sbt-native-packager

Release 1.0a1

Josh Suereth

Nov 29, 2018

Contents

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 lifecyle 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

```
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

```
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 `mappings in Universal` setting, which defines what files will be included in the package. These files are automatically moved around for the appropriate native packaging as needed.

CHAPTER 2

Getting Started

2.1 Setup

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

```
addSbtPlugin("com.typesafe.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

```
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.

```
sbt stage
./target/universal/stage/bin/<your-app>
```

2.2.2 Create a package

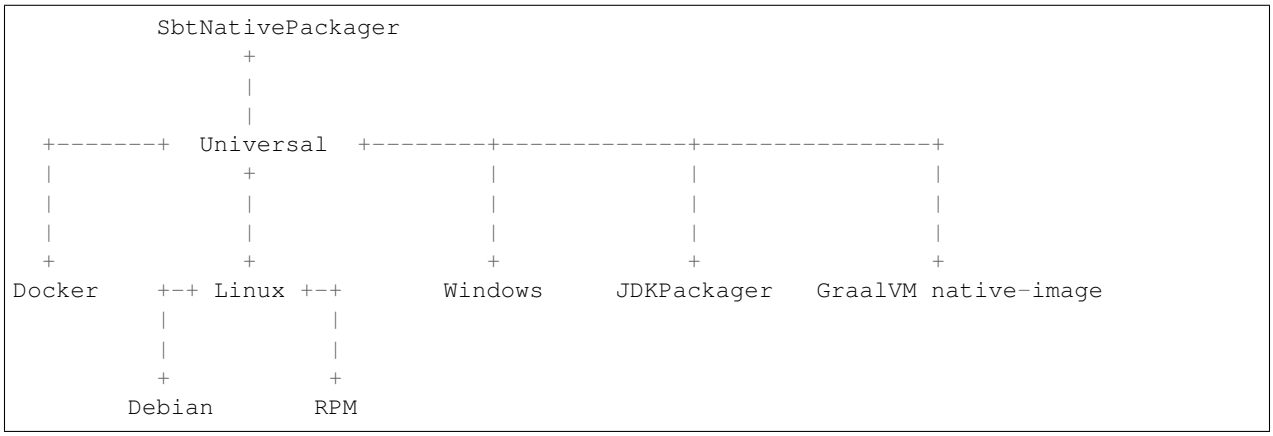
We can generate other packages via the following tasks. Note that each packaging format may needs some additional configuration and native tools available. Here's a complete list of current formats.

- `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

CHAPTER 3

Packaging Formats

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



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

```
mappings in Docker := (mappings in Universal).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

universal-docs For creating bundles of documentation

universal-src For creating bundles of source.

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

```
name in Universal := name.value

name in UniversalDocs <<= name in Universal

name in UniversalSrc <<= name in Universal

packageName in Universal := 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:

```
mappings in Universal <+= packageBin in Compile map { p => p -> "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:

```
mappings in Universal in package-zip-tarball += 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 `tgz` 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 OSX or systems with `hdiutil`.

universal-docs:package-bin Creates the `zip` universal documentation package.

universal-docs:package-zip-tarball Creates the `tgz` universal documentation package.

universal-docs:package-xz-tarball 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:

```
universalArchiveOptions in (Universal, packageZipTarball) := 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:

```
universalArchiveOptions in (Universal, packageZipTarball) :=
  (Seq("--exclude", "*~") ++ (universalArchiveOptions in (Universal,␣
↪packageZipTarball)).value)
```

Currently, these task can be customized:

universal:package-zip-tarball *universalArchiveOptions in (Universal, packageZipTarball)*

universal:package-xz-tarball *universalArchiveOptions in (Universal, packageXzTarball)*

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 `mappings in Universal` setting. Let's look at an example where we add the packaged jar of a project to the lib folder of a distribution:

```
mappings in Universal <+= (packageBin in Compile) map { jar =>
  jar -> ("lib/" + jar.getName)
}
```

The above does two things:

1. It depends on `packageBin in Compile` 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 to 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:

```
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:

```
linuxPackageMappings in Debian <+= (name in Universal, sourceDirectory in Debian) map
↪{ (name, dir) =>
  (packageMapping(
    (dir / "changelog") -> "/usr/share/doc/sbt/changelog.gz"
  ) withUser "root" withGroup "root" withPerms "0644" gzipped) asDocs()
}
```

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

```
topLevelDirectory := None
```

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

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

Or just a plain hardcoded string

```
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`

```
mappings in (Compile, packageDoc) := 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

```
import NativePackagerHelper._
```

plugin version 0.8.x or lower

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

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

```
mappings in Universal ++= directory("src/main/resources/cache")

mappings in Universal ++= contentOf("src/main/resources/docs")

mappings in Universal <++= sourceDirectory map (src => directory(src / "main" /
↪"resources" / "cache"))

mappings in Universal <++= sourceDirectory map (src => contentOf(src / "main" /
↪"resources" / "docs"))
```

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 `packageBin in Compile` 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:

```
mappings in Universal ++= 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:

```
mappings in Universal := (mappings in Universal).value ++ directory(target.value /
↪"scala-2.10" / "api")
}
```

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

```
mappings in Universal <++= (packageBin in Compile, target ) map { (_, target) =>
    val dir = target / "scala-2.10" / "api"
    (dir.***) 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:

if target = `/Users/you/dev/fantasticApp/src/scala/fantasticApp-0.1-HOTFIX01`

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

```
somedata.csv
README
```

Then the code above will produce this mapping:

```
((/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)

```
mappings in Universal <++= (packageBin in Compile, target ) map { (_, target) =>
    val dir = target / "scala-2.10" / "api"
    (dir.*** --- 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.

tl;dr how to remove stuff

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

(continues on next page)

(continued from previous page)

```
}

// sbt 0.12 syntax
mappings in Universal <<= (mappings in Universal, assembly in Compile) map {␣
↪(universalMappings, fatJar) => /* same logic */}
```

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
jarName in assembly := "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
mappings in Universal := {
    val universalMappings = (mappings in Universal).value
    val fatJar = (assembly in Compile).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( (jarName in assembly).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`.

```
name in Linux := name.value
```

3.2.4 Settings

The required fields for any linux distribution are:

`name in Linux` 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

```
(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 attached 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 instances 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 LinuxPackageMapping

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

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 specifc 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␣
↪case (file, name) => true  pattern is satisfied
        }

        // 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
        )
```

(continues on next page)

(continued from previous page)

```
        // returns a fresh LinuxPackageMapping based on the above
        linuxPackage.copy(
            mappings = filtered,
            fileData = fileData
        )
    } filter {
        linuxPackage => linuxPackage.mappings.nonEmpty // return all mappings that␣
↪are nonEmpty (this effectively removes all empty linuxPackageMappings)
    }
}

// sbt 0.12.x syntax
linuxPackageMappings <<= linuxPackageMappings map { mappings =>
    /* stuff. see above */
    mappings
}
```

The ordering in which you apply the tasks is important.

Add Mappings

To add an arbitrary file in your build path

```
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.

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

linuxPackageMappings in Rpm += {
  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.

```
// this is equal to
// linuxPackageMappings <<= linuxPackageMappings map { mappings => /* stuff */␣
↪mappings }
linuxPackageMappings := {
    // first get the current mappings. mapping is of type Seq[LinuxPackageMapping]
    val mappings = linuxPackageMappings.value
```

(continues on next page)

(continued from previous page)

```
    // 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
        }

        // now we copy the mapping but replace the mappings
        mapping.copy(mappings = filtered)

    } filter {
        // only keep those mappings that are nonEmpty (_.mappings.nonEmpty == true)
        _.mappings.nonEmpty
    }
}
```

Alter LinuxPackageMapping

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

```
// Altering permissions for configs
linuxPackageMappings := {
    val mappings = linuxPackageMappings.value
    // Changing the group for all configs
    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 => linuxPackage
    }
}
```

Alter LinuxSymlinks

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

```
// The same as linuxPackageMappings
linuxPackageSymlinks := {
    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.

```
// Add an empty folder to mappings
linuxPackageMappings += packageTemplateMapping(s"/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

```
.\" 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 SYNOPSIS
.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

```
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
- dpkg-genchanges
- lintian
- fakeroot

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*.

