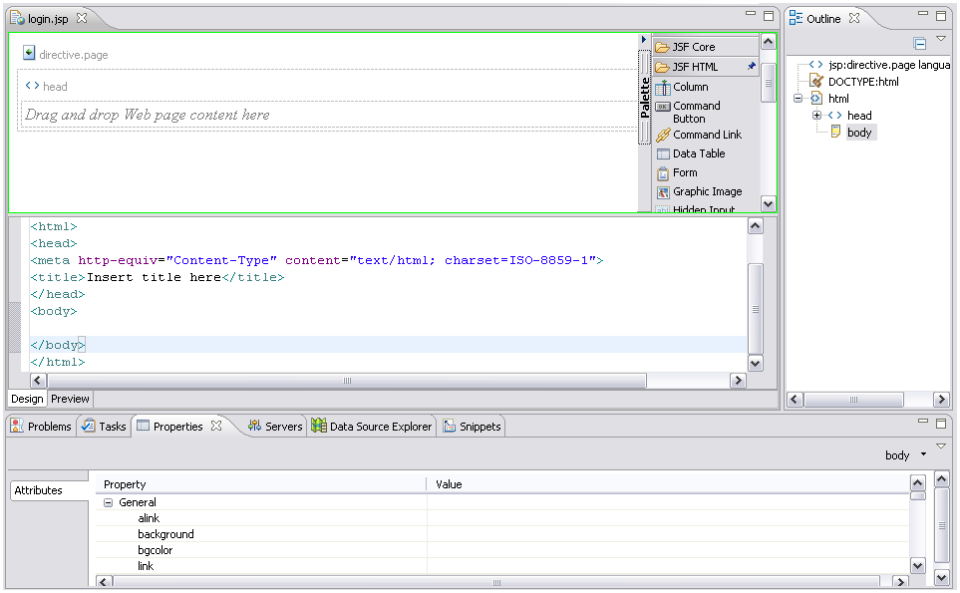
TEMA 4 – COMPONENTES DA PLATAFORMA JAVA EE

Neste tema, serão abordados os componentes da plataforma Java EE e suas funcionalidades.

4.1 JavaServer *Faces*

A JSF (*JavaServer Faces*) é um componente *Web* da plataforma Java EE utilizado para o desenvolvimento da interface gráfica da aplicação. Essa tecnologia permite criar uma página *Web* de forma rápida e prática, pois fornece um conjunto de componentes pré-fabricados que podem ser adicionados a um formulário. Cada componente possui um conjunto de eventos que permitem ao desenvolvedor saber qual ação foi executada pelo usuário e implementar a rotina pertinente ao evento executado.

Figura 6 – Interface do Eclipse para criação de páginas JSF



Na figura anterior, temos a interface do Eclipse para a criação de páginas JSF. Nessa tela, o desenvolvedor arrasta os componentes pré-fabricados como botões, tabelas, links, imagens, entre outros componentes para o corpo da página, criando, assim, de forma rápida, a interface de uma página *Web*.

4.2 JavaServer *Pages*

A JSP (*Java Server Pages*) é outro componente *Web* da plataforma Java EE utilizado para o desenvolvimento da interface gráfica da página *Web*. Essa tecnologia permite desenvolver páginas dinâmicas, embutindo código Java dentro do código HTML por meio dos *scriptlets* (bloco de comandos compreendidos entre as tags “<%” e “%>”) ou das *taglibs* da biblioteca JSTL. No quadro a seguir, é exibido o código de uma página JSP utilizando *scriptlets* para exibir uma informação na tela.

