sbt-native-packager

Release 1.0a1

Josh Suereth

Aug 16, 2022
# Contents

1 Introduction
   1.1 Goals .................................................. 1
   1.2 Scope .................................................. 1
   1.3 Core Concepts ........................................... 2

2 Getting Started
   2.1 Setup .................................................... 5
   2.2 Your first package ...................................... 5

3 Packaging Formats
   3.1 Universal Plugin ......................................... 8
   3.2 Linux Plugin ............................................. 15
   3.3 Debian Plugin ........................................... 21
   3.4 Rpm Plugin .............................................. 25
   3.5 Docker Plugin ........................................... 31
   3.6 Windows Plugin ......................................... 38
   3.7 JDKPackager Plugin ..................................... 41
   3.8 GraalVM Native Image Plugin ............................ 44

4 Project Archetypes
   4.1 Java Application Archetype .......................... 47
   4.2 Java Server Application Archetype .................. 54
   4.3 Systemloaders ........................................... 59
   4.4 Configuration Archetypes .............................. 62
   4.5 Jlink Plugin ............................................ 62
   4.6 Archetype Cheatsheet ................................... 64

5 Recipes
   5.1 Custom Package Formats ............................... 69
   5.2 Dealing with long classpaths ......................... 73
   5.3 Play 2 Packaging ....................................... 74
   5.4 Deployment .............................................. 75
   5.5 Scala JS packaging ..................................... 78
   5.6 Build the same package with different configs ...... 78
   5.7 Embedding JVM in Universal ........................... 81
   5.8 Setting the umask for your process .................. 82
CHAPTER 1

Introduction

SBT native packager lets you build application packages in native formats and offers different archetypes for common configurations, such as simple Java apps or server applications.

This section provides a general overview of native packager and its core concepts. If you want a quick start, go to the getting started section. However we recommend understanding the core concepts, which will help you to get started even quicker.

1.1 Goals

Native packager defines project goals in order to set expectations and scope for this project.

1. **Native formats should build on their respective platform** This allows native packager to support a wide range of formats as the packaging plugin serves as a wrapper around the actual packaging tool. However, alternative packaging plugins maybe provided if a java/scala implementation exists. As an example debian packages should always build on debian systems, however native packager provides an additional plugin that integrates JDeb for a platform independent packaging strategy.

2. **Provide archetypes for zero configuration builds** While packaging plugins provide the how a package is created, archetypes provide the configuration for what gets packaged. Archetypes configure your build to create a package for a certain purpose. While an archetype may not support all packaging formats, it should work without configuration for the supported formats.

3. **Enforce best-practices** There is no single way to create a package. Native packager *tries* to create packages following best practices, e.g. for file names, installation paths or script layouts.

1.2 Scope

While native packager provides a wide range of formats and archetype configurations, its scope is relatively narrow. Native packager only takes care of packaging, the act of putting a list of mappings (source file to install target path) into a distinct package format (zip, rpm, etc.).
Archetypes like *Java Application Archetype* or *Java Server Application Archetype* only add additional files to the mappings enriching the created package, but they don’t provide any new features for native-packager core functionality. Much like the *packaging format plugins*, the archetypes rely on functionality already available on your deploy target.

These things are **out of native packagers scope**

1. **Providing application lifecycle management.** The *Java Server Application Archetype* provides *configurations* for common system-loaders like SystemV, Upstart or SystemD. However, creating a custom solution which includes stop scripts, PID management, etc. are not part of native packager.

2. **Providing deployment configurations** Native packager produces artifacts with the `packageBin` task. What you do with these is part of another step in your process.

### 1.3 Core Concepts

Native packager is based on a few simple concepts. If you understand these, you will be able to customize your build, create own packaging formats and deploy more effectively.

1. **Separation of concerns** with two kinds of plugins
   - *format plugins* define how a package is created
   - *archetype plugins* define what a package should contain

2. **Mappings** define how your build files should be organized on the target system.

   Mappings are a `Seq[(File, String)]`, which translates to “a list of tuples, where each tuple defines a source file that gets mapped to a path on the target system”.

The following sections describe these concepts in more detail.

### 1.3.1 Format Plugins

Format plugins provide the implementation to create package, the **how** a package is created. For example the *Debian Plugin* provides a way to package debian packages. Each format plugin has its own documentation. Each plugin provides a common set of features:

1. **Provide a new configuration scope** Formats define their own configuration scope to be able to customize every shared setting or task.

2. **Provide package format related settings and tasks** Each format plugin may add additional settings or tasks that are only used by this plugin. Normally these settings start with the plugin name, e.g. `rpmXYZ`.

3. **Implement package task** `packageBin` or `publishLocal` tasks provide the actual action to create a package.

By enabling a format plugin only with

```
enablePlugins(SomePackageFormatPlugin)
```

the resulting package will be empty because a format plugin doesn’t provide any configuration other than the default settings for the format plugin’s specific settings.
1.3.2 Archetype Plugins

While format plugins provide the **how**, archetypes provide the **what** gets packaged. An archetype changes the configuration in all supported package format scopes; they don’t add configuration scopes.

A full list of archetypes can be found [here](#).

**An archetype may provide the following:**

1. Archetype related settings and tasks
2. New files in your package

By enabling an archetype plugin with

```scala
enablePlugins(SomeArchetypePlugin)
```

all configuration changes will be applied as well as all supported format plugins will be enabled.

**Tip**: An archetype plugin should be the starting point for creating packages!

---

1.3.3 Mappings

Mappings are the heart of native packager. This task defines what files in your build should be mapped where on the target system. The type signature for the mappings task is

```scala
mappings: TaskKey[Seq[(File, String)]]
```

The `file` part of the tuple must be available during the packaging phase. The String part represents the path inside the installation directory.

The *Universal Plugin* represents the base for all other plugins. It has a big section on how to customize mappings.

1.3.4 Architecture

The architecture can be summarized with this diagram

When using the full power of the plugin, all of the packaging is driven from the *Universal / mappings* setting, which defines what files will be included in the package. These files are automatically moved around for the appropriate native packaging as needed.

---

1.3. Core Concepts
2.1 Setup

Sbt-native-packager is an AutoPlugin. Add it to your plugins.sbt

```scala
dобавитьSbtPlugin("com.github.sbt" % "sbt-native-packager" % "x.y.z")
```

2.1.1 Native Tools

Depending on the package format you want to create, you may need additional tools available on your machine. Each packaging format has a requirements section.

2.2 Your first package

Native packager provides packaging format plugins and archetype plugins to separate configuration and actual packaging. To get started we use the basic Java Application Archetype. For more archetypes see the archetypes page.

In your build.sbt you need to enable the archetype like this

```scala
enablePlugins(JavaAppPackaging)
```

This will also enable all supported format plugins.

2.2.1 Run the app

Native packager can stage your app so you can run it locally without having the app packaged.

```bash
sbt stage
d./target/universal/stage/bin/<your-app>
```
2.2.2 Create a package

We can generate other packages via the following tasks. Note that each packaging format may need some additional configuration and native tools available. Here’s a complete list of current formats.

- Universal/packageBin - Generates a universal zip file
- Universal/packageZipTarball - Generates a universal tgz file
- Debian/packageBin - Generates a deb
- Docker/publishLocal - Builds a Docker image using the local Docker server
- Rpm/packageBin - Generates an rpm
- Universal/packageOsxDmg - Generates a DMG file with the same contents as the universal zip/tgz.
- Windows/packageBin - Generates an MSI
There is a plugin for each packaging format that native-packager supports. These plugins can rely on each other to reuse existing functionality. Currently the autoplugin hierarchy looks like this

```
SbtNativePackager
  +
  |
+-------+ Universal +--------+-------------+----------------+
 | + | | | | | | | |
 | | | | | | | | |
 + + + + + + + + +
Docker +-+ Linux +-+ Windows +-+ JDKPackager +-+ GraalVM native-image
|
|
|
+ + + + +
Debian  RPM
```

If you enable the DebianPlugin all plugins that depend on the DebianPlugin will be enabled as well (LinuxPlugin, UniversalPlugin and SbtNativePackager).

Each packaging format defines its own scope for settings and tasks, so you can customize your build on a packaging level. The settings and tasks must be explicitly inherited. For the mappings task this looks like this

```
Docker / mappings := (Universal / mappings).value
```

To learn more about a specific plugin, read the appropriate doc.

**Tip:** You may also need to read the docs of the dependent plugins. We recommend always that you read the *Universal Plugin* documentation because all plugins rely on this one.
### 3.1 Universal Plugin

The Universal Plugin creates a generic, or “universal” distribution package. This is called “universal packaging.” Universal packaging just takes a plain mappings configuration and generates various package files in the output format specified. Because it creates a distribution that is not tied to any particular platform it may require manual labor (more work from your users) to correctly install and set up.

#### 3.1.1 Related Plugins

- *Linux Plugin*
- *Docker Plugin*
- *Windows Plugin*

#### 3.1.2 Requirements

Depending on what output format you want to use, you need one of the following applications installed:

- zip (if native)
- gzip
- xz
- tar
- hdiutil (for dmg)

#### 3.1.3 Build

There is a task for each output format:

- **Zip**
  
  ```
  sbt Universal/packageBin
  ```

- **Tar**
  
  ```
  sbt Universal/packageZipTarball
  ```

- **Xz**
  
  ```
  sbt Universal/packageXzTarball
  ```

- **Dmg**
  
  ```
  sbt Universal/packageOsxDmg
  ```

#### Required Settings

The Universal Plugin has no mandatory fields.

Enable the universal plugin.
enablePlugins(UniversalPlugin)

3.1.4 Configurations

Settings and Tasks inherited from parent plugins can be scoped with Universal.

Universal packaging provides three Configurations:

- **Universal** For creating full distributions
- **UniversalDocs** For creating bundles of documentation
- **UniversalSrc** For creating bundles of source.

Here is how the values for name and packageName are used by the three configurations:

Universal / name := name.value

UniversalDocs / name := (Universal / name).value

UniversalSrc / name := (Universal / name).value

Universal / packageName := packageName.value

3.1.5 Settings

As we showed before, the universal packages are completely configured through the use of mappings. Simply specify the desired mappings for a given configuration. For example:

Universal / mappings += (Compile / packageBin).value -> "lib/foo.jar"

However, sometimes it may be advantageous to customize the files for each archive separately. For example, perhaps the .tar.gz has an additional README plaintext file in addition to a README.html. To add this just to the .tar.gz file, use the task-scope feature of sbt:

Universal / packageZipTarball / mappings += file("README") -> "README"

Besides mappings, the name, sourceDirectory and target configurations are all respected by universal packaging.

**Note:** The Universal plugin will make anything in a bin/ directory executable. This is to work around issues with JVM and file system manipulations.

3.1.6 Tasks

- **Universal / package-bin** Creates the zip universal package.
- **Universal / package-zip-tarball** Creates the tar universal package.
- **Universal / package-xz-tarball** Creates the txz universal package. The xz command can get better compression for some types of archives.
- **Universal / package-osx-dmg** Creates the dmg universal package. This only work on macOS or systems with hdiutil.
- **UniversalDocs / packageBin** Creates the zip universal documentation package.
sbt-native-packager, Release 1.0a1

**UniversalDocs / packageZipTarball** Creates the *tgz* universal documentation package.

**UniversalDocs / packageXzTarball** Creates the *txz* universal documentation package. The *xz* command can get better compression for some types of archives.

## 3.1.7 Customize

### Universal Archive Options

You can customize the commandline options (if used) for the different zip formats. If you want to force local for the *tgz* output add this line:

```
Universal / packageZipTarball / universalArchiveOptions := Seq("--force-local", "-pcvf 
˓→")
```

This will set the cli options for the `packageZipTarball` task in the `Universal` plugin to use the options `--force-local` and `pcvf`. Be aware that the above line will overwrite the default options. You may want to prepend your options, doing something like:

```
Universal / packageZipTarball / universalArchiveOptions :=
(Seq("--exclude", "+--") ++ (Universal / packageZipTarball / 
˓→universalArchiveOptions).value)
```

Currently, these task can be customized:

- **Universal/package-zip-tarball** `Universal / packageZipTarball / universalArchiveOptions`
- **Universal/package-xz-tarball** `Universal / packageXzTarball / universalArchiveOptions`

### Getting Started with Universal Packaging

By default, all files found in the `src/universal` directory are included in the distribution. So, the first step in creating a distribution is to place files in this directory and organize them as you’d like in them to be in the distributed package. If your output format is a zip file, for example, although the distribution will consist of just one zip file, the files and directories within that zip file will reflect the same organization and structure as `src/universal`.

To add files generated by the build task to a distribution, simply add a *mapping* to the `Universal / mappings` setting. Let’s look at an example where we add the packaged jar of a project to the *lib* folder of a distribution:

```
Universal / mappings += {
   val jar = (Compile / packageBin).value
   jar -> ("lib/" + jar.getName)
}
```

The above does two things:

1. It depends on `Compile / packageBin` which will generate a jar file form the project.
2. It creates a *mapping* (a `Tuple2[File, String]`) which denotes the file and the location in the distribution as a string.

You can use this pattern to add anything you desire to the package.

**Note**

If you are using an application archetype or the playframework, the jar mapping is already defined and you should not include these in your `build.sbt`. issue 227
**Universal Conventions**

This plugin has a set of conventions for universal packages that enable the automatic generation of native packages. The universal convention has the following package layout:

```
bin/
  <scripts and things you want on the path>
lib/
  <shared libraries>
conf/
  <configuration files that should be accessible using platform standard config locations.>
doc/
  <Documentation files that should be easily accessible. (index.html treated specially)>
```

If your plugin matches these conventions, you can enable the settings to automatically generate native layouts based on your universal package. To do so, add the following to your build.sbt:

```scala
mapGenericFilesToLinux
mapGenericFilesToWinows
```

In Linux, this mapping creates symlinks from platform locations to the install location of the universal package. For example, given the following packaging:

```
bin/
  cool-tool
lib/
  cool-tool.jar
conf/
  cool-tool.conf
```

The `mapGenericFilesToLinux` settings will create the following package (symlinks denoted with ->):

```
/usr/share/<pkg-name>/
  bin/
    cool-tool
  lib/
    cool-tool.jar
  conf/
    cool-tool.conf
/usr/bin/
  cool-tool -> /usr/share/<package-name>/bin/cool-tool
/etc/<pkg-name> -> /usr/share/<package-name>/conf
```

The `mapGenericFilesToWindows` will construct an MSI that installs the application in `<Platform Program Files>\<Package Name>` and include the bin directory on Windows PATH environment variable (optionally disabled).

While these mappings provide a great start to nice packaging, it still may be necessary to customize the native packaging for each platform. This can be done by configuring those settings directly.

For example, even using generic mapping, debian has a requirement for changelog files to be fully formed. Using the above generic mapping, we can configure just this changelog in addition to the generic packaging by first defining a changelog in `src/debian/changelog` and then adding the following setting:

---

3.1. Universal Plugin
Notice how we’re only modifying the package mappings for Debian linux packages.
For more information on the underlying packaging settings, see Windows Plugin and Linux Plugin documentation.

Change/Remove Top Level Directory in Output

Your output package (zip, tar, gz) by default contains a single folder with your application. If you want to change this folder or remove this top level directory completely use the `topLevelDirectory` setting.

Removing the top level directory

```scala
topLevelDirectory := None
```

Changing it to another value, e.g. the `packageName` without the version

```scala
topLevelDirectory := Some(packageName.value)
```

Or just a plain hardcoded string

```scala
topLevelDirectory := Some("awesome-app")
```

Skip packageDoc task on stage

The `stage` task forces a `javadoc.jar` build, which could slow down `stage` tasks performance. In order to deactivate this behaviour, add this to your `build.sbt`

```scala
compile / packageDoc / mappings := Seq()
```

Source issue 651.

MappingsHelper

The `MappingsHelper` class provides a set of helper functions to make mapping directories easier.

**sbt 0.13.5 and plugin 1.0.x or higher**

```scala
import NativePackagerHelper._
```

**plugin version 0.8.x or lower**

```scala
import com.typesafe.sbt.SbtNativePackager._
import NativePackagerHelper._
```

You get a set of methods which will help you to create mappings very easily.
Mapping Examples

SBT provides the IO and Path APIs, which help make defining custom mappings easy. The files will appear in the generate universal zip, but also in your debian/rpm/msi/dmg builds as described above in the conventions.

The Compile / packageBin dependency is only needed if your files get generated during the packageBin command or before. For static files you can remove it.

