

JSF, JPA, and EJB



Will Provost
Ken Kousen

Version 5.0

JSF+EJB. JSF, JPA, and EJB

Version 5.0

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of the author.

A publication of Capstone Courseware, LLC.

877-227-2477

www.capstonecourseware.com

© 2007 Will Provost. All rights reserved.

Published in the United States.

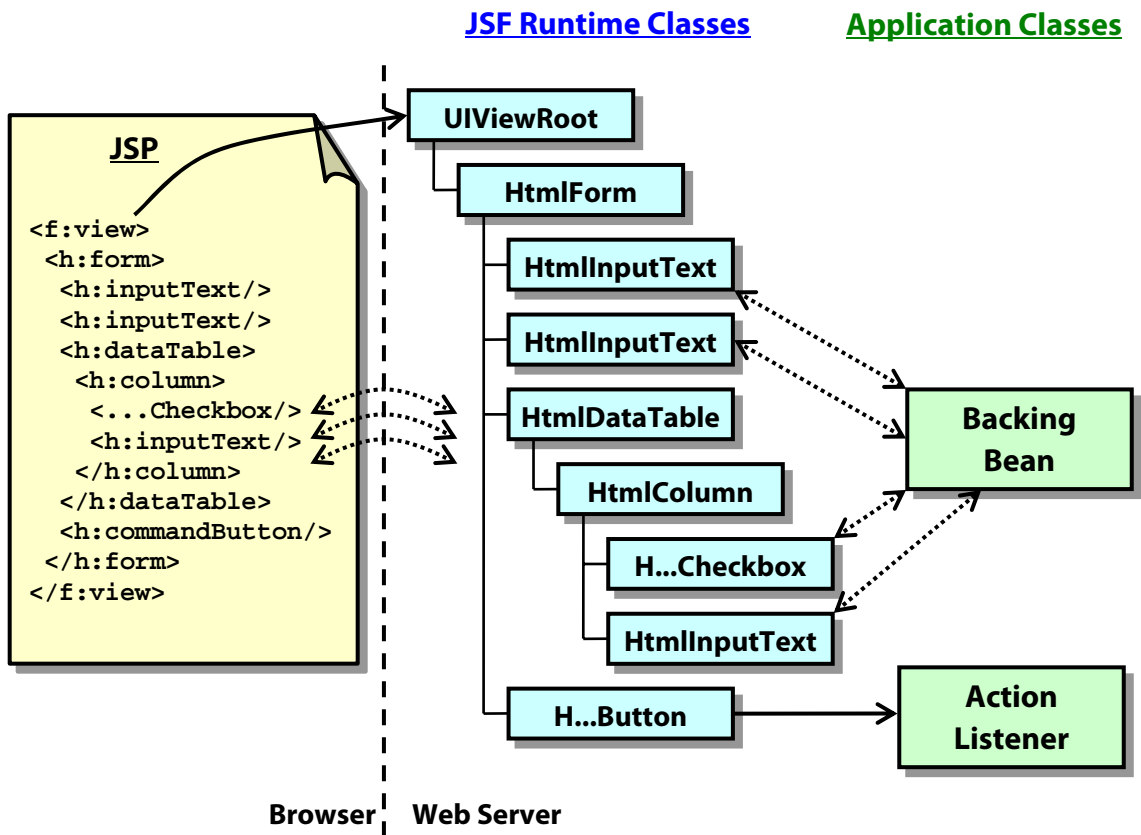
This book is printed on 100% recycled paper.

Introduction

- **This presentation illustrates the use of three Java-EE-5.0 APIs in concert:**
 - **JavaServer Faces, or JSF** – version 1.2
 - **Enterprise JavaBeans, or EJB** – version 3.0
 - **The Java Persistence API, or the JPA**
- **Though it may be interesting on its own (to those with experience in one or more of the above standards), we've built it primarily to compliment Capstone courses on these subjects.**
 - It will be most effective as a sort of encore presentation at the end of a class that uses one or both of those courses.
 - It uses a build-and-deploy infrastructure almost identical to that found in the EJB course, and requires the same application server.
- **We discuss each of these three APIs very briefly before addressing the issues involved in integrating them.**

JavaServer Faces

- **JavaServer Faces** represents a fresh approach to the problem of modeling and implementing web user interfaces.



