J2EE - Java 2
Enterprise Edition
Version 1.4

Peter Tröger
Hasso-Plattner-Institute, University of Potsdam
Agenda

- Partially based on Sun’s J2EE 1.4 Tutorial
- Available on java.sun.com (1500 pages)
- Multiple examples for Sun Application Server
- Other sources: Sang Shin, Introduction to J2EE; Spielman and Kunnumpurath, Pro J2EE 1.4
- Today: J2EE Overview, Web Tier
- Next Session: J2EE Business Tier
- Session after next: Future of J2EE (EJB 3, Spring)
Enterprise Application Frameworks

• Presentation logic, business logic, data model and access logic, system services

• Single-Tier: Mainframe-based, dumb terminals, centralized model, monolithic application

• Two-Tier: Fat client talking to back-end database, standardized data access

• Three-Tier: Thin client for presentation, middle-tier server for system services, database tier, business logic and data model changeable
Three-Tier Model

- **RPC-based**: Tight coupling of client and middle-tier (think of implementation change)

- **Object-based**: business logic and data model encapsulated in objects (CORBA, RMI)

- **Trend**: web server-based three tier model
Problems to be solved

- (Distributed) transactions
- Remoting functionality
- Security (Authentication, Authorization)
- Persistency (object-relational gap, caching)
- Pooling (database connections, threads, objects)
- Scalability
- Legacy integration
J2EE Motivation

• Component-based distributed applications for the enterprise

• Consider security, speed, transactional and reliable behavior

• Avoid vendor-lock

• Fast application design and development

• “Write once, run everywhere”
Java 2 Platform

- J2SE - Java 2 Standard Edition (desktop)
- J2ME - Java 2 Micro Edition (consumer device)
- J2EE - Java 2 Enterprise Edition (server)
J2EE Platform

- First introduction in 1999, J2EE 1.4 approved by Java Community Process in November 2003
- API compatibility for Enterprise Application Servers
- Standardized development and deployment of portable, distributed enterprise applications
- Multi-tier model for enterprise applications
  - Standardized communication of application parts enables distributed applications
- Component-based development
  - Re-usable discrete modules
- Portability through standards
- Integration of existing information systems
What Makes Up J2EE

- API and technology specification
- Development and deployment platform
- Reference implementation as part of the SDK
- Compatibility test suite
- J2EE brands, J2EE blueprints and best practices
- Sample codes
End-to-end Platform

B2B Applications

B2C Applications

Web Services

Wireless Applications

Application Server

Existing Applications

J2EE Application Server

Web Server JSP, Servlets

Enterprise Information Systems (EIS):
Relational Database,
Legacy Applications,
ERP Systems

Client Tier

Middle Tier

Client

Client

Other Services:
JNDI, JMS,
JavaMail™

Client

Client

Client

HTML/XML

Enterprise JavaBeans™

Enterprise JavaBeans™
J2EE Components

- Self-contained functional software unit, written in Java
  - Assembled into a J2EE application
  - Communicates with other components
  - Executed and managed by application server
- Client-side components: applications, applets
- Web components: Servlets, JavaServer Pages (JSP)
- Business components: Enterprise JavaBeans (EJB)
J2EE Modules

- Contains of one or more J2EE components for the same container type and one deployment descriptor
- Technically all JAR files, with own deployment descriptor
  - EJB modules (.JAR) - EJB class files
  - Web modules (.WAR) - Servlet class files, JSP files, class files, GIFs, HTML files
  - Application client modules (.JAR) - class files
  - Resource adapter modules (.RAR) - class files, native libraries, documentation; intended for JCA (EIS tier)
J2EE Containers

- Management of components in containers
- Standardized runtime environment
- Interpose on all method calls
- Provide specific J2EE services, which can be expected by the component
- Enables platform-independent deployment
- Container settings as part of the assembled application (e.g. security, connectivity)
J2EE Containers
Packaging

