

JEP 392: Packaging Tool

Owner	Andy Herrick
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	16
Component	tools/jpackage
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	S
Duration	S
Relates to	JEP 343: Packaging Tool (Incubator)
Reviewed by	Alexander Matveev, Alexey Semenyuk, Kevin Rushforth, Philip Race
Endorsed by	Brian Goetz
Created	2020/06/17 14:57
Updated	2021/02/19 20:03
Issue	8247768

Summary

Provide the jpackage tool, for packaging self-contained Java applications.

History

The jpackage tool was introduced as an incubating tool in JDK 14 by JEP 343. It remained an incubating tool in JDK 15, to allow time for additional feedback. It is now ready to be promoted from incubation to a production-ready feature. As a consequence of this transition, the name of the jpackage module will change from `jdk.incubator.jpackage` to `jdk.jpackage`.

The only substantive change relative to JEP 343 is that we replaced the `--bind-services` option with the more general `--jlink-options` option, [described below](#).

Goals

Create a packaging tool, based on the legacy JavaFX `javapackager` tool, that:

- Supports native packaging formats to give end users a natural installation experience. These formats include `msi` and `exe` on Windows, `pkg` and `dmg` on macOS, and `deb` and `rpm` on Linux.
 - Allows launch-time parameters to be specified at packaging time.
 - Can be invoked directly, from the command line, or programmatically, via the `ToolProvider API`.
- There is no GUI for the tool; a command-line interface (CLI) is sufficient.
- There is no support for cross compilation. For example, in order to create Windows packages one must run the tool on Windows. The packaging tool will depend upon platform-specific tools.
- There is no special support for legal files beyond what is already provided in JMOD files. There will be no aggregation of individual license files.
- There is no native splash screen support.
- There is no auto-update mechanism.

Non-Goals

- The following features of the legacy `javapackager` tool are not supported:
 - Java Web Start application support,
 - JavaFX-specific features,
 - `jdeps` usage for determining required modules, and
 - the Ant plugin.
- There is no GUI for the tool; a command-line interface (CLI) is sufficient.
- There is no support for cross compilation. For example, in order to create Windows packages one must run the tool on Windows. The packaging tool will depend upon platform-specific tools.
- There is no special support for legal files beyond what is already provided in JMOD files. There will be no aggregation of individual license files.
- There is no native splash screen support.
- There is no auto-update mechanism.

Motivation

Many Java applications need to be installed on native platforms in a first-class way, rather than simply being placed on the class path or the module path. It is not sufficient for the application developer to deliver a simple JAR file; they must deliver an installable package suitable for the native platform. This allows Java applications to be distributed, installed, and uninstalled in a manner that is familiar to users. For example, on Windows users expect to be able to double-click on a package to install their software, and then use the control panel to remove the software; on macOS, users expect to be able to double-click on a DMG file and drag their application to the Application folder.

The jpackage tool can also help fill gaps left by past technologies such as Java Web Start, which was removed from Oracle's JDK 11, and `pack200`, which was removed in JDK 14 (JEP 367). Developers can use `jlink` to strip the JDK down to the minimal set of modules that are needed, and then use the packaging tool to produce a compressed, installable image that can be deployed to target machines.

To address these requirements previously, a packaging tool called `javapackager` was distributed with Oracle's JDK 8. However, it was removed from Oracle's JDK 11 as part of the removal of JavaFX.

Description

The jpackage tool packages a Java application into a platform-specific package that includes all of the necessary dependencies. The application may be provided as a collection of ordinary JAR files or as a collection of modules. The supported platform-specific package formats are:

- Linux: `deb` and `rpm`
- macOS: `pkg` and `dmg`
- Windows: `msi` and `exe`

By default, jpackage produces a package in the format most appropriate for the system on which it is run.

Basic usage: Non-modular applications

Suppose you have an application composed of JAR files, all in a directory named `lib`, and that `lib/main.jar` contains the main class. Then the command

```
$ jpackage --name myapp --input lib --main-jar main.jar
```

will package the application in the local system's default format, leaving the resulting package file in the current directory. If the `MANIFEST.MF` file in `main.jar` does not have a `Main-Class` attribute then you must specify the main class explicitly:

```
$ jpackage --name myapp --input lib --main-jar main.jar \
--main-class myapp.Main
```

The name of the package will be `myapp`, though the name of the package file itself will be longer, and end with the package type (e.g., `myapp.exe`). The package will include a launcher for the application, also called `myapp`. To start the application, the launcher will place every JAR file that was copied from the input directory on the class path of the JVM.

If you wish to produce a package in a format other than the default, then use the `-type` option. For example, to produce a `pkg` file rather than `dmg` file on macOS:

```
$ jpackage --name myapp --input lib --main-jar main.jar --type pkg
```

Basic usage: Modular applications

If you have a modular application, composed of modular JAR files and/or JMOD files in a `lib` directory, with the main class in the module `myapp`, then the command

```
$ jpackage --name myapp --module-path lib -m myapp
```

will package it. If the `myapp` module does not identify its main class then, again, you must specify that explicitly:

```
$ jpackage --name myapp --module-path lib -m myapp/myapp.Main
```

(When creating a modular JAR or a JMOD file you can specify the main class with the `--main-class` option to the `jar` and `jmod` tools.)