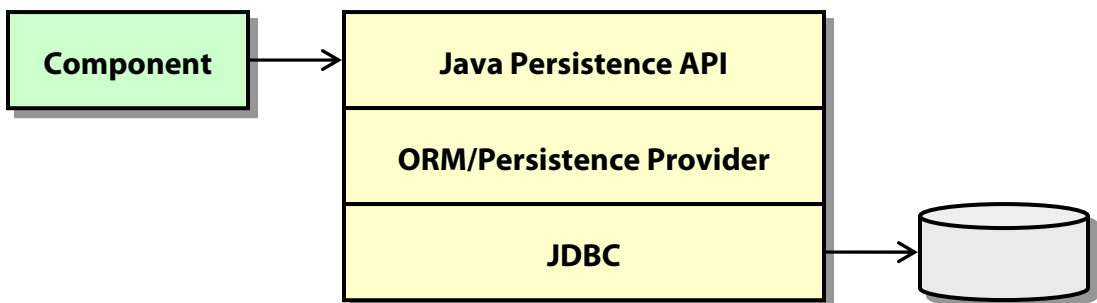
- JSP (or other technology) defines a **component tree** that has as much to do with AWT/JFC GUI-building as it does with traditional request/response and web-MVC frameworks.
- Components rely on **backing beans** for presentation state and business logic (state and behavior).

Enterprise JavaBeans

- **Enterprise JavaBeans** defines a standard for scalable business and data-access objects that relies on a dedicated **container** to:
 - Manage component **lifecycle**
 - **Mediate** conversations between clients and contained objects – and between multiple contained objects – so as to implement various **enterprise features** on behalf of the objects
- EJBs come in three flavors:
 - **Session beans** are front-line business objects, and these are further classified as **stateful** or **stateless** with regard to the client conversations or use cases that they implement.
 - **Entity beans** – or, in the newer parlance, simply **entities** – are **persistent state objects**; they act as DAOs but can also exist in transient and detached states and so are usable in business and even presentation tiers.
 - **Message-driven beans** exist specifically to handle incoming messages by asynchronous systems such as the **Java Message Service (JMS)**.
- EJB relies heavily on **metadata** to inform the container as to how a given bean should be hosted and handled; this metadata may be provided in either or both of two forms:
 - **Annotations** in the Java source code
 - **XML deployment descriptors** external to the Java class(es)

The Java Persistence API

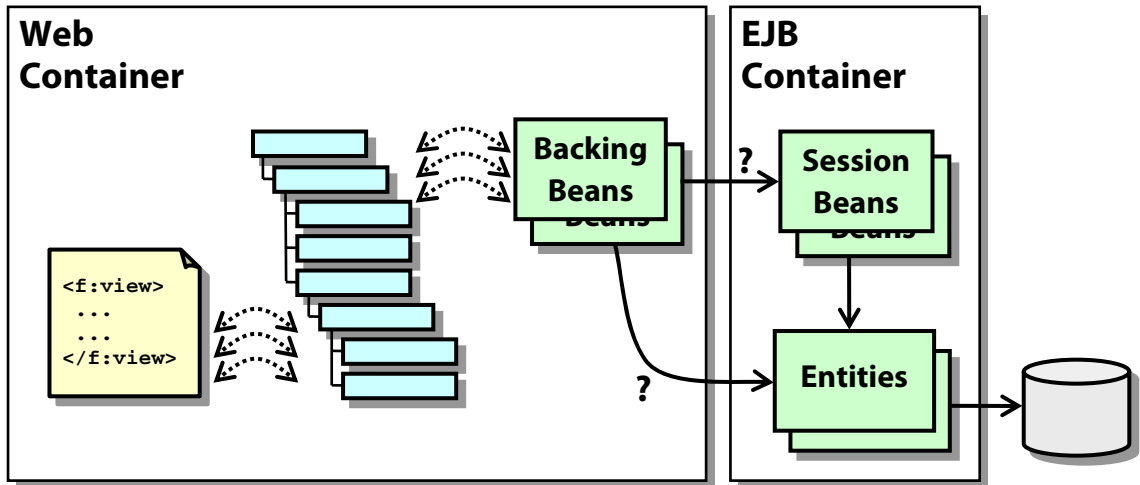
- Part of the EJB 3.0 specification is known as the **Java Persistence API, or JPA**.
 - The EJB 3.0 specification is split into three separate documents, one of which defines the JPA.
 - In its next iteration, the JPA will become a stand-alone specification for persistence in both Java SE and Java EE applications.
 - It is already usable by Java SE applications, using the Java EE JARs but not requiring the use of an EJB container.
- The JPA incorporates into Java EE a full **object/relational mapping (ORM)** layer.