Quadro 1 – Código fonte de uma JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
   <head>
      <title>Universidade</title>
   </head>
   <body>
      <h1>
         <% out.println("Olá Mundo!"); %>
      </h1>
   </body>
</html>
```

Diferentemente da especificação JSF, a JavaServer *Pages* requer do desenvolvedor um conhecimento mais apurado para o desenvolvimento da interface da página *Web*, uma vez que ele terá que implementar e estilizar os seus próprios componentes. Para isso, ele deve conhecer as três linguagens listadas a seguir.

- **HTML**: linguagem de marcação utilizada para inserir conteúdo e definir a estrutura da página *Web*.
- **JavaScript**: linguagem de script utilizada para implementar os eventos dos componentes HTML.
- **CSS**: linguagem de estilo utilizada para alterar os elementos visuais dos componentes HTML.

4.3 *Servlets*

A *servlet* é um componente da plataforma Java EE que permite ao desenvolvedor efetuar o processamento das operações realizadas pelo usuário dentro da aplicação. Para cada operação efetuada no cliente, será enviada ao servidor uma requisição HTTP com as informações necessárias para o seu processamento. Para isso, o desenvolvedor deverá implementar uma *servlet* para atender tal requisição e retornar uma resposta para o cliente, com as informações referentes ao seu processamento.

Baseado nas informações contidas na resposta do servidor, é possível identificar se a requisição HTTP foi processada com sucesso ou se ocorreu algum erro durante o seu processamento. Com isso, a aplicação consegue dar um *feedback* adequado para o usuário a respeito da ação efetuada. Uma *servlet* deve implementar a interface HttpServlet e tanto a requisição como a resposta HTTP pode ser acessada através dos objetos das classes HttpServletRequest e HttpServletResponse, respectivamente. No quadro a seguir, é exibido o código de uma *servlet*.

Quadro 2 – Código fonte de uma *servlet*

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {
   private static final long serialVersionUID = 1L;

   // Requisição HTTP método GET
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      // Bloco de comandos
   }

   // Requisição HTTP método POST
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      // Bloco de comandos
   }
}
```

4.4 Enterprise Java *Beans*

O Enterprise Java *Beans* (EJB) é um componente da camada de negócio que permite o desenvolvimento rápido e simplificado de aplicações distribuídas, transacionais, seguras e portáteis dentro da plataforma Java EE. Os EJBs estão divididos em duas categorias: *Session Beans* e *Message-driven Beans*. A configuração dos EJB pode ser realizada por meio de anotações (*Annotations*) ou por meio do arquivo web.xml, conhecido como *Deployment Descriptor*.

4.4.1 *Message-Driven Bean*

O *bean* do tipo *Message-Driven* (MDB) permite que as aplicações Java EE processem mensagens de forma assíncrona. Esse tipo de *bean* pode implementar qualquer tipo de mensagem, sendo mais comum o processamento de mensagens do tipo JMS (*Java Message Service*). Um *Message-Driven Bean* implementa a *interface* MessageListener e é identificado pela anotação @MessageDriven, na qual deve ser especificado o JNDI do JMS destino, por meio do atributo mappedName. Sempre que esse *bean* for invocado, será executado o método onMessage, que contém a lógica de negócio a ser processada pela aplicação.

Quadro 3 – Código fonte de um *Message-Driven Bean*

```
@MessageDriven(mappedName = "destino")
public class MeuMessageBean implements MessageListener {
   @Override
   public void onMessage(Message message) {
      try {
         // processamento da mensagem
      } catch (JMSException ex) {
         // tratamento das exceções
      }
   }
}
```

4.4.2 *Session Beans*

Os *Session Beans* implementam as regras de negócio da aplicação. Existem três tipos de *Session Beans* que são classificados de acordo com o seu ciclo de vida: *Stateless*, *Stateful* e *Singleton*.

4.4.2.1 *Stateless Session Bean*

O *bean* sem estado de sessão (*Stateless Session Bean*) possui um ciclo de vida correspondente ao tempo de execução do método invocado. Esse tipo de *bean* é utilizado na implementação de classes cujos métodos não exigem o armazenamento de dados entre uma requisição e outra. Isso significa dizer que o *Stateless Session Bean* não mantém um estado conversacional com o cliente. As classes sem estado de sessão são identificadas com a anotação @Stateless, conforme podemos verificar no quadro a seguir.

Quadro 4 – Código fonte de um *Stateless Session Bean*

```
@Stateless
public class Calculadora {
   public float adicao(float operador1, float operador2) {
      return operador1 + operador2;
   }

   public float subtracao(float operador1, float operador2) {
         return operador1 - operador2;
   }

   public float multiplicacao(float operador1, float operador2) {
         return operador1 * operador2;
   }

   public float divisao(float operador1, float operador2) {
         return operador1 / operador2;
   }
}
```

4.4.2.2 *Stateful Session Bean*

Diferentemente do *Stateless Session Bean*, o *bean* com estado de sessão (*Stateless Session Bean*) mantém o estado conversacional com o cliente. É importante destacar que um *bean* do tipo *Stateful* não pode ser compartilhado entre clientes. Um exemplo clássico de *Stateful Session Bean* é o carrinho de compras de um *e-commerce*. Ao adicionar um produto ao carrinho de compras, a aplicação deve manter o seu estado para que o usuário possa continuar navegando pelo *site* e adicionar novos produtos ao carrinho de compras. As classes com estado de sessão são identificadas com a anotação @Stateful, conforme podemos verificar no quadro a seguir.