Mapping a complete directory

There are some helper methods so you can create a mapping for a complete directory:

For static content, you can just add the directory to the mapping:

```scala
Universal / mappings +++ directory("SomeDirectoryNameToInclude")
```

If you want to add everything in a directory where the path for the directory is dynamic, e.g. the scala-2.10/api directory that is nested under in the target directory, and target is defined in a task:

```scala
(Universal / mappings) ~= (_ ++ directory(target.value / "scala-2.10" / "api"))
```

You can also use the following approach if, for example, you need more flexibility:

```scala
(Universal / mappings) +++ {
    val dir = target.value / "scala-2.10" / "api"
    (dir ** AllPassFilter) pair relativeTo(dir.getParentFile)
}
```

Here is what happens in this code:

- dir.*** is a PathFinder method that creates a sequence of every file under a directory, including the directory itself.
- relativeTo() returns a String that is the path relative to whatever you pass to it.
- dir.getParentFile returns the parent of dir. In this example, it’s the parent directory of whatever target is.
- pair is a PathFinder method that takes a function and applies it to every file (in the sequence), and returns a (file, function-result) tuple.

Putting it all together, this creates a map of every file under target/scala-2.10/api (including the directory target/scala-2.10/api itself) with a string that is the path to the parent of target. This is a mapping for every file and a string that tells the universal packager where it is located.

For example:

```scala
if target = /Users/you/dev/fantasticApp/src/scala/fantasticApp-0.1-HOTFIX01
```
sbt-native-packager, Release 1.0a1

and fantasticApp-0.1-HOTFIX01/scala-2.10/api/ contains the files

|somedata.csv
|README |

Then the code above will produce this mapping:

```scala
g((/Users/you/dev/fantasticApp/src/scala/fantasticApp-0.1-HOTFIX01,fantasticApp-0.1-HOTFIX01),
  (/Users/you/dev/fantasticApp/src/scala/fantasticApp-0.1-HOTFIX01/README,fantasticApp-0.1-HOTFIX01/README),
  ((/Users/you/dev/fantasticApp/src/scala/fantasticApp-0.1-HOTFIX01/somedata.csv,fantasticApp-0.1-HOTFIX01/somedata.csv))
```

Note that the first item of each pair is the full path to where the file exists on the system /Users/you, and the second part is the just the path starting after .../scala. That second part is what is returned from `<each file>.relativeTo(dir.getParentFile)`.

### Mapping the content of a directory (excluding the directory itself)

```scala
universal / mappings +++ {
  val dir = target.value / "scala-2.10" / "api"
  (dir ** AllPassFilter --- dir) pair relativeTo(dir)
}
```

The `dir` gets excluded and is used as root for `relativeTo(dir)`.

### Filter/Remove mappings

If you want to remove mappings, you have to filter the current list of mappings. This example demonstrates how to build a fat jar with sbt-assembly, but using all the convenience of the sbt native packager archetypes.

```
// removes all jar mappings in universal and appends the fat jar
universal / mappings := {
  // universalMappings: Seq[(File,String)]
  val universalMappings = (universal / mappings).value
  val fatJar = (compile / assembly).value

  // removing means filtering
  // notice the "!" - it means NOT, so only keep those that do NOT have a name ending with "jar"
  val filtered = universalMappings filter {
    case (file, name) => !name.endsWith(".jar")
  }

  // add the fat jar to our sequence of things that we've filtered
  filtered :+ (fatJar -> ("lib/" + fatJar.getName))
}
```

The complete `build.sbt` should contain these settings if you want a single assembled fat jar.
// the assembly settings
assemblySettings

// we specify the name for our fat jar
assembly / jarName := "assembly-project.jar"

// using the java server for this application. java_application would be fine, too
packageArchetype.java_server

// removes all jar mappings in universal and appends the fat jar
Universal / mappings := {
    val universalMappings = (Universal / mappings).value
    val fatJar = (Compile / assembly).value
    val filtered = universalMappings filter { case (file, name) => ! name.endsWith(".jar") }
    filtered ++ (fatJar -> ("lib/" + fatJar.getName))
}

// the bash scripts classpath only needs the fat jar
scriptClasspath := Seq( (assembly / jarName).value )

3.2 Linux Plugin

The native packager plugin is designed so that linux packages look similar but can contain distribution specific information.

Note: The linux plugin depends on the Universal Plugin.

3.2.1 Related Plugins

- Debian Plugin
- Rpm Plugin

3.2.2 Build

The linux plugin is just a top level plugin for linux packaging formats. The Linux scope contains settings which can be used by the plugins depending on the linux plugin.
sbt "show Linux / linuxPackageMappings"

Required Settings

A linux package needs some mandatory settings to be valid. Make sure you have these settings in your build:

name := "Linux Example"
version := "1.0"
maintainer := "Max Smith <max.smith@yourcompany.io>"
packageSummary := "Hello World Debian Package"
packageDescription := """A fun package description of our software, with multiple lines."""

Enable the linux plugin to activate the native package implementation.

enablePlugins(LinuxPlugin)

### 3.2.3 Configurations

Settings and tasks inherited from parent plugins can be scoped with **Linux**.

**Linux** / name := name.value

### 3.2.4 Settings

The required fields for any linux distribution are:

- **Linux** / name: The name given the package for installation.
- maintainer: The name of the maintainer of the package (important for ownership and signing).
- packageSummary: A one-sentence short summary of what the package does.
- packageDescription: A longer description of what the package does and what it includes.
- linuxPackageMappings: A list of files and their desired installation locations for the package, as well as other metainformation.
- fileDescriptorLimit: Maximum number of open file descriptors for the spawned application. The default value is 1024.

### 3.2.5 Customize

**Package Mappings**

Most of the work in generating a linux package is constructing package mappings. These ‘map’ a file to a location on disk where it should reside as well as information about that file. Package mappings allow the specification of file ownership, permissions and whether or not the file can be considered “configuration”.

Note that while the *sbt-native-packager* plugin allows you to specify all of this information, not all platforms will make use of the information. It’s best to be specific about how you want files handled and run tests on each platform you wish to deploy to.

A package mapping takes this general form

```scala
(packageMapping(
    file -> "/usr/share/man/man1/sbt.1.gz"
) withPerms "0644" gzipped) asDocs()
```

Let’s look at each of the methods supported in the packageMapping ‘library’.
**packageMapping (mappings: (File, String)*)** This method takes a variable number of 
*File -> String* pairs. The *File* should be a locally available file that can be bundled, and the 
*String* is the installation location on disk for that file. This returns a new *PackageMapping*
that supports the remaining methods.

**withPerms (mask: String)** This function adjusts the installation permissions of the associated 
files. The flags passed should be of the form of a mask, e.g. 0755.

**gzipped** This ensures that the files are written in compressed format to the destination. This is a 
convenience for distributions that want files zipped.

**asDocs** This denotes that the mapped files are documentation files. *Note: I believe these are only used 
for ‘RPM’*’s.

**withConfig (value: String="true")** This denotes whether or not a *%config* attribute is at-
tached to the given files in the generated rpm SPEC. Any value other than "true" will be 
placed inside the *%config()* definition. For example `withConfig("noreplace")` results 
in `%config(noreplace)` attribute in the rpm spec.

**withUser (user: String)** This denotes which user should be the owner of the given files in the 
resulting package.

**withGroup (group: String)** This denotes which group should be the owner of the given files in the 
resulting package.

---

**The LinuxPackageMapping Models**

All classes are located in the `com.typesafe.sbt.packager.linux` package. So if you want to create in-
stances yourself you have to add `import com.typesafe.sbt.packager.linux._` to your build file.

A *LinuxPackageMapping* contains the following fields:

- `mappings: Traversable[(File, String)]` A list of mappings aggregated by this Linux-
  PackageMapping
- `fileData: LinuxFileMetaData` Permissions for all the defined mappings. Default = “root:root  
  755”
- `zipped: Boolean` Are the mappings zipped? Default = false

All mappings are stored in the task `linuxPackageMappings` which returns a `Seq[LinuxPackageMapping]`. 
To display the contents (value), open the sbt console and call

```
show linuxPackageMappings
```

The *LinuxFileMetaData* has the following fields

- `user: String` The user owning all the mappings. Default = “root”
- `group: String` The group owning all the mappings. Default = “root”
- `permissions: String` Access permissions for all the mappings. Default = “755”
- `config: String` Are the mappings config files. Default = “false”
- `docs: Boolean` Are the mappings docs. Default = false

Last but not least there are the `linuxPackageSymlinks`, which encapsulate symlinks on your destination system. 
A *LinuxSymlink* contains only two fields

- `link: String` The actual link that points to destination
- `destination: String` The link destination

---

**3.2. Linux Plugin**
You can see all currently configured symlinks with this simple command. `linuxPackageSymlinks` is just a `Seq[LinuxSymlink]`

```
show linuxPackageSymlinks
```

**Modifying Mappings in General**

Adding, filtering and altering mappings are always simple methods on a `Seq[LinuxPackageMapping]` sequence. This section shows you the general way to add, modify, or filter mappings. The following sections have specific examples.

The basic construct for **adding** a mapping is

```
// simple
linuxPackageMappings += packageMapping( (theFile, "/absolute/path/somefile.txt") )

// specialized
linuxPackageMappings += packageMapping( (theFile, "/absolute/path/somefile.txt") )
  → withPerms("644") asDocs()
```

To **filter** or **modify** a mapping, you generally create a new mapping by copying an existing one (or occasionally by creating a new blank one), then filter or modify it, and then return that filtered or modified mapping. Here’s an example that shows a number of things you can **possibly** do. See the next section for specific examples. (Basic scala collections operations are used in the code. Here is an explanation of the filter method.)

```
// sbt 0.13.0 syntax
linuxPackageMappings := {
  // mappings: Seq[LinuxPackageMapping]
  val mappings = linuxPackageMappings.value

  // this process will must return another Seq[LinuxPackageMapping]
  mappings map { linuxPackage =>
    // each mapping element is a Seq[(java.io.File, String)]
    val filtered = linuxPackage.mappings map {
      case (file, name) => file -> name // alter stuff here
    } filter {
      case (file, name) => true // filter anything from the mapping where the...
    }

    // Copy values from the mapping: (Include only what you need)
    val fileData = linuxPackage.fileData.copy{
      user = "new user",
      group = "another group",
      permissions = "444",
      config = "false",
      docs = false
    }

    // returns a fresh LinuxPackageMapping based on the above
    linuxPackage.copy(
      mappings = filtered,
      fileData = fileData
    )
  ) filter {
  }
}
```

(continues on next page)
The ordering in which you apply the tasks is important.

**Add Mappings**

To add an arbitrary file in your build path

```scala
linuxPackageMappings += {
  val file = sourceDirectory.value / "resources" / "somefile.txt"
  packageMapping( (file, "/absolute/path/somefile.txt") )
}
```

`linuxPackageMappings` can be scoped to **Rpm** or **Debian** if you want to add mappings only for a single packaging type.

```scala
Debian / linuxPackageMappings += {
  val file = sourceDirectory.value / "resources" / "debian-somefile.txt"
  packageMapping( (file, "/absolute/path/somefile.txt") )
}
```

```scala
Rpm / linuxPackageMappings += {
  val file = sourceDirectory.value / "resources" / "rpm-somefile.txt"
  packageMapping( (file, "/absolute/path/somefile.txt") )
}
```

**Filter/Remove Mappings**

If you want to remove some mappings you have to filter the current list of `linuxPackageMappings`. As `linuxPackageMappings` is a task, the order of your settings is important. Here are some examples on how to filter mappings.

```scala
// this is equal to
// linuxPackageMappings <<= linuxPackageMappings map { mappings => /* stuff */
//   mappings }
```

```scala
linuxPackageMappings := {
  // first get the current mappings. mapping is of type Seq[LinuxPackageMapping]
  val mappings = linuxPackageMappings.value

  // map over the mappings if you want to change them
  mappings map { mapping =>
    // we remove everything besides files that end with ".conf"
    val filtered = mapping.mappings filter { case (file, name) => name.endsWith ".conf" }

    // only elements where this is true are kept
    ... }
```
Alter LinuxPackageMapping

To alter the permissions for all LinuxPackageMapping s that match a specific criteria:

```scala
// Altering permissions for configs
val mappings = linuxPackageMappings.value

mappings map {
  case linuxPackage if linuxPackage.fileData.config equals "true" =>
    // altering the group
    val newFileData = linuxPackage.fileData.copy(
      group = "appdocs"
    )
    // altering the LinuxPackageMapping
    linuxPackage.copy(
      fileData = newFileData
    )
  case _ => linuxPackage
}
```

Alter LinuxSymlinks

To alter the permissions for all LinuxPackageMapping s that match a specific criteria:

```scala
// The same as linuxPackageMappings
val links = linuxPackageSymlinks.value

links filter { /* remove stuff */ } map { /* change stuff */ }
```

Add Empty Directories

There is a special helper function that allows you to add empty directories to the package mappings. This might be useful if the service needs some location to store files.

```scala
// Add an empty folder to mappings
linuxPackageMappings += packageTemplateMapping("/usr/share/$name.value/lib/native ...")() withUser(name.value) withGroup(name.value)
```
Man Pages

There are many ways to document your projects, and many ways to expose them. While the native packager places no limit on WHAT is included in a package, there are some things which receive special treatment.

Specifically: linux man pages.

To create a linux man page for the application, let’s create a src/linux/usr/share/man/man1/example-cli.1 file

```bash
" Process this file with
" groff -man -Tascii example-cli.1
".
.TH EXAMPLE_CLI 1 "NOVEMBER 2011" Linux "User Manuals"
.SH NAME
example-cli \- Example CLI
.SH SYNOPISIS
.B example-cli [-h]
```

Notice the location of the file. Any file under src/linux is automatically included, relative to /, in linux packages (deb, rpm). That means the man file will not appear in the universal package (confusing linux users).

Now that the man page is created, we can use a few tasks provided to view it in sbt. Let’s look in the sbt console

```scala
sbt generateManPages
```

We can use this task to work on the man pages and ensure they’ll look OK. You can also directly use groff to view changes in your man pages.

In addition to providing the means to view the man page, the native packager will also automatically gzip man pages for the distribution. The resulting man page is stored in /usr/share/man/man1/example-cli.1.gz in linux distributions.

3.3 Debian Plugin

The debian package specification is very robust and powerful. If you wish to do any advanced features, it’s best to understand how the underlying packaging system works. Debian Binary Package Building HOWTO by Chr. Clemens Lee is an excellent tutorial.

SBT Native Packager provides two ways to build debian packages:

1. A native implementation, where you need dpkg-deb installed, or
2. A java, platform independent approach with jdeb.

By default the native implementation is activated.

**Note:** The debian plugin depends on the Linux Plugin.

3.3.1 Requirements

If you use the native debian package implementation you need the following applications installed:

- dpkg-deb
- dpkg-sig
3.3.2 Build

`sbt debian/packageBin`

**Required Settings**

A debian package needs some mandatory settings to be valid. Make sure you have these settings in your build:

```
name := "Debian Example"
version := "1.0"
maintainer := "Max Smith <max.smith@yourcompany.io>"
packageSummary := "Hello World Debian Package"
packageDescription := """A fun package description of our software, with multiple lines."""
```

It’s not exactly mandatory, but still highly recommended to add relevant **JRE dependency**, for example:

```
debianPackageDependencies := Seq("java8-runtime-headless")
```

Enable the debian plugin to activate the native package implementation.

```
enablePlugins(DebianPlugin)
```

**JRE Dependencies**

By default, a Debian package would have no dependencies, even for the Java Runtime Environment (JRE). A startup script will seek JRE in several popular locations, and, JRE is not found, the following message would be displayed:

```
No java installations was detected.
Please go to http://www.java.com/getjava/ and download
```

To build a Debian package that integrates properly with Debian repository environment, i.e. depends on a package that provides JRE, one needs to specify JRE dependency using `debianPackageDependencies`. Debian (Ubuntu and compatible distributions as well) provides two families of virtual packages to do that:

- **javaN-runtime** Regular (full) JRE packages, with GUI support. Use this for applications requiring AWT/Swing support, OpenGL, sound, etc.

- **javaN-runtime-headless** Minimal JRE packages without GUI support, useful for server installation to avoid pulling large set of X.org-related packages. Use this for console-only applications, services, networked / web applications, etc.

`N` in `javaN` should be replaced with minimal JRE version required by the packaged application. It usually depends on a Scala version used:
• Scala 2.11.x or earlier requires Java 6
• Scala 2.12.x requires Java 8

Note that these are virtual packages, which are provided by a set of real packages. This means, for example, while installing a .deb package that depends on java6-runtime-headless:

• If end-user has no suitable JRE installed, it would automatically pull and install some “sane default” package which provides thing functionality (typically, it would be openjdk-8-jre-headless).
• If end-user does not like default suggested JRE for some reason, it’s possible to install any alternative implementation.
• If end-user has some existing JRE installation that is sufficient to play that role (for example, openjdk-9-jre, which provides, along others, java8-runtime-headless too), it would be used.

