GlobalEducation

Converging Education and Technology

Mini-curso Java 2 Enterprise Edition

Introdução

ÍNDICE

Java Startup

Relacionamos abaixo alguns dos motivos pelos quais Java e J2EE são tecnologias de sucesso no desenvolvimento de softwares:

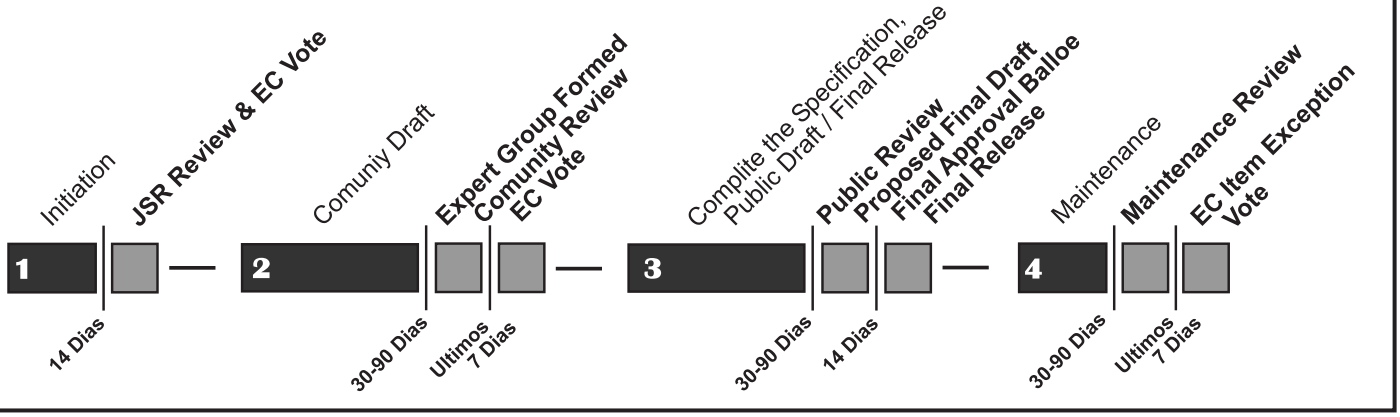
Portátil / Independente de Plataforma

** Compilada e interpretada com excelente performance.*
** Os sistemas operacionais dos principais fabricantes de software e hardware do mercado possuem máquina virtual Java (HP/Compaq, Sun, IBM, Linux, Microsoft etc.).*

Java Community Process

** Processo estabelecido pela Sun responsável pela evolução do Java. Composto por empresas, universidades, "Java Individual Expert" e público em geral.*
** Qualquer um pode participar, porém a melhor maneira é tornando-se um membro da comunidade.*
** Os projetos que são mantidos pelo JCP são votados pela comunidade e pelo comitê executivo que é composto por diversas empresas. A votação que elege os novos membros do comitê ocorre de 3 em 3 anos.*

Java Community Process Timeline

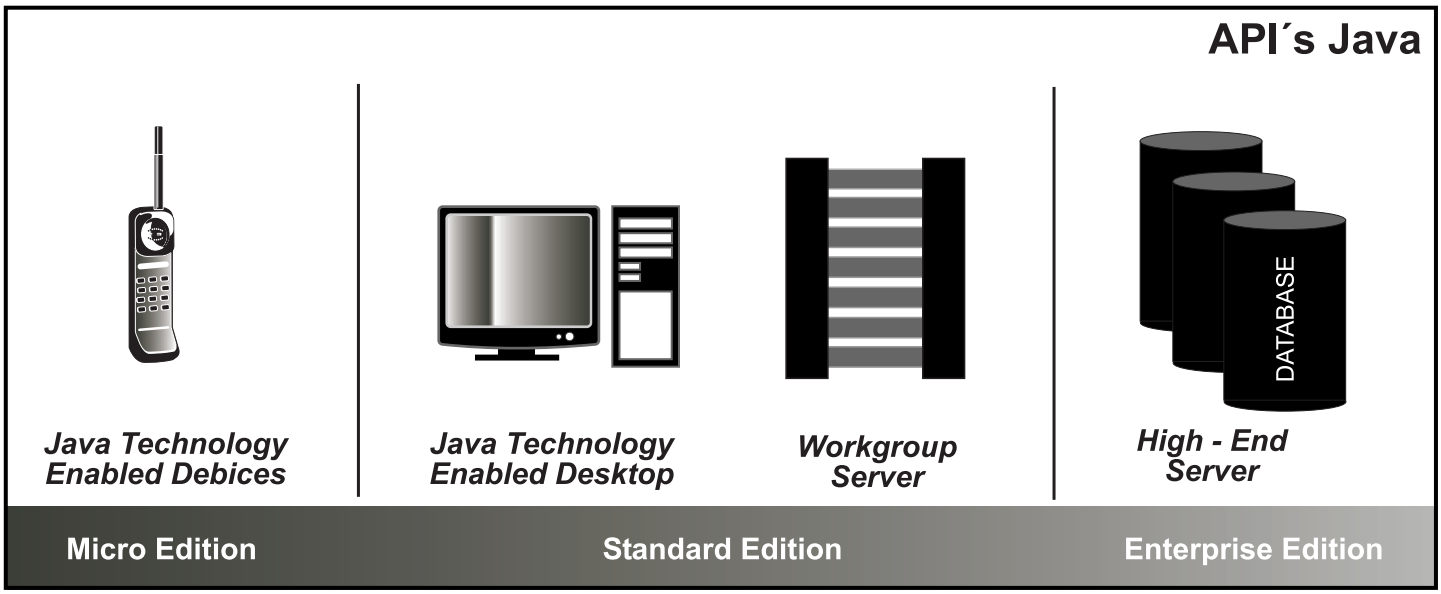


JCP Standard/Enterprise Edition Executive Committee

* Number of eligible voters: 406. Percent voting members casting votes: 22.2% The top two Members have been elected and will serve for the next three years.
Macromedia, Inc.: 20%
Nokia Networks: 17.4%
WebGain: 11.6%
SAP AG Walldorf: 10.3%
Art Technology Group, Inc (ATG): 10.3%
Silverstream Software: 7%
Nortel Networks: 5.8%
Hitachi, Ltd.: 5.8%
Dresdner Kleinwort Wasserstein: 5.1%
Tower Technology Corporation: 3.2%
SeeBeyond Technology Corp.: 2.5%
IOPSIS Software Inc.: .6%

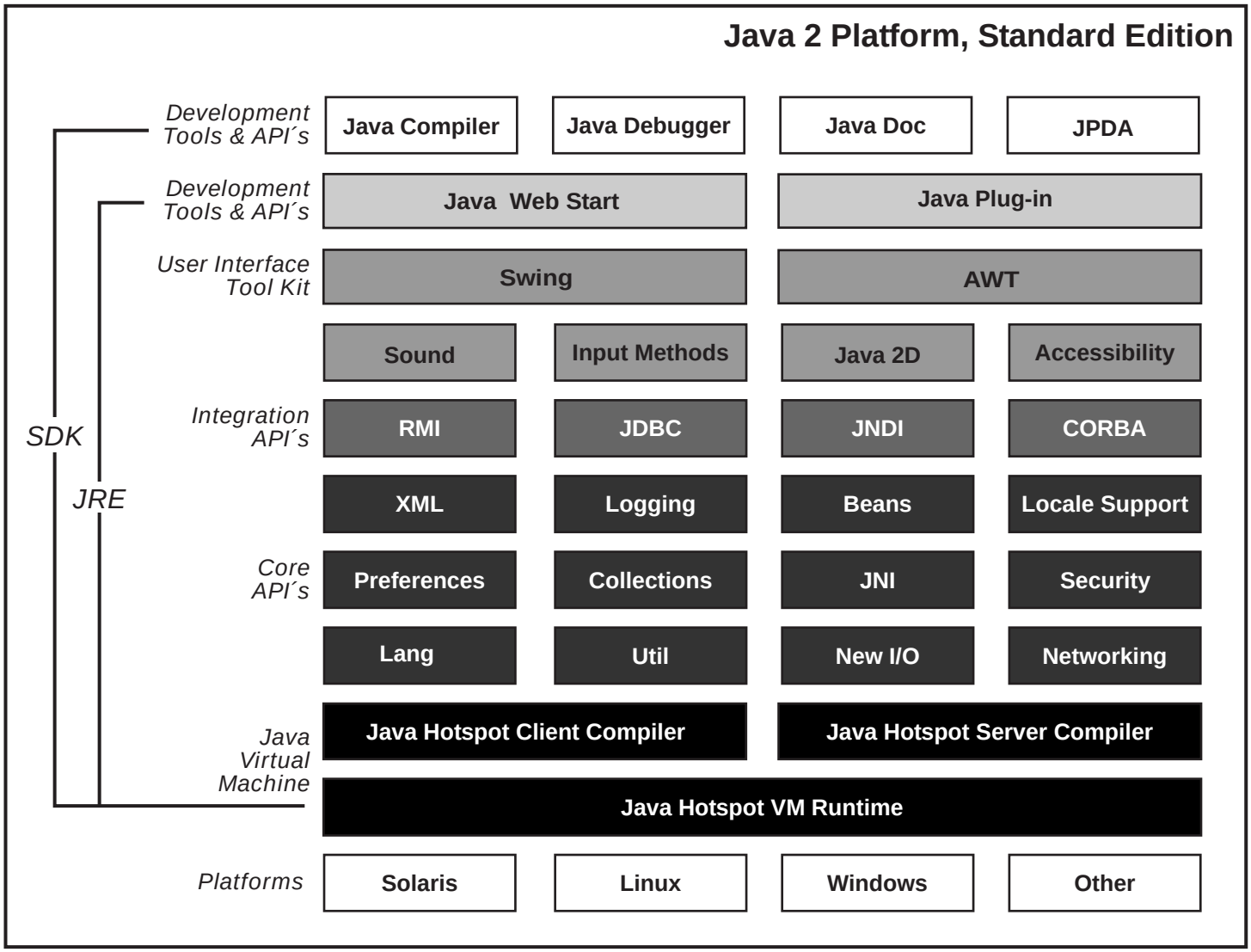
API's Java

Com um processo de padronização bem estabelecido o número de API's cresce continuamente, para termos uma idéia, em apenas dois anos a linguagem se tornou a linguagem com maior número de API's do mercado.
Java do ponto de vista de API's é dividida em 3 partes: Standard, Enterprise e Micro Edition.

