J2EE[tm] BluePrints: Questions and Answers:

Printed: Jan 22, 2003
From: http://java.sun.com/blueprints/qanda/ejb_tier/restrictions.html#sockets

Standard Version

Java™ BluePrints

Guidelines, Patterns, and code for end-to-end Java applications.

Questions and Answers - Enterprise JavaBeans Tier - EJB Restrictions

What restrictions are imposed on enterprise beans? Why are these restrictions in place?

The EJB container is responsible for managing system-related functionality such as security, threading, resource pooling, and so on. In order to control these facets of component operation, the container places certain restrictions on the components it manages.

There is quite a list of restrictions on what enterprise beans can do. The restrictions' underlying intent is to ensure the portability, scalability, and interoperability of the application. The restrictions are discussed in more detail below. Each restriction falls into one or more of several categories:

- ***Run-time distribution.*** Enterprise bean instances may be distributed; that is, running in separate Java virtual machines or on separate machines. Some restrictions are logical consequences of distributed programming semantics. For example, see the following discussion on static fields.
- ***Security.*** Many of the restrictions are imposed to maintain the integrity of the enterprise bean security model.
- ***Container control.*** The container is responsible for coordinating system services, controlling threads, managing security, bean lifecycle, and so on. Some restrictions prevent enterprise bean classes from interfering with proper container operation. For example, see the following discussion on threads.
- ***Server model.*** The EJB server, EJB container, and EJB component have clearly-specified roles in the enterprise bean programming model. Some of these restrictions prevent enterprise bean classes from usurping the responsibilities of the server or container. For example, see the following discussion on server sockets.

Specifically, enterprise beans should not:

- use the `java.lang.reflect` Java Reflection API to access information unavailable by way of the security rules of the Java runtime environment
- read or write nonfinal static fields
- use `this` to refer to the instance in a method parameter or result
- access packages (and classes) that are otherwise made unavailable by the rules of Java programming language
- define a class in a package
- use the `java.awt` package to create a user interface
- create or modify class loaders and security managers
- redirect input, output, and error streams
- obtain security policy information for a code source

- access or modify the security configuration objects
- create or manage threads
- use thread synchronization primitives to synchronize access with other enterprise bean instances
- stop the Java virtual machine
- load a native library
- listen on, accept connections on, or multicast from a network socket
- change socket factories in `java.net.Socket` or `java.net.ServerSocket`, or change the stream handler factory of `java.net.URL`.
- directly read or write a file descriptor
- create, modify, or delete files in the filesystem
- use the subclass and object substitution features of the Java serialization protocol

For more on these restrictions, see § 18.1.2 of the EJB 1.1 specification.

Are the restrictions mandatory? How are they enforced?

The restrictions are mandatory, since they are laid out in the specification as not optional. There are three basic types of enforcement of these restrictions.

- ***Compatibility Test Suite (CTS) and branding.*** Only servers that pass the CTS for the platform are allowed to brand themselves as J2EE servers. The purpose of this branding control is to ensure the highest level of compatibility between vendors' servers. The result is that vendors collaborate on designs, and then compete on implementations. The CTS and J2EE branding control indicate that the branded server does indeed implement the specification, including enforcing restrictions on enterprise beans.
- ***Pre-deployment checking tools.*** Enterprise development tools have varying levels of checking tools that ensure that enterprise beans being deployed are well-behaved (meaning they abide by the restrictions). Developers can then be sure they are not deploying enterprise beans that compromise server performance, security, or availability, etc.
- ***Security model restriction enforcement.*** The EJB specification defines that the EJB container implementation should enforce the restrictions described above by way of the standard Java security model. For example, an enterprise bean that tries to create an AWT `Frame` will fail, since the default security setting for the container denies `java.awt.AWTPermission`.

  Most EJB Container implementations will allow changes to the default security settings, so enterprise bean programming restrictions can usually be relaxed in specific cases. It's important to understand, though, the implications of violating the restrictions, by understanding why the restrictions are in place.

For more on the specifics of EJB container security, see section 18.2.1.1 of the EJB 1.1 specification; especially table 10.

Do the restrictions that apply to EJBs also apply to helper and dependent classes ?