• J2EE application is delivered as Enterprise Archive File (EAR) - a JAR file with new extension

• Contains of J2EE modules and deployment descriptors
Deployment Descriptors

- XML document describing deployment setting
- J2EE deployment descriptor
  - Defined by J2EE specification
  - Own specification for each module type
- Runtime deployment descriptor
  - J2EE implementation-specific settings
Responsibilities

Containers Handle
- Concurrency
- Security
- Availability
- Scalability
- Persistence
- Transaction
- Life-cycle management
- Management

Components Handle
- Presentation
- Business Logic
Application Settings

• J2EE security model
  • Restrict user access to system resources
• J2EE transactional model
  • Specification of related methods as transactional unit
• JNDI lookup service
  • Unified interface to naming and directory services
• J2EE remote connectivity model
  • Management of client -> EJB method invocation
Container Settings

- EJB and service life cycles
- Database connection resource pooling
- Data persistence
  - May be overridden by BMP entity beans
J2EE Role Model

- J2EE Product Provider
  - Tool Provider
- Application Component Provider
  - Enterprise Bean Developer, Web Component Developer, Application Client Developer, Application Assembler
- Application Deployer and Administrator
J2EE Products

- Multiple compatible products:
  Apache Geronimo 1.0, BEA WebLogic Server 9.0,
  IBM WebSphere AS 6.0, JBoss AS 4.0, Sun AS 8,
  Oracle AS 10g

- J2EE Compatibility Test Suite (CTS)

- Java Application Verification Kit (AVK)
  for the Enterprise

- J2EE reference implementation from Sun
  (source and binary)
J2EE 1.4 APIs

Figure 1–7 illustrates the availability of the J2EE 1.4 platform APIs in each J2EE container. The diagram shows the integration of various components such as Applet Container, Web Container, EJB Container, Application Client Container, and Database. The key takeaway is that you do not have to write any SQL code or use the JDBC™ API (see JDBC API, page 22) directly to perform certain tasks in J2EE 1.4.
J2EE APIs

- Enterprise JavaBeans Technology (EJB)
- Java Servlet Technology
- JavaServer Pages Technology (JSP)
- Java Message Service API (JMS)
- Java Transaction API (JTA)
- Java Mail API & SPI / Java Activation Framework (JAF)
- Java API for XML Processing (JAXP)
- Java API for XML-based RPC (JAX-RPC)
- SOAP with Attachments API for Java (SAAJ)
- Java API for XML Registries (JAXR)
- J2EE Connector Architecture (JCA)
- Java DataBase Connection (JDBC) API and SPI
- Java Naming and Directory Interface (JNDI)
- Java Authentication and Authorization Service (JASS)
- Java Authorization Contract for Containers API (JACC)
J2EE Tiere ;-)
Client Tier

- Web clients ("thin client")
  - Render web pages provided by web tier
  - Pages may contain Java applets (embedded client application, running in the browser)
- Application clients ("rich client")
  - Direct connection to business tier (EJB)
Client Tier
Web Tier

- Servlets: Java classes, which dynamically process requests and construct responses

- JavaServer pages: Text-based documents, includes servlet snippets
  - Static text data - HTML, WML, XML
  - JSP elements for dynamic content

- Static HTML, applets, utility classes - bundled with J2EE application
Web Tier

Web Browser
Web Pages, Applets, and Optional JavaBeans Components

Application Client and Optional JavaBeans Components

JSP Pages Servlets

JavaBeans Components (Optional)

Business Tier

J2EE Server
Business Tier

• Contains components with business code
  • program logic which solves the need of a particular business domain functionality

• Business components are reflected as Enterprise JavaBean (EJB)
  • Business logic as reusable component
  • Different Bean types (Session, Entity, Message)
Business Tier

- Web Browser
  - Web Pages, Applets, and Optional JavaBeans Components
- Application Client and Optional JavaBeans Components
- JSP Pages Servlets
- JavaBeans Components (Optional)
- Entity Beans
  - Session Beans
  - Message-Driven Beans
- Business Tier
- J2EE Server
- EIS Tier
  - Database and Legacy Systems
EIS Tier

