Using Java 9 Modularization to Ship Zero-Dependency Native Apps

Posted on October 20, 2017 | Steve Perkins

"Why can't I just build an .EXE?"

When Java was first introduced, mainstream programming languages mostly either compiled to standalone executables (e.g. C/C++, COBOL), or else ran in an interpreter (e.g. Perl, Tcl). For many programmers, Java's need for both a bytecode compiler *and* a runtime interpreter was a shift in thought. The compilation model made Java better suited for business programming than "scripting" languages. Yet the runtime model required a suitable JVM to be deployed and available on each target machine.

People bristled at this somewhat (at least *I* remember doing so!). Early web forums, and later StackOverflow questions, were full of developers looking for some way to ship their Java applications as "native" executables. To avoid the need for a Java runtime to be installed on the target machine prior to deployment.

There have been solutions from nearly the beginning. Excelsior JET (https://www.excelsiorjet.com/) is an ahead-of-time (AOT) Java compiler, providing a more-or-less C++ style experience. However, with licensing costs in the thousands of dollars, it has always been a niche option. On the free-as-in-beer end, there is Launch4j (http://launch4j.sourceforge.net/), and JDK 8's javapackager (https://docs.oracle.com/javase/8/docs/technotes/guides/deploy/packager.html) tool. These allow you to bundle a Java Runtime Environment, with a launcher executable for starting your app with that JRE. However, embedding a JRE adds roughly 200 megabytes. It's difficult to trim that down, due to technical reasons as well as tricky licensing issues.

Along Comes Java 9

The most publicized new feature in Java 9 is the new modularization system, known as Project Jigsaw (http://openjdk.java.net/projects/jigsaw/). The full scope of this warrants many blog articles, if not complete books. However, in a nutshell, the new module system is about isolating chunks of code and their dependencies.

This applies not only for external libraries, but even the Java standard library itself. Meaning that your application can declare which *parts* of the standard library it really needs, and potentially *exclude all the other parts*.

This potential is realized through the `jlink` tool that now ships with the JDK (http://openjdk.java.net/jeps/282). At a superficial first glance, `jlink` is similar to `javapackager`. It generates a bundle, consisting of:

1. your application code and dependencies,
2. an embedded Java Runtime Environment, and
3. a native launcher (i.e. bash script or Windows batch file) for launching your application with the embedded JRE.

However, `jlink` establishes "link time" as a new optional phase, in between compile-time and run-time, for performing optimizations such as removing unreachable code. Meaning that unlike `javapackager`, which bundles the entire standard library, `jlink` bundles a stripped-down JRE with only those modules that your application needs.

A Demonstration

The difference between `jlink` and it's older alternatives is striking. To illustrate, let's look at a sample project:

https://github.com/steve-perkins/jlink-demo (https://github.com/steve-perkins/jlink-demo)

(1) Create a modularized project

This repo contains a multi-project Gradle build. The `cli` subdirectory is a "Hello World" command-line application, while `gui` is a JavaFX desktop app. For both, notice that the `build.gradle` file configures each project for Java 9 compatibility with this line:

```
sourceCompatibility = 1.9
```

This, along with creating a `module-info.java` file, sets up each project for modularization.

/cli/src/main/java/module-info.java:

```
module cli {
}
```

/gui/src/main/java/module-info.java:

```
module gui {
    requires javafx.graphics;
    requires javafx.controls;

    exports gui;
}
```

Our CLI application is just a glorified `System.out.println()` call, so it depends only on the `java.base` module (which is always implicit, and needs no declaration).

Not all applications use JavaFX, however, so our GUI app must declare its dependency on the `javafx.graphics` and `javafx.controls` modules. Moreover, because of the way that JavaFX works, the low-level library needs access to our code. So the module's `exports gui` line grants this visibility to itself.

It's going to take some time for Java developers (myself included!) to get a feel for the new standard library modules and what they contain. The JDK includes a `jdeps` tool (https://docs.oracle.com/javase/9/tools/jdeps.htm#JSWOR690) that can help with this. However, once a project is setup for modularization, IntelliJ is great at recognizing missing declarations and helping auto-complete them. I assume that if Eclipse and NetBeans don't have similar support already, then they soon will.