- Persistence providers will be tools such as **TopLink** or **Hibernate**.

The Big Picture

- So a multi-tier, Java-EE application might take advantage of all of this technology:



- JSF helps us meet the challenges of the **presentation tier**.
- EJBs, and specifically session beans, may be our **business tier**.
- JPA entities make up a **persistence tier**.
- Not shown in the diagram but part of the upcoming examples are **JMS** and **message-driven beans**, integrating with the session beans to connect to the same business and persistence logic.
- The rest of this presentation discusses the challenges of combining this technology – specifically:
 - Using **session beans** from a JSF web tier, such that the business tier of the application is hosted by the EJB container, as shown
 - Using **JPA entities** directly from JSF – in this case the business logic is implemented in POJOs and runs in the web container

Using Annotations in JSF

- Most of the connectivity between JSF and either EJB or JPA will be implemented by expressing the appropriate metadata to the web container.
- Essentially, we're talking about **dependency injection**: the ability of a container to connect a dependent component to a collaborating component based on metadata.
 - A Java-EE-5.0 web container is aware of the full set of **annotations** defined for use with EJB, JPA, and other APIs.
 - **XML** metadata is also possible – and even preferred in some cases, as we discuss in our EJB coursebook.
- Only certain **managed objects** can take advantage of dependency injection.
 - This is true for the web as well as the EJB container, though with the latter it's less obvious since most hosted components are EJBs!
- Specifically, in JSF, one may define injectable dependencies in the following places:
 - **Servlets**
 - **Servlet filters**
 - Web context, request, and session **event listeners**
 - Custom JSP **tag handlers**
 - JSF **managed beans**
- It's the last of these that will be of the greatest interest to most JSF application developers.

EJB Annotations

- A JSF managed bean can make calls on an EJB most easily by:
 - **Defining a field** of that EJB's type
 - **Annotating** that field, or the corresponding mutator method, as a reference to an **@EJB**

```
@EJB private MyBeanType myBean;
```

- The usual range of dependency-injection rules and options obtain for a JSF managed bean.
 - **Injection by type** will work when only one implementation (class) of the given bean type (interface) is found by the container.
 - **Injection by name** will be necessary otherwise, and here the syntax gets a bit more involved, using either a **beanName** attribute on the annotation ...

```
@EJB (beanName="Fred") private MyBeanType myBean;
```

- ... or a **name** attribute that connects to XML metadata:

```
@EJB (name="mine") private MyBeanType myBean;
```

```
<ejb-ref>  
  <ejb-ref-name>mine</ejb-ref-name>  
  <ejb-ref-type>Session</ejb-ref-type>  
  <remote>BeanType</remote>  
  <ejb-link>Fred</ejb-link>  
</ejb-ref>
```

- Both local and remote references are possible.