- Coupling with Enterprise Information Systems
  - Enterprise resource planning (ERP) systems
  - Mainframe transaction processing systems
  - Database systems
  - Other legacy information systems
- Integration of existing data and infrastructure
J2EE Application Anatomies
J2EE Web Tier
Web Applications

- Presentation-oriented web application (JSP pages for generation of markup)

- Service-oriented web application (web service endpoints, usually as servlet)

- Web container as runtime platform (request dispatching, security, concurrency, life-cycle management, API for components)

- Life-cycle: code development, deployment descriptor development, compilation, packaging, deployment, usage
Web Modules

- Web module as deployable unit
- .WAR file
- Portable format, described in Servlet specification
- Support for unpackaged modules
Java Servlets

- Portable way for dynamic web content

- For any kind of request-response pattern, but HTTP-specific servlet classes \( \text{javax.servlet.http} \)

- Servlet life cycle: Instantiation through container, \text{init} method, \text{service} method, \text{destroy} method

- Possibility for event listeners in the WAR file (initialization, destruction, request, activation, passivation, invalidation)

- Concurrent access, synchronization in the code
Servlet vs. CGI

CGI Based Webserver

- Request CGI1
- Request CGI2
- Request CGI1

CGI Based Webserver

- Child for CGI1
- Child for CGI2
- Child for CGI1

Servlet Based Webserver

- Request Servlet1
- Request Servlet2
- Request Servlet1

JVM

- Servlet1
- Servlet2

Request CGI1

05/15/2005
Servlet Life Cycle

- public void init() throws ServletException {}

- public void destroy() {}

- For GenericServlet, override
  public void service(ServletRequest, ServletResponse)

- For HttpServlet, override
  public void doGet(...) | doPost(...) | doPut(...) | doDelete(...)

- service() method as dispatcher

- Request object contains all relevant data (e.g. HttpServletRequest)
Servlet Example

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

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest req,
                        HttpServletResponse res)
        throws ServletException, IOException{
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("Hello World");
    }
}
```
Information Sharing

- Through private helper objects
- Through sharing of objects that are attributes of a public scope
- Through a database
- Through invocation of other web resources
- Synchronization is task of the developer
Example

```java
import java.io.*; import java.util.*;
import javax.servlet.*; import javax.servlet.http.*;

public class HolisticCounter extends HttpServlet {
    static int classCount = 0;  // shared by all instances
    int count = 0;              // separate for each servlet
    static Hashtable instances = new Hashtable();  // also shared

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        count++;
        out.println("Since loading, this servlet instance has been accessed " +
                    count + " times.");

        instances.put(this, this);
        out.println("There are currently " + instances.size() + " instances.");
        classCount++;
        out.println("Across all instances, this servlet class has been " +
                    "accessed " + classCount + " times.");
    }
}
```
Client Sessions

• Session is created automatically

• Timeout period definition (programmatically, deployment descriptor of WAR file)

• invalidate operation

```java
public class CashierServlet extends HttpServlet {

public void doGet (HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
    // Get the user's session and shopping cart
    HttpSession session = request.getSession();
    ShoppingCart cart = (ShoppingCart)session.getAttribute("cart");
    ...
    // Determine the total price of the user's books
    double total = cart.getTotal();
}
```
Scope Object Attributes

Figure 11–1 shows the scoped attributes maintained by the Duke's ... you are guaranteed that no two threads will exe-
cute concurrently in the servlet's service method. A web container can
Working with Filters

• Filters are web application components (like servlets), contained in the web archive

• Chained to the container’s processing pipeline, before request processing / after response sending

• Possibility to build up filter chains

• Add new / prototypical functionality through interception, apply adapter pattern
JavaServer Pages

- Clear separation of presentation and business logic code (HTML, XML vs. JavaBeans, own tags)

- Language for JSP pages, including access to server-side objects and custom tag libraries

- Support mixture of static and dynamic content
  - Static data: HTML, SVG, WML, XML, ...
  - JSP elements, expressed in ‘standard’ syntax (.jsp) or XML syntax (.jspx)

- In case, container translates JSP page to a servlet class and compiles the class
JSP Syntax

- JSP expression: `<% expression %>`
- JSP Scriptlet: `<% code %>`
- JSP Declaration: `<%! code %>`
- JSP page directive: `<%@ page att="val" %>`
- Always XML equivalent defined
  `<jsp:expression>
   `<jsp:directive.page att="val"/>
  </jsp:expression>`
JSP Example