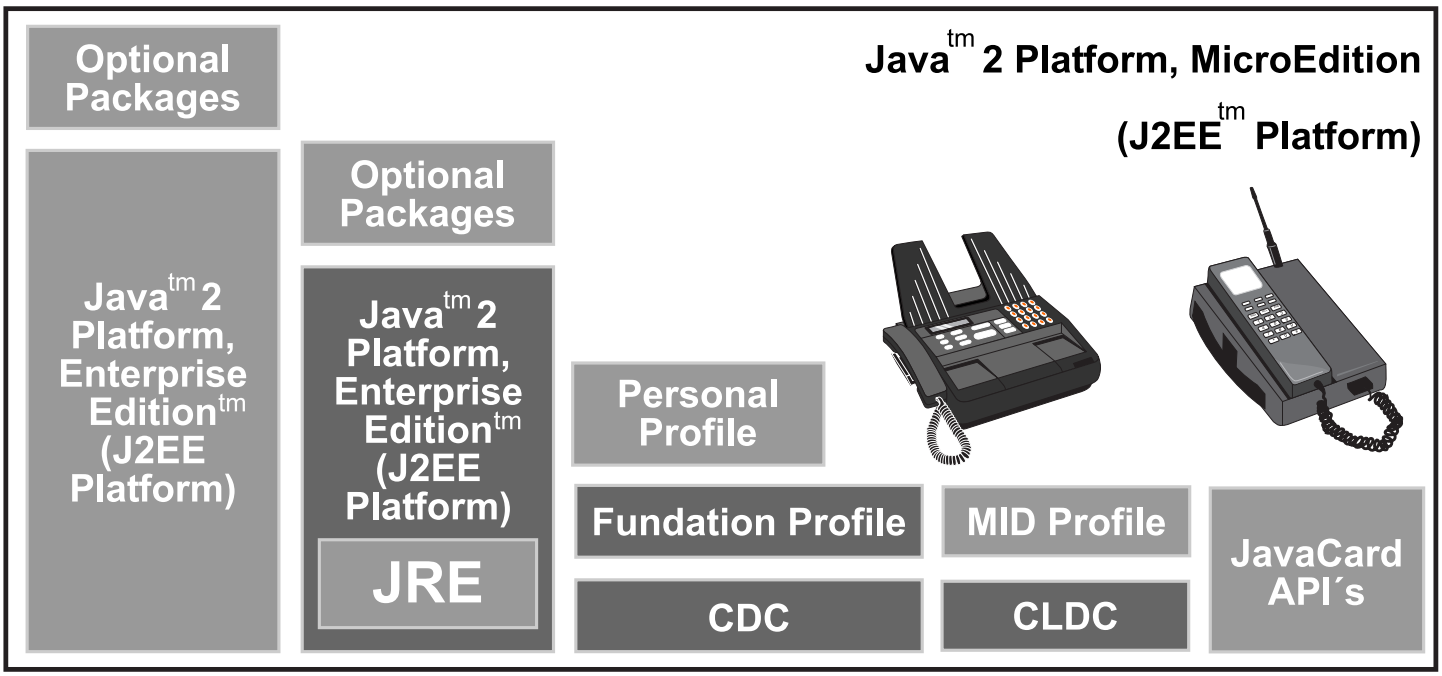


Java 2 Standard Edition

J2SE é o **"core"** da linguagem Java. É a raiz da tecnologia e está presente, mesmo que em pequenas partes, em praticamente todas aplicações Java.



Java 2 Micro Edition

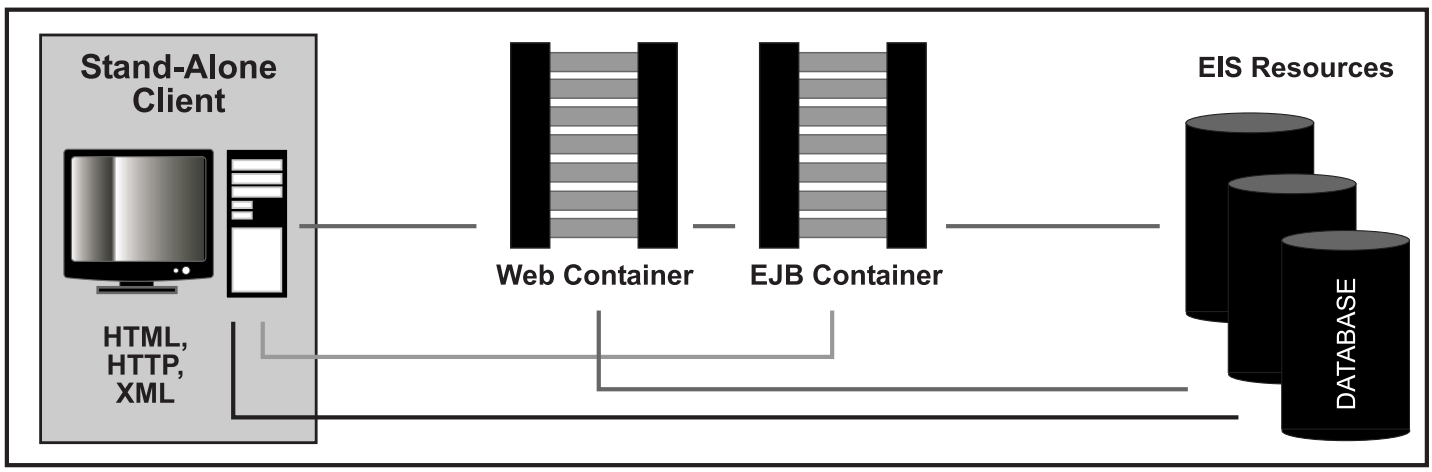


Subset do Java para micro-dispositivos

.

Aplicações para pda, celulares, pagers, mini-computadores diversos.

Java 2 Enterprise Edition



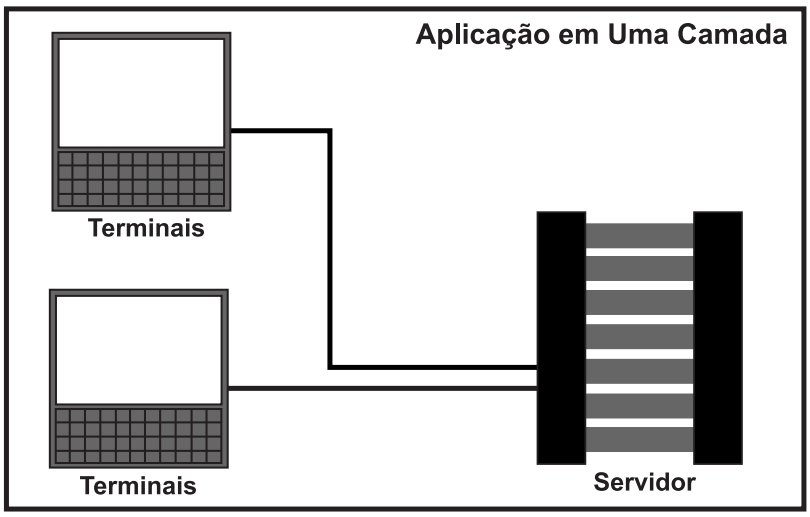
O que é J2EE?

** Um padrão composto por um conjunto de API's.*
** Um padrão para servidores de aplicação (middlewares).*
** Um padrão para desenvolvimento de aplicações multi-camadas.*
** Uma maneira de desenvolver aplicações para Web e Enterprise.*

Para uma melhor abordagem no **J2EE**, vamos relembrar os modelos de arquitetura de software e suas divisões de camadas.

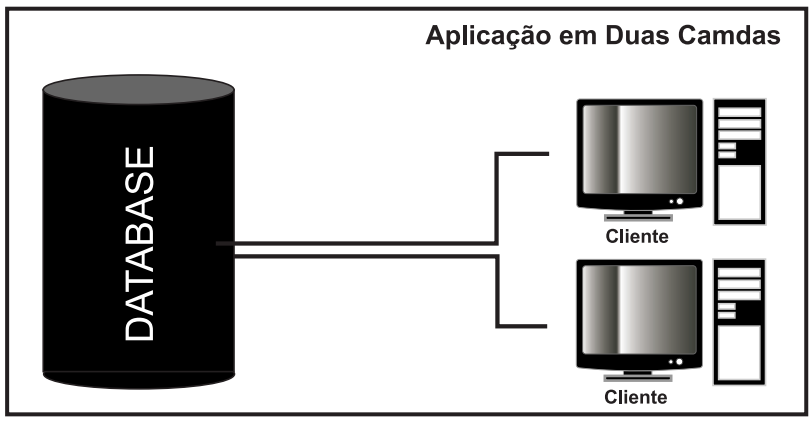
Uma Camada : Centralizado em um único servidor (sistemas monolíticos) os dados e funcionalidades eram acessados através de terminais burros.

*** Positivo:** *segurança e manutenção.*
*** Negativo:** *flexibilidade, escalabilidade e disponibilidade.*



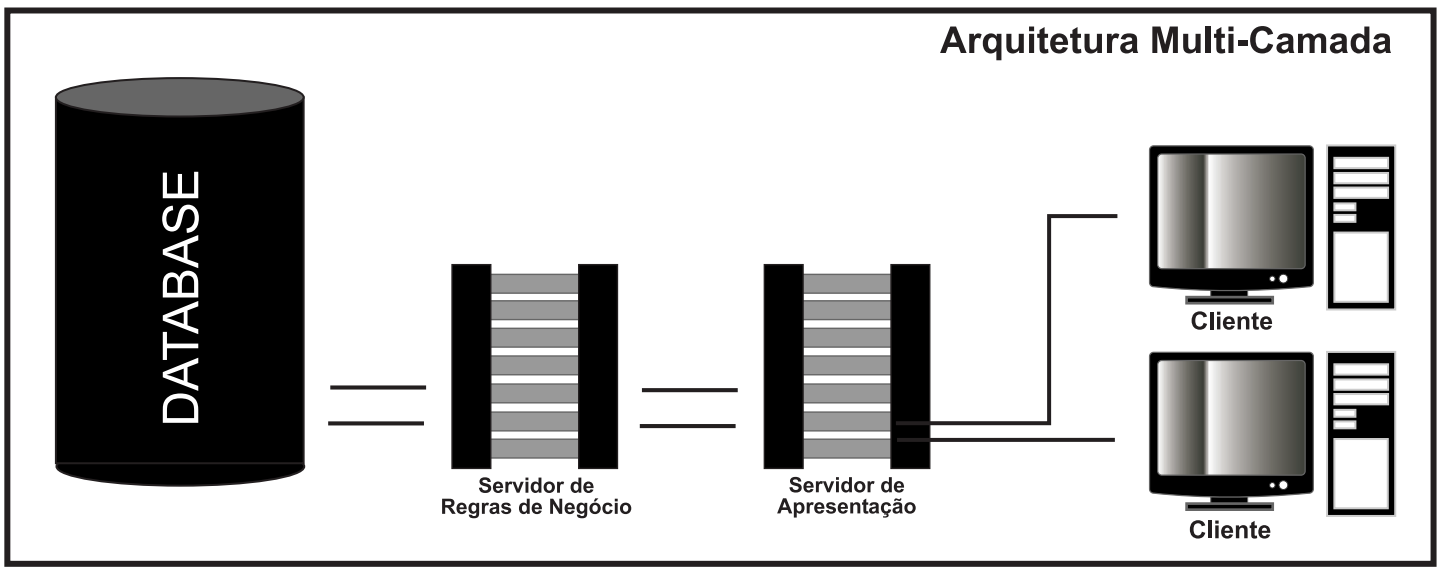
Duas Camadas : Com a explosão dos computadores pessoais de maior capacidade, as regras de apresentação (e na maioria das vezes regras de negócio também) foram migradas para o cliente restando no lado servidor somente o banco de dados e procedimentos de negócios lá embutidos **(stored procedures)**. Época onde aplicações VB + SQL, Oracle DB + Forms, Sybase + Centura dominavam o mercado.

*** Positivo :** *separar lógica de apresentação das regras de armazenamento e negócios.*
*** Negativo :** *escalabilidade e disponibilidade.*



Multi-Camadas : A necessidade de uma melhor organização no desenvolvimento de softwares e também um novo paradigma econômico e de comunicação (Internet) faz com que muitas empresas sejam obrigadas a disponibilizar seus sistemas para serem acessados em diversos dispositivos por um número exponencialmente maior de usuários. Sejam eles aplicações convencionais (fat-client), unidades de resposta audíveis (URA's), Internet Browser e outros.