(2) Build an executable JAR

To build a deployable bundle with `jlink`, you first want to package up your application into an executable JAR file. If your project has third-party library dependencies, then you'll want to use your choice of "shaded" or "fat-JAR" plugins to generate a single JAR with all dependencies included. In this case, our examples use only the standard library. So building an executable JAR is a simple matter of telling Gradle's `jar` plugin to include a `META-INF/MANIFEST.MF` file declaring the executable class:

```
jar {
    manifest {
        attributes 'Main-Class': 'cli.Main'
    }
}
```

(3) Run jlink on it

As far as I know, Gradle does not yet have a plugin offering clean and seemless integration with `jlink`. So my build scripts use an `Exec` task to run the tool in a completely separate process. It should be easy to follow, such that you can tell the command-line invocation would look like this:

```
[JAVA_HOME]/bin/jlink --module-path libs:[JAVA_HOME]/jmods --add-modules cli --launcher
cli=cli/cli.Main --output dist --strip-debug --compress 2 --no-header-files --no-man-pa
ges
```

- The `--module-path` flag is analogous to the traditional CLASSPATH. It declares where the tool should look for compiled module binaries (i.e. JAR files, or the new JMOD format). Here, we tell it to look in the project's `libs` subdirectory (because that's where Gradle puts our executable JAR), and in the JDK directory for the standard library modules.
- The `--add-modules` flag declares which modules to add to the resulting bundle. We only need to declare our own project modules (`cli` or `gui`), because the modules that *it* depends on will be pulled in as transitive dependencies.
- The resulting bundle will include a `/bin` subdirectory, with a bash script or Windows batch file for executing your application. The `--launcher` flag allows you to specify a name for this script, and which Java class it should invoke (which seems a bit redundant as this is already specified in the an executable JAR). Above, we are saying to create a script named `bin/cli`, which will invoke the class `cli.Main` in module `cli`.
- The `--output` flag, intuitively enough, specifies a subdirectory in which to place the resulting bundle. Here, we are using a target directory named `dist`.
- These final flags, `--strip-debug`, `--compress 2`, `--no-header-files`, and `--no-man-pages`, are some optimizations I've tinkered with to further reduce the resulting bundle size.

At the project root level, this Gradle command builds and links both sub-projects:

```
./gradlew linkAll
```

The resulting deployable bundles can be found at:

```
[PROJECT_ROOT]/cli/build/dist
[PROJECT_ROOT]/gui/build/dist
```

Results

Let's take a took at size of our linked CLI and GUI applications, with their stripped-down embedded JRE's:

App	Raw Size	Compressed with 7-zip
`cli`	21.7 MB	10.8 MB
`gui`	45.8 MB	29.1 MB

This is on a Windows machine, with a 64-bit JRE (Linux sizes are a bit larger, but still roughly proportionate). Some notes:

- For comparison, the full JRE on this platform is 203 megabytes.
- A "Hello World" CLI written in Go compiles to around 2 MB. Hugo (https://gohugo.io/), the website generator used to publish this blog, is a 27.2 megabyte Go executable.
- For cross-platform GUI development, a typical Qt or GTK application ships with around 15 megs of Windows DLL's for the GUI functionality alone. Plus any other shared libs, for functionality that Java provides in its base standard library. The Electron quick-start example (https://github.com/electron/electron-quick-start) produces a 131 MB deliverable.

Conclusion

To be fair, an application bundle with a launch script is not quite as clean as "*just building an .EXE*", and having a single monolithic file. Also, the JRE is comparatively sluggish at startup, as its JIT compiler warms up.

Even so, Java is now at a place where you can ship self-contained, zero-dependency applications that are comparable in size to other compiled languages (and superior to web-hybrid options like Electron). Also, Java 9 includes an experimental AOT compiler (http://openjdk.java.net/jeps/295), that might eliminate sluggish startup. While only available for 64-bit Linux initially, this `jaotc` tool will hopefully soon expand to other platforms.