Using Stateful Session Beans

- The above technique is actually appropriate only for stateless session beans, because a client needs only one reference to a stateless bean, even if it will make many calls to that bean.
- Stateful beans won't work this way.
 - Whatever injection method we choose, we will still have **one reference to one bean**.
 - How can we wedge all our different client conversations – our own sessions – into that one object? It will surely become confused!
- For a stateful bean, the general technique is to use the servlet context to look up the stateful bean each time you need it:

```
((ServletContext)
FacesContext.getCurrentInstance ()
    .getExternalContext ().getContext ())
    .lookup ("thatStatefulBean");
```

- This requires an `<ejb-local-ref>` or `<ejb-ref>` just as used in ordinary injection by name.
- This will trigger creation of a distinct bean for your session; typically this reference would be saved at session scope in the web application as well.
- In our upcoming example, we use a stateless bean that acts as a factory for stateful beans.
 - This simplifies life for the web application, as it can use a simple **@EJB** annotation and then call the factory method as needed.

Tools and Environment

- For our practical exercises with EJB, we'll rely on the Java EE 5.0 SDK from Sun Microsystems.
- The Java EE 5.0 SDK must be installed on your system, typically in **c:\Sun\SDK**.
- Other tools are deployed with the example code, under **c:/Capstone/Tools**:
 - The **JSTL** libraries are in **JSTL1.1**.
 - **MySQL** and its JDBC driver are in **MySQL5.0**.
- Be sure that the following environmental settings are correct:
 - The executable path includes the **bin** and **jdk/bin** directories under the SDK installation root.
 - The environment variable **JAVA_HOME** is set to the **jdk** directory.
 - The environment variable **JAVA_EE_HOME** is set to the SDK installation root itself.
 - An environment variable **CC_MODULE** is set to **c:\Capstone\JSF+EJB**.
- Copy the file **mysql-connector-java-5.0.4-bin.jar** from **c:/Capstone/Tools/MySQL5.0/lib** to **%JAVA_EE_HOME%/lib**.
 - This will assure that the Java EE server can find the MySQL JDBC driver on the server's own classpath.

Starting the MySQL and Java EE Servers

- To start the MySQL RDBMS, simply run the following command from **c:/Capstone/Tools/MySQL5.0/bin**:

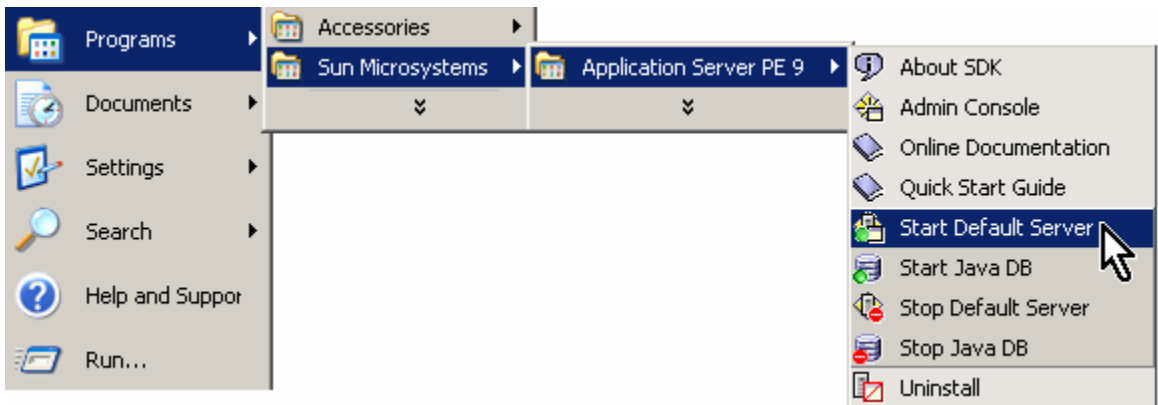
`mysqld`

- Note that the console you use to run this command will appear to hang; this is the MySQL daemon process listening for connections at its default port 3306.