```jsp
<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Using JavaServer Pages</title>
</head>
<body bgcolor="#FDF5E6" text="#000000" link="#0000EE">
<ul>
<li><b>Expression.</b><br>Your hostname: <%= request.getRemoteHost() %>.</li>

<li><b>Scriptlet.</b><br>\<% out.println("Attached GET data: "+request.getQueryString()); %></li>

<li><b>Declaration (plus expression).</b><br>\<%! private int accessCount = 0; %>
Accesses to page since server reboot: <%= ++accessCount %></li>

<li><b>Directive (plus expression).</b><br>\<%@ page import = "java.util.*" %>
Current date: <%= new Date() %></li>
</ul>
</body>
</html>
```
Further JSP Features

- HTTP request forwarding (jsp:forward)

- Use JavaBeans within the JSP page

  ```
  <jsp:useBean id="test" class="hall.SimpleBean" />
  <jsp:setProperty name="test" property="message" value="Hello WWW" />
  <jsp:getProperty name="test" property="message" />
  ```

- Scope attribute (PageContext (default), ServletRequest, HttpSession, ServletContext)

- Inclusion on compile / execution time

  ```
  <%@ include file="url" %>
  <jsp:include page="relative URL" />
  ```
Servlet-based Technologies

- JavaServer Pages Standard Tag Library (JSTL)
  - Flow control, XML manipulation, internationalization, database access, ...

- JavaServer Faces
  - API for UI representation and state handling
  - Management of page navigation issues
  - Realized as JSP tag library
  - Works with ‘backing beans’, associated to a component
<HTML>
<HEAD> <title>Hello</title></HEAD>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<body bgcolor="white">
<f:view>
<h:form id="helloForm">
<h2>Hi. My name is Duke. I'm thinking of a number from
<h:outputText value="#{UserNumberBean.minimum}"/>
<h:outputText value="#{UserNumberBean.maximum}"/>
Can you guess it?</h2>
<h:graphicImage id="waveImg" url="/wave.med.gif" />
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}"
<f:validateLongRange minimum="#{UserNumberBean.minimum}" maximum="#{UserNumberBean.maximum}" />
</h:inputText>
<h:commandButton id="submit" action="success" value="Submit" />
<h:message style="color: red" id="errors1" for="userNo"/>
</h:form>
</f:view>
</HTML>
JSF and the Container

Web Container

- Bookcashier.jsp
  - id="name"
  - id="fanClub" binding
  - id="add" action
  - #{cashier.name}
  - #{cashier.specialOffer}
  - #{catalog.add}

- CashierBean
  - name
  - specialOffer

- CatalogBean
  - add

- ShoppingCart

- NameChanged Listener
  - name component
  - value

- FanClub component
JSF Life Cycle

The life cycle of a JSF application's components and other parts is as follows:

1. **Faces Request**
   - Restore View
   - Apply Request Values
   - Process Events
   - Process Validations
   - Process Events
   - Response Complete

2. **Response Complete**
   - Response Complete

3. **Faces Response**
   - Render Response
   - Response Complete

4. **Response Complete**
   - Update Model Values
   - Validation / Conversion Errors / Render Response

5. **Response Complete**
   - Response Complete

The diagram illustrates the flow of events and the lifecycle of the JSF application, showing how different components interact within this cycle.
Internationalization

• Tailoring of messages and labels
  • Dispatcher servlet, JSP page in each language
  • Isolate locale-sensitive data into resource bundles (Java property file)
• Date and number formatting
• Character encoding
<application>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>en</supported-locale>
    <supported-locale>en_US</supported-locale>
    <supported-locale>de</supported-locale>
  </locale-config>
  <message-bundle>
    de.dcl.messageresource.MessageResources
  </message-bundle>
</application>

welcome=Welcome on this site.
login={0} have been logged in on {1}.