```
debianNativeBuildOptions in Debian := Nil // dpkg-deb's default compression␣
↪(currently xz)

debianNativeBuildOptions in Debian := 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 overridding *debianNativeBuildOptions* as described above.

Java based packaging

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

```
enablePlugins(JDebPackaging)
```

and this to your `plugins.sbt`:

```
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`.

```
linuxPackageMappings in Debian := linuxPackageMappings.value
```

3.3.4 Settings

Debian requires the following specific settings:

`name in Debian` The name of the package for debian (if different from general linux name).

`version in Debian` The debian-friendly version of the package. Should be of the form `x.y.z-build-aa`.

`debianPackageConflicts in Debian` The list of debian packages that this package conflicts with.

`debianPackageDependencies in Debian` The list of debian packages that this package depends on.

`debianPackageProvides in Debian` The list of debian packages that are provided by this package.

`debianPackageRecommends in Debian` The list of debian packages that are recommended to be installed with this package.

`linuxPackageMappings in Debian` 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.

`maintainerScripts in Debian`(`debianMaintainerScripts`) *DEPRECATED* use `maintainerScripts in Debian` 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.

changelog in Debian 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.

```
debianPackageDependencies in Debian ++= Seq("java2-runtime", "bash (>= 2.05a-11)")

debianPackageRecommends in Debian += "git"
```

Hook Actions into the Debian Package Lifecycle

To hook into the debian package lifecycle (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`:

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

```
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.

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

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`.

```
linuxPackageMappings in Rpm := linuxPackageMappings.value
```

3.4.4 Settings

Informational Settings

packageName in Rpm 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 `packageName in Linux`.

version in Rpm 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 sofware 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"`.

packageArchitecture in Rpm 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"`.

packageSummary in Rpm 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 `packageSummary in Linux`.

packageDescription in Rpm 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 `packageDescription in Linux`.

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.

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

rpmConflcits 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

maintainerScripts in Rpm 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:package-bin Generates the `.rpm` package for this project.

rpm:rpm-lint 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/s1-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:

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

To **override the ‘‘maintainerScripts in Rpm‘‘** 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:

```
// overriding
import RpmConstants._
maintainerScripts in Rpm := 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._
maintainerScripts in Rpm := maintainerScriptsAppend((maintainerScripts in Rpm).value)(
   Pretrans -> "echo 'hello, world'",
   Post -> s"echo 'installing ${(packageName in Rpm).value}'"
)

// appending from a different file
import RpmConstants._
maintainerScripts in Rpm := maintainerScriptsAppendFromFile((maintainerScripts in␣
↪Rpm).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`

```
...
echo "PACKAGE_PREFIX=${RPM_INSTALL_PREFIX}" > /etc/sysconfig/${app_name}
...
```

src/rpm/scriptlets/preun

```
...
rm /etc/sysconfig/${app_name}
...
```

Using scriptlet files like this *will override all previous contents.*

Scriptlet Migration from 1.0.x

Before

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

After

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

```
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:

```
linuxPackageMappings in Rpm := configWithNoReplace((linuxPackageMappings in Rpm).
↪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/.

3.5.2 Build

```
sbt docker:publishLocal
```

Required Settings

```
enablePlugins(DockerPlugin)
```

Spotify java based docker client

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

```
mappings in Docker := mappings.value
```

3.5.4 Settings

Informational Settings

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

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

maintainer in Docker 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` (`daemon`, by default).

daemonUser in Docker 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`.

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

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 compatibilty.).

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")), dockerAlias.value.withRegistryHost(Option("registry.internal.yourdomain.com")))`

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

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

```
packageName in Docker := packageName.value

version in Docker := 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 `mappings in Docker` are extracted underneath this directory.

```
defaultLinuxInstallLocation in Docker := "/opt/docker"
```

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
[info] 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:

```
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", (daemonUser in Docker).value)

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

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/ 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*.

3.6.1 Requirements

You need the following applications installed

- WIX Toolset

3.6.2 Build

```
sbt windows:packageBin
```

Required Settings

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

```
// 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-dba9dbacf424"
```

1.0 or higher

Enables the windows plugin

```
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`.

```
mappings in Windows := (mappings in Universal).value
```

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

3.6.4 Settings

`name in Windows` 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: `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.