This dependency works equally well with both free/libre OpenJDK packages supplied by Debian, and non-free JDKs supplied by Oracle and packaged as .deb using make-jpkg utility from Debian’s java-package.

Native packaging

Since JARs are by default already compressed, DebianPlugin disables additional compression of the debian package contents.

To compress the debian package, override debianNativeBuildOptions with options for dpkg-deb.

```scala
Debian / debianNativeBuildOptions := Nil // dpkg-deb's default compression (currently → xz)
Debian / debianNativeBuildOptions := Seq("-Zgzip", "-z3") // gzip compression at → level 3
```

Note that commit cee091c released in 1.1.1 disables package re-compression by default. While this works great with tools such as apt and dpkg, un-compressed package installation is bugged in python-apt 8.8 series. This bug prevents installation of the generated debian package in the following configuration:

• installation using python-apt module, used by Ansible and SaltStack for example,
• being on python-apt 8.8 series that’s on Debian Wheezy and perhaps older

It will fail with an error message like:

```
E: This is not a valid DEB archive, it has no 'data.tar.gz', 'data.tar.bz2' or 'data. → tar.lzma' member
```

Solutions include:

• upgrading to Debian Jessie,
• upgrading python-apt, note that no official backport is known
• re-enabling package re-compression in sbt-native-packager, by overriding debianNativeBuildOptions as described above.

Java based packaging

If you want to use the java based implementation, enable the following plugin:

```scala
enablePlugins(JDebPackaging)
```
and this to your `plugins.sbt`:

```scala
libraryDependencies += "org.vafer" % "jdeb" % "1.3" artifacts (Artifact("jdeb", "jar", 
    _ => "jar"))
```

JDeb is a provided dependency. You have to explicitly add it on your own. It brings a lot of dependencies that could slow your build times. This is the reason the dependency is marked as provided.

### 3.3.3 Configurations

Settings and Tasks inherited from parent plugins can be scoped with `Debian`.

```scala
Debian / linuxPackageMappings := linuxPackageMappings.value
```

### 3.3.4 Settings

Debian requires the following specific settings:

- **Debian / name** The name of the package for debian (if different from general linux name).
- **Debian / version** The debian-friendly version of the package. Should be of the form x.y.z-build-aa.
- **Debian / debianPackageConflicts** The list of debian packages that this package conflicts with.
- **Debian / debianPackageDependencies** The list of debian packages that this package depends on.
- **Debian / debianPackageProvides** The list of debian packages that are provided by this package.
- **Debian / debianPackageRecommends** The list of debian packages that are recommended to be installed with this package.
- **Debian / linuxPackageMappings** Debian requires a `/usr/share/doc/{package name}/changelog.gz` file that describes the version changes in this package. These should be appended to the base linux versions.
- **Debian / maintainerScripts** Use `Debian / maintainerScripts` instead. These are the packaging scripts themselves used by dpkg-deb to build your debian. These scripts are used when installing/uninstalling a debian, like prerm, postinstall, etc. These scripts are placed in the DEBIAN file when building. Some of these files can be autogenerated, for example when using a package archetype, like server_application. However, any autogenerated file can be overridden by placing your own files in the `src/debian/DEBIAN` directory.
- **Debian / changelog** This is the changelog used by dpkg-genchanges to create the `.changes` file. This will allow you to upload the debian package to a mirror.

### 3.3.5 Tasks

The Debian support grants the following commands:

- **Debian / package-bin** Generates the `.deb` package for this project.
**Debian / lintian** Generates the `.deb` file and runs the `lintian` command to look for issues in the package. Useful for debugging.

**Debian / gen-changes** Generates the `.changes`, and therefore the `.deb` package for this project.

### 3.3.6 Customize

This section contains examples of how you can customize your debian build.

#### Customizing Debian Metadata

A Debian package provides metadata, which includes **dependencies** and **recommendations**. This example adds a dependency on java and recommends a git installation.

```scala
Debian / debianPackageDependencies ::= Seq("java2-runtime", "bash (>= 2.05a-11)")
Debian / debianPackageRecommends ::= "git"
```

#### Hook Actions into the Debian Package Lifecycle

To hook into the debian package lifecycle ([https://wiki.debian.org/MaintainerScripts](https://wiki.debian.org/MaintainerScripts)) you can add `preinst`, `postinst`, `prerm` and/or `postrm` scripts. Just place them into `src/debian/DEBIAN`. Or you can do it programmatically in your `build.sbt`. This example adds actions to `preinst` and `postinst`:

```scala
import DebianConstants._
Debian / maintainerScripts ::= maintainerScriptsAppend((Debian / maintainerScripts).--value)(
  Preinst  -> "echo 'hello, world'",
  Postinst -> s"echo 'installed \$(Debian / packageName).value'"
)
```

The helper methods can be found in `MaintainerScriptHelper` Scaladocs.

If you use the JavaServerAppPackaging there are predefined `postinst` and `preinst` files, which start/stop the application on install/remove calls. Existing maintainer scripts will be extended not overridden.

#### Use a Different Castle Directory for your Control Scripts

Your control scripts are in a different castle.. directory? No problem.

```scala
debianControlScriptsDirectory <<= (sourceDirectory) apply (_ / "deb" / "control")
```

### 3.4 Rpm Plugin

RedHat rpm files support a number of very advanced features. To take full advantage of this environment, it’s best to understand how the rpm package system works. How to create an RPM package on the fedora project wiki is a good tutorial, but it focuses on building packages from source. The sbt-native-packager assumes that SBT has built your source and generated binary packages.
3.4.1 Requirements

You need the following applications installed

- rpm
- rpm-build

3.4.2 Build

```
sbt Rpm/packageBin
```

Required Settings

A rpm package needs some mandatory settings to be valid. Make sure you have these settings in your build:

```
rpmVendor := "typesafe"
```

1.0 or higher

Enables the rpm plugin

```
enablePlugins(RpmPlugin)
```

0.8 or lower

For this versions rpm packaging is automatically activated. See the Getting Started page for information on how to enable sbt native packager.

3.4.3 Configuration

Settings and Tasks inherited from parent plugins can be scoped with Rpm.

```
Rpm / linuxPackageMappings := linuxPackageMappings.value
```

3.4.4 Settings

Informational Settings

```
Rpm / packageName
```

The name of the package for the rpm. Its value defines the first component of the rpm file name (packageName-version-rpmRelease.packageArchitecture.rpm), as well as the Name: tag in the spec file. Its default value is drawn from Linux / packageName.
Rpm / version

The version of the package for rpm. Takes the form x.y.z, and note that there can be no dashes in this version string. It defines the second component of the rpm file name (packageName-version-rpmRelease.packageArchitecture.rpm), as well as the Version: tag in the spec file. Its default value is drawn from the project defined version.

rpmRelease

The release number is the package's version. When the software is first packaged at a particular version, the release should be "1". If the software is repackaged at the same version, the release number should be incremented, and dropped back to "1" when the software version is new. Its value defines the third component of the rpm file name (packageName-version-rpmRelease.packageArchitecture.rpm), as well as the Release: tag in the spec file. Its default value is "1".

Rpm / packageArchitecture

The build architecture for the binary rpm. Its value defines the fourth component of the rpm file name (packageName-version-rpmRelease.packageArchitecture.rpm), as well as the BuildArch: tag in the spec file. Its default value is "noarch".

Rpm / packageSummary

A brief, one-line summary of the package. Note: the summary must not contain line separators or end in a period. Its value defines the Summary: tag in the spec file, and its default value is drawn from Linux / packageSummary.

Rpm / packageDescription

A longer, multi-line description of the package. Its value defines the %description block in the spec file, and its default value is drawn from Linux / packageDescription.

rpmVendor

The name of the company/user generating the RPM.

rpmUrl

A url associated with the software in the RPM.

rpmLicense

The license associated with software in the RPM.

rpmEpoch

The epoch is the most significant number used when resolving different versions for the same RPM. For a given package, packages with the highest epoch will be used, and in the event of a tie it will fall back to comparing the version and release.

Rpm / packageBin / artifactPath

The location of the generated RPM.

Dependency Settings

rpmAutoreq

Enable or disable the automatic processing of required packages. Takes the form "yes" or "no", defaults to "yes". Defines the AutoReq: tag in the spec file.

rpmRequirements

The RPM packages that are required to be installed for this RPM to work.

rpmAutoprov

Enable or disable the automatic processing of provided packages. Takes the form "yes" or "no", defaults to "yes". Defines the AutoProv: tag in the spec file.

rpmProvides

The RPM package names that this RPM provides.

rpmPrerequisites

The RPM packages this RPM needs before installation

rpmObsoletes

The packages this RPM allows you to remove

rpmConflicts

The packages this RPM conflicts with and cannot be installed with.

rpmSetarch[SettingKey[Option[String]]] Run rpmbuild via Linux setarch command. Use this for cross-platform builds.
Meta Settings

- **rpmPrefix** The path passed set as the base for the revocable package
- **rpmChangelogFile** External file to be imported and used to generate the changelog of the RPM.

Scriptlet Settings

- **Rpm / maintainerScripts** Contains the scriptlets being injected into the specs file. Currently supports all previous scriptlets: %pretrans, %pre, %verifyscript%, %post, %posttrans,%preun and %postun
- **rpmBrpJavaRepackJars** appends __os_install_post scriptlet to rpmPre avoiding jar repackaging

SystemV Start Script Settings

- **rpmDaemonLogFile** File name of the log generated by application daemon.

3.4.5 Tasks

The Rpm plugin support grants the following commands:

- **Rpm/packageBin** Generates the .rpm package for this project.
- **Rpm/rpmLint** Generates the .rpm file and runs the rpmlint command to look for issues in the package. Useful for debugging.

3.4.6 Customize

Rpm Prefix

The rpm prefix allows you to create a relocatable package as defined by http://www.rpm.org/max-rpm/sl-rpm-reloc-prefix-tag.html. This optional setting with a handful of overrides to scriptlets and templates will allow you to create a working java_server archetype that can be relocated in the file system.

Example Settings:

```scala
defaultLinuxInstallLocation := "*/opt/package_root",
rpmPrefix := Some(defaultLinuxInstallLocation),
linuxPackageSymlinks := Seq.empty,
defaultLinuxLogsLocation := defaultLinuxInstallLocation + "/" + name
```

**rpmChangelogFile**

The rpmChangelogFile property allows you to set a source that will be imported and used on the RPM generation. So if you use rpm commands to see the changelog it brings that information. You have to create the content in the changelog file using the RPM conventions that are available here http://fedoraproject.org/wiki/Packaging:Guidelines#Changelogs.

Example Settings:
changelog := "changelog.txt"

rpmChangelogFile := Some(changelog)

* Sun Aug 24 2014 Team <contact@example.com> - 1.1.0
  - Allow to login using social networks
* Wed Aug 20 2014 Team <contact@example.com> - 1.0.1
  - Vulnerability fix.
* Tue Aug 19 2014 Team <contact@example.com> - 1.0.0
  - First version of the system

Scriptlet Changes

Changing scriptlets can be done in two ways:

1. Override the Rpm / maintainerScripts, or
2. Place new scripts in the src/rpm/scriptlets

To override the "Rpm / maintainerScripts" you can override the command string explicitly, create a command string using appends and/or replacements, or even get a command string from a file source.

For example:

```scala
// overriding
import RpmConstants._
Rpm / maintainerScripts := Map(
  Pre -> Seq("echo "pre-install""),
  Post -> Seq("echo "post-install""),
  Pretrans -> Seq("echo "pretrans""),
  Posttrans -> Seq("echo "posttrans""),
  Preun -> Seq("echo "pre-uninstall""),
  Postun -> Seq("echo "post-uninstall""
)

// appending with strings and replacements
import RpmConstants._
Rpm / maintainerScripts := maintainerScriptsAppend((Rpm / maintainerScripts).value)(
  Pretrans -> "echo 'hello, world'",
  Post -> s"echo 'installing $$((Rpm / packageName).value)''
)

// appending from a different file
import RpmConstants._
Rpm / maintainerScripts := maintainerScriptsAppendFromFile((Rpm / maintainerScripts).value)(
  Pretrans -> (sourceDirectory.value / "rpm" / "pretrans"),
  Post -> (sourceDirectory.value / "rpm" / "posttrans")
)
```

The helper methods can be found in MaintainerScriptHelper Scaladocs.

To place new scripts in the src/rpm/scriptlets folder you simply put the commands into the appropriate scriptlet file. (The scriptlet file names can be found in the RPM Scaladocs.)

src/rpm/scriptlets/preinst
Using scriptlet files like this will override all previous contents.

Scriptlet Migration from 1.0.x

Before

```scala
rpmPostun := rpmPost.value.map {
  content =>
    s"|echo "I append this to the current content
    |"\n  ".stripMargin
}.orElse {
  Option("|echo "There wasn't any previous content"
  ".stripMargin)
}
```

After

```scala
// this gives you easy access to the correct keys
import RpmConstants._
// in order to append you have to pass the initial maintainerScripts map
Rpm / maintainerScripts := maintainerScriptsAppend((Rpm / maintainerScripts).value)(
  Pretrans -> "echo 'hello, world'",
  Post -> s"echo 'installing $((Rpm / packageName).value)"
)
```

Jar Repackaging

RPM repackages jars by default in order to optimize jars. Repacking is turned off by default. In order to enable it, set:

```scala
rpmBrpJavaRepackJars := true
```

Note that this appends content to your Pre definition, so make sure not to override it. For more information on this topic follow these links:

- issue #195
- pullrequest #199
- OpenSuse issue

Marking config files as noreplace

By default, rpm replaces config files on disk when the content has changed between two version. Often, this is not desirable as configurations are often customized and should not change during updates. rpm provides a means to turn
of the default behaviour by marking config files as `noreplace` in the spec file. In order to enable this for the build, we provide a helper method that can be used to modify all config file mappings:

```scala
Rpm / linuxPackageMappings := configWithNoReplace((Rpm / linuxPackageMappings).value)
```

This will mark all config files as `noreplace` and prevent them from being changed during updates. Please note that the `linuxPackageMappings` are scoped to the `Rpm` plugin. This is necessary in order to catch all config files relevant to the rpm package and mark them correctly.

### 3.5 Docker Plugin

Docker images describe how to set up a container for running an application, including what files are present, and what program to run.

https://docs.docker.com/introduction/understanding-docker/ provides an introduction to Docker.

https://docs.docker.com/reference/builder/ describes the Dockerfile: a file which describes how to set up the image.

sbt-native-packager focuses on creating a Docker image which can “just run” the application built by SBT.

**Note:** The docker plugin depends on the *Universal Plugin*.

#### 3.5.1 Requirements

You need the version 1.10 or higher of the docker console client installed. SBT Native Packager doesn’t use the REST API, but instead uses the CLI directly.

It is currently not possible to provide authentication for Docker repositories from within the build. The `docker` binary used by the build should already have been configured with the appropriate authentication details. See https://docs.docker.com/engine/reference/commandline/login/ how to login to a Docker registry with username and password.

#### 3.5.2 Build

```sbt
sbt Docker/publishLocal
```

**Required Settings**

```sbt
enablePlugins(DockerPlugin)
```

**Spotify java based docker client**

You can also use the java-based spotify Docker client. Add this to your `build.sbt`

```sbt
enablePlugins(DockerSpotifyClientPlugin)
```

and this to your `plugins.sbt`
libraryDependencies += "com.spotify" % "docker-client" % "8.9.0"

The Docker-spotify client is a provided dependency. You have to explicitly add it on your own. It brings a lot of dependencies that could slow your build times. This is the reason the dependency is marked as provided.

### 3.5.3 Configuration

Settings and Tasks inherited from parent plugins can be scoped with `Docker`.

```scala
Docker / mappings := mappings.value
```

### 3.5.4 Settings

#### Informational Settings

- **Docker / packageName** The name of the package for Docker (if different from general name). This will only affect the image name.

- **Docker / version** The version of the package for Docker (if different from general version). Often takes the form `x.y.z`.

- **Docker / maintainer** The maintainer of the package, recommended by the Dockerfile format.

#### Environment Settings

- **dockerBaseImage** The image to use as a base for running the application. It should include binaries on the path for `chown`, `mkdir`, have a discoverable `java` binary, and include the user configured by `daemonUser` (default).

- **Docker / daemonUser** The user to use when executing the application. Files below the install path also have their ownership set to this user.

- **dockerExposedPorts** A list of TCP ports to expose from the Docker image.

- **dockerExposedUdpPorts** A list of UDP ports to expose from the Docker image.

- **dockerExposedVolumes** A list of data volumes to make available in the Docker image.

- **dockerLabels** A map of labels that will be applied to the Docker image.

- **dockerEnvVars** A map of environment variables that will be applied to the Docker image.

- **dockerEntrypoint** Overrides the default entrypoint for docker-specific service discovery tasks before running the application. Defaults to the bash executable script, available at `bin/<script name>` in the current WORKDIR of `/opt/docker`.

