

EJB DEPENDENCIES – LOOKUP VERSUS INJECTION

By Brian Repko

Introduction

This whitepaper outlines a set of issues and concerns that I have with the proposed change in the Enterprise JavaBeans specification (from version 2.1 to version 3 drafts) as it relates to dependency acquisition.

Let Me Get This Right

My mama taught me (ok, it was an old boss) that before you criticize anything, you better check your assumptions and your understanding. So lets take a moment and go over my understanding of all this EJB and dependency stuff.

What You Talkin' About Willis?

So, just what are these dependencies that we are talking about. This hasn't really been specified in the EJB3 draft specifications, but based on those documents and the existing EJB2.1 specification, I came up with the following potential list.

Resource, Resource Environment and Message References

The main class of objects that are mentioned as EJB dependencies are related to resources and other administered objects that the EJB needs in order to access those resources. These include the resource manager connection factories – JDBC DataSource objects, JMS ConnectionFactory objects (along with the Topic-specific and Queue-specific subclasses), JavaMail Session objects and URL objects. I would also include in this set the generic Java Connector ConnectionFactory objects.

In addition to the resource manager connection factories, we have other administered objects that are also needed to fully access the resource. As of J2EE 1.4, the JMS Topic and Queue are now officially called Message Destination References but they belong in here. The other group of objects that go in here are officially called Resource Environment References. Currently this only includes Java Connector CCI InteractionSpec objects.

I thought about adding the MessageEndpoint objects for Message-Driven Beans but they aren't really something that the MDB uses – the container manages / uses them for MDB activation – so they are not included.

EJB References and Web Services

The next group of objects that are EJB dependencies are those that represent other services or entities that the EJB needs in order to function. This set includes other EJBs (Session and Entity) and Web Services ServiceEndpoint objects.

There are some significant changes here as the EJB3 draft specification removes EJBHome objects and adds an EntityManager. I'll discuss this in more detail later but lets include the EntityManager as a dependency.

Environment Variables and Container Objects

The last group of EJB dependencies includes bean configuration information – its environment variables – and objects used in interacting with the EJB container. This last set includes the UserTransaction, the calling Principal, the TimerService and the ORB.

Dependency Lookup

The Current State of Affairs

A component developer using the current EJB spec (EJB2.1), gets these dependencies either by doing a component (java:comp/env) JNDI lookup or by getting the object directly from the context object (SessionContext, EntityContext or MessageDrivenContext). Within the bean code, one typically gets the object, uses it and then lets the reference go out of scope (or properly cleans up first, depending on the type of object). One may, potentially for performance reasons, decide to hold on to that object as an instance variable.

For Resource, Resource Environment, Message and EJB References, the component JNDI object is mapped to an object in a server JNDI tree through information in both the standard and server-specific deployment descriptors. For Web Services there is a separate deployment descriptor for the actual object bound to the component JNDI. Environment Variables are defined in the standard deployment descriptor and gotten from the component JNDI. The transaction and the ORB are also found in the component JNDI but always at a standard name. The last dependencies (principal and timer service) are gotten from the context object.

As an example, the current specification requires the following code:

```
public void someMethod() {
    InitialContext ctx = new InitialContext();
    DataSource ds =
        (DataSource)ctx.lookup("java:comp/env/jdbc/TestDS");
    Connection conn = ds.getConnection();
    ...
}
```

New Life for EJB

I read the New Life for EJB papers by Ganesh Prasad and Rajat Taneja [\[NewLife\]](#) and agree with their idea of doing dependency lookup through the context object. In other words, replace (or hide) the component JNDI lookup behind the context object and use it as a “one-stop shop” for EJB dependencies. This also has the side benefit of making the EJB API easier to understand. A change to the EJBContext (or subinterface) APIs could produce EJB code that looks something like this:

```
private SessionContext ctx;

public void someMethod() {
    DataSource ds = ctx.getDataSource("TestDS");
    Connection conn = ds.getConnection();
    ...
}
```

Dependency Injection

The phrases “Inversion of Control” (or IoC) and “Dependency Injection” now roll off the tongues of Java geeks everywhere since the whitepaper by Martin Fowler was put on the web [\[Fowler\]](#). That paper describes two types of object-object relationship establishment patterns – the Service Locator (which is basically what I’m calling “Dependency Lookup”) and Dependency Injection. It then goes on to identify three types of dependency injection – Constructor, Setter and Interface injection.

The EJB3 draft specification has chosen to pursue setter injection for EJB dependencies. The proposed specification would produce code that looks like this:

```
private DataSource ds;

public void setTestDS(DataSource ds) {
    this.ds = ds;
}

public void someMethod() {
    Connection conn = ds.getConnection();
    ...
}
```

In terms of EJB lifecycle, the EJB3 draft specification states that dependencies are injected (the setter is called) immediately after the `set<type>Context` method is called. This means that all dependencies are set prior to `ejbCreate` for session and message-driven beans and before the bean becomes pooled for entity beans.