`mappings in packageMsi in Windows` 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.

wix-file Generates the Wix xml file from *wixConfig* and *wixProductConfig* setings, unless overriden.

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:

```
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 `mappings in Windows` configuration. When generating a MSI, the plugin will first create a directory using all the `mappings in Windows` 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:

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

(continues on next page)

(continued from previous page)

```
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"))
)
```

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 X, 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 X 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 OS X, the image is the .app file. On Linux, the image is the directory that gets installed.
- `dmg`: Generates a DMG file for OS X.
- `pkg`: Generates a .pkg package for OS X.
- `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:

```
name := "JDKPackagerPlugin Example"

version := "0.1.0"

organization := "com.foo.bar"

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

mainClass in Compile := 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)

(continued from previous 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)

jdkPackagerType := "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.
(antPackagerTasks in JDKPackager) := (antPackagerTasks in JDKPackager).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.

> https://www.graalvm.org/docs/reference-manual/aot-compilation/ documents the AOT compilation of GraalVM.

3.8.1 Requirements

You must have `native-image` of GraalVM in your `PATH`.

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.

```
source <(curl -o - https://raw.githubusercontent.com/sbt/sbt-native-packager/master/.
↪travis/download-graalvm)
```

3.8.2 Build

```
sbt 'show graalvm-native-image:packageBin'
```

Required Settings

```
enablePlugins(GraalVMNativeImagePlugin)
```

3.8.3 Settings

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 Tasks

The GraalVM Native Image plugin provides the following commands:

graalvm-native-image:packageBin Generates a native image using GraalVM.

CHAPTER 4

Project Archetypes

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:

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

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.

`makeBashScript` Creates or discovers the bash script used by this project.

`makeBatScript` 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.

`batScriptExtraDefines` A list of extra definitions that should be written to the bat file template.

4.1.4 Start script options

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

-h | -help Prints script usage

-v | -verbose Prints out more information

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

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

-java-home <java home> Override the default JVM home, it accept variable expansions, e.g. `-java-home ${app_home}/../jre`

-main Define a custom main class

To configure the JVM these options are available

JAVA_OPTS environment variable, if unset uses "$java_opts"

-Dkey=val pass -Dkey=val directly to the java runtime

-J-X pass option -X directly to the java runtime (-J is stripped). E.g. `-J-Xmx1024`

In order to pass **application arguments** you need to separate the jvm arguments from the application arguments with `--`. For example

```
./bin/my-app -Dconfig.resource=prod.conf -- -appParam1 -appParam2
```

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 `mainClass in Compile` is set
2. One start script for the defined `mainClass in Compile` and forwarding scripts for all other main classes.

Note: What does *'forwarder script'* mean?

Native-packager's start script provides a *-main* option to override the main class that should be executed. A *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.

Multiple start scripts

No configuration is needed. SBT sets `mainClass in Compile` automatically to `None` if multiple main classes are discovered.

Example:

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
```

Single start script with forwarders

Generates a single start script for the defined main class in `mainClass in Compile` and forwarding scripts for all other `discoveredMainClasses in Compile`. 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"
mainClass in Compile := 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
  my-project
  my-project.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‘‘s 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

```
// 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.

Via build.sbt

You can specify your options via the `build.sbt`.

```
javaOptions in Universal ++= 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 `javaOptions in Universal` 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

```
# 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")
batScriptConfigLocation := 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.

```
// 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 (in order).

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.

```
// 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_MAIN_CLASS`.