- When you want to shut the server down again, run this prepared script from the same directory:

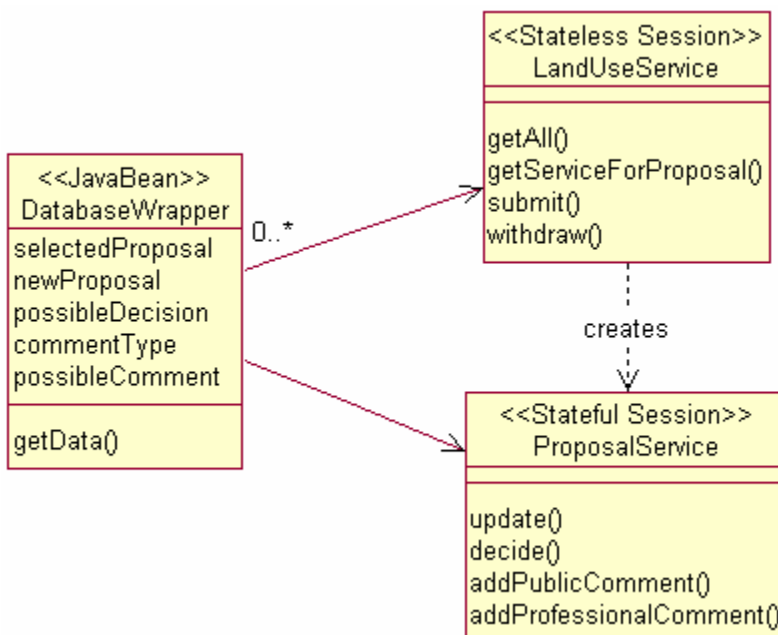
`shutdown`

- Start the application server from the Windows Start menu, as shown below ...



- You'll see confirmation that the server is running and ready for requests, and a prompt to hit any key – do so.
- You can stop the server from the menu – but leave it running for now.

- In `c:/Capstone/JSF+EJB/Examples/LandUse`, there is a version of the **LandUse** application in which a JSF GUI uses session and entity beans by way of dependency injection to a managed bean.
- In this presentation we will focus on the bridge between JSF and EJB, and see a summary of their interactions.
- We leave details of the rest of the application for further study.
 - The JSF code is almost identical to that found in the JSF course.
 - The same can be said of the EJBs and relational database as found in the EJB course.
- The bridge between the web tier and EJB tiers of this application is found in the dependencies from a single managed bean to a stateless session bean and a stateful session bean:



- First, consider the JSF configuration file, found in **docroot/WEB-INF/faces-config.xml**:

```
<managed-bean>  
  <managed-bean-name>DB</managed-bean-name>  
  <managed-bean-class>  
    gov.usda.usfs.landuse.web.DatabaseWrapper  
  </managed-bean-class>  
  <managed-bean-scope>session</managed-bean-scope>  
</managed-bean>
```

- The managed bean **DB** is the gateway to the business logic and persistent data for the application.
- In the JSF-only version of this application, this bean manages the loading and saving of a file of serialized objects.
- Here, it will connect to EJBs.

- In `src/gov/usda/usfs/landuse/web/DatabaseWrapper.java`, see the simple injection-by-type use of the `@EJB` annotation:

```
@EJB private LandUseService landUseService;
```

- This session-scope bean uses the stateless bean to get “global” information such as the list of proposals in the database.

```
private String loadData ()
{
    List<Proposal> proposals =
        landUseService.getAll ();
    ...
}
```

- And how are we assured that this method is invoked when the bean is created?
- The Java-EE annotation `@PostConstruct` assures that the `init` method will be called; this calls `loadData`.

```
@PostConstruct
public void init ()
{
    loadData ();
}
```

- This method is akin to a constructor, but will be called only after injected dependencies have been set.