<f:loadBundle basename="myPackage.MessageResources" var="msg"/>
<h:outputText value="#{msg.welcome}" />
<h:outputFormat value="#{msg.login}"
  <f:param value="Peter" />
  <f:param value="#{myBean.date}" />
</h:outputFormat>
Security in Web Applications

- HTTPS support is mandatory in J2EE-compliant web containers

- Security requirements: User identification and authentication, user account systems, authorization, audit logs, authentication schemes

- Declarative security (web.xml) vs. programmatic security (HttpServletRequest methods)
JAX-RPC

- XML-based RPC API for building web services
- No need for generating or parsing SOAP data (proxy / stub generation)
- Service Endpoint Interface (SEI) in Java, only JAX-RPC types allowed
- since June 2002 (before J2EE 1.4)
JAX-RPC Steps

- Code and compile SEI, implementation classes and interface configuration file
- Prepare package with wscompile and deploytool
- Deploy server application
- Code client classes and WSDL config file
- Use wscompile for stub files, compile and run the client application
JAX-RPC Types

• Primitive types and the according classes (boolean, byte, double, float, int, long, short)

• java.lang.String, java.math.BigDecimal, java.math.BigInteger, java.net.URI, java.util.Calendar, java.util.Date

• Arrays of supported JAX-RPC types

• Value types (members of JAX-RPC type)

• JavaBeans (properties of JAX-RPC type)
package helloservice;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloIF extends Remote {
    public String sayHello(String s) throws RemoteException;
}

package helloservice;
public class HelloImpl implements HelloIF {
    public String message = "Hello";
    public String sayHello(String s) {
        return message + s;
    }
}

JAX-RPC Example
JAX-RPC Interface
Configuration File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service
        name="MyHelloService"
        targetNamespace="urn:Foo"
        typeNamespace="urn:Foo"
        packageName="helloservice">
        <interface name="helloservice.HelloIF"/>
    </service>
</configuration>
```

```bash
wscompile -define -mapping build/mapping.xml -d build -nd build -classpath build config-interface.xml
```
JAX-RPC Packaging and Deployment

• Web service is implemented as servlet

• Deployment of web module (WAR file, web application deployment descriptor)

• Specific endpoint URI is defined on deployment time

• [URI]?wsdl is supported
Static Stub Client

- Stub: local object that acts as a proxy for remote services
- Static Stub: Created at development time
- Needs only endpoint address at runtime
- Generation of stub classes with wscompile, based on WSDL configuration file

```xml
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="http://localhost:8080/hello-jaxrpc/hello?WSDL"
        packageName="staticstub"/>
</configuration>
```
package staticstub;
import javax.xml.rpc.Stub;
public class HelloClient {
    private String endpointAddress;
    public static void main(String[] args) {
        try {
            Stub stub = (Stub)(new MyHelloService_Impl().getHelloIFPort());
            stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                             args[0]);
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello("Duke"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    private static Stub createProxy() {
        return
    }
}
Other Clients

- *Dynamic proxy client* calls remote procedure through a class that is created during runtime.

- *Dynamic invocation interface client* can call the remote procedure with all information obtained at runtime (*wscompile* not needed).

- *Application client* is a J2EE application (EAR file), which locates the (local) service through JNDI.

- JSP page can act as static stub client.

