<https://tomee.apache.org/download.html>

download Apache TomEE 8.0 , Tom EE Plus Server.

Graphical user interface

Description automatically generated

Windows – download TomEE Plus Zip. Click on the name and it will take you to download page

Graphical user interface, text, application, email

Description automatically generated

1. Unzip it and go to eclipse – server option – add it
2. Server option – select tomcat v9.0

Graphical user interface, text, application

Description automatically generated

**Java EE : Java Enterprise Edition**

Diagram

Description automatically generated

Oracle Glassfish App Server

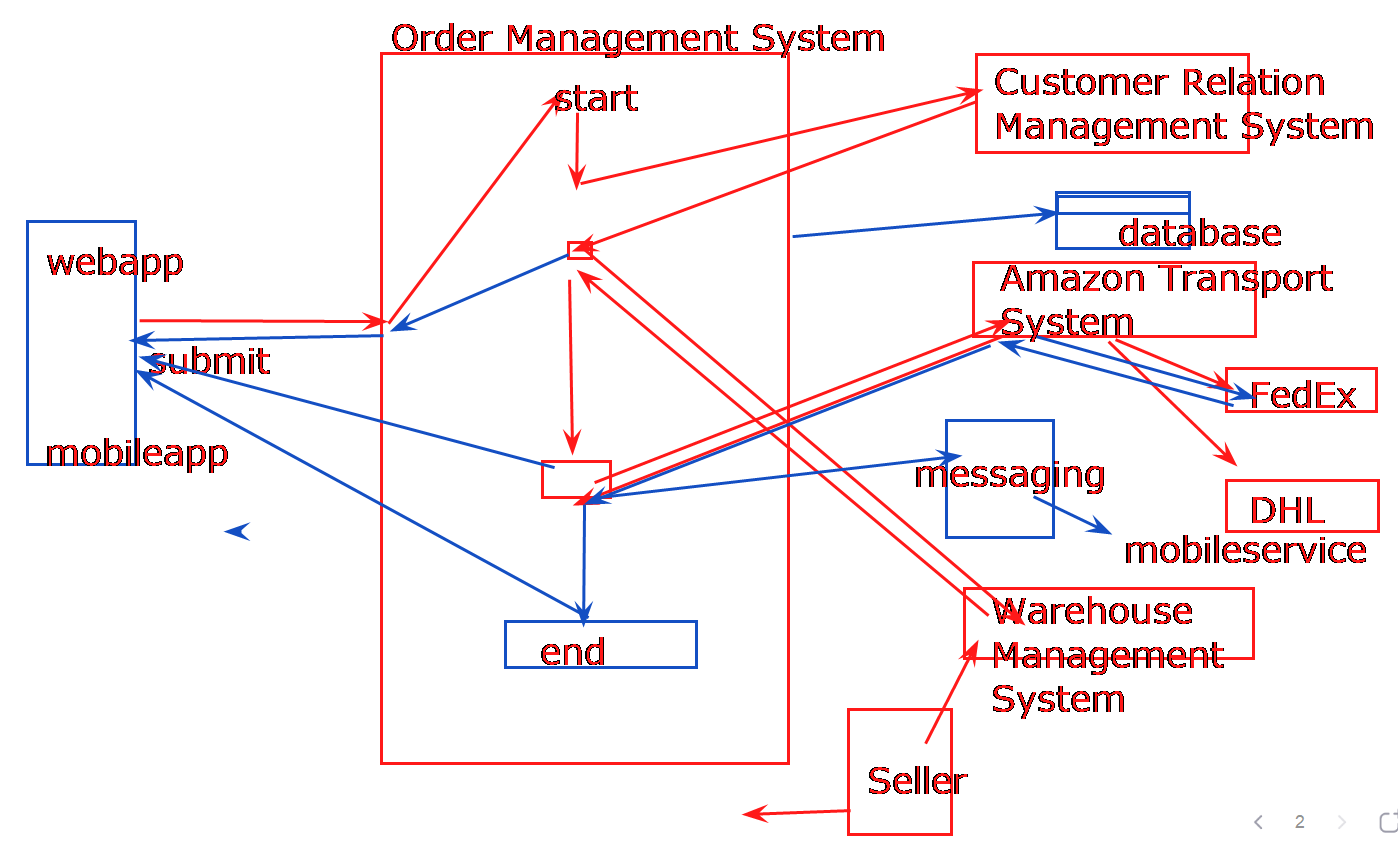
Java EE 7 Specification was the last released by Oracle Corporation.