Although Go has been very high-profile in the early wave of cloud infrastructure CLI tools (e.g. Docker (https://www.docker.com/), Kubernetes (https://kubernetes.io/), Consul (https://www.consul.io/), Vault (https://www.vaultproject.io/), etc)... Java is becoming a strong alternative, especially for shops with established Java experience. For cross-platform desktop GUI apps, I would argue that JavaFX combined with Java 9 modularization is hands-down the best choice available today.

Tags: #Programming (https://steveperkins.com//tags/programming/)

← PREVIOUS POST (HTTPS://STEVEPERKINS.COM/MIGRATING-PROJECTS-FROM-GITHUB-TO-GITLAB/)

Reika • 2 years ago

So 22 megabytes for a "hello world". This is literally inbredible.

You can fit a whole OS with GUI in less than that: http://distro.ibiblio.org/t...

16 | 5 • Reply • Share ›

Tonio Loewald ➚ Reika • 2 years ago

Hello world using Electron is far larger. (Yeah, it's still pretty awful, but then so was 92kB for hello world in Modula-2 on a Sun Workstation in 1987.)

5 | • Reply • Share ›

Reika ➚ Tonio Loewald • 2 years ago

Electron is far more useful though. What is this comparison even. "My hello world is 5 GB, but then again so is Windows 10 installation."

4 | • Reply • Share ›

Jasper Siepkes ➚ Reika • a year ago

> Electron is far more useful though.

It all depends on what your doing. For example with Electron I can't even contact a gRPC service.

Besides this exercise is about the ability of a single language with frameworks (UI, networking, IO, threading, crypto, etc.) with which you can generate a standalone executable (ie. not requiring a runtime being installed on the users PC) for 3 different platforms (Windows, OS X and Linux) in about 25MB.

I am really interested in hearing about other solutions which give you that. I suspect even QT will have a hard time since you will need to include a statically linked QT runtime.

1 | • Reply • Share ›

PAUL Miller ➚ Tonio Loewald • 2 years ago • edited

Electron is an abomination written by and used by people who *probably* don't

understand the underlying architecture. If they had a clue, they wouldn't do this to themselves or their potential users.

2 ^ | v 1 • Reply • Share ›

Tonio Loewald ➚ PAUL Miller • 2 years ago

It's no different from sticking the Java stack in your executable because you don't want to force your user to install some specific version of Java — something that plenty of Java applications do.

^ | v • Reply • Share ›

PAUL Miller ➚ Tonio Loewald • 2 years ago

Hello World is ~150 bytes in assembly (http://timelessname.com/elf...

^ | v • Reply • Share ›

Tonio Loewald ➚ PAUL Miller • 2 years ago

Really? 'Hello, world' is 12 bytes. It takes 138 bytes to return it? How inefficient.

Hello world in bash is smaller :-)

3 ^ | v • Reply • Share ›

int19h ➚ Tonio Loewald • 2 years ago

It's 20 bytes if you use the appropriate platform, which would be DOS in this case. This would be the entire binary (save as .COM).

B4 09 BA 08 01 CD 21 C3 48 65 6C 6F 20 57 6F 72 6C 64 21 24

Unpacked:

MOV AH, 09h ; B4 09
MOV DX, 0108h ; BA 08 01
INT 21h ; CD 21
RET ; C3

and then the bytes for "Hello, world!$".

Bash is technically shorter still, since echo "..." is 7 bytes + string, and the string doesn't need the terminating $. But then you also need shebang for your script to be an executable binary in and of itself, so... ~

^ | v • Reply • Share ›

sulfide ➚ PAUL Miller • 2 years ago

YA BRO I THINK WE SHOULD HEAD BUTT NOW TO COMPLETE OUR AWESOMENESS

1 ^ | v 1 • Reply • Share ›

commenter ➚ Tonio Loewald • 2 years ago

Yeah, thats why Electron sucks (not mentioning the security implications). But most of the time the app is less than 1 MB and for some reason the devs think that their user might not have a browser yet and decide to package a browser with their app.

Reasonable people use the already installed browser and build PWAs nowadays.

^ | v • Reply • Share ›

Tonio Loewald ➚ commenter • 2 years ago

An Electon app has access to the file system and can run bundled Native code and make use of the the command line etc. The 100MB overhead it pretty horrible, but take a look at the size of some of the native apps these days. Disk space is the wrong thing to optimize for.