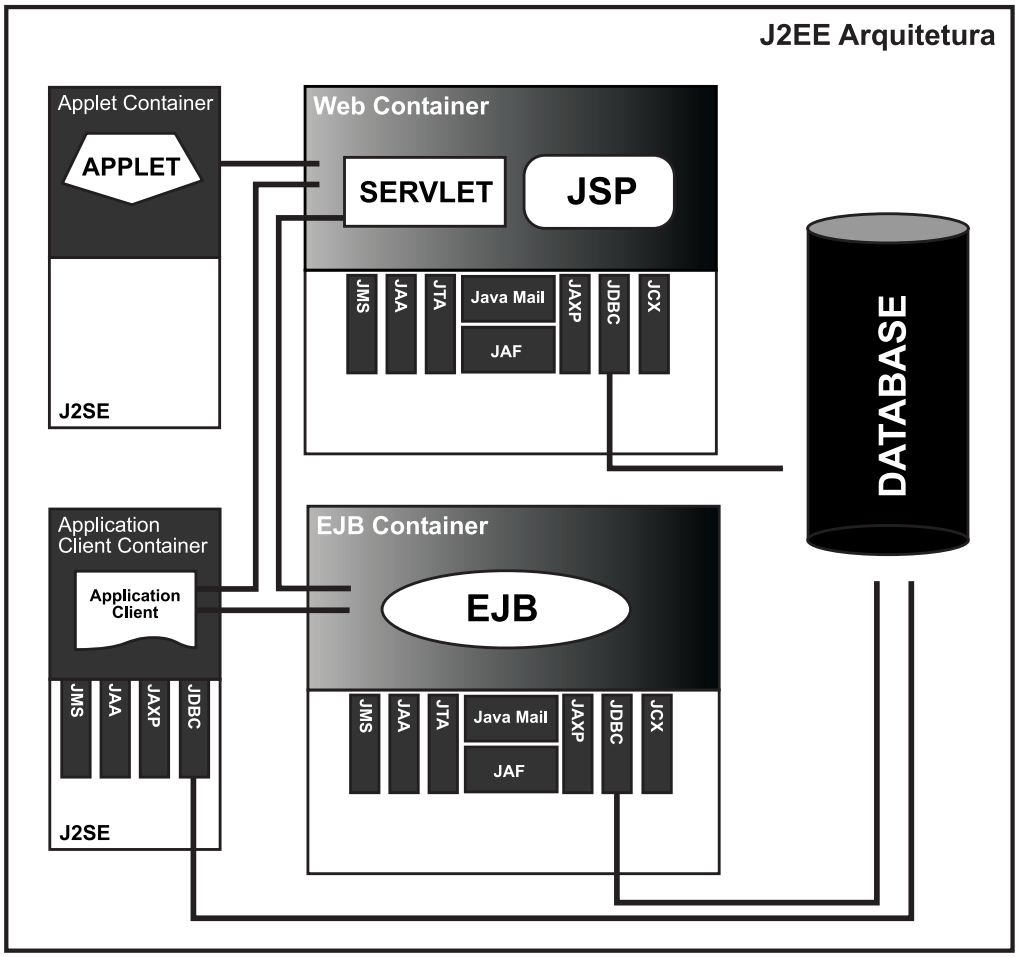
*** Positivo :** *Separação ideal de camadas, flexibilidade, escalabilidade e disponibilidade.*
*** Negativo :** *Complexidade no planejamento, arquitetura e desenvolvimento, segurança e manutenção.*



É fato que aplicações em múltiplas camadas, com serviços complexos, transações extensas que suportam centenas de acessos simultâneos por diversos tipos de cliente são extremamente difíceis de serem desenvolvidas a partir do zero. Servidores de Aplicações surgiram então com o objetivo de:

** Facilitar o desenvolvimento de sistemas com transações e regras de negócio complexas acessados por muitos usuários simultaneamente.*
** Separar definitivamente as regras de negócio das regras de apresentação e das regras de armazenamento.*
** Permitir que diversos tipos de dispositivos acessassem uma determinada funcionalidade do sistema sem reescrita de código e sim através de componentização.*
** Aumentar a segurança.*
** Facilitar manutenção.*

Arquitetura multi-camada J2EE



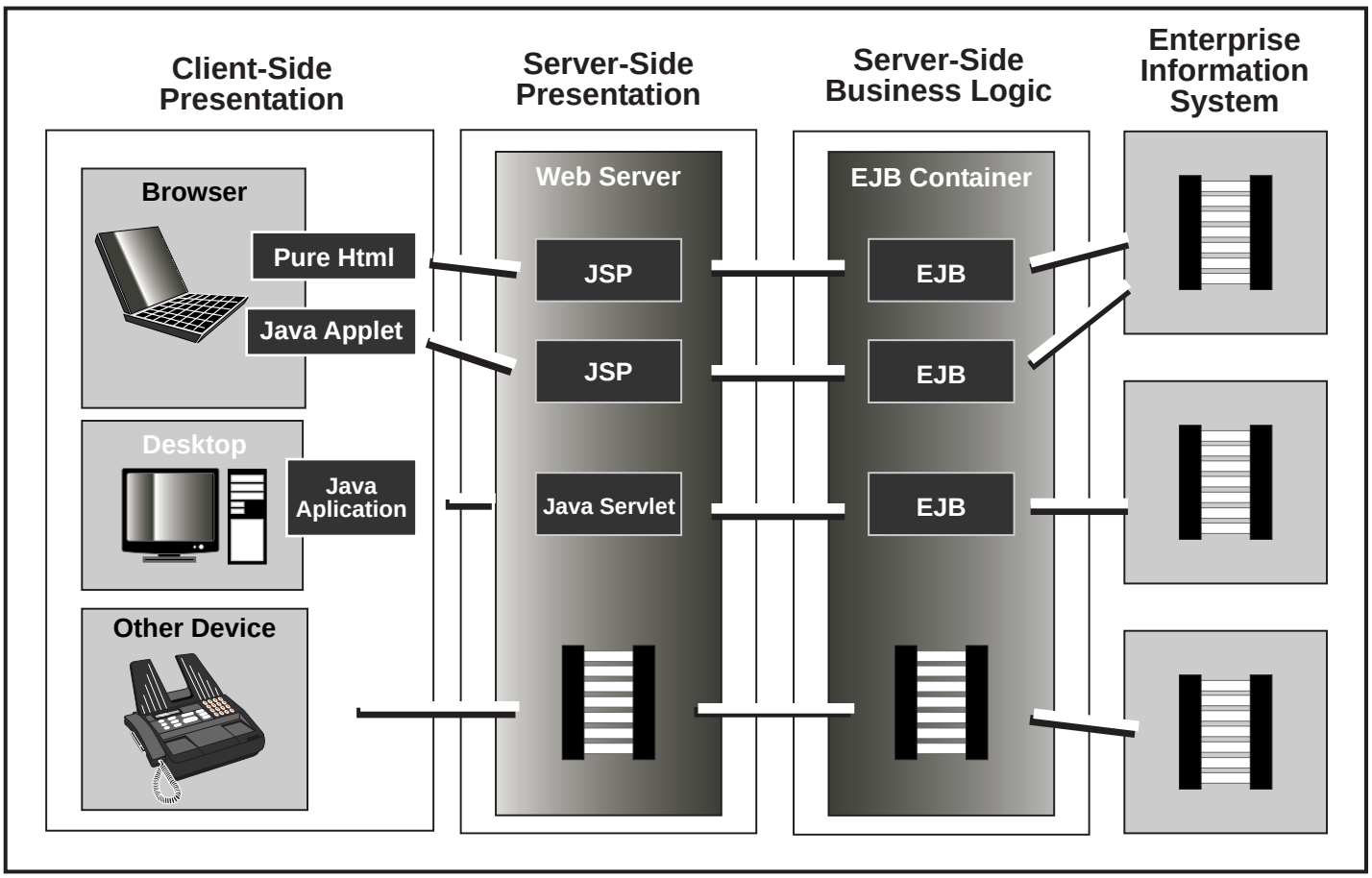
J2EE é um padrão de desenvolvimento em camadas que disponibiliza uma série de serviços de infraestrutura de alto-nível evitando o desenvolvimento de código complexo, aproximando os desenvolvedores do negócio em si.
J2EE é tipicamente consistido nas seguintes camadas:
Camada Client: Browser / HTML, Java Applets, Java Applications, Flash e outros.
Camada Web: É responsável por comunicar-se com a camada de negócio, possui lógica de apresentação e recepção de dados, tipicamente gera HTML / XML. São implementados no J2EE com Servlets e JSP.
Camada de Negócio: É a inteligência da aplicação. Na maioria das ocasiões é implementado com a API / Framework Enterprise JavaBean. O Application Server ou o container EJB vai prover serviços de persistência, transação, alocação de recursos e gerenciamento de ciclo de vida de componentes.

*** Camada de dados ou Enterprise Integration System (EIS):** *É a camada responsável pelos dados e objetos persistentes. Classes de acesso a banco de dados, conectores para ERP e funcionalidades personalizadas de I/O compõem esta camada.*

A plataforma **J2EE** simplifica o desenvolvimento de soluções enterprise através de padrões, serviços de baixo nível (infra-estrutura) e componentes modulares.
Como mostra a figura abaixo, ao término de um projeto com **J2EE** temos como resultado um conjunto de componentes para apresentação de dados e componentes de regras de negócio reusáveis.

Porque J2EE é uma boa escolha?

*** Integrável:** em vários casos, metade do tempo de um projeto Enterprise é gasto com integração de sistemas proprietários e externos. J2EE pode se integrar com outros sistemas através de diversos padrões, protocolos e técnicas: http / ssl, rmi e rmi-iiop, xml, jms, JavaMail e outros.
*** Extensível:** Tem uma linguagem Orientada por Objetos tornando reuso e comportamento "plugavel" uma realidade.
*** Modular:** Estabelece um padrão forte de componentização facilitando a modularização e também facilitando equipes e desenvolvedores a compartilharem suas API's.
*** Segura:** a plataforma Java possui suporte para as principais técnicas e protocolos de autenticação e autorização utilizadas no cotidiano. Além de possuir mecanismos de segurança integrado à máquina virtual.
*** Disponível e escalável:** com uma porção de API's maduras e uma linguagem robusta como Java, softwares escaláveis e mais disponíveis são realidade com a plataforma Java 2 Enterprise Edition.
* Protege o investimento da sua empresa, pois sua aplicação estará em conformidade com um padrão mantido por um grupo de empresas.
* A infra-estrutura inicial custa a partir de R$ 0,00.



E a plataforma Microsoft .net?

* Nova plataforma de desenvolvimento da Microsoft que copia da maneira Micro$oft a plataforma Java. Ótima plataforma para quem não tem escolha. Tem tudo o que o Java tem, porém somente 10% do que estão desenvolvendo é público ou consorciado por empresas, o restante proprietário. Custa a partir de R$ 700,00 por estação.
* Várias linguagens de programação para um mesmo código intermediário.
* Independência de plataforma ainda é mito!

E a plataforma Microsoft Windows DNA?

* Plataforma atual, bem funcional, com alguns problemas no seu padrão de componentização. Produtiva e ideal para quem já possui servidores e estações com sistema operacional Windows e quer desenvolver soluções de pequeno e médio porte não críticas.