Eclipse is Managing a new generation of Java EE technology under the brand name

Jakarta EE

Java EE Specification

* Web Application
* Enterprise Application
* Distributed Application
* Other APIs
  + Connector API
  + Messaging API



1. Web Application
   1. Servlet
   2. JSP
2. Enterprise Application
   1. EJB
   2. Web Services
      1. JAX-RS
      2. JAX-WS
3. JPA [ Skip ]
4. **Hibernate** is an implementation of JPA

**Servlet API**

Diagram

Description automatically generated

HTML is used for presentation logic only. [ Java Script & CSS or Bootstrap ]

Before Servlet, we were using CGI ( command gateway interface ) scripting language.

Servlet is an API that provides many interfaces and classes including documentation. Servlet is an interfaces. It is used to develop dynamic web pages as well.

Web Page :

**Static :** HTML

**Dynamic :** Content can be changed as per client request.

Tomcat/Jetty/Undertow Server is web server that can host only Servlet & JSP.

HTTP is the web-protocol.

GenericServlet implements Servlet

HttpServlet extends GenericServlet

Diagram

Description automatically generated

Diagram

Description automatically generated

Http Request Methods

* GET [ default ]
  + Any request submitted through the browser address bar
  + It submits the data by appending it to the target URL as query string.
  + Can be bookmarked

<https://www.google.com/search?q=Microsoft+Azure>

* POST
  + It submits the data internally without exposing it to the address bar.
  + You need to use HTML form or javascript to submit post request.

Servlet

service(request, response)

* + - doGet(request, response) if HTTP method is get type
    - doPost(request, response) if HTTP method is post type.

http://localhost:8080/JOR2-HelloWorldWeb/HelloWorld

Servlet can be used for presentation logic, but you need to embed HTML into Java coding.

**Servlet Life Cycle**

* Servlet is a singleton. Web Container is responsible for Servlet life-cycle management.

Diagram

Description automatically generated

Text, timeline

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

**Revisiting Exception : try with resource statement**

try(Resource Open Statement) // close the resources whenever you get out from try block

}catch(Exception ex){

} // no need to write finally

Resource Statement are statement that uses objects from the classes that implements AutoCloseable or Closeable interface.

package java.lang;

Interface AutoCloseable{

void close();

}

Interface Closeable extends AutoCloseable{

}

--- Java 7 , all the API that deals with I/O or database/network connection were refactored to use AutoCloseable or Closeable.

**Model – View – Controller [ MVC ] Design Pattern**

**Diagram

Description automatically generated**

**Java Server Page: JSP**

JSP is used to create web pages for User Interfaces. JSP is a server-side component and it is a servlet-based component.

JSP page consists of HTML and JSP tags. It is easier to maintain than servlet because we can separate designing & development.

JSP is an extension to Servlet. In fact, If I say it is Servlet, it will not be wrong.

<%

//scriptlet

// java coding

%>

JSP has many implicit objects.

Request , response , session, out

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<%@ page import=*"com.example.pojos.Employee"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Employee Details</title>

</head>

<body>

<h1 align=*"center"*> <u>Employee Details</u> </h1>

<%

Employee e=(Employee) session.getAttribute("emp");

**int** id=(Integer) session.getAttribute("empid");

**if**(e!=**null**) {

out.println("<p><b>Employee Id : </b>"+e.getId());

out.println("<p><b>Name : </b>"+e.getName());

out.println("<p><b>Salary : </b>"+e.getSalary());

out.println("<p><b>Dept Id : </b>"+e.getDeptId());

}**else** {

out.println("<p><b> Employee with Id "+ id+" doesnt exist");

}

%>

</body>

</html>

**JSP life cycle**

1. **First time when a request is made** to JSP page,
   1. JSP page is translated to Java Source File [ internally done by container]. This source file is a servlet file.
   2. Compile the Java Source File
   3. Class Loading loads .class file
   4. Instantiation of .class file [ A servlet object is created ]
   5. Initialization : jspInit() is invoked ---> mapped with servlet init()
   6. Request processing : jspService() ---> mapped with servlet service()
      1. Service method delegates the call to doGet or doPost
   7. Destroy method is invoked : jspDestroy() --- mapped to servlet destroy() // during garbage collection/
2. **Second Request onwards, it repeats step f.**

**Refactoring #1 emp-details.jsp page.**

<body>

<h1 align=*"center"*> <u>Employee Details</u> </h1>

<jsp:useBean id=*"emp"* scope=*"session"* class=*"com.example.pojos.Employee"*></jsp:useBean>

<jsp:useBean id=*"empid"* scope=*"session"* type=*"java.lang.Integer"*></jsp:useBean>

<%

**if**(emp!=**null**) {

out.println("<p><b>Employee Id : </b>"+emp.getId());

out.println("<p><b>Name : </b>"+emp.getName());

out.println("<p><b>Salary : </b>"+emp.getSalary());

out.println("<p><b>Dept Id : </b>"+emp.getDeptId());