2 ^ | v • Reply • Share ›

Avatar This comment was deleted.

Sepiano ➚ Guest • 2 years ago

"I guess you could easily use Inno Setup to make an installer for the program"

I'm pretty raw to packaging and did manage to produce a working jlink version of the program I'm writing. I'm wondering if there is a tutorial around that shows this "easy" process, as it applies specifically to the files created by jlink. I would find this very helpful.

^ | v • Reply • Share ›

Avatar This comment was deleted.

Sepiano ➚ Guest • 2 years ago • edited

Thanks for responding!

"If you end up with an executable that simply runs your program, you can then use Inno to do the actual installation."

That's where I'm running into problems. I'm able to generate a working executable with JLINK. It is a couple levels below a file folder named "dist". The executable requires elements in the surrounding file structure (with dist as the parent folder) in order to run. It isn't clear to me, from the Inno Setup documentation, how to present this structure for packaging. I'm going to try and take another dive into their instructions, but it sure would be nice to have a direct example.

The javapackaging examples I've come across are for Java 8, and package a complete JRE, not a customized runtime. I've not had any success trying to adapt those examples to the reduced-runtime case.

^ | v • Reply • Share ›

Avatar This comment was deleted.

Sepiano ➚ Guest • 2 years ago

Thank you again!

I misplaced the link to this thread--I meant to post and let you know that I finally figured out that the choice "Inno Setup Compile" on my start menu would launch a Wizard for generating a configuration file. I have been so

mired in command-line level and xml-file type editing (Ant mostly) that the notion that there was a Wizard to set this all up went right past me. I thought I had to take one of their examples and edit it. Doh!

Your instructions are good, and I appreciate your taking the time to write them. I also added another step. The executable was a .bat file in a /bin subfolder, so I wrote another .bat to put in the top level folder to simply call this one.

It is working, to my great relief. Some details to work out still (fussing with image vs download time tradeoffs, does the command-prompt shell have to open or can this be hidden, only some of icon assignments working). But these should be routine as far as figuring them out. Onward!

^ | v • Reply • Share ›

Avatar This comment was deleted.

BjoernKW ➚ Guest • 2 years ago • edited

Yeah, right. Wholesale abandonment of decades of software development on a reliable, high-performance platform in favour of the hyped language du jour. What could possibly go wrong?

11 ^ | v • Reply • Share ›

Avatar This comment was deleted.

xerus ➚ Guest • 2 years ago

Kotlin is neither. It's built by an established company, JetBrains, with a clear goal, full Java-interoperability, to get around the verbosity of the language while benefitting from the JVM and add some very helpful features. Additionally, it will also compile to JavaScript and Native Code for LLVM.
It has been in development for over 5 years. Why are you so easily sweeping over it?

^ | v • Reply • Share ›

Eddie Calzone ➚ Guest • 2 years ago

http://www.its-not-its.info/

^ | v • Reply • Share ›

hrzafer ➚ Guest • 2 years ago

lol :) u r funny!

^ | v • Reply • Share ›

Bleaker ➚ Guest • 2 years ago

In some cases the word dying equals to being loved by nobody, which is true for Java. I myself wrote Java for food and not love it at all, just like all of my colleagues. But Java is actually the best solution if you want to do some serious server side stuff and easy to hire med level developers.

9 ^ | v • Reply • Share ›

Avatar This comment was deleted.

Avatar This comment was deleted.

Peter Ashford ➚ Guest • 2 years ago

WxWidgets is the worst GUI library I have ever used

2 ^ | v • Reply • Share ›

Peter Ashford ➚ Guest • 2 years ago

What a ridiculous statement.You may not think Java is cool but it's BS to call it dying when it's the most popular programming language on the planet.

4 ^ | v • Reply • Share ›

Lluis Martinez Ferrando ➚ Guest • 2 years ago

Joke of the day :-)

2 ^ | v • Reply • Share ›

Torben Binder ➚ Guest • 2 years ago

Yeah i really like go for all headless use cases, but as far as i know there is no go native gui tools available. You could of course use c interop an use qt, or gtk, but you loose awesome functionality like cross compiling...

