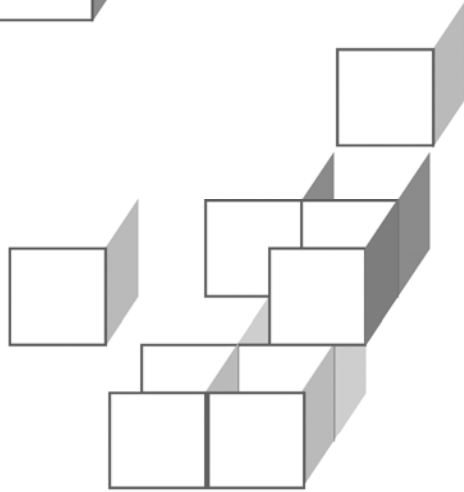




# CHAPTER 7

## DEPENDENCY INJECTION



## OBJECTIVES

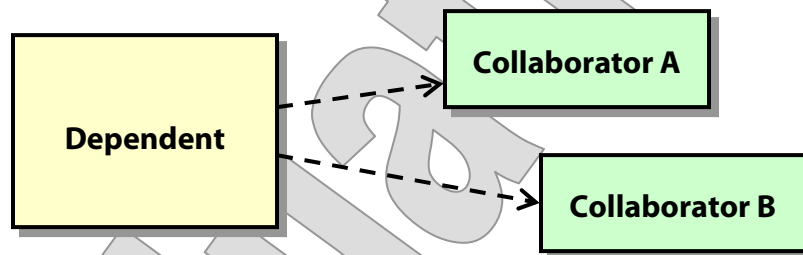
*After completing “Dependency Injection,” you will be able to:*

- Understand the need for dependency injection in a container.
- Explain the inner workings of dependency injection in EJB containers, and recognize the impact of relying on different injection methods:
  - Injection by type vs. injection by name
  - Annotations vs. deployment descriptors
- Use dependency annotations to effect injection by type.
- Use the component environment to effect injection by name.
- Understand the need to use context lookups when interacting with stateful session beans.
- Use JNDI lookups to find remote beans.
- Use appropriate strategies to make injectable dependencies such as EJB references useful to all components in web applications.

## Interdependent Systems

---

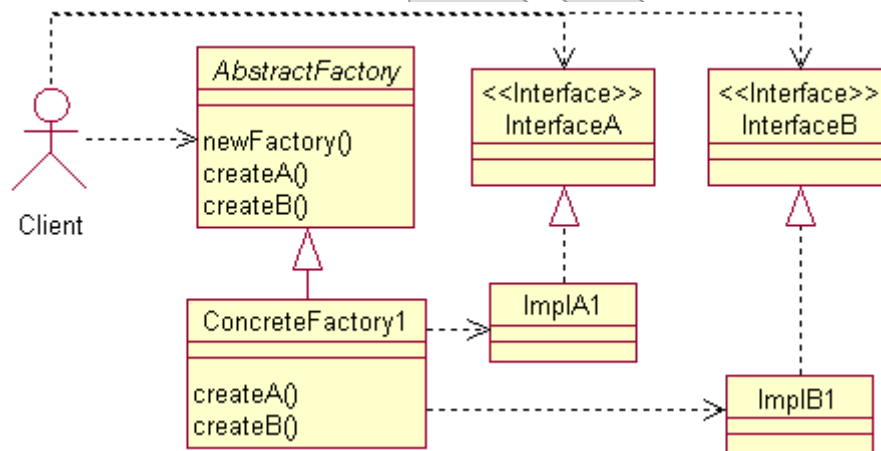
- Complex systems, by definition, function by interdependency; but at the same time dependencies bring maintenance challenges.
- A component (call it the **dependent**) that relies on another (call it the **collaborator**) for some functionality will best be defined to look to the interface of that other component.



- But as a practical matter, someone must choose a real implementation at runtime, and it's usually detrimental for the implementation to be chosen by the dependent component.
- This is **dependency on implementation type** and it makes the system less flexible, because alternate or replacement implementations might be referenced in many places.
- Also, a **dependent must find its collaborator somehow**.
  - In a single VM, this is a matter of knowing which **instance** of an object should be referenced.
  - In distributed systems, we're now talking about network-ready **names** for objects that can be published and discovered.
  - A **dependency on service location** has the same unfortunate effect on maintainability as a dependency on type: multiple points of maintenance that must be updated if the location changes.

## The Factory Patterns

- In Java, one object often creates another object explicitly, as in `HelperClass helper = new HelperClass ("module", 2);`
- Several closely-related design patterns define means of isolating the capability to create a certain object or family of objects – shown below is the **Abstract Factory** UML:

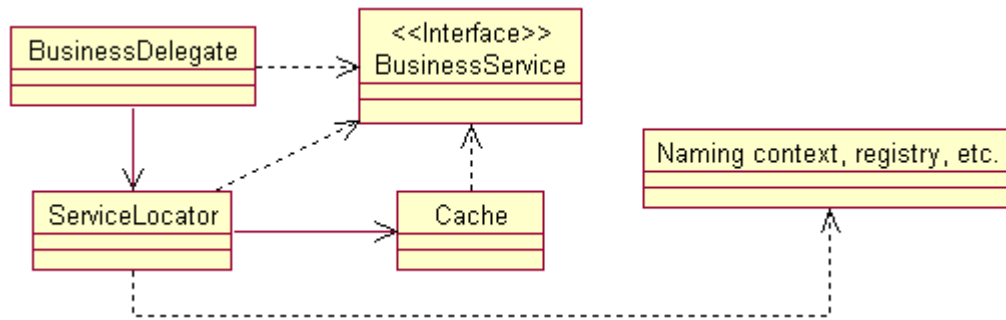


- Factories are the basic tools with which developers can get free of dependencies on implementation type.
- The key to the technique is that the body of code that is responsible for selecting the implementation type is neither the dependent nor the collaborator, but a third actor entirely.
  - The **factory method** pattern calls for a distinct method on one collaborator as a means of creating a different one – for instance a JDBC connection creating a statement and a statement creating a result set when running a query.