- Then, when the user begins working with a specific proposal, the managed bean derives a reference to the stateful session bean – see the **edit** method:

```
public String edit ()
{
    proposalService = null;
    for (int i = 0; i < data.size (); ++i)
        if (data.get (i).isSelected ())
            {
                proposalService =
                    landUseService.getServiceForProposal
                        (data.get (i).getProposal ().getID ());
                selectedProposal = proposalService.get ();
                return "detail";
            }
    ...
}
```

- When the **Done** or **Decide** button is clicked, the managed bean throws away its reference to the stateful bean with a call to the helper method **clearDetailState**, and returns to the summary page.

```
private void clearDetailState ()
{
    proposalService = null;
    possibleDecision = new Decision ();
    possibleComment = new Comment ();
}
```

- Set up, build, and deploy the application as follows:
 1. If you've deployed any of the LandUse steps from the EJB course, clean them up first:
`asant clean-DB`
 2. Create the underlying database of land-use proposals, and establish JDBC resources for the application server:
`asant init-DB`
 3. Build the application and prime the database using a simple Java SE console application:
`asant build`
`run PrimeWithData`
 4. Deploy the EAR file that holds the web and EJB tiers of the application:
`asant deploy`
- The full console output from this series of commands (starting with **asant init-DB**) can be found in **ConsoleLog.txt**.

- Open the following URL in your web browser:

`http://localhost:8080/LandUse`

Select	Applicant	Parcel	Proposal
<input type="checkbox"/>	Cranmore Paper	White Mountains NF	Selective logging
<input type="checkbox"/>	Ski USA	Green Mountain NF	Alpine park
<input type="checkbox"/>	Mines-R-Us	Tonto NF	Silver mining

Add **Edit** **Remove**

- To produce this page, the JSF component tree derives data from the managed bean – which in turn calls **getAll** on the stateless **LandUseServiceImpl**.

- Choose one of the proposals, and click **Edit**:

Proposal		Public Comments	
Parcel:	<input type="text" value="Tonto NF"/>	Contributor	Recommendation
Applicant:	<input type="text" value="Mines-R-Us"/>	Ernest DeLallo	REJECT
Application date:	<input type="text" value="9/17/06"/>	Electronics Association	ACCEPT
Proposed use:	<input type="text" value="Silver mining"/>	Sierra Club	REJECT
Proposed start date:	<input type="text" value="2/1/07"/>	<input type="button" value="Add a Public Comment"/>	
Proposed end date:	<input type="text" value="7/30/08"/>	Staff Comments	
<input type="button" value="Done"/>		Contributor	Recommendation
		Mark Fiore	NEUTRAL
		Sanjay Nagpur	NEUTRAL
		<input type="button" value="Add a Professional Comment"/>	
Decision			
Decision:	Accepted		
Date:	11/3/06		
Authority:	Jan Landry		

- This is where the stateful bean is brought into play, and from this point forward the (session-scope) managed bean will use the same stateful bean for its operations: adding comments, changing basic data, or rendering decision.

Impact on the JSF Application

- A few significant changes were required on the JSF side to work with EJBs instead of a private file of serialized objects.
 - In fact most of these have more to do with moving to a multi-tier architecture than with EJB specifically; the EJB impacts are really no more and no less than those mentioned in the example.
- State management in the web tier is a little different.
 - The JSF-only application tracks the selected proposal for a user session as an **ordinal value** into a **List**.
 - This is fine when (a) we have the list to ourselves so it's not volatile, and (b) ordinal values are meaningful!
 - For a database, ordinal values are usually unreliable, and instead it's an ID or **primary key** that matters.
 - So the managed bean tracks its **selectedProposal** as an object reference, along with the stateful-bean reference **proposalService**.
- One simple but critical change that is specific to Java EE: the **web.xml** must declare that the application uses Servlets 2.5.
 - JSF can be configured for a version-2.4 application, but **dependency injection** is a 2.5 feature.
 - Change the version back to 2.4, and watch the connection to the stateless session bean quietly fail to materialize.