Quadro 5 – Código fonte de um *Stateful Session Bean*

```
@Stateful
public class CarrinhoDeCompras {

   List<Produto> produtos;

   public CarrinhoDeCompras () {
      produtos = new ArrayList<Produto>();
   }

   public void adicionaProduto(Produto produto) {
      produtos.add(produto);
   }

   public void removeProduto(Produto produto) {
      produtos.remove(produto);
   }

   @Remove
   public void finalizaCompra() {
      produtos = null;
   }
}
```

4.4.2.3 *Singleton Session Bean*

O *bean* do tipo *Singleton* mantém o estado de conversação de um *bean* durante o tempo de execução da aplicação. Esse *Session Bean* é muito similar ao *bean* do tipo *Stateful*, com a diferença que o *bean* do tipo *Singleton* possui apenas uma única instância, a qual pode ser acessada simultaneamente por diversos clientes. No quadro a seguir, temos um EJB do tipo *Singleton* que nos permite identificar a quantidade de acessos realizados na aplicação.

Quadro 6 – Código-fonte de um *Singleton Session Bean*

```
@Singleton
public class ContadorDeAcessos {
   int contador;

   public void incrementar() {
      ++contador;
   }

   public void getAcessos() {
      return contador;
   }

   public void reiniciar() {
         contador = 0;
   }
}
```

Neste tema, serão abordadas as principais especificações e bibliotecas da plataforma Java EE.

5.1. JavaServer *Pages Standard Tag Lib*

A JSTL (*JavaServer Pages Standard Tag Lib*) é uma biblioteca composta por um conjunto de *tags* que podem ser utilizadas para o desenvolvimento da JSP como uma alternativa ao *scriptlet*, tornando o código da página mais legível e enxuto. A JSTL é composta por cinco pacotes, conforme ilustra a tabela a seguir.

Tabela 1 – Pacotes da JSTL

Pacote	Prefixo	Funcionalidade
JSTL *Core*	c	Comandos condicionais, iterativos e de atribuição
JSTL fmt	fmt	Formatação de dados
JSTL sql	sql	Persistência de dados
JSTL xml	xml	Manipulação e criação de documentos XML
JSTL *functions*	Fn	Processamento de *strings* e coleção de dados

Cada pacote JSTL conta com um conjunto de *tags* que estão agrupadas de acordo com as suas funcionalidades. Na tabela a seguir, elencaremos algumas das tags que compõem o pacote JSTL Core e suas respectivas funções.

Tabela 2 – *Tags* do pacote JSTL *Core*

Tag	Função
choose, when* e *otherwise	Utilizadas para analisar várias alternativas para uma condição específica, formam uma estrutura semelhante ao comando *switch* em Java.
if	Utilizada para exibir um determinado conteúdo na JSP caso a condição analisada seja verdadeira.
forEach	Utilizada para realizar a iteração de uma coleção.
out	Utilizada para exibir uma informação na tela.
set	Utilizada para atribuir um valor a uma variável.

Algumas *tags* exigem o preenchimento de atributos, espécie de variáveis internas da *tag*, para que elas possam ser executadas, como no caso da tag *out*. Na tabela a seguir, serão listados os atributos da *tag out*.

Tabela 3 – Atributos da *tag out*

Atributo	Descrição	Obrigatório
value	Utilizado para exibir uma informação.	Sim
default	Utilizado para exibir uma informação quando o valor do atributo *value* for igual a nulo.	Não
escapeXML	Utilizado para verificar se os caracteres especiais '&', '<' e '>' do atributo *value* devem ser convertidos em suas respectivas entidades HTML. O valor padrão desse atributo é verdadeiro.	Não

As *tags* JSTL são identificadas dentro de uma JSP pelo prefixo do pacote seguido de dois pontos mais a tag do pacote. No código a seguir, foi utilizada a *tag out* do pacote JSTL *Core* para exibir uma informação na tela. A informação a ser exibida foi informada no atributo *value* da *tag out*, como podemos observar no quadro a seguir.