The EJB3 draft specification does not remove the existing dependency lookup capabilities (with the exception of removing EJBHome objects) and does offer a change to the EJBContext interface by adding a `lookup(String name):Object` method in order to lookup dependencies that were only previously available via JNDI. While I welcome this addition of a lookup method, I think that the interface described in [\[New Life\]](#) or here is a cleaner API (no need for casting, more readable code, define the method parameters per resource type).

Summary

Basically, dependency lookup (DL) means “get me the object” and dependency injection (DI) means “make the object available to me”.

The current EJB2.1 specification does DI of the context object and DL (through the context object or via JNDI) for everything else. The EJB3 draft specification does DI for any of these objects and offers DL in its current form or as a `lookup()` method on the context.

The Airing of Grievances

So, now that I've gone over what I think I know, lets get right in to the issues.

Right Here, Right Now

The first issue that I have with using DI for EJB dependencies relates to reference scope and lifecycle.

With DI, what used to be a method local variable is now an instance variable. The obvious concern with changing a method local variable to an instance variable would be thread safety. However, this is not an issue inside the EJB container as it is a thread-managed environment. The other thing to note is that most (if not all) of the objects listed as dependencies are thread-safe.

This is more of an aesthetic issue for me – I've always been taught that resources should be acquired, used and released in the method that uses them. We do that with connections and other objects, so why not with the connection factory as well. The container should make acquiring the factory as cheap as possible.

In addition, a JNDI lookup may actually be an object creation and not just an object lookup. In particular, JavaMail Session objects are typically created on lookup. This means that the lifecycle of JavaMail Sessions will be very different with DI – the same Session object is used throughout the life of the bean as opposed to just the method using it.

Another part of this issue is reference lifecycle, in particular the release side of the “acquire-use-release” paradigm. Currently, looking up a Session EJB from another Session EJB means that you get the home and create the EJB (“acquire”), use it and remove the EJB (the EJBObject, “release”). With DI, the container will create the EJB and call a setter and you use it from there. There is no opportunity to release the dependency (i.e. call remove and set the reference to null). You can’t do that in the `ejbRemove` method either since `ejbRemove` does not get called on an exception. I do understand that SLSB remove is technically not needed – again this is more of an aesthetic issue but points to an architectural constraint on objects that can be injected.

The last part of this issue is the amount of references. With DI, each bean in the pool has a reference to every dependency of that EJB rather than just letting the references come and go when needed. Again, more aesthetic, but this can potentially become an issue with memory footprint and GC. A system of 30 SLSBs (50 per pool) and 50 EBs (200 per pool) with 5 dependencies each will require ~225k of long-lived memory to hold the references as opposed to short-lived (come and go) memory. Even if we cached those objects in a once-per-bean context object we could reduce that footprint to ~1.6k of long-lived memory.

Ch-Ch-Ch-Changes

The second issue that I have with using DI for EJB dependencies is how to handle changes to an object you have already injected (i.e. re-injection). If you factor changes into the picture, then that summary section above changes a bit. It would read – DL is “get the CURRENT object” and DI is “make the CURRENT object available to me”. Let me be clear that “change” means that the object reference has changed – not that the object state has changed. We are talking about new objects.

One way to handle dependency changes is to ignore them. The object is injected once and never re-injected (similar to “final”). However, this implies that you won’t rebind an object into the JNDI tree. That’s a pretty big restriction. In particular, it means that environment variables cannot be changed at run-time. It also will not work with the calling Principal given the current implementations of that object on various application servers (i.e. the calling Principal does not have a `ThreadLocal` within it, as does `UserTransaction`).

So let’s say that we really do have a need to re-inject the dependency. When do we call the setter on the EJB bean class? We can’t just re-inject “on change” because the dependency is an instance variable and that would normally be a race condition (who wins – the setter or the code that is using that dependency) and EJBs are not allowed to use synchronized. So we either allow for synchronized code in an EJB and abandon their single-threadedness OR we have to add DI re-injection into the lifecycle of the bean. Let’s go with lifecycle on that one. Which part of the lifecycle? Because the dependent object could change at anytime, we would have to re-inject as part of a business method call – similar to `ejbLoad`. But that would be worthless to do if the dependency hasn’t actually changed. Because EJBs are pooled objects, the issue is not if the object has changed but if it has been re-injected in to the bean being used – some beans in the pool may have the latest object (they have been used post-change) and other beans in the pool may not. Re-injection requires that all beans in the pool will require the new object at some point. This becomes a lot of infrastructure code for managing something that I already get with DL (“get the current object”). Perhaps not allowing re-injection wasn’t so bad and we just don’t let the Principal be a dependency or environment variables to be changed at run-time.