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 OSX

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

```
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

daemonGroupGid 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`.

Folder	User	Permissions	Purpose
/usr/share/**<package>**	root	755 / (655)	static, non-changeable files
/etc/default/**<package>**	root	644	default config file
/etc/**<package>**	root	644	config folder -> link to /usr/share/**<package-name>**/conf
/var/run/**<package>**	daemon	644	if the application generates a pid on its own
/var/log/**<package>**	daemon	644	log folder -> symlinked from /usr/share/**<package>**/log

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:

```
# 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.

```
// a different daemon user
daemonUser in Linux := "my-user"
// if there is an existing one you can specify the uid
daemonUserUid in Linux := Some("123")
// a different daemon group
daemonGroup in Linux := "my-group"
// if the group already exists you can specify the uid
daemonGroupGid in Linux := Some("1001")
```

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` ``:

```
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 `maintainerScripts in Rpm` 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

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

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

```
daemonUser in Linux := "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:

```
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:

```
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.

4.3.4 Upstart

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

```
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:

```
defaultLinuxStartScriptLocation in Debian := "/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 SystemLoaders you can to the `linuxScriptReplacements` map:

```
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:

```
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.

```
enablePlugins(AshScriptPlugin)
```

4.4.2 ClasspathJar & LauncherJar Plugin

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

4.5 Archetype Cheatsheet

This is a set FAQ composed on a single page.

4.5.1 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!

output path	scopes	archetype	comment
lib	all	JavaApp	
conf	all	JavaApp	
bin/`<executableScriptName>`	Global	JavaApp	
bin/`<executableScriptName>`.bat	Global	JavaApp	
bin/`<executableScriptName>`	Global		Entrypoint DockerPlugin
`<defaultLinuxInstallationLocation>`/`<packageName>`	Linux, Debian, Rpm	JavaApp	
`<defaultLinuxLogLocation>`/`<packageName>`	Linux	JavaServerApplication	
logs	Linux	JavaServerApplication	Symlink
/etc/default/`<packageName>`	Linux	JavaServerApplication	
/var/run/`<packageName>`	Linux	JavaServerApplication	
/etc/init.d/`<packageName>`	Linux, Debian, Rpm	JavaServerApplication	For SystemV
/etc/init/`<packageName>`	Linux, Debian, Rpm	JavaServerApplication	For Upstart
/usr/lib/systemd/system/`<packageName>`.service	Linux, Debian, Rpm	JavaServerApplication	For Systemd
`<defaultLinuxInstallLocation>`	Docker		Installation path inside the container

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).

4.5.2 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 `javaOptions in Universal` (JavaApp) or `javaOptions in Linux` (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.

```
bashScriptExtraDefines += """addJava "-Dconfig.file=${app_home}/../conf/app.config""""

// or more. -X options don't need to be prefixed with -J
bashScriptExtraDefines ++= Seq(
   """addJava "-Xms1024m"""",
   """addJava "-Xmx2048m""""
)
```

For information take a look at the :doc:‘ customize section for java apps </archetypes/java_app/customize>‘

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 `javaOptions in Universal` (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).

javaOpts	Scope	bashScriptConfigLocation	Archetype	mappings	comment
Nil	Universal	None	JavaApp		No jvm options
Nil	Universal	Some(appIniLocation)	JavaApp		User provides the application.ini file in `src/universal/conf/application.ini`
opts	Universal	Some(_)	JavaApp	added	creates `application.ini` but leaves `bashScriptConfigLocation` unchanged
opts	Universal	None	JavaApp	added	creates `application.ini` and sets `bashScriptConfigLocation`. If `src/universal/conf/application.ini` is present it will be overridden
Nil	Linux	None	JavaServer	added	creates `etc-default` and sets `bashScriptConfigLocation`
opts	Linux	None	JavaServer	added	creates `etc-default`, appends `javaOptions in Linux` and sets `bashScriptConfigLocation`
opts	Linux	Some(_)	JavaServer	added	creates `etc-default`, appends `javaOptions in Linux` and overrides `bashScriptConfigLocation`

4.5.3 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 `maintainer in Linux`.
- `descr` - The short description of the service; defined by `packageSummary in Linux`.
- `exec` - The script/binary to execute when starting the service; defined by `executableScriptName in Linux`.
- `chdir` - The working directory for the service; defined by `defaultLinuxInstallLocation/(packageName in Linux)`.
- `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 `packageName in Linux`.
- `version` - The software version; defined by `version`.
- `daemon_user` - The user that the service should run as; defined by `daemonUser in Linux`.
- `daemon_user_uid` - The user ID of the user that the service should run as; defined by `daemonUserUid in Linux`.
- `daemon_group` - The group of the user that the service should run as; defined by `daemonGroup in Linux`.
- `daemon_group_gid` - The group ID of the group of the user that the service should run as; defined by `daemonGroupGid in Linux`.
- `daemon_shell` - The shell of the user that the service should run as; defined by `daemonShell in Linux`.