Package metadata

The jpackage tool allows you to specify various kinds of platform independent metadata such as name and app-version, as well as platform-specific metadata for each platform.

The description of all jpackage options can be found in the jpackage [man page](#).

File associations

You can define one or more file-type associations for your application via the `--file-associations` option, which can be used more than once. The argument to this option is a properties file with values for one or more of the following keys:

- `extension` specifies the extension of files to be associated with the application,
- `mime-type` specifies the MIME type of files to be associated with the application,
- `icon` specifies an icon, within the application image, for use with this association, and
- `description` specifies a short description of the association.

Launchers

By default, the jpackage tool creates a simple native launcher for your application. You can customize the default launcher via the following options:

- `--arguments <string>` — Command-line arguments to pass to the main class if no command line arguments are given to the launcher (this option can be used multiple times)
- `--java-options <string>` — Options to pass to the JVM (this option can be used multiple times)

If your application requires additional launchers then you can add them via the `--add-launcher` option:

- `--add-launcher <launcher-name>=<file>`

The named `<file>` should be a properties file with values for one or more of the keys `app-version` `icon` `arguments` `java-options` `main-class` `main-jar` `module`, or `win-console`. The values of these keys will be interpreted as arguments to the options of the same name, but with respect to the launcher being created rather than the default launcher. The `--add-launcher` option can be used multiple times.

Application images

The jpackage tool constructs an *application image* as input to the platform-specific packaging tool that it invokes in its final step. Normally this image is a temporary artifact, but sometimes you need to customize it before packaging it. You can, therefore, run the jpackage tool in two steps. First, create the initial application image with the special package type `app-image`:

```
$ jpackage --name myapp --module-path lib -m myapp --type app-image
```

This will produce an application image in the `myapp` directory. Customize that image as needed, and then create the final package via the `--app-image` option:

```
$ jpackage --name myapp --app-image myapp
```

Runtime images

An application image contains both the files comprising your application as well as the JDK *runtime image* that will run your application. By default, the jpackage tool invokes the the [jlink tool](#) to create the runtime image. The content of the image depends upon the type of the application:

- For a non-modular application composed of JAR files, the runtime image contains the same set of JDK modules that is [provided to class-path applications in the unnamed module](#) by the regular java launcher.
- For a modular application composed of modular JAR files and/or JMOD files, the runtime image contains the application's main module and the transitive closure of all of its dependencies.

The default set of jlink options used by jpackage is

```
--strip-native-commands --strip-debug --no-man-pages --no-header-files
```

but this can be changed via the `--jlink-options` option. The resulting image will not include all available service providers; if you want those to be bound then use `--jlink-options` and include `--bind-services` in the list of jlink options.

In either case, if you want additional modules to be included to the runtime image you can use the jpackage tool's `--add-modules` option. The list of modules in a runtime image is available in the image's release file.

Runtime images created by the jpackage tool do not contain a `src.zip` file.

If you wish to customize the runtime image further then you can invoke `jlink` yourself and pass the resulting image to the jpackage tool via the `--runtime-image` option. For example, if you've used the [jdeps tool](#) to determine that your non-modular application only needs the `java.base` and `java.sql` modules, you could reduce the size of your package significantly:

```
$ jlink --add-modules java.base,java.sql --output myjre
$ jpackage --name myapp --input lib --main-jar main.jar --runtime-image myjre
```

Application-image layout and content

The layout and content of application images are platform-specific. Actual images contain some files not show in the layouts below; such files are implementation details that are subject to change at any time.

Linux

```
myapp/
bin/                // Application launcher(s)
  myapp
lib/
  app/
    myapp.cfg       // Configuration info, created by jpackage
    myapp.jar       // JAR files, copied from the --input directory
    mylib.jar
    ...
  runtime/          // JDK runtime image
```

The default installation directory on Linux is `/opt`. This can be overridden via the `--install-dir` option.

macOS

```
MyApp.app/
Contents/
  Info.plist        // Application launcher(s)
  MacOS/
    MyApp
Resources/         // Icons, etc.
  app/
    MyApp.cfg       // Configuration info, created by jpackage
    myapp.jar       // JAR files, copied from the --input directory
    mylib.jar
    ...
  runtime/          // JDK runtime image
```

The default installation directory on macOS is `/Applications`. This can be overridden via the `--install-dir` option.

Delivering jpackage

The jpackage tool is delivered in the JDK in a module named `jdk.jpackage`.

The command-line interface conforms to JEP 293 ([Guidelines for JDK Command-Line Tool Options](#)).

In addition to the command-line interface, jpackage is accessible via the [ToolProvider API](#) (`java.util.spi.ToolProvider`) under the name "jpackage".

Testing

Most tests can be done with automated scripts, but there are a few considerations to be aware of:

- Testing native packages may require optional tools to be installed; those tests will need to be written such that they are skipped on systems without the necessary tools.
- Verifying some types of native packages (e.g., `exe` on Windows or `dmg` on macOS) may require some manual testing.
- We need to ensure that native packages can be installed and uninstalled cleanly, so that developers can test in their local environment without fear of polluting their systems.

Dependencies

Native packages are generated using tools available on the host platform. On Windows, developers must install the third-party “Wix” tool in order to generate `msi` or `exe` packages.