^ | v • Reply • Share ›

sulfide ➚ Guest • 2 years ago

why would anyone want to move to go? google literally made it for newbie cogs

The key point here is our programmers are Googlers, they're not researchers. They're typically, fairly young, fresh out of school, probably learned Java, maybe learned C or C++, probably learned Python. They're not capable of understanding a brilliant language but we want to use them to build good software. So, the language that we give them has to be easy for them to understand and easy to adopt. – Rob Pike

^ | v • Reply • Share ›

Michele Adduci • 2 years ago

fun fact: if you add JavaFX and JavaFX WebView, you have already an extra +70MB on your 'lean' redistributable.

3 ^ | v • Reply • Share ›

Steve Perkins Mod ➚ Michele Adduci • 2 years ago

Sure. But JavaFX WebView means that you're embedding the WebKit browser engine into your application. In other words, you've all but written an Electron app (which is about 131 MB).

Moral to the story: Think carefully whether you want to embed a web browser into your desktop app.

4 ^ | v • Reply • Share ›

Luis Trigueiros • 2 years ago

This the first article I found about something unlocked by Java 9, loved this article.
Thank you for sharing.

3 ^ | v • Reply • Share ›

amihaiemil • 2 years ago • edited

I am speechless these days. Why are we going back? Why are we even talking about monolithic applications? Is having a JRE deployed on all environments a problem? What is Docker for, then?

What about JavaEE? These days I see people going nuts over Spring Boot and what else. Why? Is it so hard to understand the platform you're working with (Glassfish, Jboss, Wildfly etc) and develop a thin, JavaEE-compliant .war/.ear?

It seems to me that we are throwing away many years of progress.

4 ^ | v 3 • Reply • Share ›

sarnobat ➚ amihaiemil • 2 years ago • edited

I disagree that monolithic apps are undesirable. Could you imagine the most popular Unix command line tools like grep being bundles? There's something called "dependency hell" that monolithic executables avoid. And no, Docker is overkill.

^ | v • Reply • Share ›

A new Dev ➚ amihaiemil • 2 years ago

Nowadays you want deploy as fast as possible, 213545304gb docker / jboss or what ever is a BIG NO

^ | v • Reply • Share ›

amihaiemil ➚ A new Dev • 2 years ago • edited

I'm sorry, I don't see how a clean JavaEE platform with one or a few thin .war files deployed on it has 213545304 gb.

And how can you move fast when you have to handle a 500mb monolith which probably contains a few conflicting Java libraries as well?

^ | v • Reply • Share ›

A new Dev ➚ amihaiemil • 2 years ago

There is one thing that go understood, is the need of small binary for quick deployment, as cheap as possible (instance with low cpu/ram/disk) depend on the context and application of course; we don't live in a static world anymore, where you can sit and wait to move, so yes we need small/fast executable so we can deploy and execute as quick and cheap as possible

^ | v • Reply • Share ›

amihaiemil ➚ A new Dev • 2 years ago • edited

> so yes we need small/fast executable so we can deploy and execute as quick and cheap as possible

Bingo, this is what I am saying. And I'm saying that Project Jigsaw, which is encouraging the fat, monolithic way, is not supporting that endeavour. Neither does Spring boot and the fact that people are using Docker only as

Neither does Spring boot and the fact that people are using Docker only as an OS wrapper. Instead of letting Docker take care of the runtime environment, people are shoving the runtime dependencies inside the deployable, making it fat and heavy.

2 ^ | v • Reply • Share ›

A new Dev ➚ amihaiemil • 2 years ago

This is why AOT + self contained binary that only contains needed code is the way to go, not whole modules, we just want code we need, nothing else, you never needed to deploy fast tiny app so you can't understand

People use go for a reason you seems to not want to understand

^ | v • Reply • Share ›

amihaiemil ➚ A new Dev • 2 years ago

We are discussing Java here, not golang. And in Java, the same idea of self contained binaries is very bad due to how the language and the ecosystem around it are working.

In Java, if you do not pay attention to what dependencies you are bringing, you can mess things up really bad and get those kind of bugs that are not predictable or consistently reproductible -- the best example is bringing in 2 versions of the same class, from different packages; then, the classloader may be confused and pick the wrong class at runtime, causing all sorts of dubious bugs.