- `term_timeout` - The timeout for the service to respond to a TERM signal; defined by `termTimeout in Linux`, defaults to `60`.
- `kill_timeout` - The timeout for the service to respond to a KILL signal; defined by `killTimeout in Linux`, defaults to `30`.
- `start_facilities` - Intended for the `Required-Start:` line in the `INIT INFO` block. Its value is automatically generated with respect to the chosen system loader.
- `stop_facilities` - Intended for the `Required-Stop:` line in the `INIT INFO` block. Its value is automatically generated with respect to the chosen system loader.
- `start_runlevels` - Intended for the `Default-Start:` line in the `INIT INFO` block. Its value is automatically generated with respect to the chosen system loader.
- `stop_runlevels` - Intended for the `Default-Stop:` line in the `INIT INFO` block. Its value is automatically generated with respect to the chosen system loader.

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.

CHAPTER 5

Recipes

This section provides recipes for common configurations. If you can't find what you are looking for, take a look at sbt-native-packager examples github page.

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.

```
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:

```
import AssemblyKeys._

// the assembly settings
assemblySettings
```

(continues on next page)

(continued from previous page)

```
// we specify the name for our fat jar
jarName in assembly := "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
mappings in Universal := {
    // universalMappings: Seq[(File,String)]
    val universalMappings = (mappings in Universal).value
    val fatJar = (assembly in Compile).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( (jarName in assembly).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.typesafe.sbt" % "sbt-proguard" % "0.2.2")
```

Then configure the proguard options in `build.sbt`:

```
import com.typesafe.sbt.SbtProguard.ProguardKeys

// initialize the proguard settings
proguardSettings

// to configure proguard for scala, see
// http://proguard.sourceforge.net/manual/examples.html#scala
ProguardKeys.options in Proguard ++= Seq(
    "--dontwarn scala.**",
    "--dontwarn ch.qos.**"
    // ...
)

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

(continues on next page)

(continued from previous page)

```
// Java 8 requires a newer version of proguard than sbt-proguard's default
ProguardKeys.proguardVersion in Proguard := "5.2.1"

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

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

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

Now when you package your project using a command such as `sbt universal:package-zip-tarball`, 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`

```
import NativePackagerKeys._

name := "mukis-fullstack"

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

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

scalaVersion in ThisBuild := "2.11.2"

// common dependencies
libraryDependencies in ThisBuild ++= 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(
```

(continues on next page)

(continued from previous page)

```
        maintainer := "John Smith <john.smith@example.com>",
        packageDescription := "Fullstack Application",
        packageSummary := "Fullstack Application",
        // entrypoint
        mainClass in Compile := 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")
```

(continues on next page)

(continued from previous page)

```
val root = project.in(file("."))
    // adding your custom configuration scope
    .configs( TxtFormat )
    .settings(packageArchetype.java_server:_*)
    .settings(
        name := "mukis-custom-package",
        version := "1.0",
        mainClass in Compile := Some("de.mukis.ConfigApp"),
        maintainer in Linux := "Nepomuk Seiler <nepomuk.seiler@mukis.de>",
        packageSummary in Linux := "Custom application configuration",
        packageDescription := "Custom application configuration",
        // defining your custom configuration
        packageBin in TxtFormat := {
            val fileMappings = (mappings in Universal).value
            val output = target.value / s"${packageName.value}.txt"
            // create the is with the mappings. Note this is not the ISO format -.-
            IO.write(output, "# Filemappings\n")
            // append all mappings to the list
            fileMappings foreach {
                case (file, name) => IO.append(output, s"${file.getAbsolutePath}\t
↪$name${IO.Newline}")
            }
            output
        }
    )
```

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 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:

```
javaOptions in Universal ++= Seq(
  // JVM memory tuning
  "-J-Xmx1024m",
  "-J-Xms512m",

  // Since play uses separate pidfile we have to provide it with a proper path
  // name of the pid file must be play.pid
  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.

```
# 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`

```
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 automatised 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`

```
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:

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

This will make possible to push the `RPM` with:

```
`sbt rpm:publish
```

Debian

Enabled with:

```
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:

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

Then, pushing the package is

```
sbt windows:publish
```

5.4.4 Custom Configurations

You could configure only what you need as well.