Quadro 7 – JSP com *taglib*

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
   <head>
      <title>Universidade</title>
   </head>
   <body>
      <c:out value="Olá mundo com JSTL!"/>
   </body>
</html>
```

5.2 *Expression Language*

A *Expression Language* (EL) permite ao desenvolvedor acessar dinamicamente os dados dos componentes Java *Beans* e criar expressões lógicas e aritméticas. Além disso, através da EL, é possível acessar diversas informações da página como *cookies*, cabeçalhos, parâmetros, variáveis de

escopo e diversos outros atributos por meio dos chamados objetos implícitos. A EL possui a seguinte sintaxe:

Quadro 8 – Sintaxe de uma EL

```
${ expressão }
```

No quadro a seguir, temos o código de uma JSP com EL que exibe uma mensagem de boas-vindas para o usuário logado no sistema.

Quadro 9 – Código JSP com EL

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
       <title>Universidade</title>
    </head>
    <body>
       <h1>Seja bem-vindo ${usuario.nome}</h1>
    </body>
</html>
```

5.3 Java *Persistence* API

A JPA (Java *Persistence* API) é um serviço para realizar a persistência de dados da aplicação através do Mapeamento Objeto-Relacional (ORM – *Object-Relational Mapping*). Para isso, devemos utilizar as anotações do pacote javax.persistence.

No Quadro 9, temos um exemplo do Mapeamento Objeto-Relacional por meio do JPA, na qual temos as seguintes anotações.

- @Entity: especifica que a classe é uma entidade, portanto, os dados dessa classe serão armazenados em uma determinada tabela no banco de dados. Caso o nome da classe seja diferente do nome da tabela, basta informar no atributo *name* dessa anotação o nome da respectiva tabela.
- @Id: especifica quais são os atributos que compõem a chave primária da entidade.
- @GeneratedValue: especifica como será gerado o valor do atributo-chave da classe. Neste caso, foi adotada estratégia GenerationType.AUTO, na

qual o provedor de persistência criará o identificador de forma automática de acordo com o SGBD adotado.

- @Column: especifica que o dado do atributo será armazeno em uma determinada coluna da tabela. Caso o nome do atributo da classe seja diferente do nome da coluna, basta informar no atributo *name* dessa anotação o nome da respectiva coluna.
- @Temporal: especifica que o atributo é do tipo temporal: data, hora ou timestamp (data e hora).

Na documentação da JPA, você encontra mais informações a respeito das anotações e seus respectivos atributos.

Quadro 10 – Código fonte de uma classe com JPA

```
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Pessoa {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  @Column(nullable = false)
  private String nome;
  private String cpf;
  @Temporal(TemporalType.DATE)
  @Column(name = "dataNasc", nullable = false)
  private Date dataNascimento;
  @Column

  // declarar os getters e setters
}
```

5.4 JavaMail

As aplicações Java EE utilizam a API JavaMail para o envio de notificações por *e-mail*. No Quadro 11, é exibida a classe JavaMail que implementa esse serviço, adotando como exemplo uma conta do Gmail. Para que essa classe possa funcionar corretamente, deve-se definir o conteúdo das seguintes variáveis.

- *E-mail*: informar o *e-mail* que fará o envio das notificações.
- Senha: informar a senha do *e-mail* que fará o envio das notificações.

Quadro 11 – Código-fonte de uma classe com JavaMail

```
import java.util.Properties;
import javax.mail.Address;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class ServicoDeEmail {
   public static void enviaNotificacao(String destinatario, String
assunto, String conteudo) {
      Properties props = new Properties();

      // Dados do remetente
      String email = "", senha = "";

      // Configuração do servidor
      props.put("mail.smtp.host", "smtp.gmail.com");
      props.put("mail.smtp.socketFactory.port", "465");
      props.put("mail.smtp.socketFactory.class",
"javax.net.ssl.SSLSocketFactory");
      props.put("mail.smtp.auth", "true");
      props.put("mail.smtp.port", "465");

      // Autenticação
      Session session = Session.getDefaultInstance(props,
         new javax.mail.Authenticator() {
            protected PasswordAuthentication  getPasswordAuthentication(){
            return new PasswordAuthentication(email, senha);
         }
      });

      try {
         // Envio da notificação por e-mail
         Message message = new MimeMessage(session);
         message.setFrom(new InternetAddress(email));
         Address[] toUser = InternetAddress.parse(destinatario);

         message.setRecipients(Message.RecipientType.TO, toUser);
         message.setSubject(assunto);
         message.setText(conteudo);

         Transport.send(message);
      } catch (MessagingException e) {
         throw new RuntimeException(e);
      }
   }
}
```

