

# Pragmatic SBT

© 2019 Hermann Hueck

<https://github.com/hermannhueck/pragmatic-sbt>

# Abstract

This presentation gives a pragmatic introduction to sbt in several examples. Each example is a build in its own root directory: `./example??`. Beginning with very simple sbt examples the later examples are becoming more structured and more complex showing multi-project builds, cross version builds, packaging and publishing, custom *Settings* and *Tasks* and the integration of the Ammonite REPL into your build. We also look at *InputTasks*, *Commands* and plugin development.

(This presentation has been developed with sbt 1.2.8 in July 2019. sbt 1.3.0 was RC3 at that time and Scala 2.13.0 was released a few weeks ago.)

Recommended bootstrap tutorial for sbt:

- <https://www.scala-sbt.org/1.x/docs/sbt-by-example.html>

To understand the sbt core concepts see this talk:

- <https://www.youtube.com/watch?v=-shamsTC7rQ>

# Agenda

1. sbt without build file
2. sbt new
3. Creating an application from the template
4. Scala REPL
5. Aliases
6. Plugins
7. Multi-project Builds
8. Cross Builds
9. Version specific source code
10. Library Dependencies
11. Packaging
12. Publishing
13. Installable Packages with *sbt-native-packager*
14. Predefined Settings and Tasks
15. Custom Settings and Tasks
16. Global Settings
17. Ammonite REPL
18. Input Tasks
19. Commands
20. Plugin Development
21. References

# 1. sbt without build file

*.../example01*

# No build file

sbt uses the same kind of directory structure like Maven and Gradle.

We have a Scala/Java source structure in our root directory but no *build.sbt*.

```
example01 $ tree
.
└── src
    ├── main
    │   └── scala
    │       └── example
    │           └── HelloApp.scala
    └── test
        └── scala
            └── example
                └── HelloSpec.scala
```

# No build file

sbt uses the same kind of directory structure like Maven and Gradle.

We have a Scala/Java source structure in our root directory but no *build.sbt*.

```
example01 $ tree
.
└── src
    ├── main
    │   └── scala
    │       └── example
    │           └── HelloApp.scala
    └── test
        └── scala
            └── example
                └── HelloSpec.scala
```

sbt provides a sensible default for every setting.

# No build file (check versions)

```
example01 $ sbt
[warn] No sbt.version set ...
[info] Loading settings ...
[info] Set current project to example01 ...
[info] sbt server started at ...
```

# No build file (check versions)

```
example01 $ sbt
[warn] No sbt.version set ...
[info] Loading settings ...
[info] Set current project to example01 ...
[info] sbt server started at ...
```

```
sbt:example01> sbtVersion
[info] 1.2.8
```

```
sbt:example01> scalaVersion
[info] 2.12.7
```

```
sbt:example01> name
[info] example01
```

```
sbt:example01> projects
[info] In file:.../example01/
[info]      * example01
```

```
sbt:example01> version
[info] 0.1.0-SNAPSHOT
```

# No build file

Compile, run, test ...

# No build file

Compile, run, test ...

```
sbt:example01> compile
[info] Compiling 1 Scala source to .../example01/target/scala-2.12/classes ...
[info] Done compiling.
[success] Total time: ...
```

```
sbt:example01> run
[info] Packaging .../example01/target/scala-2.12/example01_2.12-0.1.0-SNAPSHOT.jar
[info] Done packaging.
[info] Running example.HelloApp

Hello World!
```

```
sbt:example01> test:compile
[info] Compiling 1 Scala source to .../example01/target/scala-2.12/test-classes ...
[error] .../example01/src/test/scala/example/HelloSpec.scala:3:12: object scalates
[error] import org.scalatest._
[error]           ^
```

Test code doesn't compile. The test library is missing.

# No build file

Add `scalatest` to `libraryDependencies` and test again.

# No build file

Add `scalatest` to `libraryDependencies` and test again.

```
sbt:example01> set libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.8"
[info] Defining libraryDependencies
[info] The new value will be used by allDependencies, dependencyPositions, depende
[info] Reapplying settings...
[info] Set current project to example01 (in build file:.../example01/)
```

```
sbt:example01> test:compile
[info] Compiling 1 Scala source to .../example01/target/scala-2.12/test-classes ...
[info] Done compiling.
```

```
sbt:example01> test
[info] HelloSpec:
[info] The HelloApp object
[info] - should say 'Hello World!'
[info] Run completed in 368 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: ...
```

# No build file

The '*save session*' command writes manually set settings to a minimal *build.sbt*.