This is why they implemented JavaEE as a platform, so people would have to deal with as few dependencies as possible -- trouble is, most people did not understand the "platform" part at all, to most of them JavaEE is just... Java with a few annotations, "you know, the webapps thing".

And neverthelss, even in golang, I still don't understand (I honestly don't) how having a fat deployable is better than having a light one and let docker take care of the runtime, let docker be the platform.

^ | v • Reply • Share ›

Peter Hansson • 2 years ago

I think the idea of distributing a private JRE with your application is far superior to anything else. Disk size is not so important. I don't think a 'price' of an additional 150 Mb disk space consumption is worth mentioning. As for distribution size (the size of the download) you can actually cut down a JRE8 down to just 25 Mb if you don't need JavaFX. Figure is for Win64. A little more for other OSes. For full details: https://netbeansscribbles.w...

The point here is that even prior to the modularisation feature in Java9 you could actually go pretty low if the concern is download size.

1 ^ | v • Reply • Share ›

js • 2 years ago

"and superior to web-hybrid options like
Electron"

Superior in regards to file size? That's the only thing you mentioned in this article?

1 ^ | v • Reply • Share ›

1 ^ | v • Reply • Share ›

Apokaliptor • 2 years ago • edited

Hey, by any chance you tried javapackager with modules?
When I do to create OSX app , the final app something.app comes with 96mb, instead of 45mb, the 'modules' for the app '72.7mb' and the 'modules' for runtime image is '26.7mb', mostly because 'these tags '--strip-debug', '--compress', '2', '--no-header-files', '--no-man-pages' are not working with javapackager, a tutorial on packaging with Java9 modules would be great, because in this context (creating JavaFX app) what really matters for us is the final apps, not runtime images, .msi for windows, .dmg for OSX, etc

Thanks, and great tutorial!

1 ^ | v • Reply • Share ›

void256 ➚ Apokaliptor • 2 years ago

Exactly! I'd like to know the same. Somehow javapackager does not support jlink specific parameters which is unfortunate. I hope somewhere exists a feature request for that and we'll get support for it soon. In the javapackager code it looks like there is some kind of plugin mechanism which actually supports passing of parameters from the javapackager commandline to the specific plugin in question but it seems not to support jlink (and the code was rather confusing there too so I'm not sure).

Who wants a shell script / batch file based jlink runtime image if you can get an actual native application with javapackager? :)

^ | v • Reply • Share ›

Dieter Hubau • 2 years ago

Great article, makes me think of making a JavaFX Application that is comparable to other Electron apps but much smaller (with less memory footprint too since Electron eats memory for breakfast, lunch and dinner)

1 ^ | v • Reply • Share ›

mamoulian • 2 years ago

Interesting.

I dislike the idea of listing dependencies in two places though - gradle and module-info. Sounds like this needs something like proguard's shrinker to automatically work out what's needed and strip everything else out - from the JRE as well as the fat jar.

Is there a benefit to this in a server environment? With docker the options are a prebuilt docker image containing a JRE and add our fat jar, or a base Linux docker image and add this embedded JRE package.

1 ^ | v • Reply • Share ›

Steve Perkins Mod ➚ mamoulian • 2 years ago

I know that Oracle has started putting out a Docker image with Alpine Linux and Oracle's JRE. That is nice, as most people have avoided potential licensing issues by using Alpine Linux with OpenJDK (which doesn't offer as much support for JVM optimizations).

However, even so... if we assume that your application JAR is 10 megs:

1. Your JAR + full Oracle JRE + Alpine Linux = 10 + 200 + 5 = 215 MB

(mailto:steve@steveperkins.com) (https://github.com/steve-perkins)
(https://gitlab.com/steve-perkins) (https://twitter.com/stevedperkins)
(https://linkedin.com/in/perkinssteve)
(https://stackoverflow.com/users/461800/steve-perkins)

Hugo v0.57.2 (http://gohugo.io) powered • Theme by Beautiful Jekyll (http://deanattali.com/beautiful-jekyll/) adapted to Beautiful Hugo (https://github.com/halogenica/beautifulhugo)