Yes. From the point of view of the container, the enterprise bean and its helper classes form a single functional unit. If an enterprise bean were not allowed to, say, open a file in the filesystem, and yet had access to a helper class that was allowed to do so, the restriction on the bean would be meaningless. Therefore, EJB component programming restrictions also apply to any helper or dependent classes the bean may use.

Why can't I use nonfinal static fields in my enterprise bean?

Nonfinal static class fields are disallowed in EJBs because such fields make an enterprise bean difficult or impossible to distribute. Static class fields are shared among all instances of a particular class, but only within a single Java Virtual Machine (JVM™). Updating a static class field implies an intent to share the field's value among all instances of the class. But if a class is running in several JVMs simultaneously, only those instances running in the same JVM as the updating instance will have access to the new value. In other words, a nonfinal static class field will behave differently if running in a single JVM, than it will running in multiple JVMs. The EJB container reserves the option of distributing enterprise beans across multiple JVMs (running on the same server, or on any of a cluster of servers). Nonfinal static class fields are disallowed because enterprise bean instances will behave differently depending on whether or not they are distributed.

It is acceptable practice to use static class fields if those fields are marked as `final`. Since final fields cannot be updated, instances of the enterprise bean can be distributed by the container without concern for those fields' values becoming unsynchronized.

Why is thread creation and management disallowed?

The EJB specification assigns to the EJB container the responsibility for managing threads. Allowing enterprise bean instances to create and manage threads would interfere with the container's ability to control its components' lifecycle. Thread management is not a business function, it is an implementation detail, and is typically complicated and platform-specific. Letting the container manage threads relieves the enterprise bean developer of dealing with threading issues. Multithreaded applications are still possible, but control of multithreading is located in the container, not in the enterprise bean.

Why can an enterprise bean not listen to or accept connections on a socket?

Because if an enterprise bean is listening on a socket, it can't be passivated -- it must always be available. Enterprise beans *can* be network socket clients, and so they can use other network resources (including other enterprise bean servers) to do their jobs. Just as with a database connection, don't hang on to open client sockets across method calls; instead, open them, communicate through the socket, and close it before returning from the method.

Why can't EJBs read and write files and directories in the filesystem? And why can't they access file descriptors?

Enterprise beans aren't allowed to access files primarily because files are not transactional resources. Allowing EJBs to access files or directories in the filesystem, or to use file descriptors, would compromise component distributability, and would be a security hazard.

Another reason is deployability. The EJB container can choose to place an enterprise bean in any JVM, on any machine in a cluster. Yet the contents of a filesystem are not part of a deployment, and are therefore outside of the EJB container's control. File systems, directories, files, and especially file descriptors tend to be machine-local resources. If an enterprise bean running in a JVM on a particular machine is using or holding an open file descriptor to a file in the filesystem, that enterprise bean cannot easily be moved from one JVM or machine to another, without losing its reference to the file.

Furthermore, giving EJBs access to the filesystem is a security hazard, since the enterprise bean could potentially read and broadcast the contents of sensitive files, or even upload and overwrite the JVM runtime binary for malicious purposes.

Files are not an appropriate mechanism for storing business data for use by components, because they tend to be unstructured, are not under the control of the server environment, and typically don't provide distributed transactional access or fine-grained locking. Business data is better managed using a persistence interface such as JDBC, whose implementations usually provide these benefits. Read-only data can, however, be stored in files in a deployment JAR, and accessed with the `getResource()` or `getResourceAsStream()` methods of `java.lang.Class`.

Why isn't AWT available from within an enterprise bean?

Because EJBs are intended to be business-functionality-specific server extensions, not clients with user interfaces. The purpose of an enterprise bean is to perform some service on the server in response to a service request. This is a separate function from managing user interaction. Note that AWT and JFC/Swing may still be used to create user interfaces that access enterprise beans through a remote enterprise bean reference.

Why is there a restriction against using the Java Reflection APIs to obtain declared member information that the Java language security rules would not allow? Doesn't Java automatically enforce those rules?

Contrary to common belief, most of the Java Reflection API can be used from EJB components. For example, `loadClass()` and `invoke()` can both be used by enterprise beans. Only certain reflection methods are forbidden.