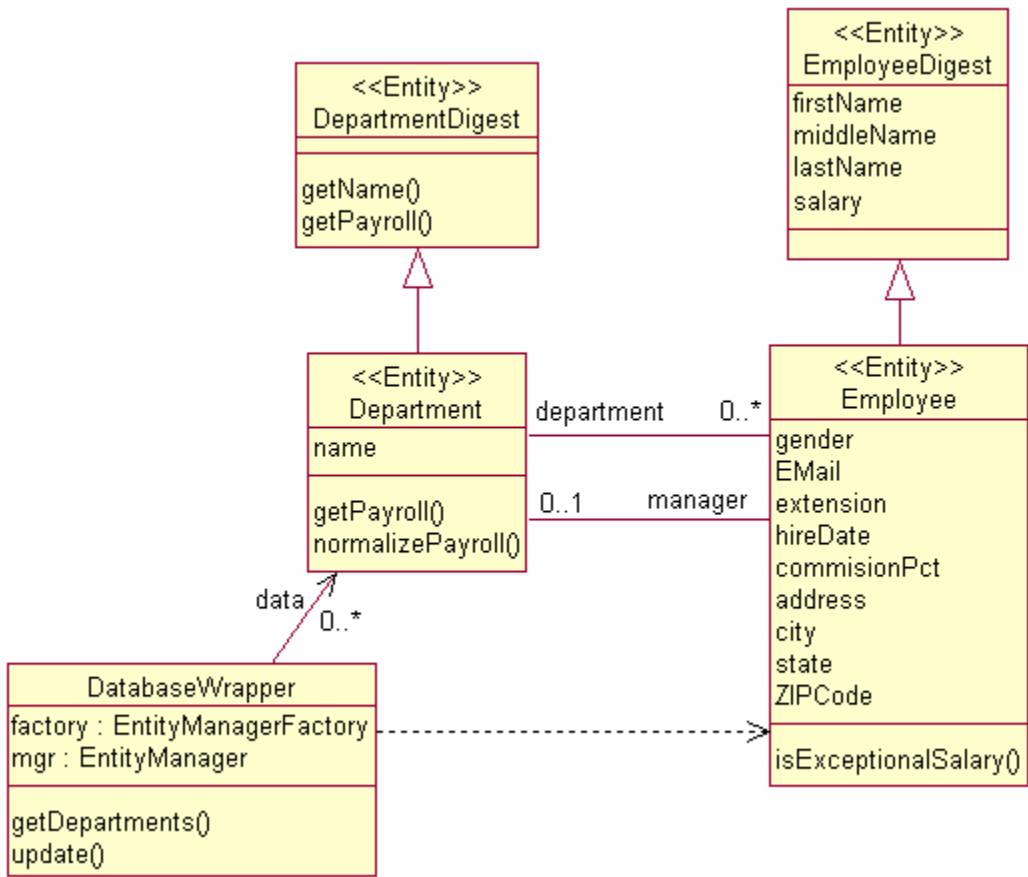
JPA Annotations

- With a full Java-EE library on hand – but even without an EJB container – Java code can take direct advantage of the JPA.
 - Thus EJBs are not essential to JPA persistence, but rather one means of implementing that persistence.
- Implementing a persistent type using JPA involves a wide range of annotations beginning with **@Entity** and also including definitions of table and column mappings, association cardinality, and eager vs. lazy fetching strategies.
 - This is beyond our scope.
- To use a JPA entity class to carry out database operations is a simpler matter:
 - The entry point is an object of type **EntityManager**.
 - It represents an abstraction known as the **persistence context**.
 - A persistence context is an instance of a **persistence unit**.
- A **persistence.xml** file will define one or more persistence units.
 - For EJB JARs this is found in the JAR's **META-INF** directory.
 - For standalone applications it is found in a **META-INF** directory anywhere off the class path.
 - For web applications it goes in the **WEB-INF** directory.
- The JPA implementation will find this definition and seek out a context when the application attempts to create an **EntityManager**.

Using JPA from Web Components

- When EJBs use JPA entities, they typically adopt a fully declarative strategy by which the EJB container manages transaction boundaries and persistence contexts.
 - Transaction and context will live for a single method call.
 - The bean can declare different requirements, such as no transactions or a context that lives longer than a method call.
- For web components, things are different:
 - Web containers will not manage transactions.
 - There is no analog to the per-method scope for persistence contexts; and we often need longer durations such as for the life of a user session.
- Therefore a JSF application adopts a different strategy – partly declarative and using dependency injection, but governing transaction and persistence-context boundaries manually.
 - Declare an injectable **EntityManagerFactory** using the **@PersistenceUnit** annotation.
 - Create and close **EntityManagers** as needed.
 - A sound approach uses lifecycle methods on a managed bean, identified by **@PostConstruct** and **@PreDestroy** annotations.
 - Control transaction boundaries using the **EntityTransaction** interface, an instance of which is exposed by **EntityManager**.