- **dockerPermissionStrategy** The strategy that decides how file permissions are set for the working directory inside the Docker image
  - `DockerPermissionStrategy.MultiStage` (default) uses multi-stage Docker build to call `chmod` ahead of time.
  - `DockerPermissionStrategy.None` does not attempt to change the file permissions, and use the host machine’s file mode bits.
  - `DockerPermissionStrategy.Run` calls `RUN` in the Dockerfile. This has regression on the resulting Docker image file size.
• DockerPermissionStrategy.CopyChown calls COPY --chown in the Dockerfile. Provided as a backward compatibility.

dockerChmodType The file permissions for the files copied into Docker image when MultiStage or Run strategy is used.

• DockerChmodType.UserGroupReadExecute (default): chmod u=rX,g=rX
• DockerChmodType.UserGroupRead: chmod u=r,g=r
• DockerChmodType.UserGroupWriteExecute: chmod u=rwX,g=rwX
• DockerChmodType.SyncGroupToUser: chmod g=u
• DockerChmodType.UserGroupPlusExecute: chmod u+x,g+x (This is for dockerAdditionalPermissions)
• DockerChmodType.Custom: Custom argument provided by the user.

dockerAdditionalPermissions Additional permissions typically used to give chmod +x rights for the executable files. By default generated Bash scripts are given DockerChmodType.UserGroupPlusExecute.

dockerVersion The docker server version. Used to leverage new docker features while maintaining backwards compatibility.

dockerApiVersion The docker server API version. Used to leverage new docker features while maintaining backwards compatibility.

dockerGroupLayers The function mapping files into separate layers to increase docker cache hits. Lower index means the file would be a part of an earlier layer. The main idea behind this is to COPY dependencies *.jar’s first as they should change rarely. In separate command COPY the application *.jar’s that should change more often. Defaults to map the project artifacts and its dependencies to separate layers. To disable layers map all files to no layer using Docker / dockerGroupLayers := PartialFunction.empty.

Publishing Settings

dockerRepository The repository to which the image is pushed when the Docker / publish task is run. This should be of the form [repository.host[:repository.port]] (assumes use of the index.docker.io repository) or [repository.host[:repository.port]][/username] (discouraged, but available for backwards compatibility.).

dockerUsername The username or organization to which the image is pushed when the Docker / publish task is run. This should be of the form [username] or [organization].

dockerUpdateLatest The flag to automatic update the latest tag when the Docker / publish task is run. Default value is FALSE. In order to use this setting, the minimum docker console version required is 1.10. See https://github.com/sbt/sbt-native-packager/issues/871 for a detailed explanation.

dockerAlias The alias to be used for tagging the resulting image of the Docker build. The type of the setting key is DockerAlias. Defaults to [dockerRepository/][dockerUsername/][packageName]:[version].

dockerAliases The list of aliases to be used for tagging the resulting image of the Docker build. The type of the setting key is Seq[DockerAlias]. Alias values are in format of [dockerRepository/][dockerUsername/][packageName]:[tag] where tags are list of including your project version and latest tag(if dockerUpdateLatest is enabled). To append additional aliases to this list, you can add them by extending dockerAlias.

dockerAliases ++= Seq(dockerAlias.value.withTag(Option("stable")),

3.5. Docker Plugin 33
dockerAlias.value.withRegistryHost(Option("registry.internal.
yourdomain.com"))

dockerBuildInit Whether the --init build option should be passed to the Docker build. See Init support for when this may be useful. Defaults to false.

dockerBuildOptions Overrides the default Docker build options. Defaults to Seq("--force-rm", "-t", [dockerAlias]"). This default is expanded if either dockerUpdateLatest or dockerBuildInit is set to true.

dockerExecCommand Overrides the default Docker exec command. Defaults to Seq("docker")

dockerBuildCommand Overrides the default Docker build command. The reason for this is that many systems restrict docker execution to root, and while the accepted guidance is to alias the docker command alias docker='/usr/bin/docker', neither Java nor Scala support passing aliases to sub-processes, and most build systems run builds using a non-login, non-interactive shell, which also have limited support for aliases, which means that the only viable option is to use sudo docker directly. Defaults to Seq([dockerExecCommand], "build", [dockerBuildOptions], ".").

dockerRmiCommand Overrides the default Docker rmi command. This may be used if force flags or other options need to be passed to the command docker rmi. Defaults to Seq([dockerExecCommand], "rmi") and will be directly appended with the image name and tag.

dockerAutoremoveMultiStageIntermediateImages If intermediate images should be automatically removed when MultiStage strategy is used. Intermediate images usually aren’t needed after packaging is finished and therefore defaults to true. All intermediate images are labeled snp-multi-stage=intermediate. If set to false and you want to remove all intermediate images at a later point, you can therefore do that by filtering for this label: docker image prune -f --filter label=snp-multi-stage=intermediate

3.5.5 Tasks

The Docker plugin provides the following commands:

- **Docker / stage** Generates a directory with the Dockerfile and environment prepared for creating a Docker image.
- **Docker / publishLocal** Builds an image using the local Docker server.
- **Docker / publish** Builds an image using the local Docker server, and pushes it to the configured remote repository.
- **Docker / clean** Removes the built image from the local Docker server.

3.5.6 Customize

There are some predefined settings which you can easily customize. These settings are explained in some detail in the next sections. If you want to describe your Dockerfile completely yourself, you can provide your own docker commands as described in Custom Dockerfile.

Docker Image Name and Version
Docker / packageName := packageName.value
Docker / version := version.value

Docker Base Image

dockerBaseImage := "openjdk"

Docker Repository

dockerRepository := Some("dockeruser")

Docker Image Customization

dockerExposedPorts := Seq(9000, 9443)
dockerExposedVolumes := Seq("/opt/docker/logs")

In order to work properly with USER daemon the exposed volumes are first created (if they do not exist) and then chowned.

Install Location

The path to which the application is written can be changed with the location setting. The files from Docker / mappings are extracted underneath this directory.

Docker / defaultLinuxInstallLocation := "/opt/docker"

Daemon User

By default, sbt Native Packager will create a daemon user named demiourgos728 whose UID is set to 1001, and emit USER 1001 since running as non-root is considered the best practice.

The following can be used to emit USER daemon instead:

Docker / daemonUserUid := None
Docker / daemonUser := "daemon"

File Permission

By default, the working directory inside the Docker image is given read-only file permissions set using multi-stage Docker build, which requires Docker 17.5 or later (watch out if you’re using older Minikube).

If you want to make the working directory writable by the running process, here’s the setting:

import com.typesafe.sbt.packager.docker.DockerChmodType
dockerChmodType := DockerChmodType.UserGroupWriteExecute

3.5. Docker Plugin
By default, the shell scripts generated by SBT Native Packager are given `chmod +x` rights. Here’s the setting to do so for other files:

```scala
import com.typesafe.sbt.packager.docker.DockerChmodType

dockerAdditionalPermissions += (DockerChmodType.UserGroupPlusExecute, "/opt/docker/bin/hello")
```

If you don’t want SBT Native Packager to change the file permissions at all here’s a strategy you can choose:

```scala
import com.typesafe.sbt.packager.docker.DockerPermissionStrategy

dockerPermissionStrategy := DockerPermissionStrategy.None
```

This will inherit the file mode bits set in your machine. Given that Kubernetes implementations like OpenShift will use an arbitrary user, remember to set both the user bits and group bits when running `chmod` yourself.

### Custom Dockerfile

All settings before are used to create a single sequence of docker commands. You have the option to write all of them on your own, filter or change existing commands or simply add some.

First of all you should take a look what you docker commands look like. In your sbt console type

```
> show dockerCommands
```

```
List(Cmd(FROM,openjdk:8), Cmd(LABEL,MAINTAINER=Your Name <y.n@yourcompany.com> ...
```

#### Remove Commands

SBT Native Packager adds commands you may not need. For example, the chowning of a exposed volume:

```scala
import com.typesafe.sbt.packager.docker._

// we want to filter the chown command for '/data'
dockerExposedVolumes += "/data"

// use filterNot to return all items that do NOT meet the criteria
dockerCommands := dockerCommands.value.filterNot {
    // ExecCmd is a case class, and args is a varargs variable, so you need to bind it
    with @
    case ExecCmd("RUN", args @ _) => args.contains("chown") && args.contains("/data")

    // don’t filter the rest; don’t filter out anything that doesn’t match a pattern
    case cmd => false
}
```

#### Add Commands

Since `dockerCommands` is just a `Sequence`, adding commands is straightforward:
import com.typesafe.sbt.packager.docker._

// use += to add an item to a Sequence
dockerCommands += Cmd("USER", (Docker / daemonUser).value)

// use ++= to merge a sequence with an existing sequence
dockerCommands ++= Seq{
  // setting the run script executable
  ExecCmd("RUN",
    "chmod", "u+x",
    s"${(Docker / defaultLinuxInstallLocation).value}/bin/${executableScriptName._1.value}"),
  // setting a daemon user
  Cmd("USER", "daemon")
}

Write from Scratch

You can simply wipe out all docker commands with

dockerCommands := Seq()

Now let’s start adding some Docker commands.

import com.typesafe.sbt.packager.docker._

dockerCommands := Seq{
  Cmd("FROM", "openjdk:8"),
  Cmd("LABEL", s"MAINTAINER="${maintainer.value}"""""
  ExecCmd("CMD", "echo", "Hello, World from Docker")
}

Busybox/Ash Support

Busybox is a popular minimal Docker base image that uses ash, a much more limited shell than bash. By default, the Java archetype (Java Application Archetype) generates two files for shell support: a bash file, and a Windows .bat file. If you build a Docker image for Busybox using the defaults, the generated bash launch script will likely not work.

To handle this, you can use AshScriptPlugin, an ash-compatible archetype that is derived from the Java Application Archetype archetype. Enable this by including:

enablePlugins(AshScriptPlugin)

With this plugin enabled an ash-compatible launch script will be generated in your Docker image.

Just like for Java Application Archetype, you have the option of overriding the default script by supplying your own src/templates/ash-template file. When overriding the file don’t forget to include ${{template declares}} somewhere to populate $app_classpath $app_mainclass from your sbt project. You’ll likely need these to launch your program.

Init support

By default, Java will run with PID 1 when you run your docker container. The JVM behaves differently when its PID is 1 compared to other PIDs, most notably, it doesn’t respond to some signals. These include the signals usually used
to instruct a Java process to dump its threads or its heap. If you want to be able to debug a running Java container, the inability to take thread or heap dumps can be a problem.

Docker has a convenient solution to this, it can configure a separate init process for you. This process will start your Java process, and it will also do some other useful things that init processes are meant to do like cleaning up orphaned processes in the container. But most importantly it will ensure that your Java process is not PID 1, which will in turn ensure that your Java process is able to respond to signals for debugging. The command docker uses is tini, which as its name suggests, is tiny, only 23kb in size.

To tell docker to configure a separate init process using tini, set the `dockerBuildInit` setting to `true`:

```scala
dockerBuildInit := true
```

### 3.6 Windows Plugin

Windows packaging is completely tied to the WIX installer toolset. For any non-trivial package, it’s important to understand how WIX works. [http://wix.tramontana.co.hu/](http://wix.tramontana.co.hu/) is an excellent tutorial to how to create packages using wix. However, the native-packager provides a simple layer on top of wix that may be enough for most projects. If it is not enough, just override `wixConfig` or `wixFiles` tasks. Let’s look at the layer above direct xml configuration.

**Note:** The windows plugin depends on the [Universal Plugin](https://developer.typesafe.com/releases/).

#### 3.6.1 Requirements

You need the following applications installed

- WIX Toolset

#### 3.6.2 Build

```scala
sbt Windows/packageBin
```

**Required Settings**

A windows package needs some mandatory settings to be valid. Make sure you have these settings in your build:

```scala
// general package information (can be scoped to Windows)
maintainer := "Josh Suereth <joshua.suereth@typesafe.com>"
packageSummary := "test-windows"
packageDescription := """Test Windows MSI.""

// wix build information
wixProductId := "ce07be71-510d-414a-92d4-dff47631848a"
wixProductUpgradeId := "4552fb0e-e257-4dbd-9ecb-dba9dbcf424"
```
1.0 or higher

Enables the windows plugin

```scala
enablePlugins(WindowsPlugin)
```

0.8 or lower

For these versions windows packaging is automatically activated. See the Getting Started page for information on how to enable sbt-native-packager.

3.6.3 Configuration

Settings and Tasks inherited from parent plugins can be scoped with Universal.

```scala
Windows / mappings := (Universal / mappings).value
```

Now, let’s look at the full set of windows settings.

3.6.4 Settings

- **Windows / name** The name of the generated msi file.
- **candleOptions** the list of options to pass to the candle.exe command.
- **lightOptions** the list of options to pass to the light.exe command. Most likely setting is:
  ```scala
  Seq("-ext", "WixUIExtension", "-cultures:en-us")
  ```
  for UI.
- **wixMajorVersion** the major version of the Wix tool-set (e.g. when using Wix 4.0.1, major version is 4). Default is 3.
- **wixProductId** The GUID to use to identify the windows package/product.
- **wixProductUpgradeId** The GUID to use to identify the windows package/product upgrade identifier (See the wix docs on upgrades).
- **wixPackageInfo** The information used to autoconstruct the `<Product><Package/>` portion of the wix xml. Note: unused if "wixConfig" is overridden
- **wixProductLicense** An (optional) rtf file to display as the product license during installation. Defaults to src/windows/License.rtf
- **wixFeatures** A set of windows features that users can install with this package. Note: unused if "wixConfig" is overridden
- **wixProductConfig** inline XML to use for wix configuration. This is everything nested inside the `<Product>` element.
- **wixConfig** inline XML to use for wix configuration. This is used if the wixFiles task is not specified.
- **wixFiles** WIX xml source files (wxs) that define the build.

"Windows / packageMsi / mappings" A list of file->location pairs. This list is used to move files into a location where WIX can pick up the files and generate a cab or embedded cab for the msi. The WIX xml should use the relative locations in this mappings when referencing files for the package.
3.6.5 Tasks

**Windows/packageBin** Creates the msi package.

**wixFile** Generates the Wix xml file from wixConfig and wixProductConfig settings, unless overridden.

The native-packager plugin provides a few handy utilities for generating Wix XML. These utilities are located in the com.typesafe.packager.windows.WixHelper object. Among these are the following functions:

- `cleanStringForId(String): String` Takes in a string and returns a wix-friendly identifier. Note: truncates to 50 characters.
- `cleanFileName(String): String` Takes in a file name and replaces any $ with $$ to make it past the Wix preprocessor.
- `generateComponentsAndDirectoryXml(File): (Seq[String], scala.xml.Node)` This method will take a file and generate `<Directory>`, `<Component>` and `<File>` XML elements for all files/directories contained in the given file. It will return the Id settings for any generated components. This is a handy way to package a large directory of files for usage in the Features of an MSI.

3.6.6 Customize

Feature configuration

The abstraction over wix allows you to configure “features” that users may optionally install. These feature are higher level things, like a set of files or menu links. The currently supported components of features are:

1. Files (ComponentFile)
2. Path Configuration (AddDirectoryToPath)
3. Menu Shortcuts (AddShortCuts)

To create a new feature, simple instantiate the WindowsFeature class with the desired feature components that are included.

Here’s an example feature that installs a binary file (`cool.jar`) and a script (`cool.bat`), and adds a directory to the PATH:

```scala
wixFeatures += WindowsFeature(
    id="BinaryAndPath",
    title="My Project’s Binaries and updated PATH settings",
    desc="Update PATH environment variables (requires restart).",
    components = Seq(
        ComponentFile("bin/cool.bat"),
        ComponentFile("lib/cool.jar"),
        AddDirectoryToPath("bin")
    )
)
```

All file references should line up exactly with those found in the Windows / mappings configuration. When generating a MSI, the plugin will first create a directory using all the Windows / mappings and configure this for inclusion in a cab file. If you’d like to add files to include, these must first be added to the mappings, and then to a feature. For example, if we complete the above setting to include file mappings, we’d have the following:

```scala
Windows / mappings +++ (Compile / packageBin, Windows / sourceDirectory) map { (jar, _ --dir) =>
    Seq(jar -> "lib/cool.jar", (dir / "cool.bat") -> "bin/cool.bat")
}
```

(continues on next page)
Right now this layer is very limited in what it can accomplish, and hasn’t been heavily debugged. If you’re interested in helping contribute, please do so! However, for most command line tools, it should be sufficient for generating a basic .msi that Windows users can install.

### 3.7 JDKPackager Plugin

This plugin builds on Oracle’s javapackager tool to generate native application launchers and installers for macOS, Windows, and Linux. This plugin takes the settings and staged output from Java Application Archetype and passes them through javapackager to create native formats per Oracle’s provided features.