Debian

```
makeDeploymentSettings(Debian, packageBin in Debian, "deb")

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

RPM

```
makeDeploymentSettings(Rpm, packageBin in Rpm, "rpm")
```

Windows

```
makeDeploymentSettings(Windows, packageBin in Windows, "msi")
```

Universal

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

makeDeploymentSettings(UniversalDocs, packageBin in UniversalDocs, "zip")

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

(continues on next page)

(continued from previous page)

```
// additional txz
addPackage(UniversalDocs, packageXzTarball in UniversalDocs, "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

```
(managedClasspath in Runtime) += (packageBin in previewJVM in Assets).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
    resourceDirectory in Compile := (resourceDirectory in (app, Compile)).value,
    mappings in Universal += {
      ((resourceDirectory in Compile).value / "test.conf") -> "conf/application.conf"
    }
  )
  .dependsOn(app)

// bascially identical despite the configuration differences
lazy val stagePackage = project
  .in(file("build/stage"))
  .enablePlugins(JavaAppPackaging)
  .settings(
    resourceDirectory in Compile := (resourceDirectory in (app, Compile)).value,
    mappings in Universal += {
      ((resourceDirectory in Compile).value / "stage.conf") -> "conf/application.conf"
    }
  )
  .dependsOn(app)

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

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

```
# 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.

```
import sbt._
import sbt.Keys._
import sbt.plugins.JvmPlugin

/** sets the build environment */
```

(continues on next page)

(continued from previous page)

```
object BuildEnvPlugin extends AutoPlugin {

  // make sure it triggers automatically
  override def trigger = AllRequirements
  override def requires = JvmPlugin

  object autoImport {
    object BuildEnv extends Enumeration {
      val Production, Stage, Test, Developement = Value
    }

    val buildEnv = settingKey[BuildEnv.Value]("the current build environment")
  }
  import autoImport._

  override def projectSettings: Seq[Setting[_]] = Seq(
    buildEnv := {
      sys.props.get("env")
         .orElse(sys.env.get("BUILD_ENV"))
         .flatMap {
           case "prod" => Some(BuildEnv.Production)
           case "stage" => Some(BuildEnv.Stage)
           case "test" => Some(BuildEnv.Test)
           case "dev" => Some(BuildEnv.Developement)
           case unkown => None
         }
         .getOrElse(BuildEnv.Developement)
    },
    // give feed back
    onLoadMessage := {
      // depend on the old message as well
      val defaultMessage = onLoadMessage.value
      val env = buildEnv.value
      s"""|$defaultMessage
          |Running in build environment: $env""".stripMargin
    }
  )

}
```

This plugin allows you to start sbt for example like

```
sbt -Denv=prod
[info] Set current project to my-app (in build file: ...)
[info] Running in build environment: Production
> show buildEnv
[info] Production
```

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

```
mappings in Universal += {
  val confFile = buildEnv.value match {
    case BuildEnv.Developement => "dev.conf"
    case BuildEnv.Test => "test.conf"
    case BuildEnv.Stage => "stage.conf"
```

(continues on next page)

(continued from previous page)

```
    case BuildEnv.Production => "prod.conf"
  }
  ((resourceDirectory in Compile).value / confFile) -> "conf/application.conf"
}
```

Ofcourse you can change all other settings, package names, etc. as well. Building different output packages would look like this

```
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

```
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.

```
// 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 acutally want to change
  mappings += ((resourceDirectory in Compile).value / "prod.conf") -> "conf/
↪application.conf"
)))
```

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

```
import NativePackagerHelper._

...

mappings in Universal ++= {
  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.

```
javaOptions in Universal ++= Seq(
  // Your original options

 "-java-home ${app_home}/../jre"
)
```