- JAX-RPC allows advanced features (e.g. authentication).
package dynamicproxy;
import java.net.URL; ...
import dynamicproxy.HelloIF;

public class HelloClient {
    public static void main(String[] args) {
        try {
            String UrlString = args[0] + "?WSDL";
            String nameSpaceUri = "urn:Foo";
            String serviceName = "MyHelloService";
            String portName = "HelloIFPort";
            URL helloWsdlUrl = new URL(UrlString);

            ServiceFactory serviceFactory = ServiceFactory.newInstance();
            Service helloService = serviceFactory.createService(helloWsdlUrl,
                new QName(nameSpaceUri, serviceName));

            dynamicproxy.HelloIF myProxy = (dynamicproxy.HelloIF)
                helloService.getPort(
                    new QName(nameSpaceUri, portName),
                    dynamicproxy.HelloIF.class);

            System.out.println(myProxy.sayHello("Buzz");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }}}
package dii;
import javax.xml.rpc.Call; ...

public class HelloClient {
    private static String qnameService = "MyHelloService";
    private static String qnamePort = "HelloIF";
    private static String BODY_NAMESPACE_VALUE = "urn:Foo";
    private static String ENCODING_STYLEPROPERTY = "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
    private static String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";

    public static void main(String[] args) {
        try {
            ServiceFactory factory = ServiceFactory.newInstance();
            Service service = factory.createService(new QName(qnameService));
            QName port = new QName(qnamePort);
            Call call = service.createCall(port);
            call.setTargetEndpointAddress(args[0]);
            QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
            call.setReturnType(QNAME_TYPE_STRING);
            call.setOperationName(new QName(BODY_NAMESPACE_VALUE, "sayHello"));
            call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.IN);
            String[] params = {"Murph!");
            String result = (String)call.invoke(params);
        } catch (Exception ex) {ex.printStackTrace();}}}

DII Client
package appclient;
import javax.xml.rpc.Stub;
import javax.naming.*;
public class HelloClient {
  private String endpointAddress;
  public static void main(String[] args) {
    try {
      Context ic = new InitialContext();
      MyHelloService myHelloService = (MyHelloService)
          ic.lookup("java:comp/env/service/MyJAXRPCHello");
      appclient.HelloIF helloPort = myHelloService.getHelloIFPort();
      ((Stub)helloPort)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
      System.out.println(helloPort.sayHello("Jake!
      
    } catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
  }
}
JAX-RPC Summary

- J2EE clients should use stubs (loss of portability between JAX-RPC implementations)
- No service-specific exceptions with DII
- One-way communication only with DII
- J2ME supports only static stubs
- J2EE container manages service access
  \( \text{(service.getPort())} \)
Java API for XML Registries

- JAXR: Uniform API to XML registries for web services
- ebXML registry and repository standard
- UDDI standard
J2EE Business Tier
Enterprise Java Beans

- Development and deployment of scalable, transactional, server-side business logic components
- EJB container as runtime environment
- EJB client: servlet, application, other bean
- The ‘heart’ of J2EE
- EJB 2.1 specification, 646 pages
EJB as Java Component

- Every EJB is also a remote object (through RMI)
- Every EJB component is also a JavaBean component
  - Introspectible properties with accessor methods
- EJBs run in a standardized runtime environment (container) and have a uniform deployment format
  - Can rely on the existence of container services
J2EE & EJB

Web Components
EJBs

Browser

EJB clients talking to EJB server
DB & EIS Resources

web-tier EJB client talking to EJB server
Standalone EJB client talking to EJB server
EJB Design Principles

- EJB applications are loosely coupled
  - Access to other components / services (from other vendors) through arbitrary names
  - Can be authored without detailed knowledge of the environment
  - Assembling to an application without code change
- EJB behavior is entirely specified by interfaces
  - Increases portability and modularity
EJB Design Principles

• EJB applications never manage resources

• Access to external resource through their container

• Efficient resource management (allocation, deallocation, sharing, pooling) by container