The actual mechanism used by this plugin is the support provided by the lib/ant-javafx.jar Ant task library, which provides more capabilities than the javapackager command line version, but the idea is the same.

This plugin’s most relevant addition to the core sbt-native-packager capabilities is the generation of macOS App bundles and associated .dmg and .pkg package formats. With this plugin complete drag-and-drop installable application bundles are possible, including the embedding of the JRE. It can also generate Windows .exe and .msi installers provided the requisite tools are available on the Windows build platform (see below). While Linux package formats are also possible via this plugin, it is likely the native sbt-native-packager support for .deb and .rpm formats will provide more configurability.

**Note:** The JDKPackagerPlugin depends on the Universal Plugin, Java Application Archetype and Launcher Plugin

#### 3.7.1 Requirements

The ant-javafx.jar library comes with Oracle JDK 8, found in the lib directory along with tools.jar and friends. If sbt is running under the JVM in Oracle JDK 8, then the plugin should be able to find the path to ant-javafx.jar. If sbt is running under a different JVM, then the path to the tool will have to be specified via the JdkPackager / antPackagerTasks setting.

This plugin must be run on the platform of the target installer. The Oracle tooling does not provide a means of creating, say, Windows installers on MacOS, or MacOS on Linux, etc.

To use create Windows launchers & installers, the either the WIX Toolset (msi) or Inno Setup (exe) is required:

- WIX Toolset
- Inno Setup

For further details on the capabilities of javapackager, see the Windows and Unix references. (Note: only a few of the possible settings are exposed through this plugin. Please submit a Github issue or pull request if something specific is desired.)
3.7.2 Enabling

The plugin is enabled via the `AutoPlugins` facility:

```
enablePlugins(JDKPackagerPlugin)
```

3.7.3 Build

To use, first get your application working per `Java Application Archetype` instructions (including the `mainClass` setting). Once that is working, run

```
sbt JdkPackager/packageBin
```

By default, the plugin makes the installer type that is native to the current build platform. The installer is put in the directory `target/jdkpackager/bundles`. The key `jdkPackageType` can be used to modify this behavior. Run `help jdkPackageType` in `sbt` for details. The most popular setting is likely to be `jdkAppIcon`.

3.7.4 Settings

*For the latest documentation reference the key descriptions in `sbt`.*

**`jdkPackagerBasename`** Filename without the extension for the generated installer package.

**`jdkPackagerType`** Value passed as the `native` attribute to `fx:deploy` task.

Per `javapackager` documentation, this may be one of the following:

- **all**: Runs all of the installers for the platform on which it is running, and creates a disk image for the application.
- **installer**: Runs all of the installers for the platform on which it is running.
- **image**: Creates a disk image for the application. On macOS, the image is the .app file. On Linux, the image is the directory that gets installed.
- **dmg**: Generates a DMG file for macOS.
- **pkg**: Generates a .pkg package for macOS.
- **mac.appStore**: Generates a package for the Mac App Store.
- **rpm**: Generates an RPM package for Linux.
- **deb**: Generates a Debian package for Linux.
- **exe**: Generates a Windows .exe package.
- **msi**: Generates a Windows Installer package.

**Note:** Because only a subset of the possible settings are exposed through the plugin, updates are likely required to fully make use of all formats. `dmg` is currently the most tested type.

**`jdkAppIcon`** Path to platform-specific application icon:

- **icns**: MacOS
- **ico**: Windows
- **png**: Linux
Defaults to a generically bland Java icon. Oracle javafx ant task reference

**jdkPackagerToolkit** GUI toolkit used in app. Either JavaFXToolkit (default) or SwingToolkit.

**jdkPackagerJVMArgs** Sequence of arguments to pass to the JVM.

- Default: Seq("-Xmx768m").
- Oracle JVM argument docs

**jdkPackagerAppArgs** List of command line arguments to pass to the application on launch.

- Default: Seq.empty
- Oracle arguments docs

**jdkPackagerProperties** Map of System properties to define in application.

- Default: Map.empty
- Oracle properties docs

**jdkPackagerAssociations** Set of application file associations to register for the application.

- Example: jdkPackagerAssociations := Seq(FileAssociation("foo", "application/x-foo", Foo Data File", iconPath))
- Default: Seq.empty
- Note: Requires JDK >= 8 build 40.
- Oracle associations docs

### 3.7.5 Example

To take it for a test spin, run sbt JdkPackager/packageBin in the test-project-jdkpackager directory of the sbt-native-packager source. Then look in the target/jdkpackager/bundles directory for the result (specific name depends on platform built).

Here’s what the build file looks like:

```scala
name := "JDKPackagerPlugin Example"
version := "0.1.0"
organization := "com.foo.bar"

libraryDependencies ++= Seq(
  "com.typesafe" % "config" % "1.2.1"
)

Compile / mainClass := Some("ExampleApp")

enablePlugins(JDKPackagerPlugin)

maintainer := "Previously Owned Cats, Inc."

packageSummary := "JDKPackagerPlugin example package thingy"

packageDescription := "A test package using Oracle's JDK bundled javapackager tool."

lazy val iconGlob = sys.props("os.name").toLowerCase match {
  (continues on next page)
case os if os.contains("mac") "*.icns"
case os if os.contains("win") "*.ico"
case _ "*.png"
}

jdkAppIcon := (sourceDirectory.value ++ iconGlob).getPaths.headOption.map(file)

d clandestinoJdkPackagerType := "installer"
jdkPackagerJVMArgs := Seq("-Xmx1g")
jdkPackagerProperties := Map("app.name" -> name.value, "app.version" -> version.value)

jdkPackagerAppArgs := Seq(maintainer.value, packageSummary.value, packageDescription.value)

jdkPackagerAssociations := Seq(
  FileAssociation("foobar", "application/foobar", "Foobar file type"),
  FileAssociation("barbaz", "application/barbaz", "Barbaz file type", jdkAppIcon.value)
)

// Example of specifying a fallback location of `ant-javafx.jar` if plugin can’t find it.
(JDKPackager / antPackagerTasks) := (JDKPackager / antPackagerTasks).value orElse {
  for { f <- Some(file("/usr/lib/jvm/java-8-oracle/lib/ant-javafx.jar")) if f.exists() } yield f
}

3.7.6 Debugging

If you are having trouble figuring out how certain features affect the generated package, you can find the Ant-based build definition file in target/jdkpackager/build.xml. You should be able to run Ant directly in that file assuming JdkPackager / packageBin has been run at least once.

3.8 GraalVM Native Image Plugin

GraalVM’s native-image compiles Java programs AOT (ahead-of-time) into native binaries.


The plugin supports both using a local installation of the GraalVM native-image utility, or building inside a Docker container. If you intend to run the native image on Linux, then building inside a Docker container is recommended since GraalVM native images can only be built for the platform they are built on. By building in a Docker container, you can build Linux native images not just on Linux but also on Windows and macOS.

3.8.1 Requirements

To build using a local installation of GraalVM, you must have the native-image utility of GraalVM in your PATH.
native-image quick installation

To get started quickly, eg make native-image available in your PATH, you may reuse the script that is used for sbt-native-packager’s continuous integration. To do so, run the following. It will install GraalVM 1.0.0-rc8.

```
```

### 3.8.2 Build

```
sbt 'show GraalVMNativeImage/packageBin'
```

### Required Settings

```
enablePlugins(GraalVMNativeImagePlugin)
```

### 3.8.3 Settings

**native-image Executable Command (Pay attention if you are using Windows OS)**

**graalVMNativeImageCommand** Set this parameter to point to native-image or native-image.cmd. Set this parameter if it is inconvenient to make native-image available in your PATH.

For example:

```
graalVMNativeImageCommand := "C:/Program Files/Java/graalvm/bin/native-image.cmd"
```

**Docker Image Build Settings**

By default, a local build will be done, expecting the native-image command to be on your PATH. This can be customized using the following settings.

**graalVMNativeImageGraalVersion** Setting this enables generating a Docker container to build the native image, and then building it in that container. It must correspond to a valid version of the Oracle GraalVM Community Edition Docker image. This setting has no effect if containerBuildImage is explicitly set.

For example:

```
graalVMNativeImageGraalVersion := Some("19.1.1")
```

**containerBuildImage**

Explicitly set a build image to use. The image must execute the Graal native-image command as its entry point. It can be configured like so:

```
containerBuildImage := Some("my-docker-username/graalvm-ce-native-image:19.1.1")
```
A helper is provided to automatically generate a container build image from a base image that contains a Graal installation. For example, if you have a GraalVM enterprise edition docker image, you can turn it into a native image builder like so:

```scala
containerBuildImage := GraalVMNativeImagePlugin.
  →generateContainerBuildImage("example.com/my-username/graalvm-
  →ee:latest")
```

The plugin will not build the native image container builder if it finds it in the local Docker registry already. The native image builders tag name can be seen in the logs if you wish to delete it to force a rebuild, in the above case, the name will be `example.com-my-username-graalvm-ee:latest`.

### Publishing Settings

`graalVMNativeImageOptions` Extra options that will be passed to the `native-image` command. By default, this includes the name of the main class.

### 3.8.4 GraalVM Resources

If you are building the image in a docker container, and you have any resources that need to be available to the `native-image` command, such as files passed to `-H:ResourceConfigurationFiles` or `-H:ReflectionConfigurationFiles`, you can place these in your projects `src/graal` directory. Any files in there will be made available to the `native-image` docker container under the path `/opt/graalvm/stage/resources`.

### 3.8.5 Tasks

The GraalVM Native Image plugin provides the following commands:

`GraalVMNativeImage / packageBin` Generates a native image using GraalVM.
Archetype plugins provide predefined configurations for your build. Like format plugins, archetype plugins can depend on other archetype plugins to extend existing functionality.

Project archetypes are default deployment scripts that try to “do the right thing” for a given type of project. Because not all projects are created equal, there is no single archetype for all native packages, but a set of them to choose from.

4.1 Java Application Archetype

Application packaging focuses on how your application is launched (via a bash or bat script), how dependencies are managed and how configuration and other auxiliary files are included in the final distributable. The JavaAppPackaging archetype provides a default application structure and executable scripts to launch your application.

Additionally there is Java Server Application Archetype which provides platform-specific functionality for installing your application in server environments. You can customize specific debian and rpm packaging for a variety of platforms and init service loaders including Upstart, System V and SystemD.

4.1.1 Features

The JavaAppPackaging archetype contains the following features.

- Default application mappings (no fat jar)
- Executable bash/bat script

4.1.2 Usage

Enable the JavaAppPackaging plugin in your build.sbt with

```
enablePlugins(JavaAppPackaging)
```
This archetype will use the `mainClass` setting of sbt (automatically discovers your main class) to generate `bat` and `bin` scripts for your project. In case you have multiple main classes you can point to a specific class with the following setting:

```
Compile / mainClass := Some("foo.bar.Main")
```

In order to generate launch scripts only for specified `mainClass`, you will need to discard automatically found main classes:

```
Compile / discoveredMainClasses := Seq()
```

To create a staging version of your package call

```
sbt stage
```

The universal layout produced in your `target/universal/stage` folder looks like the following:

```
bin/
    <app_name> <- BASH script
    <app_name>.bat <- cmd.exe script
lib/
    <Your project and dependent jar files here.>
```

You can add additional files to the project by placing things in `src/windows`, `src/universal` or `src/linux` as needed. To see if your application runs:

```
cd target/universal/stage
./bin/<app-name>
```

This plugin also enables all supported `packaging formats` as well. Currently all formats are supported by the java app archetype! For example you can build `zips`, `deb` or `docker` by just enabling `JavaAppPackaging`.

```
sbt
# create a zip file
> Universal / packageBin
# create a deb file
> Debian / packageBin
# publish a docker image to your local registry
> Docker / publishLocal
```

## 4.1.3 Settings & Tasks

This is a non extensive list of important settings and tasks this plugin provides. All settings have sensible defaults.

- **makeBashScripts** Creates or discovers the bash script used by this project.
- **makeBatScripts** Creates or discovers the bat script used by this project.
- **bashScriptTemplateLocation** The location of the bash script template.
- **batScriptTemplateLocation** The location of the bat script template.
- **bashScriptConfigLocation** The location of the bash script on the target system. **Default** ${app_home}/../conf/application.ini
- **batScriptConfigLocation** The location of the bat script on the target system. **Default** %APP_HOME%\conf\application.ini
- **bashScriptExtraDefines** A list of extra definitions that should be written to the bash file template.
\textbf{batScriptExtraDefines} A list of extra definitions that should be written to the bat file template.

\section*{4.1.4 Start script options}

The start script provides a few standard options you can pass:

- \texttt{-h \ | \ -help} Prints script usage
- \texttt{-v \ | \ -verbose} Prints out more information
- \texttt{-no-version-check} Don’t run the java version check
- \texttt{-jvm-debug <port>} Turn on JVM debugging, open at the given port
- \texttt{-java-home <java home>} Override the default JVM home, it accept variable expansions, e.g.
  \begin{verbatim}
  -java-home ${app_home}/../jre
  \end{verbatim}
- \texttt{-main} Define a custom main class

To configure the JVM these options are available

- \texttt{JAVA_OPTS} environment variable, if unset uses “\$java_opts”
- \texttt{-Dkey=val} pass -Dkey=val directly to the java runtime
- \texttt{-J-X} pass option -X directly to the java runtime (-J is stripped). E.g. \texttt{-J-Xmx1024}

In order to pass application arguments you need to separate the jvm arguments from the application arguments with \texttt{--}. For example

\begin{verbatim}
./bin/my-app -Dconfig.resource=prod.conf -- -appParam1 -appParam2
\end{verbatim}

\section*{4.1.5 Multiple Applications}

If you have multiple main classes then the JavaAppPackaging archetype provides you with two different ways of generating start scripts.

1. A start script for each entry point. This is the default behaviour, when no \texttt{Compile / mainClass} is set
2. One start script for the defined \texttt{Compile / mainClass} and forwarding scripts for all other main classes.

\textbf{Note}: What does \textit{‘forwarder script’} mean?

Native-packager’s start script provides a \texttt{-main} option to override the main class that should be executed. A \textit{forwarder script} only overrides this attribute and forwards all other parameters to the normal start script.

All customization you implemented for the main script will also apply for the forwarder scripts.

\section*{Multiple start scripts}

No configuration is needed. SBT sets \texttt{Compile / mainClass automatically to None} if multiple main classes are discovered.

\textbf{Example}:

For two main classes \texttt{com.example.FooMain} and \texttt{com.example.BarMain} sbt stage will generate these scripts:
Single start script with forwarders

Generates a single start script for the defined main class in `Compile / mainClass` and forwarding scripts for all other `Compile / discoveredMainClasses`. The forwarder scripts call the defined start script and set the `-main` parameter to the concrete main class.

The start script name uses the `executableScriptName` setting for its name. The forwarder scripts use a simplified version of the class name.

Example:
The `build.sbt` has an explicit main class set.

```
name := "my-project"
Compile / mainClass := Some("com.example.FooMain")
```

For two main classes `com.example.FooMain` and `com.example.BarMain` sbt stage will generate these scripts:

```
bin/
  bar-main
  bar-main.bat
  foo-main
  foo-main.bat
```

Now you can package your application as usual, but with multiple start scripts.

A note on script names

When this plugin generates script names from main class names, it tries to generate readable and unique names:

1. An heuristic is used to split the fully qualified class names into words:

```
pkg1.TestClass
pkg2.AnUIMainClass
pkg2.SomeXMLLoader
pkg3.TestClass
```

becomes

```
pkg-1.test-class
pkg-2.an-ui-main-class
pkg-2.some-xml-loader
pkg-3.test-class
```

2. Resulted lower-cased names are grouped by the simple class name.
   - Names from single-element groups are reduced to their lower-cased simple names.
   - Names that would otherwise collide by their simple names are used as is (that is, full names) with dots replaced by underscores
So the final names will be:

```
pkg-1_test-class
an-ui-main-class
some-xml-loader
pkg-3_test-class
```

Please note that in some corner cases this may result in multiple scripts with the same name in the resulting archive, but it is not expected to happen in normal circumstances.

### 4.1.6 Customize

The application structure is customizable via the standard mappings, which is described in the *Universal Plugin Section*.

**Application and runtime configuration**

There are generally two types of configurations:

- Configuring the JVM and the process
- Configuring the application itself

You have two options to define your runtime and application configurations.

**Configuration file**

The start scripts provided by the `BatStartScriptPlugin` and `BashStartScriptPlugin` can both load an external configuration file during execution. You can define the configuration file location for both with these two settings.

- **bashScriptConfigLocation** The location of the bash script on the target system.
  - **Default** `${app_home}/../conf/application.ini`
- **batScriptConfigLocation** The location of the bat script on the target system.
  - **Default** `%APP_HOME%\conf\application.ini`

The configuration path is the path on the **target** system. This means that native-packager needs to process this path to create a valid Universal/mapping entry.

- `${app_home}/../` is removed
- `%APP_HOME%` is removed and `/` is being replaced with /

This means you can either

1. Create a configuration path relative to the application directory (recommended)
2. Create an absolute path that has to match your target and build system

**Example**

```scala
// configure two different files for bash and bat
bashScriptConfigLocation := Some("${app_home}/../conf/jvmopts-bash")
batScriptConfigLocation := Some("%APP_HOME%\conf\jvmopts-bat")
```

Now we know how to configure the location of our configuration file. The next step is to learn how to provide content for the configuration file.

### 4.1. Java Application Archetype
Via build.sbt

You can specify your options via the build.sbt.

