

by Hani Suleiman

Exposing J2EE Urban Myths

The reality behind the legends

It has become fairly common these days when looking through blogs and various opinion pieces to hear a common cry: J2EE is a terrible, unwieldy, and cumbersome specification. While documentations from Sun and other vendors praise it, there is a lot of hostility and negativity toward it “down in the trenches,” so to speak. These trenches, of course, are populated with technologists who are always on the lookout for the next big thing, and the rank and file folks who look up to the technologists for wisdom and guidance.

In my opinion, many of these complaints are misguided, spread through rumor mongering and anecdotal stories with little to no effort made to validate them or place them in context.

Without further ado, I present you with nine urban legends surrounding J2EE.

1. JNDI is awkward.

JNDI is a lightweight directory service mandated by the J2EE spec that essentially consists of a directory of services where components can look up other components. Many people will speak derisively of JNDI; the latest buzzword compliance laws decree that you must use dependency injection to have components injected in rather than explicitly looked up. While both approaches have their merits and disadvantages, it's far from clear whether there is an overall winner.

JNDI is tremendously useful when you consider the fact that many vendor implementations will throw in features like a global clustered JNDI tree. Locally scoped JNDI references also free you from having to embed resource absolute names in your source code. JNDI is also fairly straightforward and easy to use; the barrier to entry is low enough that it really does not qualify as an obstacle.

2. J2EE is difficult to develop in.

While there is some merit to the

argument that EJBs and J2EE in general do not cater much to the in-vogue “agile” and “test-driven” development mentality, a lot of the blame should be placed squarely on vendors' shoulders.

For example, there are still vendors who do not support rapid hot-deploy, exploded application directories and runtime configuration. When having to work with such servers, it's not surprising that developers are turned off of J2EE and complain that it takes 20 minutes to preview every EJB change. It doesn't help either that many of these vendors had terrible implementations that performed abysmally and forced the hapless user to jump through any number of fire hoops to get things done.

The deployment descriptor hell is also a valid argument, but tools like XDoclet and the upcoming J2EE 1.5 greatly ease this pain. Going further, there's a blossoming market of tool vendors who try to cater to “rank and file” developers. WebLogic Workshop, WSAD, and Sun's Java Enterprise Studio are all fine examples of vendors stepping in to try to cater to a certain kind of developer. It's also important to consider the target audience for J2EE. A simple rule of thumb applies: if it doesn't fit, don't use it. Many people forget the “enterprise” aspect of J2EE. Don't overassume your project's needs and, even more important, don't overassume your own skill level. J2EE is not a trivial spec; it attempts to tackle some very thorny issues that, contrary to what you might have been told, do not have a simple solution.

3. EJBs must be avoided at all costs.

EJBs have been much maligned recently. Many loud and outspoken developers have made quite a bit of a ruckus over the inadequacies of EJBs, vowing never to touch them with a 10 foot pole. Unfortunately, in many situations this has resulted in a “let's throw the baby out with the bathwater” men-

tality. EJBs are being punished for bad marketing from Sun. No, they are not a silver bullet. In a lot of cases, entity beans are inappropriate and a huge overkill.

When it comes to session beans and message-driven beans though, there are clear benefits and advantages that should not be ignored due to the frightful aura that the letters “EJB” seem to have floating around them. Don't be so eager to dismiss the benefits of a component architecture that frees you from having to worry about pooling, security, transactions, clustering, and distribution. Part of the blame, of course, has to be placed on examples such as the early Petstore examples that encouraged the use of EJBs when it very often just didn't make any sense; but more on that later.

4. J2EE is dead!

In the trenches, EJBs on the sly have had and continue to have huge market penetration; while few people brag about this, nobody is surprised to find them slowly wriggling their way into many enterprises.

It's rare to find a financial institute that does not make extensive use of JMS. It's rare to find an enterprise that supports Java that has not purchased at least one J2EE application server. It's often more than a token purchase too, with important applications and services running on these servers. Also easy to forget is that a J2EE application will often not contain a single EJB (and be the better for it!). You're still using J2EE, and an application server will still provide your application with a lot of value-add. It's not a stigma to be bound to J2EE!

5. J2EE is unportable.

From the outset, one of the principles of J2EE was that vendors were free (and encouraged) to have their own deployment descriptors that allow deployers to fine-tune an application's



Hani Suleiman is the CTO of Formicary (www.formicary.net), a consulting services and portal solution provider. He is also a developer on a number of popular open source projects.

hani@formicary.net

deployment. Many vendors foolishly implemented this by mandating their custom deployment descriptors before deploying.

