Using Native Libraries inside SAP NetWeaver Application Server Java

Applies to:

SAP NetWeaver Application Server Java 7.1 and higher.

Summary

This document describes how SAP customers can use and deliver native libraries in AS Java and use them from Java EE applications.

Authors: Pavel Genevski, Katya Todorova

Company: SAP

Created on: 10 December 2010

Authors Bio

Pavel Genevski is a Java/C++ developer with more than 7 years of experience. Throughout his career, he has worked on various projects, ranging from software for microcontrollers to CRM and ERP solutions. Currently, he is a Senior Java developer in the Technology Development Group at SAP, with responsibilities in the core server runtime and deployment.

Katya Todorova has been with SAP for 3 years. Her main interest is in the areas of class loading and components lifecycle management in SAP NetWeaver Application Server.

Table of Contents

Problem definition

If you are reading this document you probably have a good reason to use native libraries with all their powers and inconveniences. If in doubt, refer to the JNI specification, section 1.4 "When to use JNI".

What is a native library?

The term "native library" comes from the Java Native Interface (JNI) specification and means an operating system (OS) specific library, like a Windows DLL or Linux/Unix shared library (shared object). Java byte code running inside a Java Virtual Machine (JVM) can access code from native libraries, in a vendor neutral way, through JNI. The latter provides special means that allow Java and native (e.g. C++) code to interoperate and call each other safely, without violating the Java memory model.

Identifying native libraries

In both direct JNI and dynamic invocation, a native library is identified by its canonical path on the file system.

How does Java byte code call native functions?

There are two ways to do that, direct JNI and dynamic invocation through "shared stubs" (which indirectly relies on direct JNI as well).

Direct JNI

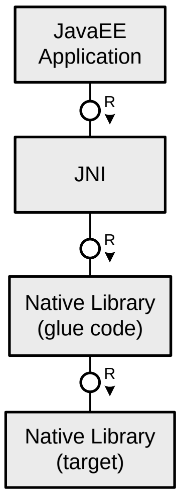


Figure 2 – Direct JNI

Using this approach, a developer has to write one native method per function they want to call. Here's what the process is:

1. Declare one or more native methods in a class. This Java code is the entry point to native code.
2. Run the javah tool on the compiled class in order to generate function headers (one or more files ending with .h and containing symbol declarations according to the JNI specification naming conventions).
3. Implement the functions from the headers in C/C++ or other language that allows creation of native libraries. (Native libraries can be implemented in any language as long as the symbols, declared in the header file are exported). The implementation is typically just glue code that further calls an existing native library and takes care of the parameter conversion between Java and native code, locking and releasing arrays in memory and vice versa.
4. Use a static block in the Java class in order to load the library:

- System.loadLIbrary() is recommended, because it adds the system depended prefixes and suffixes (such as *.dll and lib*.so) and searches the library in the predefined library path (system property named java.library.path)
- System.load() provides more flexibility because developers may specify an absolute path from which the library is loaded. It has it burdens though, because the path is system depend and the exact, OS specific file name has to be used (for example libfoo.so instead of just foo).

5. Call the native method.

This approach is tedious and requires a lot of coding. Some optimizations could be applied along the way, so that one native method can invoke several functions of the target native library, based on the parameters passed in. This idea must have ultimately led to the second approach – dynamic invocation.

Dynamic invocation (shared stubs)