```scala
Universal / javaOptions ++= Seq(  // -J params will be added as jvm parameters
  "-J-Xmx64m",
  "-J-Xms64m",

  // others will be added as app parameters
  "-Dproperty=true",
  "-port=8080",

  // you can access any build setting/task here
  s"-version=${version.value}"
)
```

For the -X settings you need to add a suffix -J so the start script will recognize these as vm config parameters.

When you use the Universal / javaOptions sbt-native-packager will generate configuration files if you haven’t set the `batScriptConfigLocation` and/or `bashScriptConfigLocation` to None.

Via Application.ini

The second option is to create `src/universal/conf/application.ini` with the following template

```ini
# Setting -X directly (-J is stripped)
# -J-X
-J-Xmx1024

# Add additional jvm parameters
-Dkey=val

# Turn on JVM debugging, open at the given port
# -jvm-debug <port>

# Don't run the java version check
# -no-version-check

# enabling debug and sending -d as app argument
# the ‘--’ prevents app-parameter swallowing when
# using a reserved parameter. See #184
# -d -- -d
```

The file will be installed to `${app_home}/conf/application.ini` and read from there by the startscript. You can use `#` for comments and new lines as you like. This file currently doesn’t has any variable substitution. We recommend using the build.sbt if you need any information from your build.

The configuration file for bash scripts takes arguments for the BASH file on each line, and allows comments which start with the `#` character. Essentially, this provides a set of default arguments when calling the script.

By default, any file in the `src/universal` directory is packaged. This is a convenient way to include things like licenses, and readmes.

If you don’t like `application.ini` as a name, you can change this in the build.sbt. The default configuration looks like this
bashScriptConfigLocation := Some("${app_home}/../conf/application.ini")
bashScriptConfigLocation := Some("%APP_HOME%\conf\application.ini")

Add code to the start scripts

The second option is to add code to the generated start scripts via these settings.

- **bashScriptExtraDefines** A list of extra definitions that should be written to the bash file template.
- **batScriptExtraDefines** A list of extra definitions that should be written to the bat file template.

**BashScript defines**

The bash script accepts extra commands via `bashScriptExtraDefines`. Generally you can add arbitrary bash commands here, but for configurations you have two methods to add jvm and app parameters.

```scala
// add jvm parameter for typesafe config
bashScriptExtraDefines += "addJava "-Dconfig.file=${app_home}/../conf/app.config"
// add application parameter
bashScriptExtraDefines += "addApp "--port=8080"
```

**Syntax**

`{{template Declares}}` Will be replaced with a series of `declare <var>` lines based on the `bashScriptDefines` key. These variables are predefined:

- **app_mainclass** - The main class entry point for the application.
- **app_classpath** - The complete classpath for the application.

**BatScript defines**

The Windows batch script accepts extra commands via `batScriptExtraDefines`. It offers two methods to add jvm and app parameters using similar syntax to the BASH script.

```bash
// add jvm parameter for typesafe config
batScriptExtraDefines += "call :add_java "-Dconfig.file=%APP_HOME%\conf\app.config"
// add application parameter
batScriptExtraDefines += "call :add_app "--port=8080"
```

**Syntax**

`@@APP_ENV_NAME@@` will be replaced with the script friendly name of your package.

`@@APP_NAME@@` will be replaced with user friendly name of your package.

`@APP_DEFINES@` will be replaced with a set of variable definitions, like `APP_MAIN_CLASS`, `APP_CLASSPATH`.

**Start script customizations**

While the native packager tries to provide robust BASH/BAT scripts for your applications, they may not always be enough. The native packager provides a mechanism where the template used to create each script can be customized or directly overridden.
Bash and Bat script extra defines

For the bat and bash script are separated settings available to add arbitrary code to the start script. See BashScript defines and BatScript defines for details.

The bashScriptExtraDefines sequence allows you to add new lines to the default bash script used to start the application. This is useful when you need a setting which isn’t mean for the command-line parameter list passed to the java process. The lines added to bashScriptExtraDefines are placed near the end of the script and have access to a number of utility bash functions (e.g. addJava, addApp, addResidual, addDebugger). You can add lines to this script as we did for the Typesafe config file above. For more complex scripts you can also inject a separate file managed in your source tree or resource directory:

```
bashScriptExtraDefines +++= IO.readLines(baseDirectory.value / "scripts" / "extra.sh")
```

This will add the contents of /scripts/extra.sh in the resource directory to the bash script. Note you should always concatenate lines to bashScriptExtraDefines as other stages in the pipeline may be include lines to the start-script.

Overriding Templates (Bash/Bat)

**Warning:** Replacing the default templates should really only be done if:

1. There is a bug in one of the script templates you need to workaround
2. There is a deficiency in the features of one of the templates you need to fix.

In general, the templates are intended to provide enough utility that customization is only necessary for truly custom scripts.

In order to override full templates, like the default bash script, you can create a file in src/templates/bash-template. Alternatively, you can use a different file location by setting bashScriptTemplateLocation. There are

Similarly the windows BAT template can be overridden by placing a new template in src/templates/bat-template. You can also use a different file location by setting batScriptTemplateLocation.

4.2 Java Server Application Archetype

**Hint:** Supports only deb and rpm packaging. No support for Windows or macOS

In the Java Application Archetype section we described how to build and customize settings related to an application. The server archetype adds additional features you may need when running your application as a service on a server. SBT Native Packager ships with a set of predefined install and uninstall scripts for various platforms and service managers.

4.2.1 Features

The JavaServerAppPackaging archetype depends on the Java Application Archetype and adds the following features

- daemon user/group support
• default mappings for server applications
  */var/log/<pkg> is symlinked from <install>/logs */
  /var/run/<pkg> owned by daemonUser
• etc-default support

4.2.2 Usage

```scala
enablePlugins(JavaServerAppPackaging)
```

Everything else works the same way as the Java Application Archetype.

**Tip:** If you want your application to be registered as a service enable a Systemloaders plugin.

4.2.3 Settings & Tasks

This is a non extensive list of important settings and tasks this plugin provides. All settings have sensible defaults.

- `daemonUser` User to start application daemon
- `daemonUserUid` UID of daemonUser
- `daemonGroup` Group to place daemonUser to
- `daemonGroupId` GID of daemonGroup
- `daemonShell` Shell provided for the daemon user
- `daemonStdoutLogFile` Filename stdout/stderr of application daemon. Now it’s supported only in SystemV

4.2.4 Default Mappings

The java server archetype creates a default package structure with the following access rights. `<package>` is a placeholder for your actual application name. By default this is `normalizedName`.

<table>
<thead>
<tr>
<th>Folder</th>
<th>User</th>
<th>Permissions</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/share/&lt;package&gt;</td>
<td>root</td>
<td>755 / 655</td>
<td>static, non-changeable files</td>
</tr>
<tr>
<td>/etc/default/&lt;package&gt;</td>
<td>root</td>
<td>644</td>
<td>default config file</td>
</tr>
<tr>
<td>/etc/&lt;package&gt;</td>
<td>root</td>
<td>644</td>
<td>config folder -&gt; link to /usr/share/&lt;package-name&gt;/conf</td>
</tr>
<tr>
<td>/var/run/&lt;package&gt;</td>
<td>daemon</td>
<td>644</td>
<td>if the application generates a pid on its own</td>
</tr>
<tr>
<td>/var/log/&lt;package&gt;</td>
<td>daemon</td>
<td>644</td>
<td>log folder -&gt; symlinked from /usr/share/&lt;package&gt;/log</td>
</tr>
</tbody>
</table>

You can read more on best practices on wikipedia filesystem hierarchy, debian policies and in this native packager discussion.

If you want to change something in this predefined structure read more about it in the linux section.

4.2.5 Customize
Application Configuration

After creating a package, the very next thing needed, usually, is the ability for users/ops to customize the application once it’s deployed. Let’s add some configuration to the newly deployed application.

There are generally two types of configurations:

- Configuring the JVM and the process
- Configuring the Application itself.

The server archetype provides you with a special feature to configure your application with a single file outside of customizing the `bash` or `bat` script for applications. As this file is OS dependent, each OS gets section.

Linux Configuration

There are different ways described in _Customizing the Application_ and can be used the same way.

The server archetype adds an additional way with an _etc-default_ file placed in _src/templates_, which currently only works for _SystemV_ and _systemd_. The file gets sourced before the actual startscript is executed. The file will be installed to `/etc/default/<normalizedName>`

Example `/etc/default/<normalizedName>` for SystemV:

```bash
# Available replacements
# ------------------------------------------------
# ${{author}} package author
# ${{descr}} package description
# ${{exec}} startup script name
# ${{chdir}} app directory
# ${{retries}} retries for startup
# ${{retryTimeout}} retry timeout
# ${{app_name}} normalized app name
# ${{daemon_user}} daemon user
# ------------------------------------------------

# Setting JAVA_OPTS
# -----------------
JAVA_OPTS="-Dpidfile.path=/var/run/$app_name/play.pid $JAVA_OPTS"

# For rpm/systemv you need to set the PIDFILE env variable as well
PIDFILE="/var/run/$app_name/play.pid"

# export env vars for 3rd party libs
# ----------------------------------
COMPANY_API_KEY=123abc
export COMPANY_API_KEY
```

Daemon User and Group

Customize the daemon user and group for your application with the following settings.

```bash
// a different daemon user
Linux / daemonUser := "my-user"
// if there is an existing one you can specify the uid
Linux / daemonUserUid := Some("123")
```

(continues on next page)
Environment variables

The usual JAVA_OPTS can be used to override settings. This is a nice way to test different jvm settings with just restarting the jvm.

Windows Configuration

Support planned.

Systemloader Configuration

See the Systemloaders documentation on how to add a systemloader (e.g. SystemV, Systemd or Upstart) to your package.

Package Lifecycle Configuration

Some scripts are covered in the standard application type. Read more on Java Application Customization. For the java_server package lifecycle scripts are customized to provide the following additional features

- Chowning directories and files correctly (if necessary)
- Create/ Delete users and groups according to your mapping
- Register application at your init system

For this purpose sbt-native-packager ships with some predefined templates. These can be overridden with different techniques, depending on the packaging system.

Partially Replace Template Functionality

Most sbt-native-packager scripts are broken up into partial templates in the resources directory. You can override these default template snippets by adding to the linuxScriptReplacements map. As an example you can change the loader-functions which starts/stop services based on a certain `ServerLoader`:

```scala
linuxScriptReplacements += "loader-functions" -> TemplateWriter.
   --generateScript(getClass.getResource("/custom-loader-functions"), Nil)
```

The custom-loader-functions file must declare the startService() and stopService() functions used in various service management scripts.

RPM Scriptlets

RPM puts all scripts into one file. To override or append settings to your scriptlets use Rpm / maintainerScripts or these `RpmConstants_`'s:
Pre %pre scriptlet
Post %post scriptlet
Pretrans %pretrans scriptlet
Posttrans %posttrans scriptlet
Preun “%preun scriptlet”
Postun %postun scriptlet
Verifyscript %verifyscript scriptlet

If you want to have your files separated from the build definition use the default location for rpm scriptlets. To override default templates in a RPM build put the new scriptlets in the rpmScriptletsDirectory (by default src/rpm/scriptlets).

RpmConstants.Scriptlets By default to src/rpm/scriptlets. Place your templates here.

Available templates are:
post-rpm pre-rpm postun-rpm preun-rpm

The corresponding maintainer file names are:
pretrans post pre postun preun verifyscript posttrans

Override Postinst scriptlet

By default the post-rpm template only starts the service, but doesn’t register it.

```
service ${app_name} start
```

For CentOS we can do

```
chkconfig ${app_name} defaults
service ${app_name} start || echo "${app_name} could not be started. Try manually with service ${app_name} start"
```

For RHEL

```
update-rc.d ${app_name} defaults
service ${app_name} start || echo "${app_name} could not be started. Try manually with service ${app_name} start"
```

Debian Control Scripts

To override default templates in a Debian build put the new control files in the debianControlScriptsDirectory (by default src/debian/DEBIAN).

```
debianControlScriptsDirectory By default to src/debian/DEBIAN. Place your templates here.
```

debianMakePreinstScript creates or discovers the preinst script used by this project.
debianMakePrermScript creates or discovers the prerm script used by this project.
debianMakePostinstScript creates or discovers the postinst script used by this project.
debianMakePostrmScript creates or discovers the postrm script used by this project.
Available templates are

```
  postinst  preinst  postun  preun
```

**Linux Replacements**

This is a list of values you can access in your templates

```
$\{\text{author}\}
$\{\text{descr}\}
$\{\text{exec}\}
$\{\text{chdir}\}
$\{\text{retries}\}
$\{\text{retryTimeout}\}
$\{\text{app_name}\}
$\{\text{daemon_user}\}
$\{\text{daemon_group}\}
```

**Attention:** Every replacement corresponds to a single setting or task. For the `linuxScriptReplacements` you need to override the setting/task in the `Linux` scope. For example

```
Linux / daemonUser := "new-user"
```

overrides the `daemon_user` in the `linuxScriptReplacements`.

**Example Configurations**

A list of very small configuration settings can be found at [sbt-native-packager-examples](#)

### 4.3 Systemloaders

SBT native packager provides support for different systemloaders in order to register your application as a service on your target system, start it automatically and provide systemloader specific configuration.

**Tip:** You can use systemloaders with the [Java Application Archetype](#) or the [Java Server Application Archetype](#)!

#### 4.3.1 Overview

There is a generic `SystemloaderPlugin` which configures default settings and requires necessary plugins. It gets triggered automatically, when you enable a specific systemloader plugin. If you want to implement your own loader, you should require the `SystemloaderPlugin`.

**General Settings**

`serverLoading` Loading system to be used for application start script (SystemV, Upstart, Systemd). This setting can be used to trigger systemloader specific behaviour in your build.
**serviceAutostart** Determines if service will be automatically started after installation. The default value is true.

**startRunlevels** Sequence of runlevels on which application will start up

**stopRunlevels** Sequence of runlevels on which application will stop

**requiredStartFacilities** Names of system services that should be provided at application start

**requiredStopFacilities** Names of system services that should be provided at application stop

**killTimeout** Timeout before sigkill on stop (after term)

**termTimeout** Timeout before sigterm on stop

**retries** Number of retries to start service

**retryTimeout** Timeout between retries in seconds

### 4.3.2 SystemV

Native packager provides different SysV scripts for `rpm` (CentOS, RHEL, Fedora) and `debian` (Debian, Ubuntu) package based systems. Enable SystemV with:

```java
enablePlugins(SystemVPlugin)
```

The *Java Server Application Archetype* provides a `daemonStdoutLogFile` setting, that you can use to redirect the systemV output into a file.

### 4.3.3 Systemd

In order to enable Systemd add this plugin:

```java
enablePlugins(SystemdPlugin)
```

**Settings**

**systemdSuccessExitStatus** Takes a list of exit status definitions that when returned by the main service process will be considered successful termination, in addition to the normal successful exit code 0 and the signals `SIGHUP`, `SIGINT`, `SIGTERM`, and `SIGPIPE`. Exit status definitions can either be numeric exit codes or termination signal names.

**systemdIsServiceFileConfig** Should file app_name.service be marked as config. Default is `true`. If it is set to `true`, file will be marked `%config` in rpm package for example.

### 4.3.4 Upstart

SystemV alternative developed by Ubuntu. Native packager adds support for rpm as well, but we recommend using Systemd if possible.