}**else** {

out.println("<p><b> Employee with Id "+ empid+" doesnt exist");

}

%>

</body>

**Refactoring #2 emp-details.jsp page.**

<body>

<h1 align=*"center"*> <u>Employee Details</u> </h1>

<jsp:useBean id=*"emp"* scope=*"session"* class=*"com.example.pojos.Employee"*></jsp:useBean>

<jsp:useBean id=*"empid"* scope=*"session"* type=*"java.lang.Integer"*></jsp:useBean>

<p><b>Employee Id : </b> <%=emp.getId()%> </p>

<p><b>Name : </b><%=emp.getName()%> </p>

<p><b>Salary : </b><%=emp.getSalary()%> </p>

<p><b>Dept Id : </b><%=emp.getDeptId()%> </p>

<!-- //comment -->

</body>

**JSP Expression Language: JSP EL**

**${object}**

**EL Operator + - / % == < > <= >= && ! empty**

**div mod eq ne lt gt le ge and or not**

**Refactoring #3 emp-details.jsp page.**

<body>

<h1 align=*"center"*> <u>Employee Details [ EL ]</u> </h1>

<p><b>Employee Id : </b> ${emp.id} </p>

<p><b>Name : </b> ${emp.name} </p>

<p><b>Salary : </b> ${emp.salary} </p>

<p><b>Dept Id : </b>${emp.deptId} </p>

<!-- //comment -->

</body>

emp.id = emp.getId();

emp.deptId = emp.getDeptId();

emp.totalSalary= emp.getTotalSalary();

**Debugging JSP page may be a tedious process.**

**JSTL : Java Standard Tag Library**

**Tag Name Location prefix**

Core Tags <http://java.sun.com/jsp/jstl/core> c

Function Tags <http://java.sun.com/jsp/jstl/functions> fn

Formatting Tags <http://java.sun.com/jsp/jstl/fmt> fmt

XML Tags <http://java.sun.com/jsp/jstl/xml> x

SQL Tags <http://java.sun.com/jsp/jstl/sql> sql

Java EE 7 onwards = replace java.sun.com as jcp.org

Core Tags <http://java.sun.com/jsp/jstl/core> c

**<c:out>**

**<c:import>**

**<c:catch>**

**<c:if>**

**<c:forEach>**

**<c:choose>**

**<c:when test=””>**

**</c:when>**

**<c:otherwise>**

**</c:otherwise>**

**</c:choose>**

**Enterprise Java Bean: EJB**

* EJB’s lifecycles are managed by the EJB container [ instantiation to destruction ]. You don’t need to use a new keyword to create the object.
* EJB containers maintain all EJB instances into a common pool. i.e EJB pool.
* Client can obtain reference of EJB instances by requesting to EJB container.
* **This client can be local client [other apps in same JVM ] or remote client [ app running on different JVM ]**
* **Must use interface**

1. Session Bean
   1. Stateless
   2. Stateful Local or Remote // distributed programming
   3. Singleton
2. Message-Driven Bean // Java Messaging Service // JMS

EJB lite:

* EJB’s lifecycles are managed by the EJB container [ instantiation to destruction ]. You don’t need to use a new keyword to create the object.
* EJB containers maintain all EJB instances into a common pool. i.e EJB pool.
* **Client to access it , can be from the same application.**
* **Doesn’t have interface implementation and this is why “No Interface View”**
* **EJB support Container Managed Transaction**

1. Stateless

Diagram

Description automatically generated

1. Stateful

Diagram

Description automatically generated

@Local

**public** **interface** EmployeeDao {

**public** List<Employee> findAll() **throws** EmployeeException;

**public** String register(Employee e) **throws** EmployeeException;

**public** Employee findById(**int** id) **throws** EmployeeException;

//public int remove(int id) throws EmployeeException();

//public String update(Employee e) throws EmployeeException;

}

**@Stateless**

**public** **class** EmployeeDaoImpl **implements** EmployeeDao{

**public** Connection getConnection() **throws** SQLException{

String jdbcURL="jdbc:mysql://localhost:3306/javaoursoul2";

Connection con=DriverManager.*getConnection*(jdbcURL,"root","admin#123");

**return** con;

}

// methods

}

**public** **class** EmployeeServlet **extends** HttpServlet{

@EJB

**private** EmployeeDao dao;

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

/\*

\*

\*/

**public** **void** doGet(HttpServletRequest req, HttpServletResponse res)**throws** ServletException, IOException {

HttpSession s=req.getSession();

**Deleted the EmployeeFactory**

**Deleted the attached Library**

**Assignment**

Diagram

Description automatically generated

**Web Services**

Web Service is an appropriate medium to propagate communication between the client and server.