5.5 Java *Beans*

Java *Beans* é uma classe extremante simples que tem como objetivo encapsular e abstrair uma entidade do sistema. Para que uma classe seja considerada um *bean*, ela deve atender aos seguintes requisitos:

- implementar a interface *Serializable*;
- possuir apenas o construtor padrão; e
- seus atributos são acessados apenas pelos métodos *get* e *set.*

No quadro a seguir, temos um exemplo de uma classe do tipo Java *Bean*.

Quadro 12 – Código fonte de um Java *Bean*

```
import java.util.Date;

public class Pessoa implements java.io.Serializable {
   private String cpf;
   private String nome;
   private Date dataDeNascimento;

   public Pessoa() {
      this.nome = null;
      this.cpf = null;
      this.date = null;
   }

   public String getCpf() {
      return cpf;
   }
   public void setCpf(String cpf) {
      this.cpf = cpf;
   }

   public Date getDataDeNascimento() {
      return dataDeNascimento;
   }
   public void setDataDeNascimento(Date dataDeNascimento) {
      this.dataDeNascimento = dataDeNascimento;
   }

   public String getNome() {
      return nome;
   }
   public void setNome(String nome) {
      this.nome = nome;
   }
}
```

5.6 Java *Transaction* API

A JTA (Java *Transaction* API) fornece uma interface padrão de alto nível para demarcação de transações por meio do pacote javax.transaction. Uma

transação consiste em uma sequência de operações que são executadas em conjunto, como se fosse uma única unidade lógica. Ela segue o conceito ACID, acrônimo para as quatros propriedades de uma transação: atomicidade, consistência, isolamento e durabilidade. A transação pode ser especificada tanto em nível de classe quanto em nível de método por meio da anotação @Transaction.

Quadro 13 – Transação em nível de classe

```
@Transactional
public class Bean {
  // Definição da classe
}
```

Quadro 14 – Transação em nível de método

```
public class Bean {
   public void metodo1() {
       // Bloco de comandos
   }

   @Transactional
   public void metodo2() {
     // Bloco de comandos
   }
}
```

5.7 Java *Database Connectivity* API

A JDBC (Java *Database Connectivity*) é uma API que fornece um conjunto de classes e interfaces para que a aplicação possa realizar a persistência de dados através de um *driver* específico para o SGBD desejado, devendo este ser importado ao projeto.

Quadro 15 – Classe de conexão com JDBC

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {
   public static Connection createConnection() throws SQLException{
      String connectionString = "jdbc:mysql://localhost:3306/[catalogo]";
      String usuario = "";
      String senha = "";

      Connection conexao = null;
      conexao = DriverManager.getConnection(connectionString, usuario,
senha);

      return conexao;
   }
}
```

Adotando o MySQL como SGBD da aplicação, no Quadro 12, temos a classe que estabelece a conexão com o banco de dados, devendo-se definir as seguintes variáveis.

- *ConnectionString*: substituir o termo [catalogo] pelo nome da base de dados que se deseja acessar.
- Usuário: informar o usuário de acesso ao banco de dados.
- Usuário: informar a senha do usuário de acesso ao banco de dados.

Para executar os comandos DML (*Data Manipulation Language*), como *insert*, *update*, *delete* e *select*, iremos utilizar o objeto PrepareStatement e para obter o resultado da consulta a interface *ResultSet*. No Quadro 16, temos o exemplo de um código referente ao processo de inserção de dados. Para isso, deve-se instanciar o objeto PreparedStatement por meio do método prepareStatement do objeto de conexão, passando por parâmetro o comando SQL a ser executado. Para executar o comando SQL, basta invocar o método execute do objeto instanciado anteriormente.

Quadro 16 – Código-fonte referente à inserção de dados