## The Service Locator Pattern

---

- A Java EE design pattern, **Service Locator**, identifies a separate component with the responsibility to find required services for a dependent component:

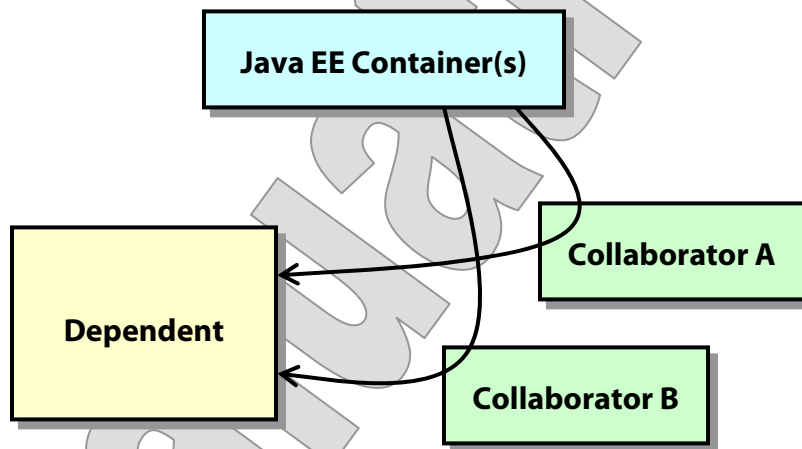


- Service locators can do more than just locate services, as the diagram above suggests.
  - They can **cache** service locations.
  - They can **retry** and **restore** connections as needed.
  - They can **authenticate** the caller or the service.
- Service locators are a classic means of managing dependencies on name or location.

## Dependency Injection

---

- EJB 3.0 – and Java EE 5.0 in general – defines the new feature of **dependency injection**:
  - A dependent declares its dependencies using Java annotations.
  - A Java-EE container resolves those dependencies by some heuristic involving class names, explicit bean names, and network locations.



- This provides a means of managing both types of dependency: dependency on implementation type and on name/location.

## Injection by Magic

---

- Dependency injection at its simplest is a joy to use and may seem like outright magic:

- A bean class is declared `@Stateless`:

```
@Stateless
public class MyClass
    implements MyInterface
```

- ... and a client declares that it needs a bean of that type:

```
@EJB private MyClass service;
```

- And, presto! they start talking.

- There's more to the story, though, and for many common connectivity scenarios the actual usage is more complicated – though still well worth the effort.
- We'll start with the dead-simple usage as above, and work our way up and out to some more complicated cases.

## Finding the RecordsService

**EXAMPLE**

- In **Examples/Wholesale/Step1** we see this “injection by magic” approach that we’ve been using for most of our EJB references so far.
  - An initialization servlet **cc.sales.web.InitServlet** publishes the injected reference to the **Records** service for all application components to use:

```
public class InitServlet
    extends HttpServlet
{
    public void init (ServletConfig config)
    {
        config.getServletContext ()
            .setAttribute ("records", service);
    }

    @EJB private Records service;
}
```

- **products.jsp** then uses this stored reference, as shown below. Other JSPs and servlets in the application all use this as their starting point for interacting with the order-fulfillment logic.

```
<c:forEach var="product"
           items="${records.allProducts}" >
```

## Injection by Type

---

- Let's get back to that word we used earlier ... "magic."
- Magic is great when it works, but unless we know the trick, we'll be lost when, for some reason, the magic stops working for us.
- So, how does dependency injection work?
- In the simple cases we've seen so far, the trick is just that the Java EE container will find a collaborating component of the correct **type** to satisfy the dependency.
  - The **@EJB** annotation modifies a field of a certain interface type.
  - The container seeks out an EJB that implements that type and, finding it, injects a reference to it to the annotated field.
- This suits a great many cases, because most often an EJB implements an interface that was designed just for it – or, to put it another way, there is usually only one class per interface.
- If there are multiple implementations of a bean interface, injection by type will not work.

```
@Stateless
public class Service1
    implements Service
...

```

```
@Stateless
public class Service2
    implements Service
...

```

- What then?

## Injection by Name

---

- To call out a specific collaborator, we use the name of a local EJB.
  - We'll come back to the remote case later in the chapter.
- The **@EJB** annotation defines a **beanName** attribute that can be used for this purpose.

```
@EJB (beanName="Service1")  
private Service service1;
```

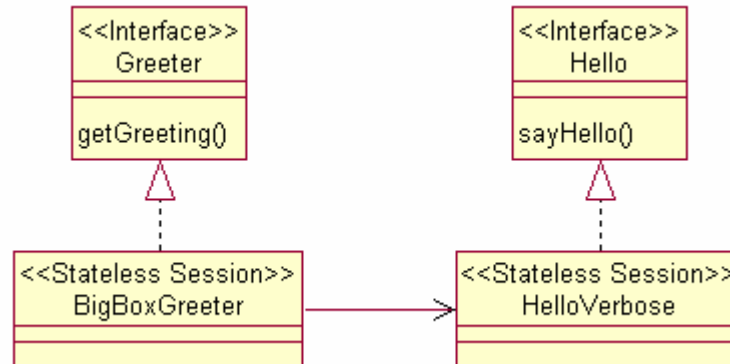
```
@EJB (beanName="Service2")  
private Service service2;
```

- Don't confuse this with the **name** attribute.
- **name** provides the name of a bean when used in the **@Stateful** or **@Stateless** annotations; on the **@EJB** annotation it defines a name for the reference, not the name of a bean to use to resolve the dependency.

## Hello, Hello

**DEMO**

- Let's look at the differences between injection by type and injection by name.
- An expanded version of the “Hello, EJB” application includes two interfaces, each with a single implementation:



- We'll observe injection by type in the starter code, and then introduce a third bean and see how that breaks the system.
  - We'll then solve the problem with injection by name.
    - Do your work in **Demos/Injection**.
    - The completed demo is in **Examples/Hello/Step4**.
1. Review the code in `src/cc/ejb/BigBoxGreeter.java`, and see that it uses the simplest possible form of dependency injection to connect to **HelloVerbose** as a delegate:

```

public String getGreeting ()
{
    return delegate.sayHello () +
        " Can I help you find something today?";
}
  
```

```
@EJB private Hello delegate;
```

## Hello, Hello

DEMO

2. Build and test the new **Client** class, which now simply looks up and calls the **BigBoxGreeter**.

**ant**

**runAC**

```
Greeting from big-box greeter:  
Well, hi, there! Nice to see you. Can I help you  
find something today?
```

3. We know that this works because the container looks for possible collaborators by the dependency type, and there is only one bean in the application that implements **Hello** right now.
4. Create a second bean class **HelloTerse** by copying **src/cc/ejb/HelloVerbose.java** and changing the class name and implementation:

```
@Stateless  
public class HelloTerse  
    implements Hello  
{  
    public String sayHello ()  
    {  
        return "Hi.";  
    }  
}
```

## Hello, Hello

**DEMO**

5. Build and test again. Hmm ...

**ant**

**runAC**

Greeting from big-box greeter:

Well, hi, there! Nice to see you. Can I help you find something today?