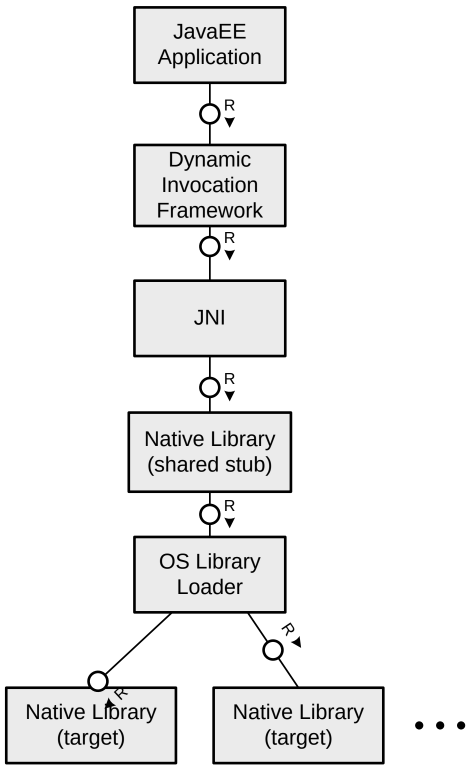


Figure 3 – Dynamic invocation

With dynamic invocation, developers use a framework that dispatches Java calls to the target native libraries without the need of native method declarations in user code. The framework itself must have a native library (shared stub) that is accessed through direct JNI. This native library then uses the OS library loader to load and invoke the target native libraries, as specified by the user.

One of the popular libraries for dynamic invocation is JNA (see the references section). Here is how JNA works from the developer's perspective:

1. Specify the name of the library as a string.
2. Declare (i.e. mirror) the native function signature in a Java interface.
3. Use the JNA API to obtain a reference to the library.
4. Call the library's functions.
5. Unload the library.

Dynamic invocation is more convenient than direct JNI, because it doesn't require any native (glue) code and does the parameter conversion for you. It comes at some performance price of course.

Life cycle of native libraries inside the JVM

As a general rule, a library may be loaded only once per OS process. However, this works differently depending on the invocation mechanism in use.

Direct JNI

The class loader of the class that calls System.load* becomes the "defining class loader" of the native library.

An attempt to load the library more than once from classes with the same defining class loader will not result in an error but will silently return. This is because the first time a library is loaded successfully the class loader is associated with it and all subsequent calls to load the same library are ignored.

An attempt to load the library more than once from classes with different defining class loaders results in an UnsatisfiedLinkError. This restriction has been in place since Java 1.2. According to the JNI specification 1.2 it is needed to ensure type safety between class loaders and non-ambiguous synchronization semantics.

A library is unloaded when its defining class loader is garbage collected. This can only happen after all the classes defined by this class loader have been garbage collected as well. Unfortunately garbage collection cannot be enforced in a vendor neutral way (if possible at all), so it cannot be guaranteed when a native library will be unloaded.

Dynamic invocation

Since native libraries are not loaded with the System.load* group of methods but with a framework specific API, the life cycle of these libraries depends on the particular framework. However, it eventually boils down to the following:

In order to load the library, the user invokes some framework specific method which results in the library being loaded by the OS library loader into the JVM process.

An attempt to load the library twice doesn't normally result in an error regardless of the initiating class loader, though it might be framework specific. This is so, because the OS library loader returns a pointer to the already loaded library instead of raising an error, should a subsequent attempt be made to load it within the same process. Frameworks are free to further restrict loading in whatever way they see fit.

A library may be unloaded either explicitly or by the garbage collector:

- Explicit unloading – the fact that the framework uses the OS library loader makes it possible for the user to close (i.e. dispose) a library explicitly much like they close file handles. This is usually the preferred way because it is more deterministic within the time domain and allows resources to be freed up immediately.
- Garbage collection – the framework may unload the library when the owning class/class loader is garbage collected. This might be the default or the backup mechanism (in case the developer forgets to dispose the library explicitly). It could be implemented with either finalizers or phantom references. In both cases the framework will try to hook on the lifecycle of an object that is related to the library and unload the library when the object is garbage collected.

In both cases, if the library is stateful, its state will be shared between all the callers inside the JVM. Also, if there are concurrent requests to the library they have to be synchronized in Java code if the library is not thread safe. From this perspective direct JNI is slightly better, because it guarantees that the library is owned by its defining class loader.

Deployment of native libraries

Delivery

As outlined above, the user may either specify the absolute path to the library being loaded with System.load() or rely on the default search path with System.loadLibrary(). Let's have a closer look at the two different options and the implications that they bring to the server environment.