*** Microsoft Windows DNA:** *Component Object Model e Distributed Component Object Model foram especificados como padrão de componentização de software, iniciado por um consórcio de empresas, melhor parte da plataforma.*

*** Microsoft Windows DNA:** *Microsoft Transaction Server é o servidor de componentes de negócio.*

*** Microsoft Windows DNA:** *Internet Information Server é o servidor HTTP CGI com ASP utilizado para gerar páginas dinâmicas baseadas em componentes locais/remotos de regras de negócio ou com dados obtidos diretamente da camada de dados.*

O que mais compõe o J2EE além de um conjunto de API's que facilitam o desenvolvimento multi-camadas?

*** "Blueprints Design Guidelines for J2EE"** *desenvolvido por técnicos altamente capacitados, reúne em documentos e exemplos de código as melhores práticas de desenvolvimento de aplicação J2EE.*

*** J2EE Compatibility Test Suite:** *Processo formal de teste de compatibilidade de Application Server J2EE garantindo o funcionamento da sua aplicação em servidores de diferentes fabricantes.*

*** J2EE Reference Implementation:** *A plataforma inclui um servidor chamado de R.I. (reference implementation) implementado com 100% das funcionalidades especificadas. É a chave para validar suas aplicações J2EE.*
100% free e com código fonte disponível.

*** J2EE:** *Todo poder e sucesso do J2SE + suporte para Enterprise Java Beans, Java Servlets API, Java Server Pages, XML e Messaging.*

O que você precisa para ter J2EE dentro da sua empresa:

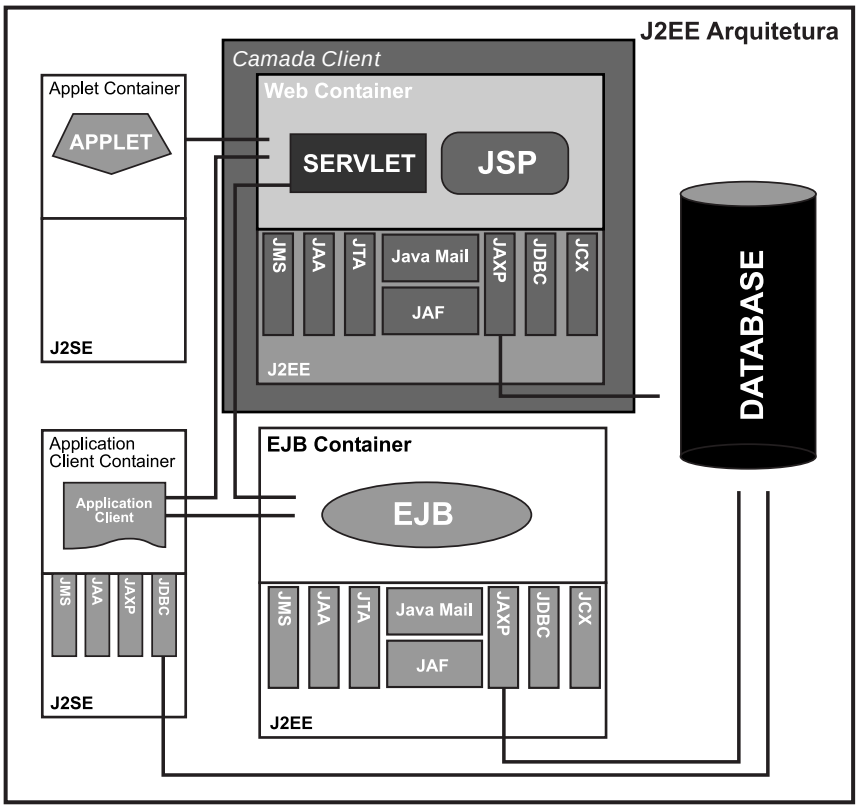
*** Necessidade:** *disponibilizar soluções / aplicações na Web, adotar um padrão de desenvolvimento de longo prazo ou migrar de plataforma como Centura & Sybase, Oracle DB & Forms, Microsoft DNA etc.*

*** Conhecimento:** *o item mais importante na adoção de qualquer tecnologia. Recomendamos que todos os conhecimentos fundamentais da linguagem Java, principalmente OOP (Programação Orientada a Objetos)sejam transmitidos para a equipe por profissionais.*

*** Softwares:** *JDK 1.3.1, Tomcat 4, JBOSS 2.3.3 e Ant.*

*** Livros:** *Core Java Vol. I e II, Core Servlets, More Core Servlets, Core Design Patterns (Sun Press), Java Server Programming (Wrox), Enterprise Java Beans (O'reilly).*

J2EE e a camada Client



Devemos concentrar nesta camada somente regras de apresentação de dados, validações prévias, tratamento de eventos ocorridos em outras camadas e outros tipos de diálogo com o usuário. Regras de negócio, código SQL, políticas de segurança e outras funcionalidades típicas de aplicações devem ser programadas fora desta camada. *Clients J2EE podem ser implementados com thin-clients (browser / http / html) ou com fat-clients (aplicações instaladas no cliente)* utilizando-se as seguintes tecnologias:

O que eu preciso de ferramentas / softwares para desenvolver camadas client para Java?

Os softwares vão variar conforme o tipo de client: Browser HTML ou Java Applet / Application. *Quando client é Browser /* HTML é recomendado utilizar um editor HTML como Macromedia *DreamWeaver, Microsoft Frontpage* ou então algum gratuito. Este editor será utilizado para desenvolver páginas HTML com código Java, Java Server Pages (JSP). **Também é recomendado instalar os dois principais navegadores do mercado:** Internet Explorer e Netscape para testar a compatibilidade das páginas desenvolvidas.

Quando for desenvolver **client Applet** ou **Java Applications** é recomendado utilizar algum **ambiente de desenvolvimento gráfico para Java** como Sun Forte, Borland Jbuilder, Oracle Jdeveloper, Toguether, etc.
Quase todos ambientes possuem uma versão gratuita, porém é necessário verificar a política adotada, pois nem todos permitem desenvolver softwares para produção.

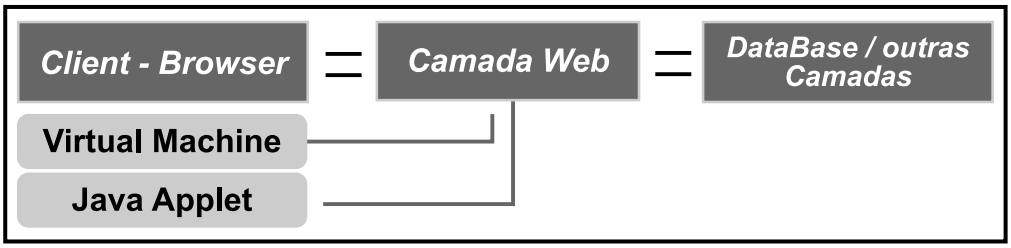
1. Browser, HTML e JavaScript

Esta composição de tecnologias para Graphical User Interfaces estão cada vez mais fazendo parte das aplicações corporativas. Com J2EE geramos o conteúdo HTML dinâmico via http facilmente com o uso de JavaServlets e JSP, temas abordados adiante.
As regras para formatação e validação de dados podem estar presente com o uso de JavaScript, ficando restrito aos recursos do browser acessíveis via JavaScript.

2. Aplicações stand-alone

Aplicações stand-alone ou “fat-client” são instaladas e executadas no cliente. Para interfaces gráficas podemos utilizar os Kits de API’s Swing e AWT.
Aplicações stand-alone podem ter acesso irrestrito a máquina do usuário e estabelecer conexões de rede de longo prazo com maior facilidade do que navegadores com http. O aspecto negativo neste tipo de cliente é o consumo de processamento e memória (300mhz com 64 mb mínimo). As dificuldades de manutenção, instalação e distribuição de releases atualmente são superadas com o uso de Java Web Start.

3. Java Applets
São como "fat-client" porém são contidas por um browser e distribuídas dinamicamente por http. A virtual machine que executa Java Applets pode estar embutida no browser ou então instalada como plug-in. Por estas aplicações estarem sendo distribuídas pela Internet / redes públicas http, existem mecanismos de segurança que impedem que Java Applets façam operações críticas nas máquinas dos usuários.



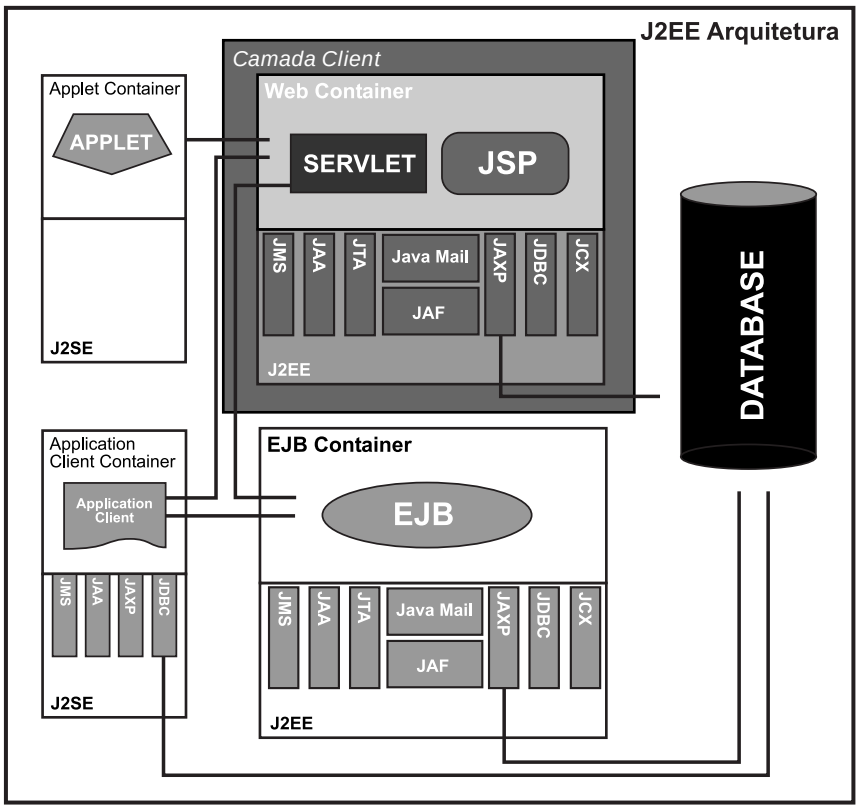
4. Outras plataformas
Outras plataformas de desenvolvimento podem sempre se comunicar com J2EE via http com técnicas de tunneling, é o que pode ocorrer quando utilizamos Flash ou Visual Basic como client.

J2EE e a camada Web

Uma das maiores procura à tecnologia Java é sem dúvida para criação de aplicações Web. Devido a grande facilidade de disponibilizar e atualizar, desenvolver sistemas / soluções para serem acessados via Web tornou-se praticamente uma obrigação para as empresas.
Uma das peças fundamentais para isso é protocolo http. Rápido e escalável proporciona confiança e performance ideal para atender diversas solicitações simultâneas além de ser simples.