- The new class is ignored – true?
- We see here a big difference between JBoss and many other Java EE 5 servers. Others would have refused to deploy this application, citing an inability to figure out which bean type to use as the collaborator for the **delegate** dependency. With no clear statement of an intent to connect to one or the other of the beans, the container would not take the chance of mis-assembling the application.
- JBoss, though, ploughs ahead and chooses one – it seems it's the last bean, in alphabetical order by bean name, that gets connected.
- Try adding a third bean named **HelloX**, and see that it winds up getting the call instead of **HelloVerbose**.
- But in no case is there any error message, and we can easily get unintended results – this is not good.

## Hello, Hello

**DEMO**

6. Change **BigBoxGreeter** to select the bean it wants:

```
@EJB (beanName="HelloVerbose")
    private Hello delegate;
```

7. Rebuild and see that you get clean behavior once again.

**ant**

**runAC**

Greeting from big-box greeter:

Well, hi, there! Nice to see you. Can I help you find something today?

Evaluation Only

## Dependency “Injection?”

---

- When we use **beanName** to identify the collaborator, is this really dependency injection?
- Not exactly. Dependency injection implies that the collaborator is chosen by a separate actor (the container); in this usage the dependent source file identifies the collaborator explicitly.
- This is not dependency injection; it’s really just a slick way to look up an EJB by name.
- The practical impact is that, should the choice of bean change, we’d have to change the dependent source file and rebuild.
  - With proper dependency injection the dependent and collaborator components’ code is unchanged in this maintenance scenario.
  - Dependency by type meets this important criterion; using **beanName** does not.
  - This does not mean injection by name is a bad thing, but we’ll see next that there are other techniques to achieve it that are more flexible and maintainable than **beanName**.

## The Component Environment

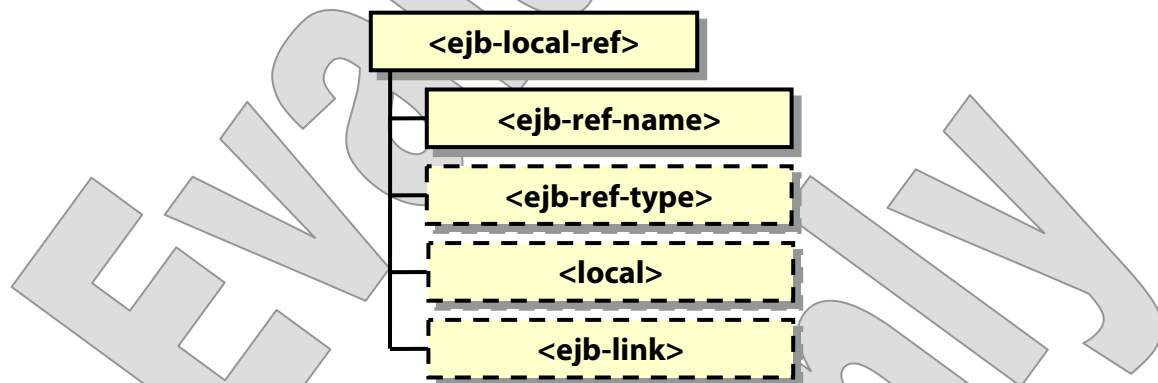
---

- Components can depend on things other than EJBs:
  - JDBC data sources
  - JMS message queues
  - Web services
  - Simple strings such as the URL of a required file
- For all these purposes, Java EE defines the concept of the **component environment**.
- This is essentially a private namespace in which useful resources can be registered.
- The mechanism is such that the dependent component can declare its environmental requirements to the container, and they can be satisfied in a variety of ways.
  - The component environment was essentially the earlier version of dependency injection for Java EE.
  - It's not as slick as Java-5 annotations, but it does essentially the same thing, which is to allow a component to connect to collaborators and resources without itself encoding dependencies on their types and locations.
  - In fact it has the advantage of being defined externally, avoiding even the naming dependency of the **beanName** technique.
- EJBs, servlets and other managed web components, and application clients all have component environments.

## EJB Deployment Descriptors

---

- The deployment descriptor was an essential – and burdensome – element of EJB 2.1 development, defining all metadata.
- An EJB-3.0 application may have no deployment descriptor, since annotations can express almost all necessary metadata now.
- But the component environment transcends individual beans – by definition – so if environment entries are required, at least a minimal descriptor will be necessary.
- The subset of the deployment-descriptor model that governs component-environment definition of local EJB references is summarized here:



- At bottom this is a mapping from the `<ejb-ref-name>` to the `<ejb-link>`, which is the name of another bean.
- A similar model obtains for web and application-client components and their environment entries.

## Finding Things

---

- The **EJBContext** interface offers the **lookup** method:  
`public Object lookup (String localName);`
  - **SessionContext** and **EntityContext** extend **EJBContext**.
- This will look up an entry in the component environment by name, and return the entry as an object reference.
- We discussed development roles in an earlier chapter, and the component environment illustrates a clear distinction between two of those roles:
  - The **bean provider** chooses the name of the reference; this is appropriate because that name is used in the Java code for the component, typically as the argument to a **lookup** call.
  - The **application assembler** works at a scope that goes beyond a single bean, and is in a position to connect that private name to a resource unknown to the bean provider: another EJB, a data source, etc.
  - In this way each participant can carry out his or her tasks without stepping on the other's toes.
- The **name** attribute to the **@EJB** annotation is a shorthand for performing a context lookup during the initialization of the dependent component.

