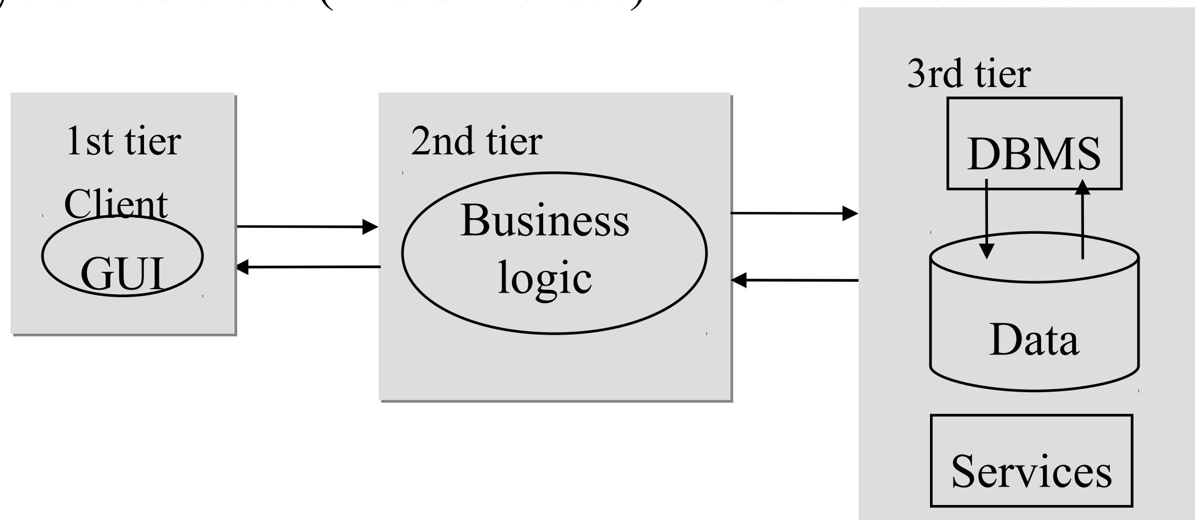
# Java EE Introduction, Content

- Component Architecture: Why and How
- Java EE: Enterprise Java

# The Three-Tier Model

- The three -tier architecture allows to maintain state information, to improve performance, scalability and availability
  - Client in the first tier - presentation layer
  - Business logic in 2nd tier - security and personalization of the client
  - System services (and databases) in 3rd tier – services and storage



# Why Component Architecture

- Rapid application development
- Reusability and portability of parts of the software system
- Decrease of the need for in-house expertise

# Why Container-Managed Components

- Avoid writing infrastructure code for non-functional requirements like navigation, validation, transactions, security and O/R-mapping.
- Frameworks are thoroughly tested and proven to work well.
- Lots of documentation, easy to get help.

# Why Container-Managed Components, Cont'd

- Non-functional requirements are difficult to code.
- Not using a framework means writing new code which means introducing new bugs.
- Callback style makes sure all calls to non-functional requirements code are made at the right time.
  - Handled by the framework.

# How Component Architecture

- ***Component***
  - a reusable program building block for an application;
  - presents a manageable, discrete chunk of logic (functionality);
  - implements a set of well-defined interfaces.
  - Examples: pricing component, billing component
- ***Container***
  - an application program or a subsystem in which the component lives;
  - Component's context;
  - creates, manages and "glues" components;
  - provides life cycle management, security, deployment, and runtime services for components it contains (component contract).
  - Examples: Web container (for JSF pages and Servlets), EJB container (for EJBs)

# How Component Architecture (cont'd)

- ***Specifications***
  - For components, containers (hosts), and tools (development, deployment)
  - Set of conventions (standards) for
    - Container (Context) Services
    - APIs (classes, interfaces, methods, constructors)
      - Names
      - Semantics

- A well-defined component architecture is a set of standards (specifications) necessary for different vendors to write the components, containers and tools

# Development and Deployment

- ***Development tools***
  - for developing components
    - NetBeans (Oracle)
    - Eclipse (eclipse.org )
- ***Deployment tools***
  - for configuring (customizing, naming) and packaging components
    - NetBeans
    - Admin console

# Application Servers

- ***An application server***
  - Run time environment for component-based applications
    - Applications are deployed and run on an application server
  - Provides containers and services for applications made of components.
    - Services: naming, connectivity, persistence, transactions, etc.
  - Provides services for clients
    - Downloadable clients (HTML)
  - Some examples:
    - GlassFish (Oracle)
    - Tomcat (Apache)