Web Service is something that can expose business functionalities and let other applications [ built on the same or different technologies] consume those services. Interoperability is the main agenda behind web services.

Web Service can use HTTP as a transport layer protocol to exchange data. A remote application can easily invoke methods running on the server.

1. XML-Based Web Service

* It works with multiple transport layer protocol. HTTP, SMTP, POP3. UDP
* XML is used for data exchange.

SOAP

XML

App2/Java

Tech

xml-Java

App1

tech

SOAP

XML

JAX-B API

XML - XSD

SOAP

WSDL = Web Service Description Language

JAX-WS API

<http://localhost:8080/hrm/webservices/EmployeeJAX?wsdl>

Client Testing Tool = SOAP

1. RESTful Service [ REST Service ] : it uses the only HTTP/s protocol.

JAXB-API

JSON data [ key:value]

JAX-RS

Representational State

GET [http://localhost:8080/hrm/emp-rest/{name}](http://localhost:8080/hrm/emp-rest/%7bname%7d)

GET http://localhost:8080/hrm/emp-rest/find/{id}

Client Testing Tool

1. SOAP REST
2. POSTMAN
3. ADVANCE REST CLIENT

**ASSIGNMENT ON HTML/JAVASCRIPT & REST SERVICE**

**Graphical user interface, application, Word

Description automatically generated**

As per RDBMS

Table = Relation

Record = Tuple

**Java Persistence API**

* It is a part of Java EE specification.
* It deals with database operation.
* It is a superset of JDBC.
* It supports Object Relational Mapping [ ORM ]
* JPA is using classes [ by mapping them to tables] to perform CRUD operation on database.
* The class is known as Entity Class.

ORM software is the implementation of JPA

1. Oracle TopLink
2. EclipseLink
3. OpenJPA
4. Hibernate

**Entity Mapping.**

1. **Annotation**
2. **Descriptor File \*\*\*hbm.xml**

save(Object o) // inserting records persist(Object o) // JPA

saveOrUpdate(Object o) // insert or update merge(tObject o) // JPA

T get(Class<T> type, Serializable id) T find(Serializable id)

T load(Class<T> type, Serializable id)

// research to find the difference between get & load.

remove(Object o) remove(Object o)

**Entity Relationship**

@OneToOne

@OneToMany

@ManyToOne

@ManyToMany

* Table that has FK constraint, owns the relationship.
* The other side of the table is known as Inverse Side.

The same concept is applicable on entities as well.

* If entity class name is same as table name , no need to use @Table

**Employee**

id

name

salary

Department dept

**Department**

id

name

location

Many One

Owning Inverse

**Uni-directional** is the default direction in that entities are linked. This means inverse side [ **Department]** details will also be fetched if you try to find a record from owning **side [ Employee ]**

A bi-Directional approach can also be used to fetch the details of each entity by each other. It is going to use existing relationship constraints. To be in Bi-Directional , entities must be in Uni-Directional.

**Department**

id

name

location

List<Employee> empList;

**Employee**

id

name

salary

Department dept

Hibernate Query language / Java Persistence Query Language [ JPQL ]

NAMED QUERY

select EMP\_ID, NAME, SALARY, SALARY \* 12 "Annual Salary" from Employee;

select e.id, e.name, e.salary, e.salary \* 12 as “Annual Salary” from Employee e;

Criteria API

* It is using set of classes & interfaces to form queries. Therefore it is compiled queries. -
* @Override
* **public** List findAllUsinCQ() **throws** HrException{
* List<Employee> dList=**null**;
* **try**(SessionFactory f=getConnection();
* Session s=f.openSession())
* {
* CriteriaBuilder cb=s.getCriteriaBuilder();
* CriteriaQuery<Employee> cr=cb.createQuery(Employee.**class**);
* Root<Employee> root=cr.from(Employee.**class**);
* cr.select(root).where(cb.gt(root.get("salary"), 3000));
* Criteria cr=s.~~createCriteria~~(Employee.**class**);
* dList = cr.list();
* }**catch** (SQLException e) {
* **throw** **new** HrException("Error",e);
* }
* **return** dList;
* }

// select e from Employee e where e.salary>3000;

**Java I/O Operation**

**InputStream**

Reading content from the source [ file/device/socket]

**OutputStream**

Writing content to the destination [ file/device/socket]

**System.out out is an object of PrintStream**

**System.in in is an object of InputStream read from console**

**Reader [ only Character Stream ]**

Reading content from the source [ file/device/socket]

**Writer [ only Character Stream ]**

Writing content to the destination [ file/device/socket]

Persistence of Object is known as Serialization.

class Employee implements Serializable {

}

Employee.class

Unique serial version id.

Employee e=new Employee(1,”Moto”,2010.00,101);

Employee e=?