## An EJB Deployment Descriptor

**EXAMPLE**

- Assume that we have an existing bean class in our application:

```
@Stateless public class CollaboratingBean
```

- Here is a hypothetical component environment for a session bean – this content would be found in a file **META-INF/ejb-jar.xml** located in the EJB JAR:

```
<ejb-jar version="3.0" ... >
  <enterprise-beans>
    <session>
      <ejb-name>DependentBean</ejb-name>
      <ejb-local-ref>
        <ejb-ref-name>myReference</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local>com.you.Collaborator</local>
        <ejb-link>CollaboratingBean</ejb-link>
      </ejb-local-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

- Code in the class **DependentBean** (or some other class annotated with a **name** of “DependentBean”) might perform a context lookup to connect to the collaborating bean:

```
private com.you.Collaborator otherBean;
@Resource private SessionContext context;

@PostConstruct public void init ()
{
  otherBean = (Collaborator)
    context.lookup ("myReference");
}
```

## A Web Application's Environment

**EXAMPLE**

- Components in a web application share a component environment, defined at the top level in **web.xml**:

```
</web-app>
...
<ejb-local-ref>
  <ejb-ref-name>otherRef</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>com.you.Collaborator</local>
  <ejb-link>CollaboratingBean</ejb-link>
</ejb-local-ref>
</web-app>
```

- In code for a servlet somewhere in the application, we might take the simpler approach of defining a named, injectable reference to an EJB:

```
@EJB (name="otherRef") public Hello service;
```

- Again, note the important distinction between **name** and **beanName**:
  - **beanName** calls out the name of a collaborator explicitly.
  - **name** is the name of the reference itself, which belongs entirely to the bean in which the **@EJB** annotation is written; it is a port into which the collaborator can be injected via a lookup in the component environment.

## True Dependency Injection

DEMO

- We'll do a quick cleanup of the Hello, EJB application in this regard – do your work in **Demos/CompEnv**.
  - The completed demo is in **Examples/Hello/Step5**.

1. In **BigBoxGreeter**, change the **beanName** attribute to a **name** attribute and use this to define a reference called “delegate”.

```
@EJB (name="delegate") private Hello delegate;
```

2. Copy the prepared deployment descriptor in **ejb-jar.xml** to the **ejb/META-INF** directory, and fill in the values as shown below. See what we're doing: we're declaring to the container that the **BigBoxGreeter** bean has a reference called **delegate** to a session bean with a business interface **cc.ejb.Hello**, and that this should be resolved to a bean named **HelloVerbose**.

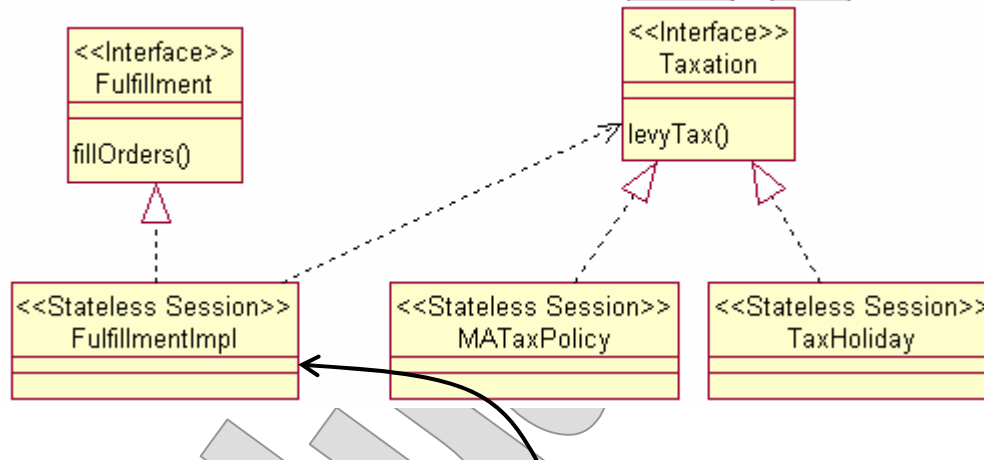
```
<session>  
  <ejb-name>BigBoxGreeter</ejb-name>  
  <ejb-ref>  
    <ejb-ref-name>delegate</ejb-ref-name>  
    <ejb-ref-type>Session</ejb-ref-type>  
    <local>cc.ejb.Hello</local>  
    <ejb-link>HelloVerbose</ejb-link>  
  </ejb-ref>  
</session>
```

- Now, we have removed the information about the collaborating bean from the dependent bean, and define it externally. Changes in the wiring of the application are now manageable from this deployment descriptor, without making changes to (who knows how many) Java source files.
3. Build, deploy, and test, and see that the behavior is unchanged.

## Configuring Tax Policy

**EXAMPLE**

- In **Examples/Wholesale/Step2**, notice that the **Fulfillment** service used by the **ProcessingServlet** requires a pluggable tax policy – and that we use injection by name to achieve this:



- See `src/cc/sales/ejb/FulfillmentImpl.java`:

```
@EJB (name="taxPolicy") private Taxation taxPolicy;
```

- ... and `ejb/META-INF/ejb-jar.xml`:

```
<ejb-local-ref>
  <ejb-ref-name>taxPolicy</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>cc.sales.ejb.Taxation</local>
  <ejb-link>MATaxPolicy</ejb-link>
</ejb-local-ref>
```

## Impact on Stateful Session Beans

---

- Injection by type or injection by name – using **beanName**, reference **name**, or context lookups – works as advertised for collaborators that are stateful or stateless session beans.
- There is one additional consideration for stateful session beans, if the dependent component needs to create multiple instances of the stateful bean – perhaps for its own purposes or perhaps because it is acting as a factory for the stateful bean.
  - Recall that the container will automatically satisfy an **@EJB** dependency by creating a distinct stateful session bean – or at least by recognizing the reference as a distinct client of the bean, even if various beans from the pool are used to serve requests.
  - It will do the same for any explicit context **lookup**.
  - To create multiple instances on demand, though, an injectable dependency is unsuitable, because it is initialized just once, during the creation of the dependent component.
  - So context lookups, though they involve a few more lines of code, are the correct technique in such a case:

```
private MyService service;

public void doGet (HttpServletRequest request,
                  (HttpServletRequest response)
                  throws ServletException, IOException
{
    request.getSession ().setAttribute ("bean",
    aJNDIContext.lookup ("someStatefulBean"));
    ...
}
```

## One Stateful Bean per Customer

LAB 7A

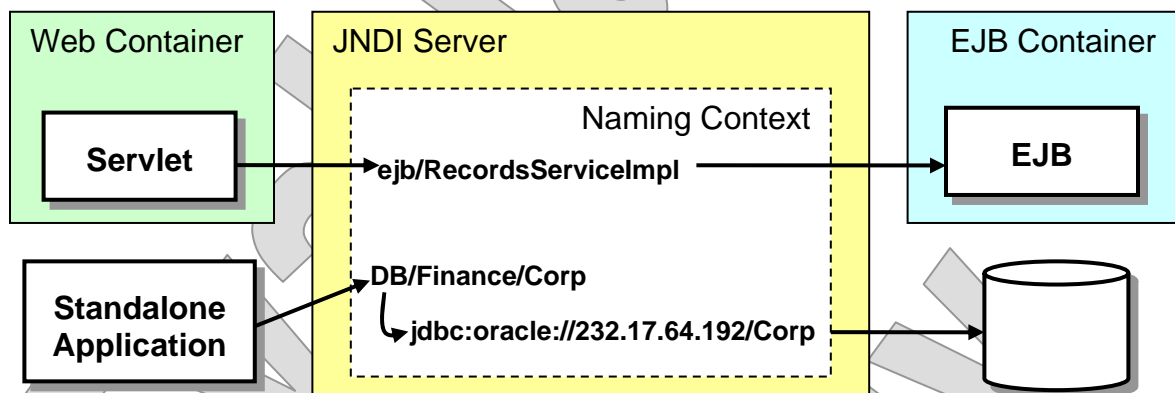
**Suggested time: 30 minutes**

In this lab you will connect or refactor `LandUse` and its bean connections, to fix the lingering problem that the stateless `LandUseServiceImpl` bean holds only one reference to the `ProposalServiceImpl` bean, and hands it out repeatedly, possibly to multiple callers who should not be sharing a stateful bean. You'll use a simple context lookup in the `getServiceForProposal` factory method; this will have the happy effect of prompting the EJB container to create a new instance of the stateful bean for each new lookup.

Detailed instructions are found at the end of the chapter.

## Remote Connectivity and JNDI

- Simple **beanName** or `<ejb-link>` lookups are limited to local dependencies.
- A dependent must find a remote beans through other means – specifically, the **Java Naming and Directory Interface**, or **JNDI**.
- JNDI makes remote objects available on a network and allows Java clients to find them based on **compound names** – a fancy term that means JNDI manages a hierarchy of named nodes.



- An EJB container publishes its remote beans in the Java EE server's **global JNDI context**.
- The name under which a bean is published is determined one of two ways:
  - By a **non-portable deployment descriptor** that attaches a specific JNDI name to a specific bean
  - By the **mappedName** annotation attribute on the bean itself (which is also non-portable)

## A JNDI Report

**EXAMPLE**

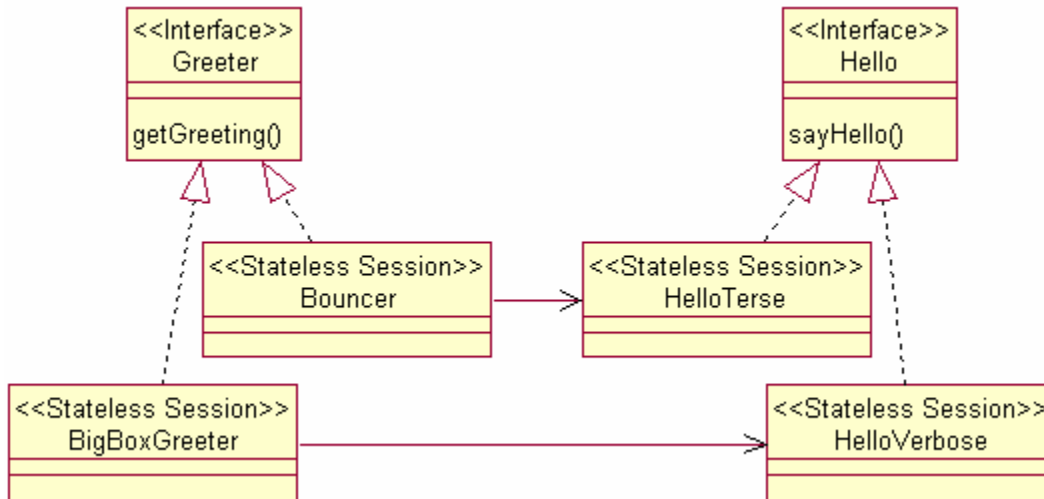
- In **Examples/JNDIReport** is a simple utility that does classic JNDI lookups using the **javax.naming** package.
  - It recurses through all published contexts and prints out a tree of names available on the local machine.
- **Build and test it now.** Depending on which code examples you've deployed, your exact results will vary:

```
ant
runAC
...
Hello
  HelloTerse
    local-cc.ejb.Hello
    local
  BigBoxGreeter
    remote
    remote-cc.ejb.Greeter
  HelloVerbose
    local-cc.ejb.Hello
    local
...
HelloClient
  UserTransaction
  metaData
  env
  Client
    bigBox
  classPathEntries
```

## JNDI Bindings

**EXAMPLE**

- In **Examples/Hello/Step6**, we see an expanded set of beans, with a second type of **Greeter** that delegates to **HelloTerse**:



- We're also taking steps to set explicit JNDI names under which the two greeter beans will be published.
  - The new **Bouncer** bean uses the simple **mappedName** technique:
 

```
@Stateless (mappedName="ejb/Bouncer")
```
  - **BigBoxGreeter** takes the more formal approach, with no attributes on its **@Stateless** annotation but instead a non-portable deployment descriptor – see **ejb/META-INF/jboss.xml**:

```

<ejb>
  <ejb-name>BigBoxGreeter</ejb-name>
  <jndi-name>ejb/BigBoxGreeter</jndi-name>
</ejb>
  
```

## JNDI Bindings

**EXAMPLE**

- The **Client** application now uses explicit **mappedName** attributes to connect its **@EJB** dependencies:

```
@EJB (mappedName="ejb/Bouncer")
    private static Greeter bouncer;
@EJB (mappedName="ejb/BigBoxGreeter")
    private static Greeter bigBox;
```

- **Build and deploy the application and give it a try:**

```
ant
```

```
runAC
```

```
Greeting from bouncer:
Hi. ID, please?
```

```
Greeting from big-box greeter:
Well, hi, there! Nice to see you. Can I help you
find something today?
```

- **Run JNDIReport again and notice the different names that are now published – working from Examples/JNDIReport:**