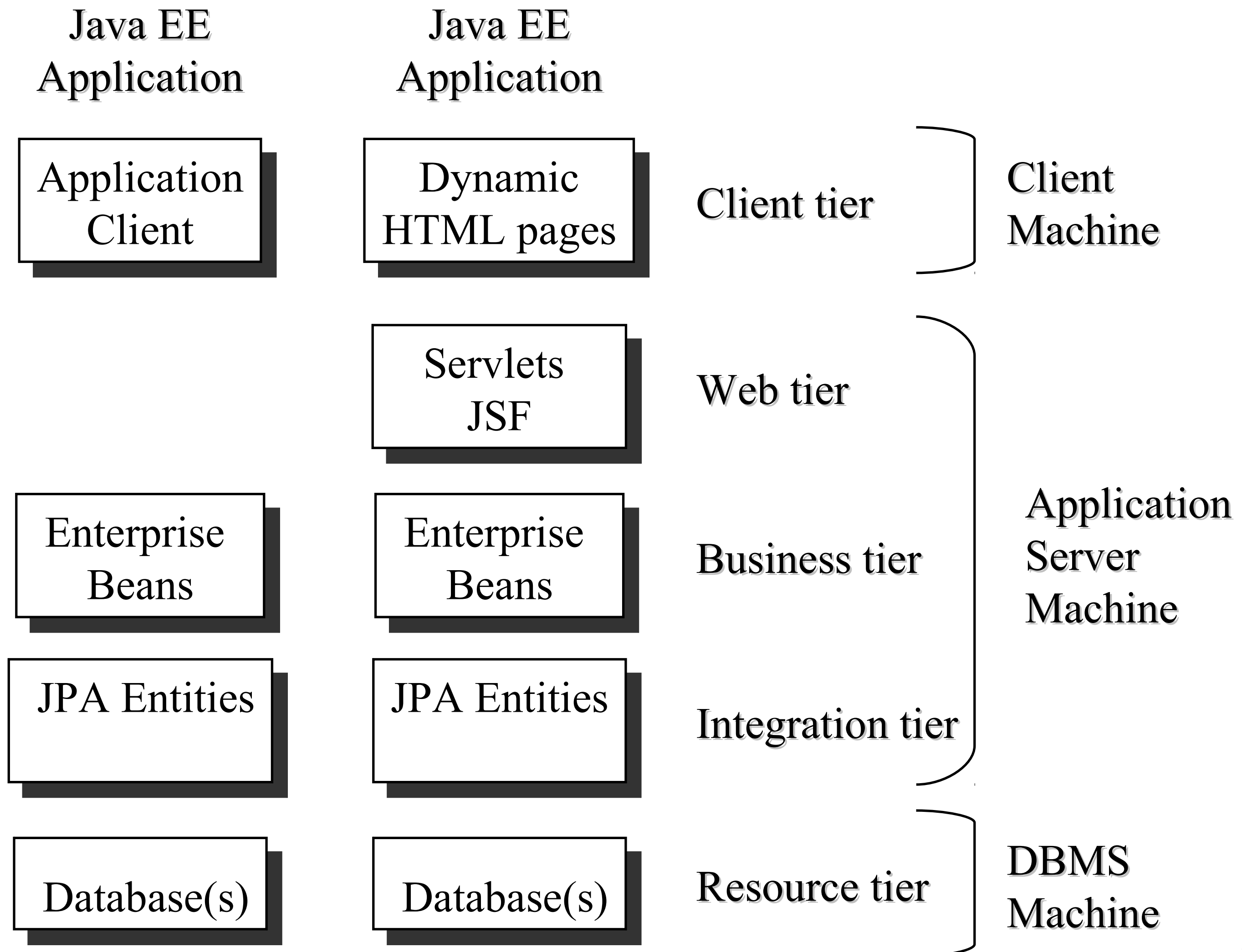
# Java Platform, Enterprise Edition (Java EE)

http://www.oracle.com/technetwork/java/javaee/overview/index.html

# Some Useful Links

- Java Platform, Enterprise Edition (Java EE)
  - http://www.oracle.com/technetwork/java/javaee/overview/index.html
- Java EE Training & Tutorials
  - http://www.oracle.com/technetwork/java/javaee/documentation/index.html
- The Java EE 6 Tutorial:
  - http://download.oracle.com/javaee/6/tutorial/doc/
- Java developer connection at
  - http://www.oracle.com/technetwork/index.html

# Multi-Tiered Java EE Applications



# The Java EE Technologies

- Four groups:
  - Enterprise Application Technologies
  - Web Application Technologies
  - Management and Security Technologies
  - Web Services Technologies

# Enterprise Application Technologies

- **Enterprise JavaBeans (EJB)**
  - EJBs are the standard building blocks for corporate server applications.
- **J2EE Connector Architecture**
  - An architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems.
- **Java Message Service API (JMS)**
  - A specification of an API for enterprise messaging services. To create, send, receive, and read messages.
- **Java Persistence API (JPA)**
  - Provides a POJO (Plain Old Java Object) persistence model for object-relational mapping. Developed and use for EJB, but can be used directly.
- **Java Transaction API (JTA)**
  - An abstract API for resource managers and transactional applications. Also used in writing JDBC drivers, EJB containers and hosts.
- **JavaMail**
  - A set of abstract classes that models a mail system.