- In `c:/Capstone/JSF+EJB/Examples/HR` there is a simple JSF application that shows department and employee records from a human-resources database, and allows limited data editing.
- A single session-scope bean of type **DatabaseWrapper** invokes persistence methods to manipulate JPA entities for **Department** and **Employee** records:



- See `src/cc/hr/web/DatabaseWrapper.java`: this class coordinates all JPA activities for the application.
 - It defines an injectable **EntityManagerFactory** and also an **EntityManager** that it will manage by itself:

```
@PersistenceUnit
private EntityManagerFactory factory;

private EntityManager mgr;
```

- It controls the creation and closing of the **EntityManager** in its lifecycle methods; so we know that the persistence context will be in force for the user session:

```
@PostConstruct
public void createEntityManager()
{
    mgr = factory.createEntityManager();
}
```

```
@PreDestroy
public void closeEntityManager()
{
    mgr.close();
}
```

- It uses the **EntityManager** for queries and updates, wrapping the updates in an **EntityTransaction**:

```
public List<Department> getDepartments ()
{
    data = (List<Department>) mgr.createQuery
        ("select x from Department x").getResultList ();
    return data;
}

public void update ()
{
    EntityTransaction tx = mgr.getTransaction ();
    for (Department department : data)
        try
        {
            tx.begin ();
            mgr.merge (department);
            tx.commit ();
        }
        catch (Exception ex)
        {
            System.out.println ("Exception during " +
                "transaction -- rolling back.");
            ex.printStackTrace ();
            try { tx.rollback (); }
            catch (Exception ex2) {}
        }
}
```

- See `docroot/WEB-INF/classes/META-INF/persistence.xml`, which defines the JPA persistence unit for the application:

```
<persistence-unit name="Earthlings"
  transaction-type="RESOURCE_LOCAL" >
  <provider>
    oracle.toplink.essentials.PersistenceProvider
  </provider>
  <jta-data-source>
    jdbc/Earthlings
  </jta-data-source>
  <properties>
    <property name="toplink.logging.level"
      value="INFO" />
  </properties>
</persistence-unit>
```

- Set up, build, and test the application as follows. (MySQL and the Java EE application server should both still be running.)

1. Build the database:

```
asant create-DB
```

2. Build and deploy the application:

```
asant
```

3. Test at the following URL:
<http://localhost:8080/HR>

Human Resources: Departments		
Department	Location	Payroll
Research	1 Grover's Mill Circle, Grover's Mill, NJ	\$1,035,000.00
Administration	791 Massachusetts Avenue, Boston, MA	\$1,515,000.00
Software Development	1045 University City Boulevard, Charlotte, NC	\$874,000.00
Hardware Development	1045 University City Boulevard, Charlotte, NC	\$792,000.00
Test And Integration	1045 University City Boulevard, Charlotte, NC	\$533,000.00
Sales	3 Peachtree Plaza, Atlanta, GA	\$1,130,000.00
HR	791 Massachusetts Avenue, Boston, MA	\$244,000.00
Facilities	791 Massachusetts Avenue, Boston, MA	\$178,000.00
Operations	3 Peachtree Plaza, Atlanta, GA	\$621,000.00

Human Resources: Employees		
Department	Employee	Job Salary
Research	Amdell, John	<input type="text" value="\$15,000.00"/>
	Angel, John	<input type="text" value="\$30,000.00"/>

4. Try changing one or more employee salaries, and see that the department payroll accurately reflects the updated salaries. This is persistent now, so new sessions, redeploys of the application, etc., all will see the durable data from the MySQL database.