```java
enablePlugins(UpstartPlugin)
```

As a side note Fedora/RHEL/Centos family of linux specifies Default requiretty in its `/etc/sudoers` file. This prevents the default Upstart script from working correctly as it uses sudo to run the application as the daemonUser. Simply disable requiretty to use Upstart or modify the Upstart template.
4.3.5 Customization

Native packager provides general settings to customize the created systemloader scripts.

Start Script Location

In order to change the location of the systemloader script/config file you need to adjust the `defaultLinuxStartScriptLocation` like this:

```scala
Debian    / defaultLinuxStartScriptLocation := "/lib/systemd/system"
```

You may need to change these paths according to your distribution. References are

- Ubuntu systemd documentation
- Debian systemd documentation
- RHEL systemd documentation

Customize Start Script

Sbt Native Packager leverages templating to customize various start/stop scripts and pre/post install tasks. As an example, to alter the `loader-functions` which manage the specific start and stop process commands for System-Loaders you can to the `linuxScriptReplacements` map:

```scala
import com.typesafe.sbt.packager.archetypes.TemplateWriter

linuxScriptReplacements += {
  val functions = sourceDirectory.value / "templates" / "custom-loader-functions"
  // Nil == replacements. If you want to replace stuff in your script put them in this Seq[(String,String)]
  "loader-functions" -> TemplateWriter.generateScript(functions.toURL, Nil)
}
```

which will add the following resource file to use start/stop instead of initctl in the post install script:

```scala
startService() {
  app_name=$1
  start $app_name
}

stopService() {
  app_name=$1
  stop $app_name
}
```

The `debian` and `redhat` pages have further information on overriding distribution specific actions.

Override Start Script

It’s also possible to override the entire script/configuration for your service manager. Create a file `src/templates/systemloader/$loader/$template` and it will be used instead.

Possible values:

- `$loader - upstart, systemv or systemd`
• $template -
  - systemv-loader-functions, start-debian-template, or start-rpm-template
  - systemd-loader-functions or start-template
  - upstart-loader-functions or start-template

Syntax
You can use ${variable_name} to reference variables when writing your script. The default set of variables is:

• descr - The description of the server.
• author - The configured author name.
• exec - The script/binary to execute when starting the server
• chdir - The working directory for the server.
• retries - The number of times to retry starting the server.
• retryTimeout - The amount of time to wait before trying to run the server.
• app_name - The name of the application (linux friendly)
• app_main_class - The main class / entry point of the application.
• app_classpath - The (ordered) classpath of the application.
• daemon_user - The user that the server should run as.
• daemon_log_file - Absolute path to daemon log file.

4.4 Configuration Archetypes

This is a small collection of additional archetypes that provide smaller enhancements.

4.4.1 AshScript Plugin

This class is an alternate to JavaAppPackaging designed to support the ash shell. Java Application Archetype generates bash-specific code that is not compatible with ash, a very stripped-down, lightweight shell used by popular micro base Docker images like BusyBox. The AshScriptPlugin will generate simple ash-compatible output.

```scala
enablePlugins(AshScriptPlugin)
```

4.4.2 ClasspathJar & LauncherJar Plugin

See the Dealing with long classpaths section for usage of these plugins.

4.5 Jlink Plugin

This plugin builds on Java’s jlink tool to embed a JVM image (a stripped-down JRE) into your package. It produces a JVM image containing only the modules that are referenced from the dependency classpath.
Note: Current implementation only detects the platform modules (that is, the ones present in the JDK used to build the image). Modular JARs and directories are packaged as specified by the UniversalPlugin.

```
enablePlugins(JlinkPlugin)
```

The plugin requires Oracle JDK 11 or OpenJDK 11. Although jlink and jdeps are also a part of the older JDK versions, those lack some of the newer features, which was not addressed in the current plugin version.

This plugin must be run on the platform of the target installer. The tooling does not provide a means of creating, say, Windows installers on MacOS, or MacOS on Linux, etc.

The plugin analyzes the dependencies between packages using jdeps, and raises an error in case of a missing dependency (e.g. for a provided transitive dependency). The missing dependencies can be suppressed on a case-by-case basis (e.g. if you are sure the missing dependency is properly handled):

```
jlinkIgnoreMissingDependency := JlinkIgnore.only(
  "foo.bar" -> "bar.baz",
  "foo.bar" -> "bar.qux"
)
```

For large projects with a lot of dependencies this can get unwieldy. You can use a more flexible ignore strategy:

```
jlinkIgnoreMissingDependency := JlinkIgnore.byPackagePrefix(
  "foo.bar" -> "bar"
)
```

Otherwise you may opt out of the check altogether (which is not recommended):

```
jlinkIgnoreMissingDependency := JlinkIgnore.everything
```

### 4.5.1 Known issues

Adding some library dependencies can lead to errors like this:

```
java.lang.module.FindException: Module paranamer not found, required by com.fasterxml.jackson.module.paranamer
```

This is often caused by depending on automatic modules. In the example above, `com.fasterxml.jackson.module.paranamer` is an explicit module (as in, it is a JAR with a module descriptor) that defines a dependency on the `paranamer` module. However, there is no explicit `paranamer` module - instead, Jackson expects us to use the `paranamer` JAR file as an automatic module. To do this, the JAR has to be on the module path. At the moment `JlinkPlugin` does not put it there automatically, so we have to do that ourselves:

```
jlinkModulePath := {
  // Get the full classpath with all the resolved dependencies.
  fullClasspath.in(jlinkBuildImage).value
  // Find the ones that have 'paranamer' as their names.
  .filter { item =>
    item.get(moduleID.key).exists { modId =>
      modId.name == "paranamer"
    }
  }
  // Get raw 'File' objects.
  .map(_.data)
}
```

### 4.5. Jlink Plugin
4.5.2 Further reading

For further details on the capabilities of jlink, see the jlink and jdeps references. (Note: only some of the possible settings are exposed through this plugin. Please submit a Github issue or pull request if something specific is desired.)

4.6 Archetype Cheatsheet

This is a set FAQ composed on a single page.

4.6.1 Path Configurations

This section describes where and how to configure different kind of paths settings like

- what is the installation location of my package
- where is the log directory created
- what is the name of my start script

**Quick Reference Table**

This table gives you a quick overview of the setting and the scope you should use. Paths which do not begin with a / are relative to the universal directory. The scopes are ordered from general to specific, so a more specific one will override the generic one. Only the listed scopes for a setting a relevant. Any changes in other scopes will have no effect!

<table>
<thead>
<tr>
<th>output path</th>
<th>scopes</th>
<th>archetype</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib</td>
<td>all</td>
<td>JavaApp</td>
<td></td>
</tr>
<tr>
<td>conf</td>
<td>all</td>
<td>JavaApp</td>
<td></td>
</tr>
<tr>
<td>bin/&lt;executableScriptName&gt;</td>
<td>Global</td>
<td>JavaApp</td>
<td></td>
</tr>
<tr>
<td>bin/&lt;executableScriptName&gt;.bat</td>
<td>Global</td>
<td>JavaApp</td>
<td></td>
</tr>
<tr>
<td>bin/&lt;executableScriptName&gt;</td>
<td>Global</td>
<td>JavaApp</td>
<td>Entry point DockerPlugin</td>
</tr>
<tr>
<td>&lt;defaultLinuxInstallationLocation&gt;</td>
<td>Global</td>
<td>JavaApp</td>
<td></td>
</tr>
<tr>
<td>&lt;defaultLinuxLogLocation&gt;/packageName</td>
<td>Linux, Debian, Rpm</td>
<td>JavaServerApplication</td>
<td></td>
</tr>
<tr>
<td>logs</td>
<td>Linux</td>
<td>JavaServerApplication</td>
<td>Symlink</td>
</tr>
<tr>
<td>/etc/default/&lt;packageName&gt;</td>
<td>Linux</td>
<td>JavaServerApplication</td>
<td></td>
</tr>
<tr>
<td>/var/run/&lt;packageName&gt;</td>
<td>Linux</td>
<td>JavaServerApplication</td>
<td></td>
</tr>
<tr>
<td>/etc/init.d/&lt;packageName&gt;</td>
<td>Linux, Debian, Rpm</td>
<td>JavaServerApplication</td>
<td>For SystemV</td>
</tr>
<tr>
<td>/etc/init/&lt;packageName&gt;</td>
<td>Linux, Debian, Rpm</td>
<td>JavaServerApplication</td>
<td>For Upstart</td>
</tr>
<tr>
<td>/usr/lib/systemd/system/&lt;packageName&gt;.service</td>
<td>Linux, Debian, Rpm</td>
<td>JavaServerApplication</td>
<td>For Systemd</td>
</tr>
<tr>
<td>&lt;defaultLinuxInstallLocation&gt;</td>
<td>Docker</td>
<td>Installation path inside the container</td>
<td></td>
</tr>
</tbody>
</table>
Settings

These settings configure the path behaviour

- **name**: Use for the normal jar generation process
- **packageName**: Defaults to `normalizedName`. Can be override in different scopes
- **executableScriptName**: Defaults to `normalizedName`. Sets the name of the executable starter script
- **defaultLinuxInstallLocation**: Defaults to `/usr/share/`. Used to determine the installation path for for linux packages (rpm, debian)
- **defaultLinuxLogsLocation**: Defaults to `/var/log/`. Used to determine the log path for linux packages (rpm, debian).

### 4.6.2 JVM Options

JVM options can be added via different mechanisms. It depends on your use case which is most suitable. The available options are

- Adding via `bashScriptExtraDefines` and `batScriptExtraDefines`
- Providing a `application.ini` (JavaApp) or `etc-default` (JavaServer) file
- Set Universal / `javaOptions` (JavaApp) or Linux / `javaOptions` (JavaServer, linux only)

**Warning**: If you want to change the location of your config keep in mind that the path in `bashScriptConfigLocation` should either - be **absolute** (e.g. `/etc/etc-default/my-config<`) or - starting with `${app_home}/../` (e.g. `${app_home}/../conf/application.ini`)

Extra Defines

With this approach you are altering the bash/bat script that gets executed. Your configuration is literally woven into it, so it applies to any archetype using this bashscript (app, akka app, server, ...).

For a bash script this could look like this.

```bash
bashScriptExtraDefines += """addJava "-Dconfig.file=${app_home}/../conf/app.config"
bashScriptExtraDefines ++= Seq(
  """addJava "-Xms1024m"
  """addJava "-Xmx2048m"
)
```

For more information take a look at the [according documentation](#).

**File - application.ini or etc-default**

Another approach would be to provide a file that is read by the bash script during execution.
**Java App**

Create a file `src/universal/conf/application.ini` (gets automatically added to the package mappings) and add this to your `build.sbt` inject the config location into the bashscript.

```
bashScriptConfigLocation := Some("${app_home}/../conf/application.ini")
```

**Java Server**

See `Server App Config - src/templates/etc-default-{systemv,systemd}`

**Setting - javaOptions**

The last option to set your java options is using Universal / javaOptions (JavaApp and Server). This will generate files according to your archetype. The following table gives you an overview what you can use and how things will be behave if you mix different options. Options lower in the table are more specific and will thus override the any previous settings (if allowed).

<table>
<thead>
<tr>
<th>javaOpt</th>
<th>Dependent on</th>
<th>bashScriptConfigLocation</th>
<th>Archetype mappings</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil</td>
<td>Universal</td>
<td>None</td>
<td>JavaApp</td>
<td>No jvm options</td>
</tr>
<tr>
<td>Nil</td>
<td>Universal</td>
<td>Some(app_home)</td>
<td>JavaApp</td>
<td>User provides the application.ini file in src/universal/conf/application.ini</td>
</tr>
<tr>
<td>opts</td>
<td>Universal</td>
<td>Some(_)</td>
<td>JavaApp</td>
<td>creates application.ini but leaves bashScriptConfigLocation unchanged</td>
</tr>
<tr>
<td>opts</td>
<td>Universal</td>
<td>None</td>
<td>JavaApp</td>
<td>creates application.ini and sets bashScriptConfigLocation. If src/universal/conf/application.ini is present it will be overridden</td>
</tr>
<tr>
<td>Nil</td>
<td>Linux</td>
<td>None</td>
<td>JavaServer</td>
<td>creates etc-default and sets bashScriptConfigLocation</td>
</tr>
<tr>
<td>opts</td>
<td>Linux</td>
<td>None</td>
<td>JavaServer</td>
<td>creates etc-default, appends Linux / javaOptions and sets bashScriptConfigLocation</td>
</tr>
<tr>
<td>opts</td>
<td>Linux</td>
<td>Some(_)</td>
<td>JavaServer</td>
<td>creates etc-default, appends Linux / javaOptions and overrides bashScriptConfigLocation</td>
</tr>
</tbody>
</table>

**4.6.3 Overriding Templates**

You can override the default template used to generate any of the scripts in any archetype. Listed below are the overridable files and variables that you can use when generating scripts.

**Bat Script - src/templates/bat-template**

Creating a file here will override the default template used to generate the `.bat` script for windows distributions.

**Syntax**
@@APP_ENV_NAME@@ - will be replaced with the script friendly name of your package.

@@APP_NAME@@ - will be replaced with user friendly name of your package.

@APP_DEFINES@ - will be replaced with a set of variable definitions, like APP_MAIN_CLASS, APP_MAIN_CLASS.

You can define additional variable definitions using batScriptExtraDefines.

**Bash Script - src/templates/bash-template**

Creating a file here will override the default template used to generate the BASH start script found in bin/<application> in the universal distribution

**Syntax**

$\{\{template declares\}\} - Will be replaced with a series of declare <var> lines based on the bashScriptDefines key. You can add more defines to the bashScriptExtraDefines that will be used in addition to the default set:

- app_mainclass - The main class entry point for the application.
- app_classpath - The complete classpath for the application (in order).

**Service Manager Templates**

It’s also possible to override the entire script/configuration templates for your service manager. These templates vary by loader type. Create a file src/templates/systemloader/$loader/$template and it will be used instead.

Possible values:

- $loader - upstart, systemv or systemd
- $template - * systemv - loader-functions, start-debian-template, or start-rpm-template * systemd - loader-functions or start-template * upstart - loader-functions or start-template

**Syntax**

You can use $\{(variable_name)\} to reference variables when writing your script. The default set of variables is:

- author - The name of the author; defined by Linux / maintainer.
- descr - The short description of the service; defined by Linux / packageSummary.
- exec - The script/binary to execute when starting the service; defined by Linux / executableScriptName.
- chdir - The working directory for the service; defined by defaultLinuxInstallLocation/(Linux / packageName).
- retries - The number of times to retry starting the server; defined to be the constant 0.
- retryTimeout - The amount of time to wait before trying to run the server; defined to be the constant 60.
- app_name - The name of the application (linux friendly); defined by Linux / packageName.
- version - The software version; defined by version.
- daemon_user - The user that the service should run as; defined by Linux / daemonUser.
**Server App Config - src/templates/etc-default-{systemv,systemd}**

Creating a file here will override the `/etc/default/<application>` template for the corresponding loader.

The file `/etc/default/<application>` is used as follows given the loader:

- **systemv**: sourced as a bourne script.
- **systemd**: used as an EnvironmentFile directive parameter (see `man systemd.exec`, section `EnvironmentFile` for a description of the expected file format).
- **upstart**: presently ignored.

If you’re only overriding `JAVA_OPTS`, your environment file could be compatible with both systemv and systemd loaders; if such is the case, you can specify a single file at `src/templates/etc-default` which will serve as an override for all loaders.
5.1 Custom Package Formats

This section provides an overview of different packaging flavors.

5.1.1 SBT Assembly

Main Goal

Create a fat-jar with sbt-assembly in order to deliver a single, self-containing jar as a package instead of the default lib/ structure.

First add the sbt-assembly plugin to your `plugins.sbt` file.

```sbt
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

The next step is to remove all the jar mappings from the normal mappings and only add the assembly jar. In this example we’ll set the assembly jar name ourself, so we know exactly what the output should look like. Finally we change the `scriptClasspath` so it only contains the assembled jar. This is what the final `build.sbt` should contain:

```sbt
import AssemblyKeys._

// the assembly settings
assemblySettings
```

(continues on next page)
// we specify the name for our fat jar
assembly / jarName := "assembly-project.jar"

// using the java server for this application. java_application is fine, too
packageArchetype.java_server

// removes all jar mappings in universal and appends the fat jar
Universal / mappings := {
  // universalMappings: Seq[(File,String)]
  val universalMappings = (Universal / mappings).value
  val fatJar = (Compile / assembly).value
  // removing means filtering
  val filtered = universalMappings filter {
    case (file, name) => ! name.endsWith(".jar")
  }
  // add the fat jar
  filtered :+ (fatJar -> ("lib/" + fatJar.getName))
}

// the bash scripts classpath only needs the fat jar
scriptClasspath := Seq( (assembly / jarName).value )

5.1.2 Proguard

Main Goal

Create a package that contains a single fat-jar that has been shrunken / optimized / obfuscated with proguard.