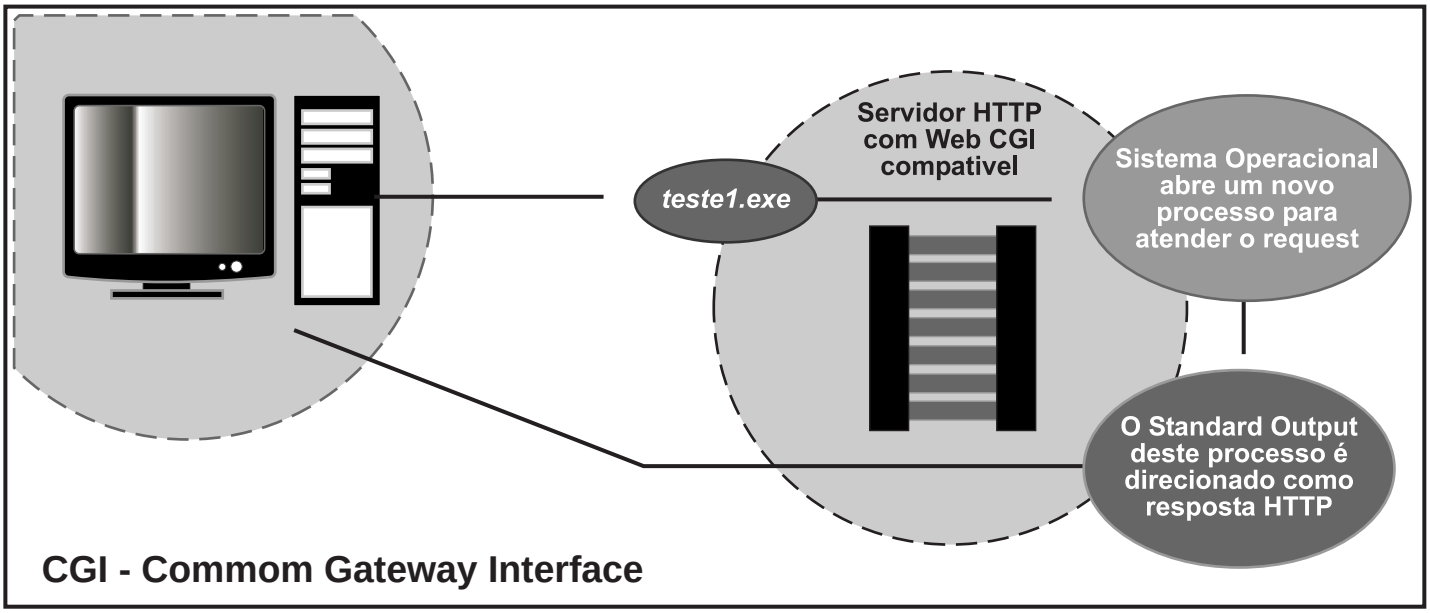
Algumas das principais características do protocolo HTTP:

** Simples e robusto.*
** As operações são baseadas em request / response.*
** É um protocolo sessionless, não possui nenhum conceito implícito de seção de usuário. Todas requisições são tratadas de maneira independente e até mesmo anônima.*
** Grande liberdade com firewalls.*
** As operações mais utilizadas das especificadas no protocolo são GET e POST.*



O protocolo **http** foi projetado no início para fornecer **conteúdo estático.** Para tornar possível o desenvolvimento de aplicações, que na prática sempre **precisam de dados dinâmicos** (databases, arquivos, diretórios etc.) foram necessárias algumas **adaptações.**

A primeira geração de aplicações para **Web** desenvolvidas com http utiliza **scripts** com **Commom Gateway Interface (CGI).** CGI é uma maneira de um **Web Client executar programas no Web Server.** Vejamos como funciona uma requisição de um navegador no cliente a um CGI no servidor Web.



Aplicações Web J2EE utilizam o conceito de container. O Web Container J2EE além de gerar conteúdo dinâmico para http é capaz de:

** Gerenciar os recursos acessados utilizados pelos componentes.*
** Multi-tarefa com threads e não processos para atender requisições simultâneas.*
** Gerencia seções de usuários.*
** Fornece estrutura de segurança para aplicação.*
** E diversas outras tarefas.*

As duas principais API's do Web Container J2EE são:
Java Servlets e Java Server Pages. A implementação de referência oficial destas API's é o **Tomcat da Apache,** recentemente renomeado para Catalina.
Existem muitos fabricantes **Web Container J2EE**. É necessário ficar atento em qual a **versão da API** que o fabricante implementou. O Tomcat possui disponível para download as seguintes versões:

Servlet	JSP	Tomcat
2.2	1.1	3.xxxx
2.3	1.2	4.xxxx

Instalando o servidor Tomcat

O processo de download e instalação do Tomcat é muito simples:

1. URL para download:
http://jakarta.apache.org/builds/jakarta-tomcat-4.0/archives/v4.0.3/bin/
http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.4/bin/

2. Caso utilize Windows execute a instalação ou caso tenha feito o download de um arquivo compacto, descompacte no diretório desejado.

3. Configure a variável de ambiente TOMCAT_HOME no seu sistema operacional atribuindo o diretório onde instalou o Tomcat a ela.

4. Execute o **startup.bat** presente no diretório **TOMCAT_HOME/bin.**

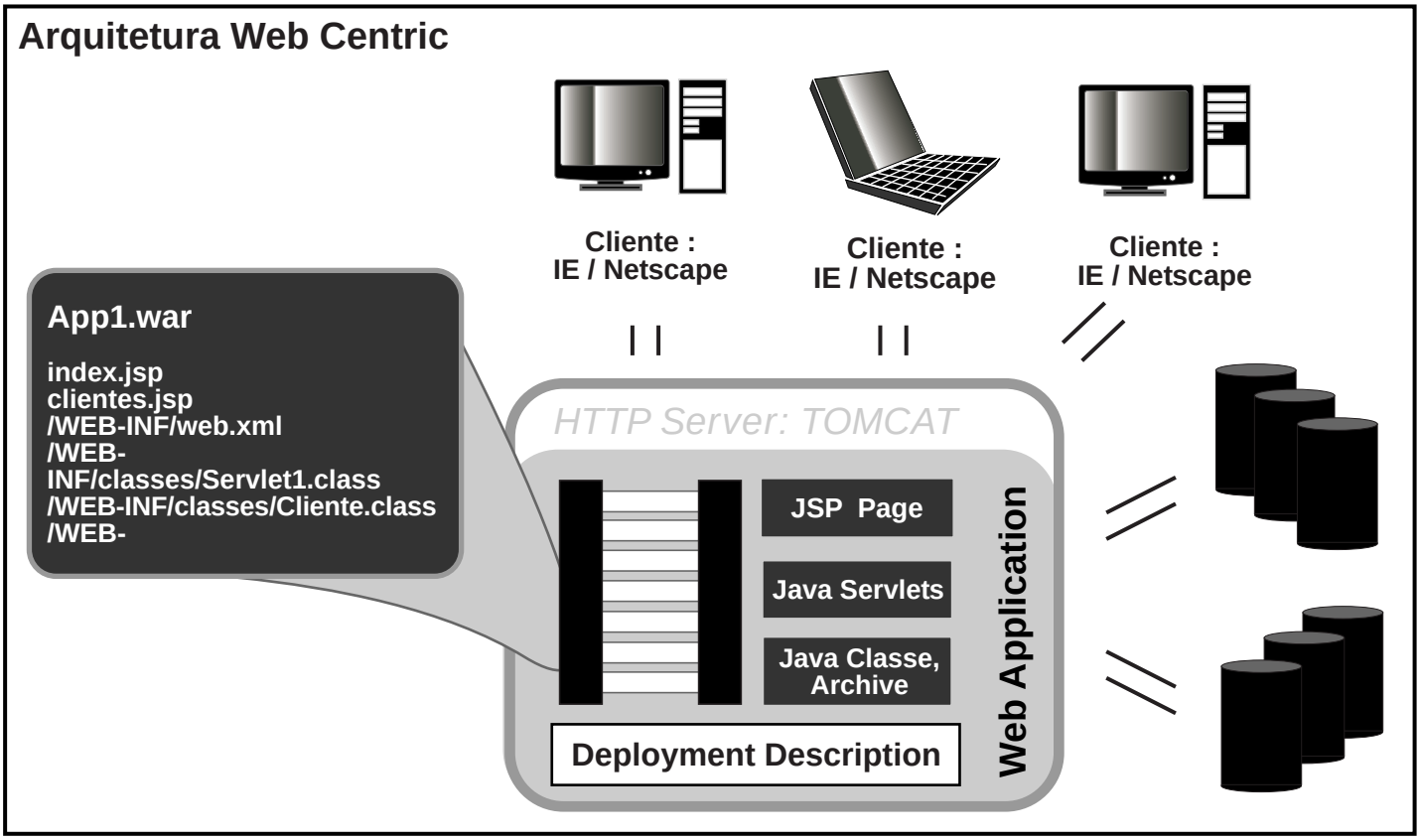
5. Por **default** o Tomcat vem configurado para trabalhar com a **porta 8080**, abra o navegador e coloque o seguinte endereço**: http://localhost:8080**

6. Se a página home do Tomcat aparecer a instalação do Tomcat foi feita com sucesso. *Agora é só criar e desenvolver suas aplicações no Tomcat. As aplicações são armazenadas no diretório TOMCAT_HOME/webapps. Cada sub-diretório nesse local representa um projeto com seu próprio contexto e URL de acesso, exemplo:*

c:\tomcat\webapps\globalcode = http://localhost:8080/globalcode
c:\tomcat\webapps\root= http://localhost:8080/
c:\tomcat\webapps\app1= http://localhost:8080/app1

Para criar uma nova aplicação, sugerimos que duplique o diretório root para que não precise criar os diretórios e arquivo xml manualmente. Após a criação de uma nova aplicação é necessário reinicializar o processo do Tomcat.
Uma aplicação Java para Web Container é uma coleção de documentos HTML/XML, componentes Java para Web (servlets e JSP) e outros recursos / arquivos. Esta coleção de arquivos é organizada em uma estrutura de diretório pré-definida conhecida como Web ARchive (WAR). Esta estrutura é esperada pelo container e devemos segui-lá rigorosamente, qualquer inconformidade resultará em um erro em tempo de instalação e às vezes de execução.

Apresentamos abaixo detalhes de uma aplicação com arquitetura Web-Centric, utilizando a camada Web do J2EE e thin-client:



index.jsp componente Java Server Page, são páginas HTML com porções de código Java.

clientes.jsp outro suposto componente JSP

web.xml esse arquivo é chamado deployment descriptor, nele podemos fazer diversas configurações para nossa aplicação no container como definir variáveis de projeto, indicar quais recursos nosso sistema utiliza do container, entre outros.

Servlet1.class component Servlet. É uma classe Java que herda HttpServlet e faz sobreposição de algum(s) método(s).

Cliente.class classe normal do Java que pode ser utilizada nos JSP's e em todas as outras classes da aplicação.