Absolute path – this implies that the user code will have to be hooked to the deployment process so that it can unpack the native library out of the deployment archive and put it on the right place on the file system. Moreover, user code will have to construct the proper, system dependent library file name (e.g. mylib.dll or libmylib.so). There are two problems with this approach:

- User code, running inside a JavaEE server is not allowed to access the file system. Normally servers are restricted just to a portion of the file system in order to avoid security breaches. Moreover, even if user code is able to get write access to a portion of the file system (e.g. the temp folder), the file might be deleted afterwards by a cleanup procedure.
- Normally, user code cannot be hooked to the deployment process inside a JavaEE server. In reality a lot of application developers use the init method of a Servlet in combination with some file system indexes or timestamps in order to execute some logic once per deployment. Therefore, such an implementation of delivery is possible in general.

Default search path – this works as long as the libraries are delivered to the JVM library search path (java.library.path) or the OS library search path (i.e. PATH on Windows and LD_LIBRARY_PATH on Linux), so that they can be loaded by the JVM process with System.loadLIbrary(). The only thing needed for this delivery method is that AS Java recognizes native libraries inside deployment archives and delivers them accordingly. AS Java supports this out-of-the-box for deployment components with software type "primary-library". Native libraries that are packed inside primary libraries with the respective descriptor are delivered to "java.library.path".

Redeployment

While initial deployment seems straightforward, redeployment poses some problems, related to the lifecycle of the library. The old version of the library has to be unloaded before the new one can be loaded. If not, the following problems may occur:

If the JVM is holding a lock on the library file, it may be impossible to delete the file in order to replace it with its new version. This is typical for Microsoft Windows operating systems. On POSIX compliant OS (Linux etc) it is normally not a problem because the old version of the library will remain loaded into the JVM process (e.g. by inode) even though it's been deleted from the file system and the new version will be successfully written.

If the library has been updated on the file system but not unloaded by the JVM this might lead to the following consequences:

- In case of "Direct JNI" an UnsatisfiedLinkError will be thrown to the caller that tries to load the library from the new class loader.
- In case of "Dynamic invocation" the callers will see the old functionality, because the OS library loader will return a pointer to the old version of the library (which is already loaded).

Out of this, we can define two strategies for native library life cycle during redeployment:

- JVM bound life cycle - lifecycle of the library is bound to the lifecycle of the JVM and it cannot be reloaded without restarting the JVM. This is the safest option, because it is guaranteed that the library will be unloaded. The downside is more TCO in case the native library has to be updated frequently (which is not a common use case).
- Application bound life cycle – if we suppose that the native library is delivered with an application archive (currently not supported by AS Java), then the lifecycle of the library is bound to the lifecycle of the application with which it is delivered. That is, the library is reloaded while the JVM is running. As outlined above, the problem here is to make sure that the library has been unloaded. One

possible solution might be to register a phantom reference on the defining class loader in order to wait until it is garbage collected and call System.gc() in order to give the JVM a hint to do the garbage collection. While this might work in some environments, there is always a risk that the garbage collection will never happen or will happen in intolerable amount of time.

It is essential to ensure that a library is unloaded before an update is triggered in order to ensure successful and consistent, platform independent redeployment.

The user perspective – Use cases

From the user's perspective, we can distinguish the following use cases:

Initial Deployment

The native library does not exist on the system. Hence application code cannot access it. Then the user deploys a deployment archive containing the native library. After successful deployment, the library shall become available to application code so that the application can call the functions in the library.

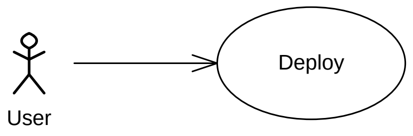


Figure 4 – Initial deployment of a native library

Usage

Provided that the library is successfully deployed to AS Java, application code shall be able to call its functions using both Direct JNI and Dynamic invocation.