• Container configuration as administrative task, no programmatic interface

• Container provides system services: persistency, security, transactions, connection pooling, lifecycle management, threading
EJB Types

- **Session Bean**
  - Represents transient communication with client app
  - When client is finished, session bean and data is gone
- **Entity Bean**
  - Represents persistent data, one row of a database table
  - When client is finished, data saving is ensured
- **Message-driven Bean**
  - Java Message Service (JMS) listener component
  - Session bean which reacts on asynchronous messages
Session Beans

- Purpose: Performs a task for a client, similar to an interactive session; hides business task complexity

- Only one client per session bean at the same time, session type is configured in the deployment descriptor

- Stateless Session Beans
  - No conversational state, container can assign any instance to a client

- Stateful Session Beans
  - Unique state information per client-bean session (conversational state)
Entity Beans

- Represents business object in a persistent storage (= row in a database table)
- Persistent, allow shared access, have primary keys, can participate in relationships with other entity beans
- Bean-managed persistence (BMP): State saving to database is implemented in the code
- Container-managed persistence (CMP): EJB container generates database calls (needs abstract schema), increases portability
- Abstract schema: part of deployment descriptor, specifies persistent fields and relationships
  - Enterprise Java Beans Query Language
  - Support for different multiplicity's and directions in relations
Message-Driven Beans

• Short-lived stateless JMS message listener component

• Messages can be send by any J2EE component

• Not intended to be directly used by clients or other components

• All instances of one message bean are equivalent

• Can be transaction-aware components
JMS

- API for performing asynchronous messaging
- Client-oriented interfaces, interaction with message-oriented middleware (MOM), e.g. IBM MQ Series
- Roles: messaging clients, messaging destinations, and a JMS-compatible messaging provider
- Point-to-point or publish-subscribe pattern
- Support for filtering and transactional messaging
- Message types: TextMessage, BytesMessage, ObjectMessage, MapMessage, StreamMessage
EJB Interface Types

- EJB must provide:
  - Home interface (local / remote):
    Used by clients to create / find specific EJB types
  - Client interface (local / remote):
    Business functionality provided to the client
  - Implementation class(es)
  - Message-driven beans need only implementation class
  - Local clients run in the same JVM, no marshalling needed
Local Interface vs. Remote Interface

• Bean should either have local or remote interfaces

• Application clients usually work with remote interface

• Component distribution vs. access latency

• Tight or loosely coupling of related beans

• Container-managed entity bean relationship demands local access (and same JAR file)

• Remote interface of a stateless session bean can also provide a web service, instead of RMI
Client Interface

• Remote client interface
  • Extends `javax.ejb.EJBObject`, which in turn extends `java.rmi.Remote`
  • Methods must throw `RMIException`
  • Arguments and return values must be serializable

• Local client interface extends `EJBLocalObject`

```java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.math.*;

public interface Converter extends EJBObject {
    public BigDecimal dollarToYen(BigDecimal dollars)
        throws RemoteException;
    public BigDecimal yenToEuro(BigDecimal yen)
        throws RemoteException;
}
```
Home Interface

• Client need a way to create an object reference

• Remote home interface extends EJBHome

• create(): Returns an object of the client interface type, multiple signatures supported

• Matching ejbCreate() object method for each create() in the home interface needed

```java
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface ConverterHome extends EJBHome {
   Converter create() throws RemoteException, CreateException;
}
```
Implementation

Class vs. Interfaces

• Container generates proxy classes for the implementation of the local / remote interfaces

• Business logic does NOT implement the local / remote interfaces, but one of the `javax.ejb.EnterpriseBean` derivations (`SessionBean`, `EntityBean`, or `MessageDrivenBean`)

• Ensures container-mediated client calls

• Binding of interfaces to implementation class through deployment descriptor
SessionBean Implementation Class