First add the sbt-proguard plugin to the plugins.sbt file:

addSbtPlugin("com.lightbend.sbt" % "sbt-proguard" % "0.3.0")

Then configure the proguard options in build.sbt:

enablePlugins(SbtProguard)

// to configure proguard for scala, see
// http://proguard.sourceforge.net/manual/examples.html#scala
Proguard / proguardOptions ::= Seq{
  
  "-dontoptimize",
  "-dontnote",
  "-dontwarn",
  "-ignorewarnings",
  // ...
}

// specify the entry point for a standalone app
Proguard / proguardOptions += ProguardOptions.keepMain("com.example.Main")

Proguard / proguardVersion := "6.0.3"
// filter out jar files from the list of generated files, while
// keeping non-jar output such as generated launch scripts
Universal / mappings := (Universal / mappings).value.
  filter {
    case (file, name) => !name.endsWith(".jar")
  }

// ... and then append the jar file emitted from the proguard task to
// the file list
Universal / mappings += (Proguard / proguard).value.map(jar =>
  jar -> ("lib/" + jar.getName))

// point the classpath to the output from the proguard task
scriptClasspath := (Proguard / proguard).value.map(jar =>
  jar.getName)

Now when you package your project using a command such as sbt Universal/packageZipTarball, it will include fat jar that has been created by proguard rather than the normal output in /lib.

### 5.1.3 Multi Module Builds

**Main Goal**

Aggregate multiple projects into one native package

If you want to aggregate different projects in a multi module build to a single package, you can specify everything in a single build.sbt

```scala
import NativePackagerKeys._
name := "mukis-fullstack"

// used like the groupId in maven
ThisBuild / organization := "de.mukis"

// all sub projects have the same version
ThisBuild / version := "1.0"

ThisBuild / scalaVersion := "2.11.2"

// common dependencies
ThisBuild / libraryDependencies += Seq(
  "com.typesafe" % "config" % "1.2.0"
)

// this is the root project, aggregating all sub projects
lazy val root = Project(
  id = "root",
  base = file("."),
  // configure your native packaging settings here
  settings = packageArchetype.java_server++ Seq(
    maintainer := "John Smith <john.smith@example.com>",
    packageDescription := "Fullstack Application",
  )
)
```

(continues on next page)
packageSummary := "Fullstack Application",
  // entrypoint
  Compile / mainClass := Some("de.mukis.frontend.ProductionServer")
),
  // always run all commands on each sub project
  aggregate = Seq(frontend, backend, api)
) dependsOn(frontend, backend, api) // this does the actual aggregation

// --------- Project Frontend ------------------
lazy val frontend = Project(
  id = "frontend",
  base = file("frontend")
) dependsOn(api)

// --------- Project Backend ----------------
lazy val backend = Project(
  id = "backend",
  base = file("backend")
) dependsOn(api)

// --------- Project API ------------------
lazy val api = Project(
  id = "api",
  base = file("api")
)

5.1.4 Custom Packaging Format

Main Goal

Use native packager to define your own custom packaging format
and reuse stuff you already like

The very core principle of native packager are the mappings. They are a sequence of File -> String tuples, that map a file on your system to a location on your install location.

Defining a custom mapping format is basically transforming these mappings into the format of you choice. To do so, we recommend the following steps

1. Create a new configuration scope for you packaging type
2. Define a packageBin task in your new scope that transforms the mappings into a package

The following examples demonstrates how to create a simple text format, which lists all your mappings inside a package format. A minimal build.sbt would look like this

import NativePackagerKeys._
val TxtFormat = config("txtFormat")
val root = project.in(file("."))
  // adding your custom configuration scope

(continues on next page)
To create your new “packageFormat” just run

```
TxtFormat / packageBin
```

If you want to read more about sbt configurations:

- sbt tasks
- sbt configurations
- custom configuration

### 5.2 Dealing with long classpaths

By default, when the native packager generates a script for starting your application, it will generate an invocation of java that passes every library on the classpath to the classpath argument, `-cp`. If you have a lot of dependencies, this may result in a very long command being executed, which, aside from being aesthetically unpleasing and difficult to work with when using tools like `ps`, causes problems on some platforms, notably Windows, that have limits to how long commands can be.

There are a few ways you can work around this in the native packager.

#### 5.2.1 Generate a launcher jar

The native packager includes a plugin that allows generating a launcher jar. This launcher jar will contain no classes, but will have your projects main class and classpath in its manifest. The script that sbt then generates executes this jar like so:

```
java -jar myproject-launcher.jar
```

To enable the launcher jar, enable the `LauncherJarPlugin`:

```
enablePlugins(LauncherJarPlugin)

5.2.2 Generate a classpath jar

The classpath jar is very similar to the launcher jar, in that it also has the classpath on its manifest, but it does not include the main class in its manifest, and so executed by the start script by invoking:

```
java -cp myproject-classpath.jar some.Main
```

To enable the classpath jar:

enablePlugins(ClasspathJarPlugin)

5.2.3 Configure a wildcard classpath

JDK 6 and above supports configuring the classpath using wildcards. To enable this, simply override the scriptClasspath task to only contain `*`, for example:

```
scriptClasspath := Seq("*")
```

One downside of this approach is that the classpath ordering will no longer match the classpath ordering that sbt uses.

5.3 Play 2 Packaging

Although Play 2 supports Sbt Native Packager, it requires some additional steps to successfully package and run your application.

Tip: there are also two sections in the play documentation that describe deploying and configuring:

- play deploying
- play prod configuration

5.3.1 Build Configuration

Depending on whether you want to package your application as a deb-package or as an rpm-package, you have to setup your build configuration accordingly. Please, refer to Debian Plugin and Rpm Plugin pages for additional information.

Note that Upstart is not supported by all available operation systems and may not always work as expected. You can always fallback to the SystemV service manager instead. For more information on service managers please refer to Java Server Application Archetype page.

5.3.2 Application Configuration

In order to run your application in production you need to provide it with at least:

- Location where it can store its pidfile
- Production configuration
One way to provide this information is to append the following content in your build definition:

```scala
Universal / javaOptions += Seq(
  // JVM memory tuning
  "-J-Xmx1024m",
  "-J-Xms512m",
  // Since play uses separate pidfile we have to provide it with a proper path
  s"-Dpidfile.path=/var/run/${packageName.value}/play.pid",
  // alternative, you can remove the PID file
  s"-Dpidfile.path=/dev/null",
  // Use separate configuration file for production environment
  s"-Dconfig.file=/usr/share/${packageName.value}/conf/production.conf",
  // Use separate logger configuration file for production environment
  s"-Dlogger.file=/usr/share/${packageName.value}/conf/production-logger.xml",
  // You may also want to include this setting if you use play evolutions
  "-DapplyEvolutions.default=true"
)
```

This way you should either store your production configuration under `${path_to_app_name}/conf/production.conf` or put it under `/usr/share/${app_name}/conf/production.conf` by hand or using some configuration management system.

**Warning:** Your pid file must be called `play.pid`

### SystemV

If you use a system using SystemV start script make sure to provide a `etc-default` in `src/templates` and set the `PIDFILE` environment variable.

```bash
# Setting JAVA_OPTS
# -----------------
# you can use this instead of the application.ini as well
# JAVA_OPTS="-Dpidfile.path=/var/run/${app_name}/play.pid $JAVA_OPTS"

# For rpm/systemv you need to set the PIDFILE env variable as well
PIDFILE="/var/run/${app_name}/play.pid"
```

See customize section for [Java Server Application Archetype](#) for more information on `application.ini` and `etc-default` template.

## 5.4 Deployment

This page shows you how to configure your build to deploy your build universal(zip, tgz, txz), rpm, debian or msi packages. For information on docker, please take a look at the docker page.
Note: The deployment settings only add artifacts to your publish task. Native packager doesn’t provide any functionality for publishing to native repositories.

5.4.1 Setup publish Task

You need a working publish task in order to use the following configurations. A good starting point is the sbt publish documentation. You should have something like this in your build.sbt:

```scala
publishTo := {
  val nexus = "https://oss.sonatype.org/
  if (version.value.trim.endsWith("SNAPSHOT"))
    Some("snapshots" at nexus + "content/repositories/snapshots")
  else
    Some("releases" at nexus + "service/local/staging/deploy/maven2")
}
```

For an automated build process are other plugins like the sbt release plugin.

5.4.2 Default Deployment

The easiest way is to add UniversalDeployPlugin to your build.sbt:

```scala
enablePlugins(JavaServerAppPackaging, UniversalDeployPlugin)
```

You are now able to publish your packaged application in both tgz and zip formats with:

- `Universal/publish` Publish the zip (or tgz/txz depending on the configuration. Default is to publish zip along with tgz) package

5.4.3 Custom Deployments

When using other package formats we need to explicitly configure the deployment setup to a more specific one.

**RPM**

Your build.sbt should contain:

```scala
enablePlugins(RpmPlugin, RpmDeployPlugin)
```

This will make possible to push the RPM with:

```bash
sbt Rpm/publish
```

**Debian**

Enabled with:

```scala
enable(DebianPlugin, DebianDeployPlugin)
```

that will make possible to publish a deb package with:
sbt Deb/publish

Windows

If using an msi packaging you need to enable:

```scala
enable(WindowsPlugin, WindowsDeployPlugin)
```

Then, pushing the package is

```scala
sbt Windows/publish
```

5.4.4 Custom Configurations

You could configure only what you need as well.

**Debian**

```scala
makeDeploymentSettings(Debian, Debian / packageBin, "deb")
```

// if you want a changes file as well
```scala
makeDeploymentSettings(Debian, Debian / genChanges, "changes")
```

**RPM**

```scala
makeDeploymentSettings(Rpm, Rpm / packageBin, "rpm")
```

**Windows**

```scala
makeDeploymentSettings(Windows, Windows / packageBin, "msi")
```

**Universal**

```scala
// zip
makeDeploymentSettings(Universal, Universal / packageBin, "zip")
```

```scala
makeDeploymentSettings(UniversalDocs, UniversalDocs / packageBin, "zip")
```

```scala
// additional tgz
addPackage(Universal, Universal / packageZipTarball, "tgz")
```

```scala
// additional txz
addPackage(UniversalDocs, UniversalDocs / packageXzTarball, "txz")
```
5.5 Scala JS packaging

**Warning:** This is no official scala js doc, but created from the native-packager community. See issue-699.

5.5.1 Package webjars and scalajs resources

In order to package all assets correctly, add this to your project

```
Runtime / managedClasspath += (Assets / previewJVM / packageBin).value
```

5.6 Build the same package with different configs

If you want to build your application with different settings, e.g. for test, staging and production, then you have three ways to do this.

**Tip:** All examples are shown in a simple build.sbt. We recommend using AutoPlugins to encapsulate certain aspects of your build.

All examples can also be found in the native-packager examples.

5.6.1 SBT sub modules

The main idea is to create a submodule per configuration. We start with a simple project build.sbt.

```
name := "my-app"
enablePlugins(JavaAppPackaging)
```

In the end we want to create three different packages (test, stage, prod) with the respective configurations. We do this by creating an application module and three packaging submodules.

```
// the application
lazy val app = project
  .in(file("."))
  .settings(
    name := "my-app",
    libraryDependencies += "com.typesafe" % "config" % "1.3.0"
  )
```

Now that our application is defined in a module, we can add the three packaging submodules. We will override the resourceDirectory setting with our app resource directory to gain easy access to the applications resources.

```
lazy val testPackage = project
  // we put the results in a build folder
  .in(file("build/test"))
  .enablePlugins(JavaAppPackaging)
  .settings(
    // override the resource directory
    Compile / resourceDirectory := (app / compile / resourceDirectory).value,
    Universal / mappings += {
    (continues on next page)
Now that you have your `build.sbt` set up, you can try building packages.

```scala
((Compile / resourceDirectory).value / "test.conf") -> "conf/application.conf"
})
.dependingOn(app)

// basically identical despite the configuration differences

```%

```scala
lazy val stagePackage = project
  .in(file("build/stage"))
  .enablePlugins(JavaAppPackaging)
  .settings(
    Compile / resourceDirectory := (app / Compile / resourceDirectory).value,
    Universal / mappings += {
      ((Compile / resourceDirectory).value / "stage.conf") -> "conf/application.conf"
    }
  )
  .dependingOn(app)

lazy val prodPackage = project
  .in(file("build/prod"))
  .enablePlugins(JavaAppPackaging)
  .settings(
    Compile / resourceDirectory := (app / Compile / resourceDirectory).value,
    Universal / mappings += {
      ((Compile / resourceDirectory).value / "prod.conf") -> "conf/application.conf"
    }
  )
  .dependingOn(app)
```

Now that you have your `build.sbt` set up, you can try building packages.

```bash
# stages a test build in build/test/target/universal/stage
testPackage/stage

# creates a zip with the test configuration
sbt testPackage/Universal/packageBin
```

This technique is a bit verbose, but communicates very clear what is being built and why.

### 5.6.2 SBT parameters and Build Environment

SBT is a java process, which means you can start it with system properties and use these in your build. This pattern may be useful in other scopes as well. First we define an `AutoPlugin` that sets a build environment.

This plugin allows you to start sbt for example like

```
sbt -Denv=prod
```

Now we can use this `buildEnv` setting to change things. For example the `mappings`. We recommend doing this in a plugin as it involves quite some logic. In this case we decide which configuration file to map as `application.conf`.

5.6. Build the same package with different configs 79
Of course you can change all other settings, package names, etc. as well. Building different output packages would look like this:

```sh
sbt -Denv=test Universal/packageBin
sbt -Denv=stage Universal/packageBin
sbt -Denv=prod Universal/packageBin
```

### 5.6.3 SBT configuration scope (not recommended)

The other option is to generate additional scopes in order to build a package like `Prod / packageBin`. Scopes behave counter-intuitive sometimes, why we don’t recommend this technique.

**Error:** This example is work in progress and doesn’t work. Unless you are not very familiar with sbt we highly recommend using another technique.

A simple start may look like this:

```scala
lazy val Prod = config("prod") extend(Universal) describedAs("scope to build → production packages")
lazy val Stage = config("stage") extend(Universal) describedAs("scope to build → staging packages")

lazy val app = project
  .in(file("."))
  .enablePlugins(JavaAppPackaging)
  .configs(Prod, Stage)
  .settings{
    name := "my-app",
    libraryDependencies += "com.typesafe" % "config" % "1.3.0"
  }
```

You would expect `Prod / packageBin` to work, but extending scopes doesn’t imply inheriting tasks and settings. This needs to be done manually. Append this to the `app` project.

```scala
// inheriting tasks and settings
.settings(inConfig(Prod) (UniversalPlugin.projectSettings))
.settings(inConfig(Prod) (JavaAppPackaging.projectSettings))
// define custom settings
.settings(inConfig(Prod) (Seq{
  // you have to override everything carefully
  packageName := "my-prod-app",
  executableScriptName := "my-prod-app",
  // this is what we actually want to change
})
```

(continues on next page)
Note that you have to know more on native-packager internals than you should, because you override all the necessary settings with the intended values. Still this doesn’t work as the universal plugin picks up the wrong mappings to build the package.

## 5.7 Embedding JVM in Universal

Sbt Native Packager supports embedding the jvm using the *JDKPackager Plugin*, however, in some cases you may want instead to embed the JVM/JRE in other formats, e.g. a tarball with one of the java archetypes.

To accomplish this you need to:

- Add the JVM/JRE of your choice to the `mappings`
- Make the launcher use the embedded `jre`

### 5.7.1 Adding the JVM

The JRE is by definition OS dependent, hence you must choose the one appropriate for your case. The example below assumes a Linux 64 JRE, whose files are at `$HOME/.jre/linux64`. The files will be copied to a `jre` directory in your distribution.

```scala
import NativePackagerHelper._

val jresDir = Path.userHome / ".jre"
val linux64Jre = jresDir.toPath.resolve("linux64")
directory(linux64Jre.toFile).map { j =>
  j._1 -> j._2.replace(jreLink, "jre")
}
```

### 5.7.2 Application Configuration

In order to run your application in production you also need to make the launcher use the jre added above. This can be done using the `-java-home` option with a relative path.

```scala
Universal / javaOptions ++= Seq(  // Your original options
  "-java-home ${app_home}/../jre"
)
```
5.8 Setting the umask for your process

In Linux, the umask, associated with the user running the process, will determine what permissions files generated by the process, such as log files, will have.

5.8.1 Within the start script

You can set the umask within the generated start script:

```
bashScriptExtraDefines += "umask 077"
```

This line will be added just before the application will be executed by the start script and therefore the application process will also inherit this umask. For more information take a look at the according documentation.

**Hint:** When using Docker this is the only reliable way right now to set the umask for the process. Docker will ignore any umask set in e.g. `/etc/login.defs` (or other config files) within a docker image.

5.8.2 System V/systemd

When using the Java Server Application Archetype usually the system default umask for daemons is fine and can be left unchanged, however sometime the need arises to override it.

As an alternative to the start script approach described in the section above, to set a custom umask for your application that is running either via a System V init script or systemd, the umask can be specified via a custom `/etc/default/` appname file.

The contents of this file can be specified by creating a template at `src/templates/etc-default` and adding a line such as the following:

```
UMASK 077
```