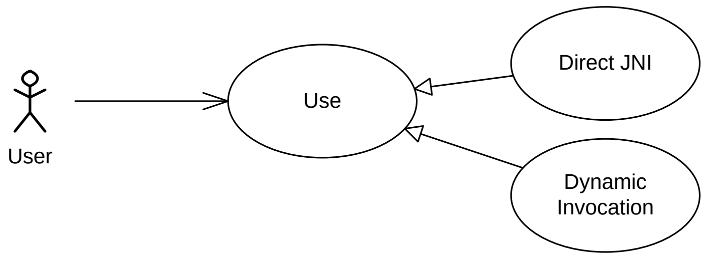


Figure 5 – Usage of the library

Redeployment (Update)

The user shall be able to update the native library with newer versions. After successful redeployment, the old library shall be unloaded and the new one shall be able to respond to function calls coming from an application

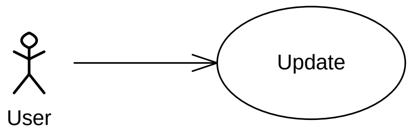


Figure 6 – Redeployment (update) of the native library

Undeployment

The user shall be able to undeploy the native library. As a result of this the library shall disappear from the library search path and application code shall not be able to access it.

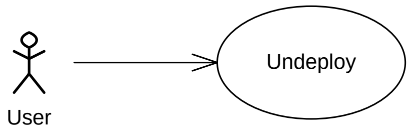


Figure 7 – Undeployment of the native library

Possible solutions

Delivery with an application

The first thing that comes to mind is to package the native library within the application itself. Here is how it looks:

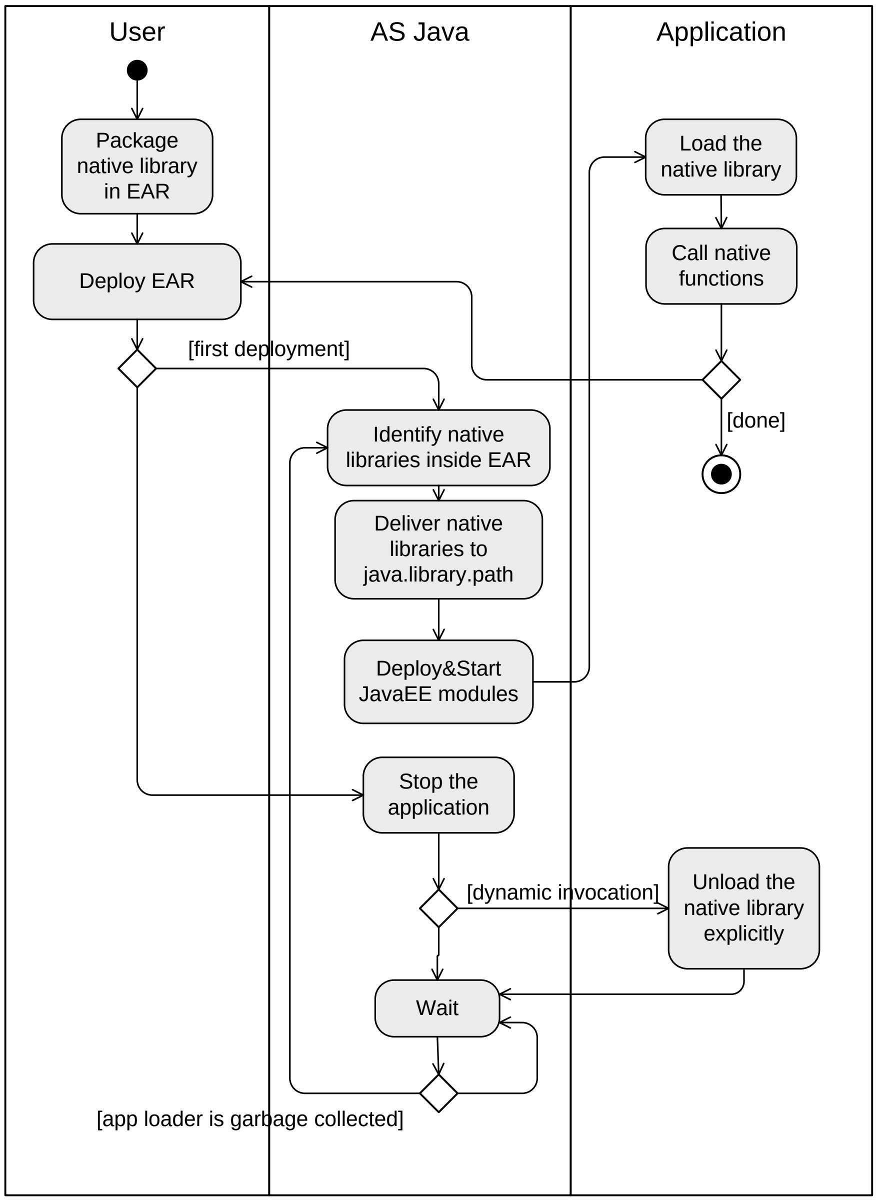

User
AS Java
Application
Package native library in EAR
Deploy EAR
[first deployment]
Identify native libraries inside EAR
Deliver native libraries to java.library.path
Deploy&Start JavaEE modules
Load the native library
Call native functions
[done]
Stop the application
[dynamic invocation]
Unload the native library explicitly
Wait
[app loader is garbage collected]

Figure 8 – Delivery of a native library with an application

As shown on the activity diagram:

The user has the responsibility to package the native library inside an EAR. This may include the creation of SAP specific deployment descriptors (e.g. META-INF/SearchRules.xml).

AS Java inspects the archive for native libraries and their respective descriptors and deliver them to "java.library.path".

The application is loading the native library explicitly with System.load* or using a dynamic invocation framework API

Upon redeployment

- The application is supposed to explicitly unload the library in case of dynamic invocation or do nothing in case of System.load*.
- AS Java has to wait until the application class loader has been garbage collected in order to proceed with the redeployment.

Pros

Customers can use standard JavaEE EAR archives - this approach does not require exposing some SAP proprietary archive formats or tools to customers. However, customers have to use an SAP proprietary deployment descriptor (e.g. META-INF/SearchRules.xml).

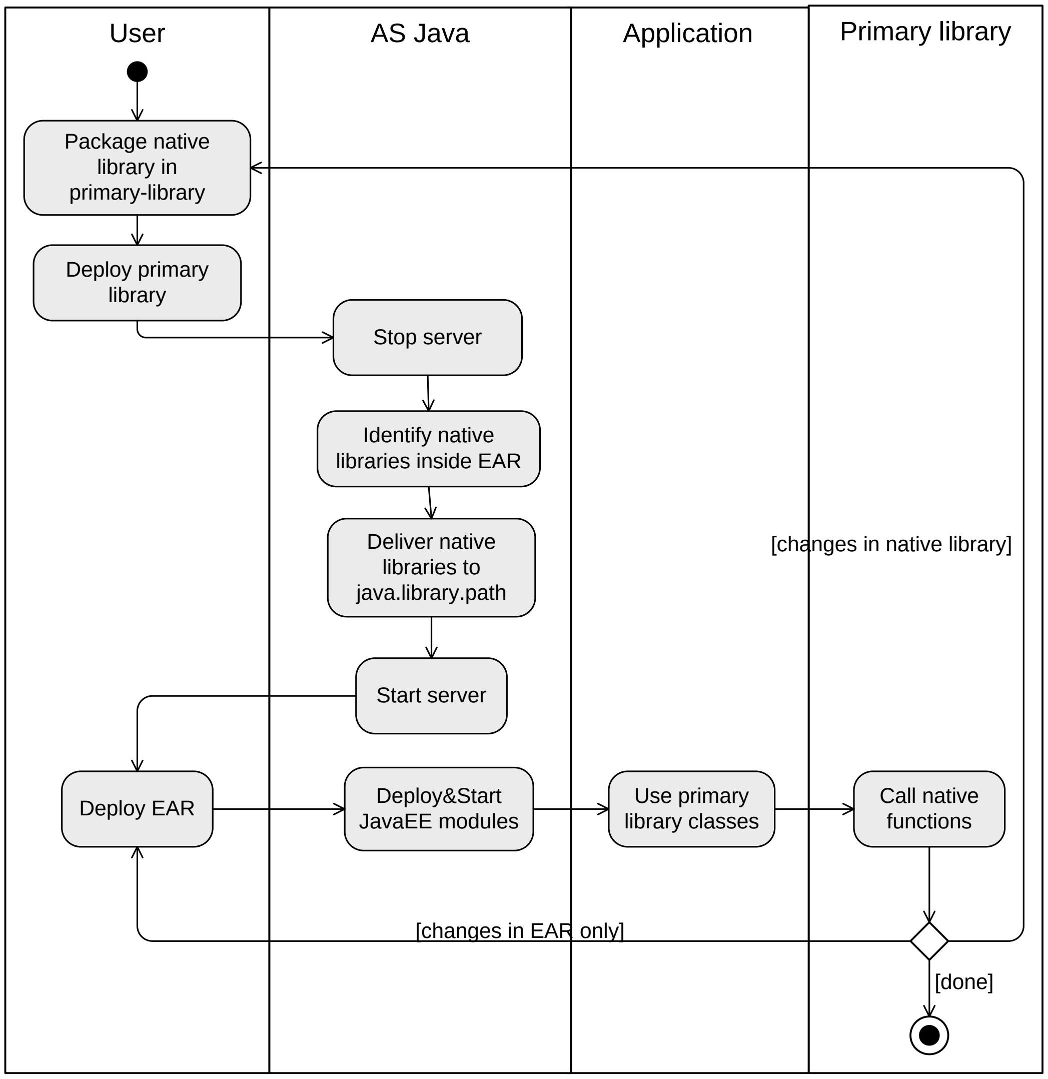
Redeployment of the EAR doesn't require a restart of the server - this benefit is applicable only to the development use case. SP updates restart the server anyway. Also, there is normally no need to update a native library during JavaEE application development. Usually JavaEE developers get a native library from third parties and just use the library inside an application, while modifying the application itself.