# Web Application Technologies

- **Java Servlets**
  - An API for extending the functionality of a web server.
  - Java classes that process requests and construct responses, usually for HTML pages
  - Replaces CGI
  - Provides a gateway between Web clients and EJBs
- **JavaServer Faces (JSF)**
  - An API for representing UI components (dynamic HTML) and managing their state; handling events from components; server-side data validation and conversion.
- **JavaServer Pages (JSP)**
  - Text-based documents that are compiled into servlets and define how dynamic content can be added to static content in HTML or other markups.
- **JavaServer Pages Standard Tag Library (JSTL)**
  - Encapsulates core functionality common to many JSP applications, e.g. iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

# Web Services Technologies

- Java API for RESTful Web Services (JAX-RS)
- Java API for XML-Based Web Services (JAX-WS)
  - Replaces JAX-RPC
- Java Architecture for XML Binding (JAXB)
  - Provides a convenient way to bind an XML schema to a representation in Java code.
- SOAP with Attachments API for Java (SAAJ)
  - Provides a standard way to send XML documents over the Internet from the Java platform.
- Streaming API for XML
  - Streaming Java-based, event-driven, pull-parsing API for reading and writing XML documents.
- Web Service Metadata for the Java Platform

# Java Servlet

**`javax.servlet`**

Servlet Home page:

http://www.oracle.com/technetwork/java/index-jsp-135475.html

# Java Servlet, Content

- Introduction
- Life Cycle
- Request Handling
- Thread Safety
- Our First Servlet
- Request
- Response
- Sessions
- Filters
- Listeners
- Servlet Context

# Introduction

- A Servlet is program running on a web server.
- Used mainly as controller in web applications
  - Receives HTTP requests and directs the request to the model component that can handle it.
  - The controller Servlet is part of a framework (e.g. JSF, Struts) and normally not written by application developer.
- Can also be used to generate HTTP response.
  - Mainly pages without text, e.g. images.

# Introduction, Cont'd

- Servlets live inside a framework, the servlet container.
  - Have no **`main`** method, only called by the container.

# Life Cycle

1. The class is loaded into the Java VM.
2. An object is instantiated.
3. The servlet's **`init()`** method is called.
4. Each HTTP call is directed to the **`service`** method, who's default implementation will call the **`doGet`** or **`doPost`** method (depending on the HTTP method).
5. If the servlet is garbage collected, its **`destroy`** method is called before garbage collection.

# Request Handling

- What happens when a HTTP request is received?
  1. The container creates new objects of the class **`HttpServletRequest`** representing the HTTP request and **`HttpServletResponse`** representing the HTTP response.
  2. The container interprets the URL and decides which servlet to call.

# Request Handling (cont)

3. The container creates a new thread and use it to call the servlet's **`service`** method, passing the objects created above.
4. The service method decides if **`doGet`** or **`doPost`** shall be called, and calls that method.
5. That method uses the request object to get information about the HTTP request and the response object to create the HTTP response.

# Request Handling (cont)

6. The container sends the HTTP response and discards the request and response objects.

# Thread Safety

- Each request to the same servlet is executed in a separate thread but uses the same servlet instance.
- Instance variables in the servlet, and objects called by the servlet, are not thread safe.
  - Avoid such fields that are not final!
  - Try not to use **`synchronized`** since it will reduce performance.

# Our First Servlet

```
package helloworld;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

# Our First Servlet, Cont'd

```
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>NEVER EVER WRITE HTML IN A SERVLET</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
}
```

# Our First Servlet (cont)

- The WebServlet annotation specifies the servlet's URL.
- All servlets should inherit **`javax.servlet.http.HttpServlet`**.
- An HTTP get request will result in a call to the **`doGet`** method.
- All that is written to the stream returned by **`response.getWriter()`** will be sent to the browser by the container.

# ***Never write HTML in a servlet!!***

- Bad cohesion since the servlet would handle both input, output and probably also act as controller.
- Bad cohesion to mix languages (Java and HTML).
- Difficult to maintain since there might be different developers for Java and HTML.
- Very messy with all line breaks in strings.
- Impossible to use HTML editors
- JSF is designed for this and contains many helpful features.

# Request

- Instances of the **`javax.servlet.http.HttpServletRequest`** class are used to represent HTTP requests sent to the servlet.
  - Passed to **`doGet`**/**`doPost.`**

# Request, Cont'd

- **`HttpServletRequest`** can be used to:
  - Get HTTP parameters.
  - Store and read Java objects (attributes).
  - Get the URL of the request.
  - Get information about the client like ip address or browser version.
  - Get HTTP headers.
  - Get cookies.
  - Get session related data.

# Request, Cont'd

- When the following form is submitted there will be three HTTP parameters, **`email`**, **`name`** and **`action`**.