• Implements SessionBean interface
  • ejbCreate(): Initializes bean state
  • ejbRemove(): Clean up of bean state
  • ejbActivate(): Restore state information
  • ejbPassivate(): Save state information
  • setSessionContext(SessionContext): Retrieve context object from container (valid for whole bean lifetime)

• Optional notification about transaction boundaries by the container (SessionSynchronization interface)
Session Bean Lifecycle

- Create / remove called by client
- Bean passivation / reactivation (in case) by container
EntityBean
Implementation Class

• Implements EntityBean interface
  • ejbFindByPrimaryKey(): Finder method
  • ejbPostCreate(): Called after according ejbCreate() and setup of the transaction context
  • ejbLoad(): Load bean state from persistent storage
  • ejbStore(): Save bean state to persistent storage
  • [set|unset]EntityContext(EntityContext): Associate / disassociate bean identity context object
Entity Bean Lifecycle

- Client or container can cause state transition
- With BMP, ejbCreate / ejbActivate must set primary key field
- No association to EJB object identity in pooled state
MessageDrivenBean Implementation Class

- Implements `MessageDrivenBean` interface
  - `ejbCreate()`: Single method, no argument
  - `setMessageDrivenContext()`: Container stores identity context object
  - `ejbRemove()`: Called before destruction

- Also implements `javax.jms.MessageListener` (onMessage() method)
Message-Driven Bean Lifecycle

- Non-existent or ready to receive messages
- Stateless implementation
Application Client

• Create initial name context (JNDI name-to-object binding set)

• Obtain environment naming context of the application client

• Retrieve name-bound reference object from JNDI

• Narrow reference object to bean home interface

```java
import java.math.*;
Context initial = new InitialContext();
Context myEnv = (Context)initial.lookup("java:comp/env");
Object objref = myEnv.lookup("ejb/SimpleConverter");
ConverterHome home =
   (ConverterHome)PortableRemoteObject.narrow(objref, ConverterHome.class);
Converter currencyConverter = home.create();
BigDecimal param = new BigDecimal("100.00");
BigDecimal amount = currencyConverter.dollarToYen(param);
```
JNDI

• Java Naming and Directory Interface

• Associate names and attributes with objects, provide way to access the objects by their name

• Binding: Association between a name and an object

• Context: Set of bindings, name can be bound to another subcontext (similar to FS directory structure)

• Entry point with `javax.naming.InitialContext` class

• Provider concept (LDAP, RMI registry, DNS, filesystem, NDS, NIS)
Web Client

- Create initial name context (JNDI name-to-object binding set)
- Retrieve name-bound reference object from JNDI
- Narrow reference object to bean home interface
- Create instance through home interface and use it

```jsp
<%@page import="javax.naming.*"%>  <%@page import="javax.rmi.*"%>
<%@page import="org.example.*"%>
...
int op1 = Integer.parseInt(request.getParameter("op1"));
int op2 = Integer.parseInt(request.getParameter("op2"));
Context initial = new InitialContext();
Object objref = initial.lookup("AddBean");
AddRemoteHome home =
(AddRemoteHome)PortableRemoteObject.narrow(objref, AddRemoteHome.class);
AddRemote addB = home.create();
%>...
<p>The addition result is <%= addB.add(op1,op2)%></p>
```
EJB Packaging

• Deployment descriptor (e.g. persistency type, transaction attributes)

• Enterprise bean classes

• Client (local / remote) and home interface

• Helper classes
EJB Naming Conventions

- EJB name: `<name>Bean`
- EJB class: `<name>Bean`
- Home interface: `<name>Home`
- Remote interface: `<name>`
- Local home interface: `<name>LocalHome`
- Local interface: `<name>Local`
More Container APIs

- Java Transaction API (JTA)
- Descriptive vs. programmatic security
- EJB Timer Service
- XML document processing
- Java Management Extensions (JMX)
- J2EE Management Services
- EJBs with web service interface
Which ONE of these SUSPECTS caused your PROJECT to fail?
Thank You