```
String query = "INSERT INTO pessoa ("
             + "   nome,"
             + "   cpf,"
             + "   dataNascimento"
             + ") VALUES ("
             + "   'Rafael',"
             + "   '99988877766',"
             + "   '2021-10-04'"
             + ")";

PreparedStatement ps = conexao.prepareStatement(query);
ps.execute();
```

Para efetuar a consulta, também deve-se instanciar o objeto PrepareStatement, porém, dessa vez, será invocado o método executeQuery, que irá retornar os registros da consulta em variável *ResulSet*, conforme podemos visualizar no Quadro 17.

Quadro 17 – Código-fonte referente à consulta de dados

```
conexao = ConnectionFactory.createConnection();

String sql = "SELECT id, nome FROM pessoa";
PreparedStatement ps = conexao.prepareStatement(sql);


ResultSet rs = ps.executeQuery();
while(rs.next()){
      int codigo = rs.getInt("id");
      String nome = rs.getString("nome");
       System.out.printf("Código %d: %s\n",codigo, nome);
}
```

Os registros da consulta podem ser acessados por meio de um laço de repetição utilizando o método *next* do objeto *ResultSet*.

FINALIZANDO

Nesta aula, abordamos a plataforma Java EE voltada para o desenvolvimento de aplicações corporativas de grande porte. Ela fornece uma série de APIs que facilitam o processo de desenvolvimento de uma aplicação, de modo que o desenvolvedor não precise se preocupar em desenvolver os requisitos não funcionais, focando apenas na implementação dos requisitos funcionais.

Além disso, foram abordados todos os elementos que compõem as diferentes camadas do ambiente de uma aplicação Java EE, enfatizando, principalmente, a camada do servidor da aplicação, tendo em vista que iremos implementar e utilizar algumas das especificações que são executadas nessa camada.

Futuramente, iremos abordar o *Spring Framework*, objeto de estudo desse curso, abordando as suas especificações e traçando comparativos com as especificações da Java EE, justificando o porquê de ele ser um dos *frameworks* de desenvolvimento Java mais utilizados do mercado.

REFERÊNCIAS

CNES. **Java Runs remote-controlled Mars rover**: Java runs remote-controlled Mars rover – CNET. set. 2021.

ECLIPSE. **WTP Tutorials – JavaServer Faces Tools Tutorial:** WTP Tutorials – JavaServer Faces Tools Tutorial. Disponível em: <eclipse.org>. Acesso em: 17 dez. 2021.

ORACLE. **Java Platform, Micro Editon (Java ME)**. 2021. Disponível em: <https://www.oracle.com/java/technologies/javameoverview.html>. Acesso em: 17 dez. 2021.

ORACLE. **Introduction to Java EE**. 2021. Disponível em: <https://javaee.github.io/tutorial/overview001.html>. Acesso em: 17 dez. 2021.

ORACLE. **Obtenha informações sobre a Tecnologia Java**. Disponível em: <https://www.java.com/pt-BR/about/>. Acesso em: 17 dez. 2021.

ORACLE. **Onde obter informações técnicas sobre o Java**. 2021. Disponível em: <https://www.java.com/pt-BR/download/help/techinfo_pt-br.html/>. Acesso em: 17 dez. 2021.

ORACLE. **Java™ EE 8 Specification APIs**. 2021. Disponível em: <https://javaee.github.io/javaee-spec/javadocs/>. Acesso em: 17 dez. 2021.

ORACLE. **JavaServer Pages Standard Tag Library 1.1 Tag Reference**: Overview (TLDDoc Generated Documentation). Disponível em: <oracle.com>. Acesso em: 17 dez. 2021.

ORACLE. **Overview of the EL**: overview of the EL. Disponível em: <javaee.github.io>. Acesso em: 17 dez. 2021.

ORACLE. **Enterprise JavaBeans (EJBs)**: enterprise JavaBeans (EJBs). Disponível em: <oracle.com>. Acesso em: 17 dez. 2021.

ORACLE. **J2EE 1.4 APIs**: J2EE 1.4 APIs. Disponível em: <oracle.com>. Acesso em: 17 dez. 2021.