```
<form action="MyServlet" method="post">
  <input type="text" name="email"/>
  <input type="text" name="name"/>
  <input type="submit" value="action"/>
</form>
```

# Request, Cont'd

- These parameters can be read like this in a servlet:

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse resp) {
  String email = request.getParameter("email");
  String name = request.getParameter("name");
  String action = request.getParameter("action");
}
```

# Request, Cont'd

- Any Java object can be used as an attribute.
- Attributes are both stored and read by servlets.
- Used to pass objects from one servlet to another when requests are forwarded.
- The **`getAttribute`**/**`setAttribute`** methods in **`HttpServletRequest`** are used for this.

# Response

- Instances of the **`javax.servlet.http.HttpServletResponse`** class are used to represent HTTP answers from the servlet.
  - Passed to **`doGet`**/**`doPost.`**

# Sessions

- HTTP is stateless, sessions are used to introduce state.
- A client is identified with a cookie or some extension to the url (url rewriting). Some data on the server is associated with the session.

# Sessions, Cont'd

- The method **`request.getSession`** is used to create a session.
- The method **`session.invalidate`** is used to destroy a session.
- It is a good idea to set a time out after which a session is destroyed.

# Sessions, Cont'd

- Data is stored in the session object with the **`setAttribute`** method and read with **`getAttribute`**.
- Data in the session object is NOT thread safe since the same session is shared between all windows of the same browser on the same computer.
  - Avoid instance variables in the session object and in objects stored as attributes in the session object.
  - Try not to use **`synchronized`** since it will reduce performance.

# Filters

- A web resource can be filtered by a chain of filters in a specific order specified on deployment.
- A filter is an object that can transform the header and content (or both) of a request or response:
  - Query the request and act accordingly;
  - Block the request-and-response pair from passing any further;
  - Modify the request headers and data;
  - Modify the response headers and data.

# Filters, Cont'd

- A filter class is defined by implementing the **`Filter`** interface and providing the **`@WebFilter`** annotation as shown below.

```
@WebFilter("/*")
public class MyFilter implements Filter
```

- The **`doFilter`** method is called before and after a resource processing any URL that matches the URL pattern specified in the **`@WebFilter`** annotation.

# Listeners

- Listener classes will receive and handle life-cycle events issued by the Web container. Context, session and request events can be handled.
- For example, a context listener, often used to initialize singletons used by servlets.

```
@WebListener
public final class ContextListener implements ServletContextListener {
    private ServletContext context = null;
    public void contextInitialized(ServletContextEvent event) {
        context = event.getServletContext();
        try {
            BookDAO bookDB = new BookDAO();
            context.setAttribute("bookDB", bookDB);
        } catch (Exception ex) { e.printStackTrace();}
    }

    public void contextDestroyed(ServletContextEvent event) {
        context = event.getServletContext();
        BookDAO bookDB = (BookDAO) context.getAttribute("bookDB");
        bookDB.remove();
        context.removeAttribute("bookDB");
    }
}
```

# Listeners, Cont'd

| Source | Event | Listener Interface |
|---|---|---|
| Web context | Initialization and destruction | javax.servlet. ServletContextListener |
| | Attribute added, removed, or replaced | javax.servlet. ServletContextAttributeListener |
| Session | Creation, invalidation, activation, passivation, and timeout | javax.servlet.http. HttpSessionListener, javax.servlet.http. HttpSessionActivationListener, |
| | Attribute added, removed, or replaced | javax.servlet.http. HttpSessionAttributeListener |
| Request | A servlet request has started being processed by web components | javax.servlet. ServletRequestListener |
| | Attribute added, removed, or replaced | javax.servlet. ServletRequestAttributeListener |

# Accessing the Web Context

- The context in which web components execute, i.e. the web container
- To get the context, call the **`getServletContext`** method on the servlet.
- The context object implements the **`ServletContext`** interface.
- The web context provides methods for accessing:
  - Initialization parameters,
  - Resources associated with the web context,
  - Attributes,
  - Logging capabilities.
- For example, retrieving an attribute set by a Context listener, see slide 41:

```
@WebServlet("/catalog")
public class CatalogServlet extends HttpServlet {
    private BookDAO bookDB;
    public void init() throws ServletException {
        bookDB = (BookDAO)getServletContext().getAttribute("bookDB");
        if (bookDB == null) {
            throw new UnavailableException("Couldn't get database.");
        }
    }
}
```