Cons

The trickiest part here is to wait until the application class loader has been garbage collected so that it is guaranteed that the native library is unloaded. As already described, there is no platform independent way to enforce garbage collection; hence AS Java might wait forever. If AS Java does not wait for the application class loader to be garbage collected, redeployment might fail or the new version of the application might see the old library instead of the new one. This limitation might actually outweight the advantage that a restart of the server is not required.

Delivery with a primary library

Within AS Java, there is an already existing mechanism for deployment of native libraries inside deployment archives of software type primary-library.


User
AS Java
Application
Primary library
Package native library in primary-library
Deploy primary library
Stop server
Identify native libraries inside EAR
Deliver native libraries to java.library.path
[changes in native library]
Start server
Deploy EAR
Deploy&Start JavaEE modules
Use primary library classes
Call native functions
[changes in EAR only]
[done]

The user packages the native library in an SDA file with software type primary-library and deploys it.

> The location of the native libraries within the SDA and their target platform (e.g. windows 32 bit) must be specified in META-INF/SearchRules.xml inside the SDA. More details on the format are given at help.sap.com.
> http://help.sap.com/saphelp_webas630/helpdata/en/b5/22123b8d92294fac207283f3e8756e/content.htm
>
> In case of direct JNI the user declares native methods in classes within the primary library SDA.

AS Java deploys the primary library SDA and delivers the native libraries to “java.library.path”. This works out-of-the-box.

The user deploys the EAR file with the JavaEE application as usual. If the application is changed, the user should redeploy only the EAR (without server restart). If the primary library is changed, it should be redeployed. This requires server restart since the library is primary.

The application uses the native library through the classes in the primary library or directly through a dynamic invocation framework.

Pros

Predictable native library lifecycle – since there is a restart of the JVM it is guaranteed that the native library will be unloaded and the new one will be loaded afterwards. Also, developers have clear notion that native library lifecycle is not bound to application lifecycle.

Cons

Redeployment of a native library causes a server restart. This might not be an issue given the fact that native libraries are not updated that often as compared to the application that uses them.

Customers have to create primary libraries. Since this format does not differ very much from the EAR application format this might not be an issue.

Related Content

JNI Specifications

JNA

Valery Silaev’s blog about creation of libraries

Packaging Native Libraries Inside SAP NetWeaver AS Java

Copyright