```
runAC
```

```
Hello
...
BigBoxGreeter
    remote
    remote-cc.ejb.Greeter
...
Bouncer
    remote-cc.ejb.Greeter
...
ejb
    BigBoxGreeter
    Bouncer
...

```

## Finding Remote Things

---

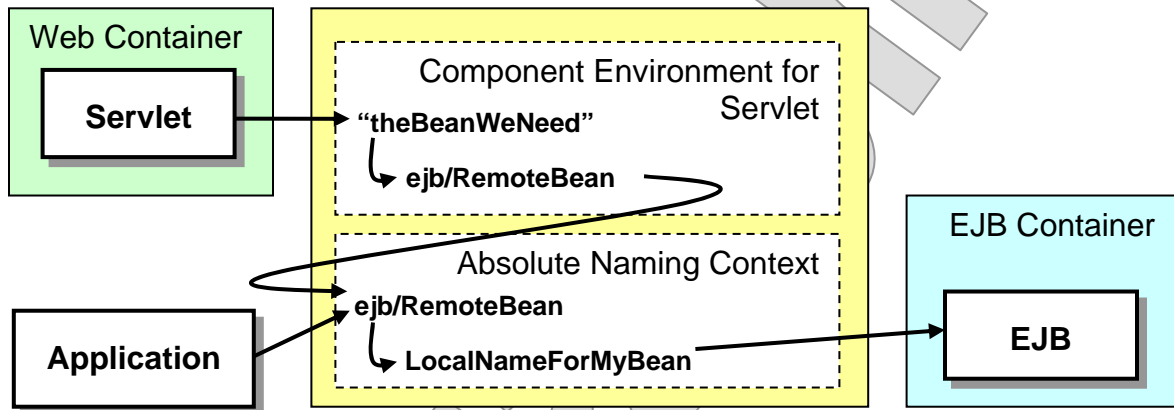
- A remote dependent must find its collaborating bean by performing a JNDI lookup.
- There are more and less complex ways in which to do this.
- The classic JNDI lookup incantation, used heavily in EJB 2.x and still possible for 3.0, is this:

```
Context ctx = new InitialContext ();
BizInterface bean = (BizInterface)
    javax.rmi.PortableRemoteObject.narrow
        (ctx.lookup ("ejb/aBean"), BizInterface.class);
```

- This has two significant disadvantages:
  - The code is intricate and error-prone. Not shown is the required exception-handling around names not found, classes not found, and so on.
  - The JNDI name is encoded in the source file of the dependent component – again, not what we want from the standpoint of best maintainability.

## JNDI for Dependency Injection

- EJB 3.0 offers an easier approach: references in the component environment can be wired to global JNDI names.



- This again requires the use of a **non-portable** deployment descriptor – now using the implicitly remote `<ejb-ref>`:

```
<ejb-ref>
  <ejb-ref-name>theBeanWeNeed</ejb-ref-name>
  <jndi-name>ejb/RemoteBean</jndi-name>
</ejb-ref>
```

- The dependent component can use **context lookups** the same way it does for local beans.

```
@Resource private SessionContext context;
...
context.lookup ("theBeanWeNeed");
```

- An important feature here is that the dependent is unaware not only of the name and location of the collaborator but even whether it is local or remote; this is known as **location transparency**.
- Again, an **@EJB** with a **name** attribute will be injected with the results of an automated context lookup.

## Using mappedName for JNDI Lookup

---

- The **mappedName** attribute may be available on the **@EJB** annotation as well as on the bean-type annotations.
  - That is, support for this attribute is optional in the specification.
- We've been using this quite a bit already, for convenience:  
**@EJB (mappedName="ejb/Bouncer")**  
**private Greeter bouncer;**
- However this is non-portable, and re-introduces the dependency on the global name of the collaborating object.
  - In other words, while it is much simpler and easier to use, it suffers the same downsides as the EJB-2.1-style JNDI lookup we saw earlier.
  - It's fine for early development cycles or proofs of concept, but not robust enough for later development and production.

## Bountiful Greetings

**DEMO**

- We'll give the application client the same treatment we gave the EJBs, and pull those specific JNDI names out of the source code.
  - In fact, we'll leave one bean as is, and refactor the other, to show that these means of working with JNDI are interchangeable on both the service and client sides of the conversation.

- **Do your work in Demos/Lookup.**

- The completed demo is in **Examples/Hello/Step7**.

1. Remove the **mappedName** attribute from the **bouncer** reference in **src/Client.java**, and replace it with a **name** for the reference itself:

```
@EJB (name="bouncer")  
private static Greeter bouncer;
```

2. Copy the file **application-client.xml** from the demo directory to **client/META-INF**, and fill in the values shown below:

```
<ejb-ref>  
  <ejb-ref-name>bouncer</ejb-ref-name>  
  <ejb-ref-type>Session</ejb-ref-type>  
  <remote>cc.ejb.Greeter</remote>  
</ejb-ref>
```

- Note that, as in our earlier demonstration, this portable descriptor only declares the **bouncer** reference; it does not resolve it.

## Bountiful Greetings

**DEMO**

3. Open `client/META-INF/jboss-client.xml` and fill in values there as well – this resolves the reference to a global JNDI name, and again we've pulled the location information out of the dependent Java source to a more maintainable external file:

```
<jndi-name>HelloClient</jndi-name>
<ejb-ref>
  <ejb-ref-name>bouncer</ejb-ref-name>
  <jndi-name>ejb/Bouncer</jndi-name>
</ejb-ref>
</jboss-client>
```

- The pre-existing `<jndi-name>` provides a name for the application client itself; JBoss names all client applications, and in fact we've been looking our client apps up, in order to run them.

4. Build and test and see that we can still contact both greeters:

```
ant
```

```
runAC
```

```
Greeting from bouncer:
Hi. ID, please?
```

```
Greeting from big-box greeter:
```

```
Well, hi, there! Nice to see you. Can I help you
find something today?
```

- Finally, notice that we're mixing our techniques freely:
  - We look up **bouncer** using an external deployment descriptor to find a bean that uses **@Stateless (mappedName)**.
  - We do just the opposite with **bigBox**, using **mappedName** on the client side to find a bean whose JNDI name is called out in a deployment descriptor.

## Summary of Techniques

---

- Let's review the options available, as we've covered quite a bit of ground in this chapter.
- On the service side, an EJB can be named in several ways:
  - Its **default local name** is the class name with no package qualifiers.
  - We can override that with an explicit **name** attribute to the bean-type annotation (**@Stateful** or **@Stateless**).
  - It's **default remote name** is derived from the fully-qualified name of the business interface – not the class.
  - We can override this with the **mappedName** attribute.
  - Or, we can define it in a **non-portable deployment descriptor**.
- Clients can find local beans in a few ways:
  - **Injection by type**, if there is only one local implementation
  - Injection by **beanName** – this is suspect because it actually sets a name dependency into the source code
  - Injection by **portable deployment descriptor** and **<ejb-link>**, using either explicit **EJBContext.lookup** or a **name** for the dependency
- For remote beans the options are similar:
  - **Injection by type**, if there is only one implementation on the host
  - Injection by **mappedName** – this too is a compromise
  - Injection by **JNDI lookup** using a non-portable descriptor, and again either by explicit lookup or a reference **name**

## Who Can Declare Dependencies

---

- So far we've focused on injection into a few types of components: EJBs, servlets, and applications running in the Java EE application-client container.
- Other objects can use dependency injection. EJB and resource references can be declared on the following component types:
  - EJBs
  - Servlets
  - Servlet filters
  - Web context, request, and session event listeners
  - Custom JSP tag handlers
  - JSF managed beans
  - Application clients in the Java EE container

## EJBs in Web Applications

---

- Conspicuously absent are POJOs running in the web container.
- These classes, even though their code may be invoked directly from a servlet, have no direct relationship with the web container.
  - They are not declared to the container in **web.xml**, nor annotated as playing any special role.
  - Therefore the container cannot index them, manage them, control their lifecycle, inject dependencies, etc.
- For servlet/JSP web applications (or those that use MVC frameworks such as Struts and Spring), a strategy is needed to assure that objects that use EJBs and resources can find them.
- Here are three common approaches:
  - If a servlet is involved in each HTTP request, it can use direct dependency injection or JNDI lookup to find collaborators, and make either them or the results of invoking their methods available as **request attributes**.
  - When working with stateful session beans, a variant of this strategy has the servlet posting the stateful bean reference as a **session attribute**, which matches up nicely to the lifecycle and availability of the stateful bean itself.
  - An **initialization servlet** can publish its injected dependencies as **application attributes** through the servlet context.
- JSF applications have an easier time of it, since they have their own dependency-injection features and these integrate easily with EJB injection.

## A Remote Web Application

LAB 7B

**Suggested time: 15 minutes**

In this lab you will observe the configuration and functioning of a simple web application deployed with the Hello/Greeter EJBs; and then configure a separate version of that web application that must function outside the main EAR deployment, contacting the target EJB remotely.

Detailed instructions are found at the end of the chapter.

Evaluation Only

## SUMMARY

- **At first blush, dependency injection looks too good to be true.**
  - Too much of the literature on EJB3 boasts about the simplicity of injection by type, while conveniently ignoring the slight complications of injection by name.
- **In fact it is a powerful feature that is well worth the trouble to implement – but it is not the mind-reading system we’re sometimes led to expect.**
  - You can have the absolute simplicity of an unattributed **@EJB** annotation.
  - You can have the declarative power of choosing a collaborating bean by name (local or JNDI).
  - You just can’t have them both for the same dependency! and the cost of injection by name is not that high – nothing like the onerous EJB 2.1 descriptor that many developers remember with some dread.
- **When dealing with stateful session beans, it’s essentially unworkable to inject the EJB reference: instead, inject a context reference and look up the bean each time it’s needed.**
  - This will trigger creation of a new stateful bean for each new client conversation.

## One Stateful Bean per Customer

**LAB 7A**

In this lab you will connect or refactor `LandUse` and its bean connections, to fix the lingering problem that the stateless `LandUseServiceImpl` bean holds only one reference to the `ProposalServiceImpl` bean, and hands it out repeatedly, possibly to multiple callers who should not be sharing a stateful bean. You'll use a simple context lookup in the `getServiceForProposal` factory method; this will have the happy effect of prompting the EJB container to create a new instance of the stateful bean for each new lookup.

<b>Lab workspace:</b>	<b>Labs/Lab7A</b>
<b>Backup of starter code:</b>	<b>Examples/LandUse/Step7</b>
<b>Answer folder(s):</b>	<b>Examples/LandUse/Step8</b>
<b>Files:</b>	<b>ejb/META-INF/ejb-jar.xml</b> (move from root directory) <b>src/gov/usda/usfs/landuse/ejb</b> <b>    /LandUseServiceImpl.java</b> <b>src/gov/usda/usfs/landuse/ejb</b> <b>    /ProposalServiceImpl.java</b>

### Instructions:

1. Build the starter version of the application. If possible, test it in two separate browsers on your machine, or, if your classroom is networked, pair up with another student and both test against one of your applications. Try doing editing on two different proposals concurrently: each open a different proposal, and intermix actions by the two clients such as adding comments or deciding the proposal. We're testing the application to see if it is capable of handling multiple concurrent clients: do one client's actions affect the other client's target proposal?

A specific test for this is as follows. Open the application in two separate browsers. Then, in browser A, select the first proposal in the list and click **Edit**. In browser B, select the second proposal, and click **Edit**. In browser A, make a change to the proposed use field, and click **Done**. What happens?

## One Stateful Bean per Customer

## LAB 7A

It works! The change goes only to the first proposal, and if you make a similar change in browser B and click **Done**, you that change assigned cleanly to the second proposal. So ... no problem? All is well?

Not exactly. There is an underlying problem, as we've discussed, which is that the stateless bean is not acting as a factory for stateful beans, but just handing out a single object reference over and over. It's hard to observe this bug without high-volume concurrency testing, because JBoss hands out multiple stateless instances to start with: that is, each distinct web session is getting a separate stateless bean, and so the fact that the stateless bean then has just one stateful bean is not a problem ... yet. Eventually, the pool of stateless beans would max out and then a stateless bean would be recycled (which is fine) – along with the stateful bean it had injected into it originally (which is not).

To make it easier to observe the bug, we're going to tweak the pooling policy JBoss uses for the stateless bean. We'll also instrument the code a bit to clarify what's going on under the hood.

2. Open **LandUseServiceImpl.java**, and import **org.jboss.ejb3.annotation.Pool** and **org.jboss.ejb3.annotation.defaults.PoolDefaults**.
3. Now add an annotation to the bean, as shown below:

```
@Stateless
@Pool (value=PoolDefaults.POOL_IMPLEMENTATION_STRICTMAX, maxSize=1)
public class LandUseServiceImpl
```

So, we're asking JBoss to use just one instance of this bean, come what may. (This is closer to the singleton policy discussed a few chapters ago, which is applied by default in some other EE servers.)

**One Stateful Bean per Customer****LAB 7A**

4. In the **getServiceForProposal** method, add code to write the object's string representation to the server console – this will include a unique object ID, so we can see exactly what instances are doing what. For example:

```
System.out.println ("getServiceForProposal() called on " + this);
```

5. Open **ProposalServiceImpl.java** and add similar code to trace calls to the **init** method.
6. Build (and you may want to **run PrimeWithData**) and again test with two browsers and alternate your actions over two proposals. Aha – now you see the concurrency failure: you see your first change in both the first and the second proposal. The server console clarifies what's going on:

```
getServiceForProposal() called on  
gov.usda.usfs.landuse.ejb.LandUseServiceImpl@b9ce56  
init() called on gov.usda.usfs.landuse.ejb.ProposalServiceImpl@17586b4  
getServiceForProposal() called on  
gov.usda.usfs.landuse.ejb.LandUseServiceImpl@b9ce56  
init() called on gov.usda.usfs.landuse.ejb.ProposalServiceImpl@17586b4
```

Again, this would happen eventually at any pool size; fixing a strict one-bean pool is just a way to expose a basic coding problem.

**One Stateful Bean per Customer****LAB 7A**

7. Open `LandUseServiceImpl.java`, and remove the `@EJB` dependency at the bottom of the class definition.
8. In its place, declare a `@Resource` dependency called `context`, which is of type `SessionContext`.
9. At the beginning of `getServiceForProposal`, add code to declare a local variable `proposalService`; this will replace the class field you've been using so far. Initialize it to the results of a call to `context.lookup`, passing "proposalService"; you'll have to downcast this result since `lookup` returns an `Object`.

The rest of the method code will work as it did before – but now it's getting a freshly-created session bean to initialize and return, instead of the same one every time.

10. Now, how is this lookup going to be resolved correctly? No magic! We have to declare the EJB reference in the deployment descriptor. Move the provided file `ejb-jar.xml` from the lab directory itself down to `ejb/META-INF`, and review the contents of the file:

```
<session>
  <ejb-name>LandUseServiceImpl</ejb-name>
  <ejb-local-ref>
    <ejb-ref-name>proposalService</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>gov.usda.usfs.landuse.ejb.ProposalService</local>
    <ejb-link>ProposalServiceImpl</ejb-link>
  </ejb-local-ref>
</session>
```

11. Build and test the application again, and you should see that separate clients can now work with the application without getting in each other's way.

You can also see the correct separation of the stateful beans in the server console now: one stateless bean is fine, but now we get one stateful for each caller.

```
getServiceForProposal() called on
gov.usda.usfs.landuse.ejb.LandUseServiceImpl@a98c44
init() called on gov.usda.usfs.landuse.ejb.ProposalServiceImpl@11fc108

getServiceForProposal() called on
gov.usda.usfs.landuse.ejb.LandUseServiceImpl@a98c44
init() called on gov.usda.usfs.landuse.ejb.ProposalServiceImpl@54dd6a
```

# A Remote Web Application

**LAB 7B**

In this lab you will observe the configuration and functioning of a simple web application deployed with the Hello/Greeter EJBs; and then configure a separate version of that web application that must function outside the main EAR deployment, contacting the target EJB remotely.

<b>Lab workspace:</b>	<b>Labs/Lab7B</b>
<b>Backup of starter code:</b>	<b>Examples/HelloRemote/Step1</b>
<b>Additional folder(s):</b>	<b>Examples/Hello/Step8</b> (local web application)
<b>Answer folder(s):</b>	<b>Examples/HelloRemote/Step2</b>
<b>Files:</b>	<b>docroot/WEB-INF/web.xml</b> <b>jboss-web.xml</b> (template) <b>docroot/WEB-INF/jboss-web.xml</b> (to be created and modified)

## Instructions:

1. The Hello application has been enhanced with a web module that calls one of the EJBs. Review the code in **Examples/Hello/Step8/src/cc/web/GreetingServlet.java**. Observe the injectable dependency on a **Greeter** EJB:

```
@EJB (name="greeterService") public Greeter service;
```

2. From the directory **Examples/Hello/Step8**, build and deploy the application.

**ant**

3. Test at the following URL, and see that the **GreetingServlet** has gotten a greeting and made it available to the **greeting.jsp** to be viewed:

**http://localhost:8080/Hello**

## Hello Web Client

Greetings from the EJB application:

**Hi. ID, please?**

## A Remote Web Application

## LAB 7B

How does the servlet get attached to the correct EJB? The portable deployment descriptor connects the “greeterService” reference to the **Bouncer** bean – see **Examples/Hello/Step8/docroot/WEB-INF/web.xml**:

```
<ejb-ref>
  <ejb-ref-name>greeterService</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <remote>cc.ejb.Greeter</remote>
  <ejb-link>Bouncer</ejb-link>
</ejb-ref>
```

4. Now, look over the code in your lab directory, and see that just a few files have been selected from the larger **Hello** application as a starting point for **HelloRemote**:
  - The **docroot** directory, including the JSP and **web.xml**
  - The **META-INF** directory with **application.xml**
  - The servlet class in **src/cc/web/GreetingServlet.java**
  - The business interface in **src/cc/ejb/Greeter.java**

These files are almost identical to their counterparts in the **Hello** application you just tested: the only changes have been in filenames and URLs so that this application is deployed independently under the name **HelloRemote**, and in user-readable strings so that its presentation can be distinguished from the one you just saw.

5. Of course if you try to build and deploy this application, you won't get very far, because the **<ejb-link>** to the **Bouncer** bean will be irresolvable. You'll have to refactor this to connect to a remote bean instead of a local one. Start by removing that line of code from **web.xml**.
6. Create a new file **docroot/WEB-INF/jboss-web.xml** by copying the one in the root lab directory. This is the non-portable deployment descriptor, which is necessary for remote connections since these require JNDI. Fill in the necessary information for the remote EJB reference as an **<ejb-ref>** element – you can use the **jboss-client.xml** file from the “Bountiful Greetings” demo as a template for this new content, but the reference name is now **greeterService**, and let's connect to the **BigBoxGreeter** instead of the **Bouncer**.

**A Remote Web Application****LAB 7B**

7. Now you've refactored the application to look for a remote bean by its JNDI name instead of a local bean by its local name. Notice that you did this without touching either the client or the EJB source code. Build and test the new web application, and see the verbose greeting of the **BigBoxGreeter** at the following URL:

**ant**

**`http://localhost:8080/HelloRemote`**

### Hello Remote Web Client

Greeting from the remote EJB application:

**Well, hi, there! Nice to see you. Can I help you find something today?**