This restriction refers to the enabling of the security permission `ReflectPermission`. The `suppressAccessChecks` permission target name, when enabled, allows a class to use reflection to inspect, access, and modify protected and private members and methods in other classes. Obviously, if EJB components are using private or protected members or methods to manage sensitive information, this facility could be used to violate security mechanisms. Therefore, the EJB specification explicitly restricts usage of `suppressAccessChecks`, to prevent the security hole that would result. Denial of `ReflectPermission` is part of the standard security policy for an EJB container.

Why all the restrictions on creating class loaders and redirection of input, output, and error streams?

Class loading is allowed, but creating custom loaders is not, for security reasons. These restrictions exist because the EJB container has responsibility for class loading and I/O control. Allowing the EJB to perform these functions would interfere with proper operation of the Container, and are a security hazard.

The Java Pet Store has code that loads classes from inside an enterprise bean class using `Class.forName()`, in `StateMachine`.

Why can't I load native code?

Native code, if allowed in enterprise beans, could cause a host of problems, including but not limited to:

- ***Loss of portability.*** Using native code implies implementing that native code on every platform you use, or ever will use. Using native code in enterprise beans would tie those beans to the platforms on which the code runs.
- ***Loss of system stability.*** Native code is typically written in C, and so will suffer from the liabilities of C programs, such as pointer errors, memory leaks, invalid memory references, and so forth. Furthermore, the code is likely to be less stable than the JVM, if only because the JVM code base is likely to be more mature. If enterprise beans were allowed to use native code, a native code crash would probably bring the application server down.
- ***Loss of scalability.*** Remember that enterprise beans are multithreaded. If enterprise beans were allowed to use native code, that code would either have to be entirely reentrant, with thread management code at appopriate places; or access to the native calls would have to be synchronized by the Container. Long-running native calls would lock enterprise bean instances into memory, preventing them from being candidates for passivation.
- ***Compromised security.*** The EJB container is somewhat like the applet "sandbox", in that its security policies define what the contained instances can do. One of the things the container

> does is prevent enterprise beans from scribbling on the filesystem anywhere they please. Native code has no such restriction. So, for example, if someone breaks the deployment security in your enterprise bean environment, they have full access to your filesystem. They can read files, read directories, broadcast file contents, or even download and replace the server's JVM with a security-compromised (or worse) runtime executable to commit mayhem or simply make other security breaches easier to create.

How does the container prevent native code from being called? The container has a `SecurityManager` whose standard policies (section § 18.2.1.1 of the specification, Table 10) don't set a `checkLink` permission for any native library. So, attempts at executing the `load()` or `loadLibrary()` methods of `java.lang.Runtime` result in a `SecurityException`.

The EJB specification states:

"Some Containers may allow the Deployer to grant more, or fewer, permissions to the enterprise bean instances than specified in Table 10. Support for this is not required by the EJB specification. Enterprise beans that rely on more or fewer permissions will not be portable across all EJB Containers."

This means that, if your container will let you grant permission to run native code, and you don't mind that your enterprise bean component is not portable not only by platform, but also possibly not by server, then you can run native methods in enterprise beans. Just be aware that you're locking yourself into a box, and taking on the liabilities (problems with security, stability, portability, scalability, etc.) that use of native code implies. we strongly recommend against using native code in enterprise beans.

By the way, it won't help to have your enterprise bean defer native calls to other classes, because instances of those classes will running in the same container, and so will share the enterprise bean's security restrictions. The enterprise bean specification states explicitly that helper classes of enterprise beans are under the same restrictions as enterprise beans themselves.

If you absolutely must have access to native code in your system, but want to keep your enterprise beans portable, consider wrapping the native code as an RMI technology-enabled object, and then using the service from your enterprise bean. This solution won't solve the security problem, and may still affect scalability, but at least if the native code crashes, it won't bring your entire application server down. Also, EJB servers from different vendors running on multiple platforms can use this new service. When in some future time portability becomes an issue (as it invariably does, for systems that make it into production), your enterprise bean functionality will port to new app servers or platforms with a minimum of pain, and you can focus on reimplementing native functionality as necessary.