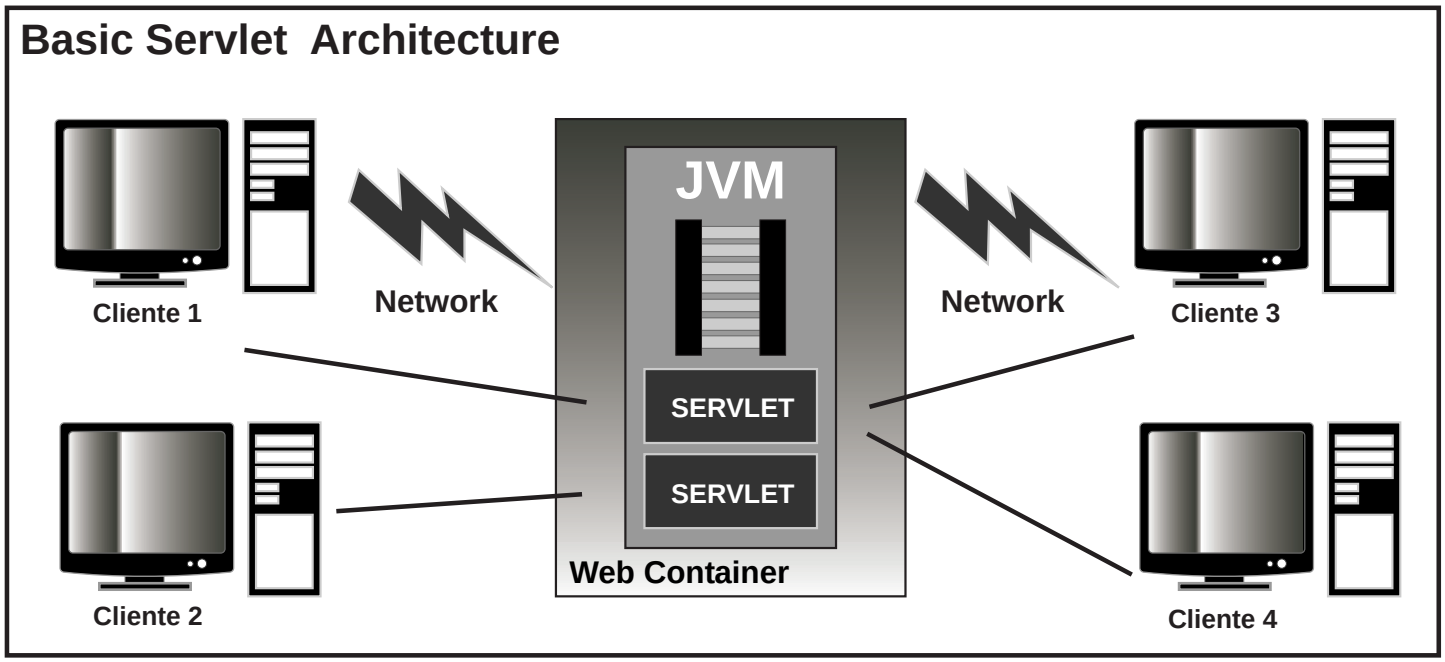
Database.class classe convencional Java com métodos de acesso ao banco de dados.

Agora que sabemos os fundamentos de http, como instalar um Web Container e criar um novo projeto /aplicação, vamos detalhar os tópicos Java Servlet e Java Server Pages.

Java Servlets

Servlets são módulos executados dentro de um serviço orientado por request / response. A API Servlet permite que servidores *(request / response)* tenham suas funcionalidades estendidas com programas Java. Apesar da especificação permitir que outros protocolos e servidores implementem Servlets, somente servidores http utilizam com popularidade.

Vejamos o desenho abaixo do funcionamento geral de um servlet:



Código do servlet:
Para criarmos um servlet devemos criar uma **sub-classe de HttpServlet**. Para cada operação do

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class OlaMundo extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

protocolo http temos um método na classe HttpServlet que podemos sobrepor. No caso do exemplo acima estamos **sobrepondo o método void doGet** (*HttpServletRequest, HttpServletResponse)* para atender as solicitações feitas através da operação HTTP GET que é a operação **"default" de um navegador.**

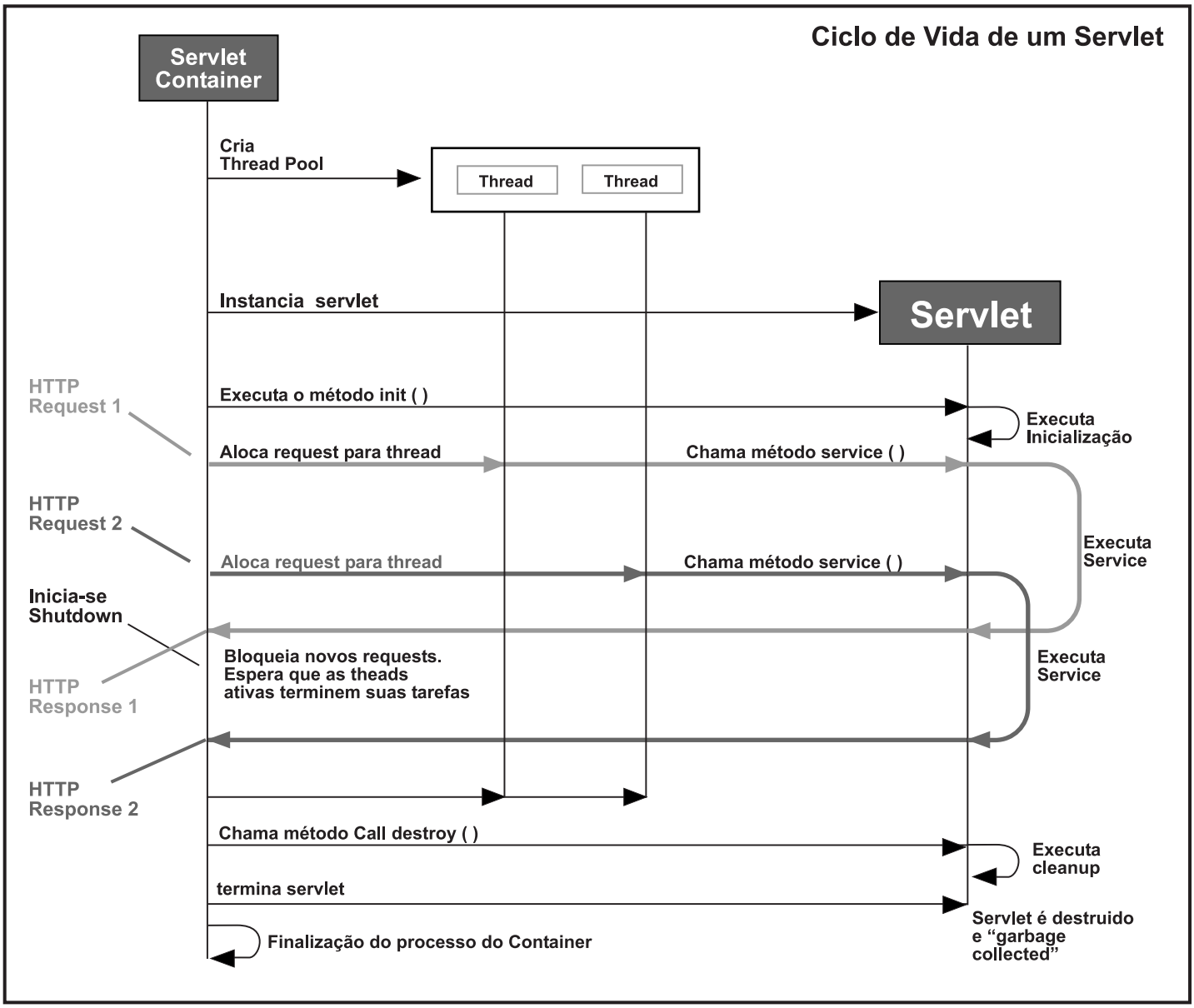
Os seguintes métodos representam as operações do protocolo HTTP e podem ser sobrepostos:

** doDelete, doPost, doGet, doTrace, doPut e doOptions todos com a mesma assinatura (HttpServletRequest, HttpServletResponse)*
** Os dois únicos métodos realmente utilizados são GET e POST e veremos a diferença entre eles adiante.*

Outros dois métodos comuns em servlets que apóiam o gerenciamento do ciclo de vida do objeto são:

** init() / init(ServletConfig): executado pelo Container depois de criar o objeto servlet. "Construtor" do servlet.*
** destroy(): executado pelo Web Container antes de retirar o objeto servlet da memória. "Destrutor" do servlet.*

Compilando e instalando servlets no Tomcat



Vimos o código-fonte de um servlet simples e um overview dos principais métodos que podemos sobrepor. Vamos agora compilar e testar nosso servlet:

1. Vamos criar o arquivo **OlaMundo.Java** no próprio diretório de **"produção"** do Tomcat: **TOMCAT_HOME\webapps\minhaapp\WEB-INF\classes\OlaMundo.java**

2. Copiamos o código apresentado na página anterior para o arquivo criado.

3. É necessário indicar no **classpath** onde se encontra o **jar da API servlet**, sugerimos que utilize o **j2ee.jar** que é distribuído no **JDK Enterprise Edition*. Compile o arquivo.**

4. Agora vamos testa-lo no navegador. **Para acessar um servlet**, devemos colocar a url da nossa aplicação **(http://servidor:8080/minhaapp)** mais /servlet/NomeClasseServlet **sem** colocar **.class,** para nosso caso a URL é:

http://localhost:8080/minhaapp/servlet/servlets.OlaMundo

Se o servlet estiver dentro de um pacote é necessário especificar o **nome completo da classe**, ou seja, com o nome do pacote como prefixo.

Exemplo: http://localhost:8080/minhaapp/servlet/pacote1.OlaMundo

*Se preferir é possível baixar somente as classes da API servlet no site:
http://java.sun.com/products/servlet/download.html

More Servlets

Um objeto da classe servlet é instanciado na primeira requisição feita ao servidor por um cliente. Este objeto servlet será utilizado para atender todos os próximos requests sendo que o container iniciará uma thread para cliente.
Vamos conferir:

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Contador extends HttpServlet {
    int conta=0;
    public void doGet(HttpServletRequest request, HttpServletResponse   response) throws
IOException, ServletException
    {
        conta++;
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Contador</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>A pagina foi acessada: " + conta + " </p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Reparem que a variável conta **é acumulada, mesmo se tivéssemos acessando este servlet de outro computador.**

doGet / doPost / request / response

As informações trafegadas do cliente para o servidor via http **GET** são transmitidas na URL:

http://localhost:8080/minhaapp/MeuServlet?cor_letra=1&cor_fundo=2

Utilizamos este método quando **são poucas informações e não necessitamos de segurança nos dados.**

Já o método http **POST** transfere os dados via stream ou seja, no pacote http de request ao servidor. Este método é muito utilizado para envio de dados de html forms, envio de arquivos do cliente para o servidor e para aumentar a segurança como em rotinas de login.

Na classe servlet, ambos os métodos doGet e doPost recebem como argumento:

HttpServletRequest request
O objeto request representa todas as informações da requisição e os dados enviados do cliente para o servidor. Os principais métodos disponibilizados pela classe HttpServletRequest são:

* *String getParameter(String name) lê parâmetros enviados por URL ou por POST de formulários html.*
* *Cookie[ ] getCookies( ) retorna uma matriz de objeto que representa os cookies enviados pelo client.*
* *HttpSession getSession() retorna a seção do usuário, se não existe uma seção para o usuário, uma seção é criada automaticamente.*
* *Java.security.Principal getUserPrincipal() retorna um objeto que contém o nome do usuário autenticado*
* *ServletInputStream getInputStream() retorna um stream que é tipicamente utilizado para fazer upload do cliente para o servidor http via form html com o tag <input type="FILE">*

HttpServletResponse response
Este objeto representa a via de resposta para o cliente. Com ele obtemos o stream para gerar a resposta. Podemos gerar qualquer tipo de conteúdo: binário ou texto, geralmente utilizamos servlet para gerar XML's, gráficos e às vezes HTML.
Os principais métodos disponibilizados pela classe HttpServletResponse são:

* *PrintWriter getWriter() obtém um stream texto para gerar a resposta para o cliente, utilizado nos exemplos acima.*
* *void setContentType(String type) configura o formato do conteúdo que esta sendo gerado de resposta. Exemplo text/html, text/xml, image/gif, etc.*
* *ServletOutputStream getOutputStream()obtém um stream binário para gerar a resposta para o cliente. Utilizado para gerar gráficos dinâmicos, documentos Excel e outros.*
* *void addCookie(Cockie c) permite o envio de um cookie para o cliente. Cookies são pequenas porções de dados persistidos no navegador do cliente por um tempo que pode ser configurado.*
* *void sendRedirect(String location) redireciona a para outra URL.*
* *void setStatus(int sc) configura um código de retorno para o cliente. Os código estão definidos em constantes na interface HttpServletResponse.*

Vamos agora analisar diversos exemplos de servlets com os métodos apresentados acima.