Realistically, you *can* have your applications be portable. Expecting it to be a trivial matter of dropping in the .ear file is naive and akin to expecting a Windows-developed Swing application to work flawlessly on an OS X machine without a single misplaced pixel. Sure it's doable once you do it a few times, but it takes knowledge of both Swing implementations to get it right.

To be fair to Sun, there have been a number of JSRs to help ease this problem. JSR-77 and JSR-88 (J2EE management and J2EE deployment, respectively) both attempt to reduce the vendor variance in these two areas.

Part of the problem also lies in the confusion about J2EE roles. While in reality it might be one developer who develops, assembles, and deploys an application, it pays to keep in mind that those three tasks are performed by different roles. A deployer, for example, would not know about the code that is being deployed, but would be able to spend the time and effort required to ensure the application is configured correctly to be deployed into a particular container and environment.

6. *J2EE is expensive.*

Yes, it's hugely expensive if you go with the big name vendors. Even though some have dropped their prices, don't expect to feel you've gotten a good deal if you go for the IBMs and BEAs of this world. There are a number of both cheap and free alternatives that work just as well (and in some cases, just as badly!) as the "big" players out there. Whatever your budget, you're likely to find a vendor who caters to it.

The hidden aspect of expense is the nonmonetary costs associated with it. Yes, you do need someone comfortable in a J2EE environment. For some of the free offerings, you certainly need to invest the time and effort it takes to get some of the more obscure features you're after to work. Do not neglect to factor in the cost of the time and effort you'd be wasting by going to a third-party library and spending hours reconfiguring it for every new application.

8. *Petstore is a reference implementation.*

Incredibly, many people use Petstore to "prove" that J2EE is overly complex and awkward. Petstore is not a reference implementation; it spends a lot of effort utilizing every approach and API possible. Taken as a whole, it's a collection of blueprints artificially packaged in one solution. Only a madman would use *all* the blueprints when developing his own app. Simply pick a few that make sense to you and use those, keeping in mind the project's requirements.

Sun seems to have realized the faults with Petstore and has probably come to regret the early hacked-together versions. The new Adventure Builder blueprints application seems closer to a "real-world" solution in terms of implementation, I'm told. If you really want a Petstore kind of example, look beyond Sun's to the myriad of implementations used to demo various frameworks and tools. With very few exceptions, they're generally better written and make a lot more sense.

9. *J2EE is useless without extra frameworks.*

The first thing that most people do before embarking on any application, of any size or scope, is to try to use the chance to utilize any number of popular frameworks. In these days of framework mass hysteria, "naked" J2EE is perceived as being bereft of all functionality and quite useless out of the box.

Not so! While many larger applications would see a huge benefit in committing to a particular framework (be it open source or otherwise), smaller ones often do not merit the extra learning curve, maintenance cost, and configuration nightmares associated with many of these frameworks. Be flexible, consider your needs, and pick an intelligent solution that fits the problem at hand. If any of these frameworks was a true "one size fits all" solution, it'd be part of the spec.

This in fact does seem to have happened, with the introduction of JavaServer Faces. Unfortunately while well intentioned, the specification is far from adequate in its current incarnation. I am told that the expert group is aware of these issues and is hard at work addressing them.

“ Could it be that people don't know that every J2EE container is required to provide data sources that can be looked up?”

7. *API "X" is too complex/unclear; it's easier to roll your own instead.*

It's amazing how people choose to do their own connection pooling, or try to manage transactions "manually." Could it be that people don't know that every J2EE container is required to provide data sources that can be looked up? Do they not know that every container supports pooling out of the box, that more often than not is incredibly trivial and straightforward to set up?

While it's enjoyable to tinker with various open source projects and play at integration with them, invest the time up front to use these features as provided by your application server. Yes, it's a lot less sexy; yes, it's a lot more boring. However, it will save time in both the short and long term, reduce your maintenance, and eliminate the "keeping up with the Jones'" aspect of tracking third-party sources. J2EE mandates many of these services, including but not limited to transaction management and connection pooling.

Conclusion

What's our take-home lesson? Put succinctly, take everything you hear with a grain of salt! It's often easy to get caught up in the hype (both positive and negative), but to date, nobody has come up with a solution that absolves us developers from having to think, consider, and apply our core skill to all our decisions: problem-solving in innovative and efficient ways.

J2EE is an evolving spec, albeit at a slower pace than some would like. It has areas it will mature greatly in, and despite many assertions to the contrary, it's a mature platform backed by a strong specification and huge industry commitment. Investing in it might no longer seem to be glamorous or innovative, but that's the sign of a mature and well-established specification. The technologists might have moved on, but for the rest of us enterprise application developers, we have a powerful and compelling tool in our arsenal that's useful far more often than not. ☺