You're So Far Away

The third issue is really the same issue as the last but applied to remote objects. In particular JMS Destinations and ConnectionFactories (a remote JMS Server) or remote EJBs. We've just taken the problem of managing re-injection and made it a cross-JNDI or cross-JVM issue. That is even more infrastructure code for managing something that exists already.

Its Good To Be King

The fourth issue that I have with using DI for EJB dependencies is exception handling. With DL, you control when the lookup is done and can try/catch for it. Since you are in a business method, you can determine how the EJB client sees a failure to get the dependency – is it an application exception? is the transaction rolled back or not? In other words, you control what happens on failure.

With DI, the container looks up the dependency during the lifecycle of the bean and injects it into the bean. So what happens if the container can't lookup the dependency? The first EJB3 draft specification has a statement that the method using the dependency would throw a RuntimeException. This statement is removed in the second EJB3 draft specification. Sometimes, you don't want control to be inverted.

There's No Place Like Home

The fifth issue relates to the bigger issue of removing EJBHome objects and how a client gets access to an EJB.

For Session EJBs, the EJB3 draft specification refers the reader to Chapter 8 for information on how the client gets access to a session bean. Chapter 8 discusses the idea and syntax of DI in detail. This makes a huge assumption on EJB clients – they must be running in an IoC container that understands the EJB injection syntax. J2SE with no additional software must be a viable EJB client environment. There should be no assumed dependency on a container for clients (and here I mean client as in client-server, not J2EE Application Clients, which do have a container). In addition, there is no syntax for injection of remote (meaning from a different JNDI) EJBs. The remoteness of EJBs has to be preserved.

In the case of Entity EJBs, the many home objects will be replaced with a single EntityManager object. The EJB3 draft specification does not include an injection syntax for the EntityManager. This syntax needs to be specified so that the object is available to clients of Entity EJBs (i.e. other EJBs, J2EE Application Clients and J2SE). The specification will also need to include syntax for injection of remote EntityManager objects. In addition, a client that accesses two different application servers may have to manage injection of potentially two different EntityManagers. Again, there is nothing for how that would work in the EJB3 draft specifications.

Lastly, it is unclear from the EJB3 draft specifications if EJBHome objects actually go away or what specific object counts as an EJB dependency – Home or Object. The DI examples in Chapter 8 include injection of EJBHome objects. This has to be made clear.

Containers are from Mars, Test Harnesses are from Venus

The last issue that I have with using DI for EJB dependencies is the purported reason given for doing it – that it is the only way to do testing outside of the container. I find this reason to be a joke. Using DL, with the proposed API changes, one could create a test harness with mock context objects and have them do whatever is needed to get the dependent object. You'd have to create a mock context object anyway for access to container services and objects (Timer, Transaction, Security, entity PrimaryKey) so it just wouldn't be that hard to do dependencies that way as well.

You Want Fries With That?

Moving past the issues, the concern that I have with DI for EJB dependencies, now that the idea has been released to the public, is that it will stay in the specification but as an option. DL and JNDI will not go away, because code will still have to be backward compatible and JNDI may be the only way to get remote objects. If DI is included as an option, it will have to be fully worked out and specified; however, I feel that there are so many issues with the option, for no new architectural features, that working through those issues is just wasted effort that could be better spent on other more important and needed improvements to the EJB specification.

Conclusion

Dependency lookup works now – we can make it better with some simple API changes. In the end, I find the proposed change in the EJB3 draft specification neither aesthetically pleasing nor technologically pleasing. I feel that adding dependency injection solves no problem that can't be solved more easily, and it potentially introduces new problems. I would prefer that the EJB3 Expert Group spend their time on other items that are not currently included in the EJB3 draft specifications (e.g. common library packaging or instance-based security). So as not to be mistaken, I am a proponent of using IoC and dependency injection, but not in the case of EJB dependencies and architecture. I humbly request that dependency injection (except for the context object) be removed from the EJB3 draft specification and that dependency lookup be made simpler with changes to the API.

References

[NewLife] <http://today.java.net/pub/a/today/2004/08/05/ejbnewlife.html>
[Fowler] <http://martinfowler.com/articles/injection.html>

About the Author

Brian Repko is one of many Java geeks at Object Partners, Inc. – a consultancy located in Minneapolis, Minnesota. He grew up on EJB going back to when “WebLogic” was the name of the company and “Tengah” was the name of the server and is currently enjoying making EJB, Spring and Hibernate all work together. He'd like to thank all his co-workers and paper reviewers for their comments. Brian can be reached via email at brian.repko@objectpartners.com.