Servlet para ler os dados enviados via HTML Form

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LerDados extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException{
      String nome=null;
      nome = request.getParameter("textNome");
      /* poderíamos então gravar o nome em um banco de dados e redirecionar este a
      resposta para um html de sucesso!*/
}
```

HTML que faz a chamada ao serlvet:

```
<html>
<head>
<title>GlobalEducation - J2EE Web</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF" text="#000000">
<form name="form1" method="GET" action="servlet/servlets.LerDados">
  <p>Entre com seu nome:
    <input type="text" name="textNome">
  </p>
  <p>
    <input type="submit" name="Submit" value="Enviar">
    <input type="reset" name="Reset" value="Reset">
  </p>
</form>
</body>
</html>
```

Reparem que podemos especificar a forma de transmissão dos dados no tag html form, se mudarmos o html para **POST** obrigatoriamente teríamos que ter o método **doPost** em nosso servlet.

Servlet para uploading de arquivos via HTML Form

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FileUpload extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException {
        BufferedReader bf=request.getReader();
    PrintWriter out=response.getWriter();
        response.setContentType("text/html");
        out.println("<html>");
        out.println("<body>");
        String line=null;
        while((line=bf.readLine())!=null) {
       out.println("<p>" + line + "</p>");
        }
        /* Ou mais legível:
        line = bf.readLine();
        while(line !=null) {
            out.println("<p>" + bf.readLine() + "</p>");
            line = bf.readLine();
        }*/
        out.println("</body>");
        out.println("</html>");
        out.close();
}
```

HTML que faz a chamada ao serlvet

```
<html>
<head>
<title>GlobalEducation - J2EE Web</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF" text="#000000">
<form name="form1" method="POST" enctype="multipart/form-data"
action="servlet/servlets.FileUpload">
  <p>Selecione o arquivo:
    <input type="file" name="arquivoParaUpload">
  </p>
  <p>
    <input type="submit" name="Submit" value="Enviar">
    <input type="reset" name="Reset" value="Reset">
  </p>
</form>
</body>
```

Reparem que podemos especificar a forma de transmissão dos dados no tag html form, se mudarmos o html para **GET** obrigatoriamente teríamos que ter o método **doGet** em nosso servlet.

Servlet & Multithreading

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MultiTarefa extends HttpServlet {
  public void operacaoDemorada() {
    System.out.println("Inicio da suposta operação com multi-threading " +
      Thread.currentThread().getName());
    for(long y=0;y<100000000;y++) {}
    }
  }
  public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException {
    System.out.println("Inicio ao atendimento a solicitação do browser: " +
      Thread.currentThread().getName());
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>MultiTarefa</title></head>");
    out.println("<body>");
    out.println("<h1>O objeto atende mais de uma solicitação pois abre
      threads" + x++ + "</h1>");
    out.println("<p>Laço grande...Thread:" +
      Thread.currentThread().getName() + ")...</p>");
    this.operacaoDemorada();
    out.println("<p>Terminado.</p>");
    out.println("</body></html>");
    out.close();
    System.out.println("Termino do atendimento a solicitação do browser: "
      + Thread.currentThread().getName());
    }
  }
}
```

Servlet & Multithreading – Thread Safe

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MultiTarefa extends HttpServlet implements SingleThreadModel {
  public void operacaoDemorada() {
  System.out.println("Inicio da suposta operação com Threadsafe " +
      Thread.currentThread().getName());
    for(long y=0;y<100000000;y++) {}
    }
  }
  public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException {
    System.out.println("Inicio ao atendimento a solicitação do browser: " +
      Thread.currentThread().getName());
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>MultiTarefa - Threadsafe</title></head>");
    out.println("<body>");
    out.println("<h1>O objeto atende mais de uma solicitação pois abre
      threads" + x++ + "</h1>");
    out.println("<p>Laço grande...Thread:" +
      Thread.currentThread().getName() + ")...</p>");
    this.operacaoDemorada();
    out.println("<p>Terminado.</p>");
    out.println("</body></html>");
    out.close();
    System.out.println("Termino do atendimento a solicitação do browser: "
      + Thread.currentThread().getName());
    }
  }
}
```

Servlet & Seção HTTP

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Secao extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException {
    String nome;
    nome=request.getParameter("textNome");
    response.setContentType("text/html");
    PrintWriter out= response.getWriter();
    Login l;
    HttpSession s=request.getSession(false);
    if(s==null) {
      out.println("<p>Criando seção</p>");
      l=new Login();
      l.nome=nome;
      s=request.getSession(true);
      s.putValue("dados", l);
    }
    else
    {
      l=(Login) s.getValue("dados");
    }
    out.println("<p>Welcome to Internet Application, " + l.nome + "</p>");
    out.println("<p>Seu contador: " + l.contador++ + "</p>");
    s.putValue("dados", l);
  }
}
```

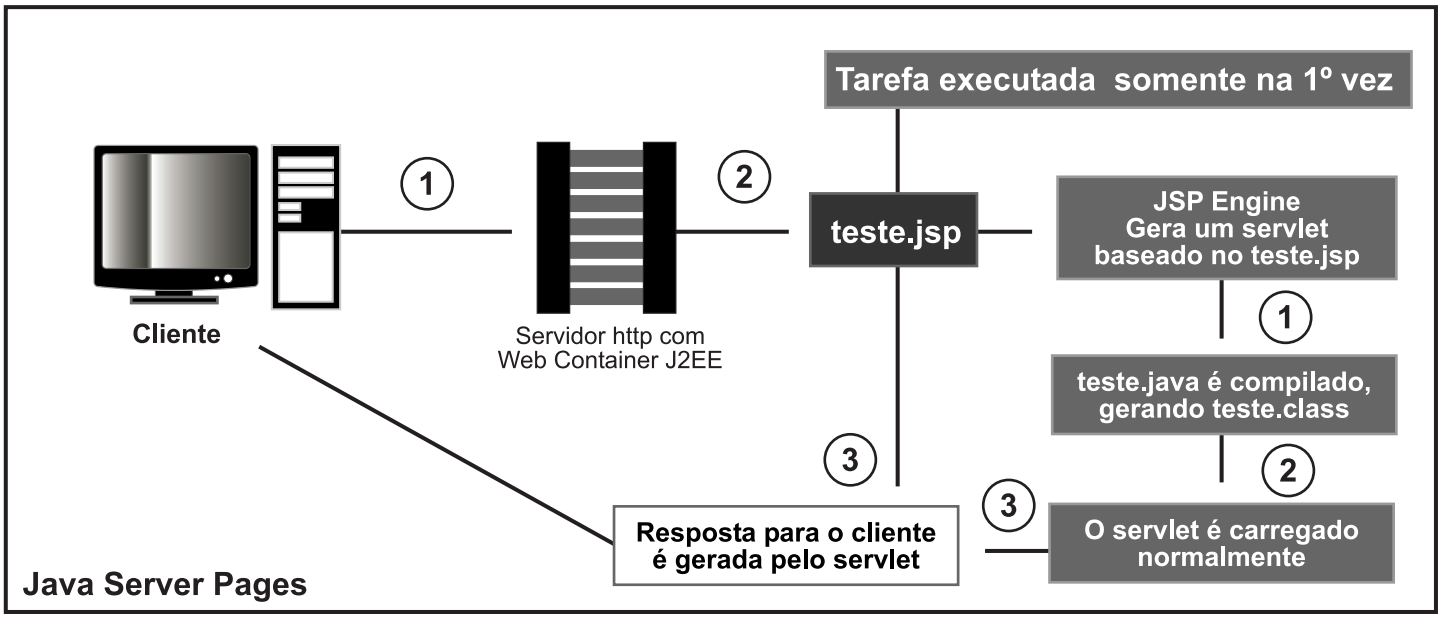
Agora podemos ter contadores por usuário utilizando seções. Não existe nenhum truque para gerenciar seções com o protocolo http, na verdade o Web Container gerou um ID e enviou para o cliente / browser. Desta maneira todas as próximas requisições que o browser fizer para a URL ele mesmo se identificará. Isso é feito através de cookies ou havendo restrições, com reescrita de URL.

Java Server Pages

Java Server Pages são páginas HTML com uma porção de código Java. É uma solução similar ao Active Server Pages (ASP) da Microsoft ou então PHP. Veja o exemplo abaixo:

```
<hmtl>
<body>
<% for(int x=0;x<100;x++) {%>
<p> O valor de x é:<%=x %> </p>
<% } %>
</body>
</html>
```

Um JSP possui um processo de execução um pouco diferente do processo de um Servlet, vejamos a ilustração a seguir:



Como vemos o resultado final de um JSP é uma classe Java que é **gerada por um JSP Engine.** Logo após a classe é compilada com o **compilador convencional** do Java através da variável de ambiente JAVA_HOME.
O **engine de JSP é sensível a modificações ou seja, se você mudar o JSP ele vai gerar novamente a classe servlet.**
Podemos dizer então que JSP é uma implementação de Servlets? Sim, é um servlet com alguns **comportamentos adicionais** definidos nas interfaces **javax.servlet.jsp.HttpJspPage e JspPage.**

JSP Directives

As diretivas JSP são instruções de configuração do nosso JSP em geral. Através das diretivas podemos fazer import de classes, herdar outro JSP, configurar o tipo de conteúdo de saída, fazer include de outros arquivos entre outras configurações.
São três as principais diretivas JSP:

** A diretiva PAGE é uma das principais diretivas, pois permiti a configuração de atributos importantes do nosso JSP, sua sintaxe é:*

```
<%@ page <atributo>=”<valor>” <atributo>=”<valor>” ...%>

Exemplo:
<%@ page import=”java.util.Vector, java.rmi.*”
         errorPage=”erros.jsp” %>
```

Os principais atributos que podem ser modificados por essa diretiva são: extends, import, session, buffer, autoFlush, isThreadSafe, info, errorPage, isErrorPage e contentType.

** Uma diretiva muito útil é a DIRETIVA INCLUDE, com ela podemos estruturar nossos arquivos para fazer reuso de pequenas porções de html e jsp. Abaixo sua sintaxe e exemplo:*

```
<%@ include file=”<filename>” %>

Exemplo:
<%@ include file=”botaoVoltar.html” %>
```

O único atributo disponível é o atributo file para a diretiva include.

E para finalizar, temos a diretiva taglib para utilização de bibliotecas de tags para Web que veremos mais adiante. Sua sintaxe:

```
<%@ taglib uri=”tagLibraryURI” prefixo=”tagPrefix” %>
```

Scripts JSP

Scripts no JSP permitem adicionar código Java no HTML: definir variáveis, métodos e scriptlets (porções de código Java para gerar conteúdo).

Scripts JSP – Declarações

```
<%! JAVA CODE %>

Exemplo:

<html>
  <head>
    <title>J2EE Course – JSP</title>
  </head>
  <body>
    <%= exemploMetodo() %>
  </body>
</html>

<%!
  String valor="Exemplo de Variável String";
  public String exemploMetodo() {
    return valor;
  }
%>
```

Declarações em JSP são blocos de código Java que podem conter declarações de variáveis e métodos que serão utilizados no escopo da classe.

Scripts JSP – Scriptlets

É um bloco de código Java executado durante o processo de request-processing e declarado dentro de tags <% %>.
Este pedaço de código fará parte do restante do código que monta a resposta ao cliente. Vejamos o exemplo:

```
<%
//JAVA CODE
%>

Exemplo:

<html>
 <head>
    <title>J2EE Course – JSP</title>
  </head>
 <body>
    <%
      for(int x=0; x<100;x++) {
        out.println("<p>Veja que temos out.println, valor de x=" + x + "</p>");
      }
    %>
  </body>
</html>
```

Scripts JSP – Expressions

É um Scriptlet com uma expressão abreviada que envia um valor para o cliente. A expressão indicada no Tag será processada em tempo de http request e o valor será convertido para String. Se o resultado da expressão é um objeto, então o método toString() será utilizado.
Exemplo:

```
<%!
      String testeExpression="J2EE";
%>
<html>
  <head>
    <title>J2EE Course – JSP</title>
  </head>
  <body>
    <p>O valor de testeExpression é <%= testeExpression %></p>
  </body>
</html>
```

Scripts JSP – Implicit Objects

Baseado na API de servlets JSP's disponibilizam diversos objetos convenientes que são muito utilizados na prática. Abaixo citamos os principais:

* **request** idem ao HttpServletRequest do método doGet ou doPost do servlet e como nos servlet's podemos obter parâmetros e dados enviados pelo cliente, cookies e outros.
* **response** idem ao HttpServletResponse de um servlet. Tipicamente utilizado para gerar código de status http e header.
* **session** representa uma seção http onde podemos ler, incluir e remover dados específicos de um usuário conectado no sistema Web. Caso a seção não exista, será criada automaticamente ao menos que tenha utilizado a diretiva session="false".
* **application** representa o ServletContext, encapsula o acesso a uma collection onde podemos colocar objetos / dados que são utilizados em toda aplicação Web.

Exemplo de uso de objetos implícitos:

```
<%
response.setIntHeader("Refresh", 3);
String valor= request.getParameter("teste");
if(valor == null || valor.equals("")) {
      valor = "Nenhum valor enviado"
}
%>

<html>
  <head>
    <title>J2EE Course – JSP</title>
  </head>
  <body>
   <p>O valor enviado foi <%= valor %></p>
  </body>
</html>
```

JSP Tag Extensions

Tag Extensions ou custom tags é um recurso significativo introduzido na versão 1.1 do JSP. Com ele

é possível criarmos nossos próprios tags para utilizarmos em nossos JSP's. Custom tags é uma maneira prática e eficiente para componentização da camada de apresentação (presentation tier). Podemos encapsular scriptles em componentes como esse e reusarmos em diversos JSP's. Por exemplo, poderíamos criar um custom tag que montasse uma caixa de combinação HTML com dados vindos de um SQL server.

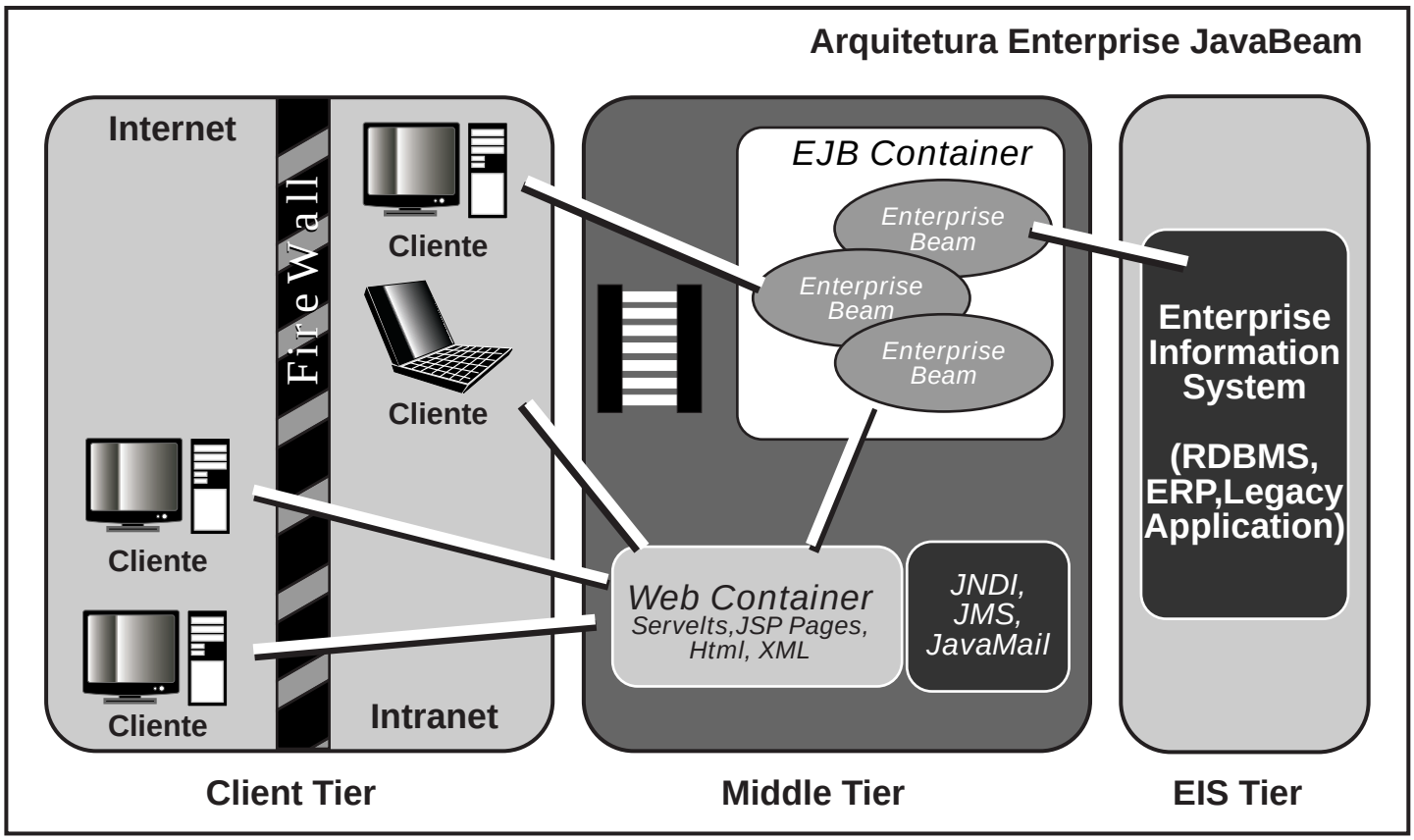
Servlets ou JSP's?

Agora que vimos os principais recursos e funcionalidades de servlets e JSP poderão surgir dúvidas quanto a melhor maneira para o desenvolvimento de aplicações para Web, já que ambos permitem trabalhar com conteúdos dinâmicos.
Bem, uma aplicação Web pode ter sucesso somente com Servets ou somente com JSP's, portanto, não existe maneira correta e sim algumas melhores práticas como as relacionadas abaixo:

1. Nunca gerar HTML no Servlet:.
2. Nunca utilize JSP para gerar dados binários.
3. Procure sempre utilizar um servlet para recepcionar um HTML form.
4. E outras mais.

J2EE e a camada de negócio

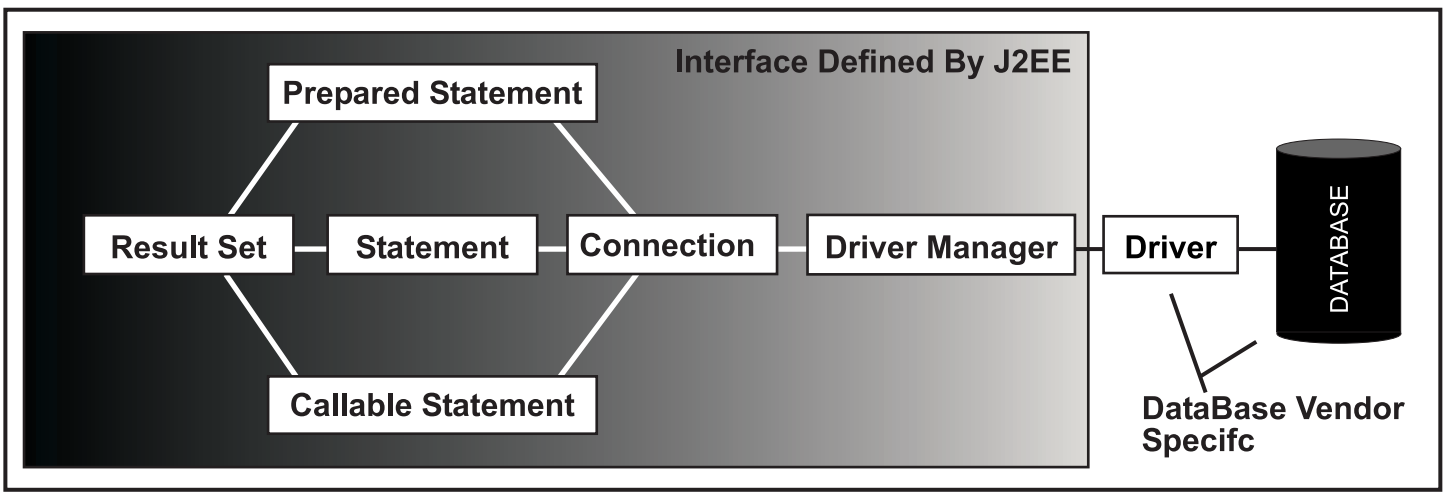
Esta é a camada mais sofisticada da arquitetura, componentes com regras e lógica de negócio podem ser distribuídos e gerenciados por servidores de aplicação que possuem um container Enterprise Java Bean.



O maior benefício que o uso de um servidor de regras de negócio pode proporcionar é a escalabilidade. Isso porque um Web Container não possui sistemas de gerenciamento de entidades, persistencia e processos transacionais.
A principal conseqüência de uma arquitetura Web Centric é que temos um grande desperdício de viagens ao banco de dados. Com arquitetura Web Centric se uma página JSP precisa de uma lista das entidades clientes, uma viagem ao banco de dados é feita para recuperar as informações exclusivamente para aquele request.
Temos três tipos de Enterprise Java Beans:
1. Session Bean: voltados para processos e transações
2. Entity Bean: objetos que representam uma entidade e necessitam de persistência.
3. Message Driven Bean: são processos disparados por filas assíncronas *(Message Queue Server).*

J2EE e a camada de Dados

A maneira mais popular de se comunicar com a camada de dados é através da API JDBC. JDBC foi amplamente aceito por quase todos os fornecedores de banco de dados. Um banco de dados compatível com JDBC é um banco de dados que possui um "driver" de comunicação do Java com Banco de Dados implementados com TCP/IP. Outra característica desta API é sua eficiência com banco de dados relacionais.
Recomendamos que você adote como melhor prática de desenvolvimento, o isolamento de todos os métodos que trabalham com o banco de dados em uma camada chamada de **DAO – Data Access Object.** As mudanças em bancos de dados *(marca e modelagem)* na maioria das vezes são imprevisíveis, portanto o isolamento de código SQL das entidades reduz o impacto quando mudanças no banco de dados ocorrerem.



Existe também uma API chamada **Connector** que trabalha com o **mesmo conceito de driver do JDBC**, porém esta é voltada para fabricantes de softwares em geral, ou seja, **não só banco de dados. ERPs famosos como SAP**, softwares de transações de sistema de grande porte e outros possuem conectores **J2EE.**