# No build file

The '*save session*' command writes manually set settings to a minimal *build.sbt*.

```
sbt:example01> session save  
[info] Reapplying settings...  
[info] Set current project to example01 (in build file:.../example01/)
```

```
sbt:example01> exit  
[info] shutting down server
```

```
example01 $ cat build.sbt  
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.8" % Test
```

```
example01 $ cat project/build.properties  
sbt.version=1.2.8
```

# No build file

The '*save session*' command writes manually set settings to a minimal *build.sbt*.

```
sbt:example01> session save  
[info] Reapplying settings...  
[info] Set current project to example01 (in build file:.../example01/)
```

```
sbt:example01> exit  
[info] shutting down server
```

```
example01 $ cat build.sbt  
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.8" % Test
```

```
example01 $ cat project/build.properties  
sbt.version=1.2.8
```

This is a possible but not the usual way to set up an sbt project.

## 2. sbt new

*.../example02*

# `sbt new [ gitter8-template ]`

quickly sets up a new sbt project using the specified gitter8 template.

Many useful templates for different purposes can be found here:

<https://github.com/foundweekends/giter8/wiki/giter8-templates>

# `sbt new [ gitter8-template ]`

quickly sets up a new sbt project using the specified gitter8 template.

Many useful templates for different purposes can be found here:

<https://github.com/foundweekends/giter8/wiki/giter8-templates>

*scala/scala-seed.g8*

... a useful template for a simple project setup.

```
pragmatic-sbt $ sbt new scala/scala-seed.g8  
[info] Set current project to ...
```

A minimal Scala project.

```
name [Scala Seed Project]: example02
```

```
Template applied in .../example02
```

```
pragmatic-sbt $ sbt new scala/scala-seed.g8  
[info] Set current project to ...
```

A minimal Scala project.

```
name [Scala Seed Project]: example02
```

```
Template applied in .../example02
```

```
pragmatic-sbt $ cd example02  
example02 $ tree  
. .  
├── build.sbt  
├── project  
│   ├── Dependencies.scala  
│   └── build.properties  
└── src  
    ├── main  
    │   └── scala  
    │       └── example  
    │           └── Hello.scala  
    └── test  
        └── scala  
            └── example  
                └── HelloSpec.scala
```

# Generated files

```
// project/build.properties  
sbt.version=1.2.8
```

```
// project/Dependencies.scala  
import sbt._  
  
object Dependencies {  
    lazy val scalaTest = "org.scalatest" %% "scalatest" % "3.0.8"  
}
```

```
// build.sbt  
import Dependencies._  
  
ThisBuild / scalaVersion      := "2.13.0"  
ThisBuild / version           := "0.1.0-SNAPSHOT"  
ThisBuild / organization       := "com.example"  
ThisBuild / organizationName := "example"  
  
lazy val root = (project in file("."))  
  .settings(  
    name := "example02",  
    libraryDependencies += scalaTest % Test  
  )
```

# ThisBuild

Settings in *ThisBuild* are the default settings for subprojects. This saves us from specifying the settings repeatedly in different subprojects.

A subproject can redefine any setting specified in *ThisBuild*.

# 3. Creating an application from the template project

*.../example03*

# Weather Application

The app queries the weather from <https://www.metaweather.com/api/> using gigahorse-okhttp as HTTP client and play-json to parse the JSON response.

# Weather Application

The app queries the weather from <https://www.metaweather.com/api/> using gigahorse-okhttp as HTTP client and play-json to parse the JSON response.

```
example03 $ tree
.
├── build.sbt
└── project
    ├── Dependencies.scala
    └── build.properties
└── src
    ├── main
    │   └── scala
    │       ├── weather
    │       │   ├── Weather.scala
    │       │   └── WeatherApp.scala
    │       └── weather.sc
    └── test
        └── scala
            └── weather
                └── WeatherSpec.scala
```

## project/Dependencies.scala

```
import sbt._

object Dependencies {
    lazy val playJson = "com.typesafe.play" %% "play-json" % "2.7.3"
    lazy val okHttp = "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0"
    lazy val scalaTest = "org.scalatest" %% "scalatest" % "3.0.5"
}
```

## build.sbt

```
import Dependencies._

ThisBuild / scalaVersion      := "2.12.8"
ThisBuild / version           := "0.1.0"
ThisBuild / organization       := "com.example"
ThisBuild / organizationName := "example"

lazy val root = (project in file("."))
  .settings(
    name := "Example03",
    libraryDependencies ++= Seq(okHttp, playJson, scalaTest % Test)
  )
```

## src/main/scala/weather/WeatherLib.scala

```
object WeatherLib {  
  
  def weatherOf(locationName: String, longFormat: Boolean = false)  
    (implicit ec: ExecutionContext): Future[String] = {  
    ??? // some impl  
  }  
}
```

(tested by: src/test/scala/weather/WeatherSpec.scala)

## src/main/scala/weather/WeatherApp.scala

```
object WeatherApp extends App {  
  
  import WeatherLib._  
  
  implicit val ec: ExecutionContext = ExecutionContext.global  
  
  def showWeatherOf(location: String): String = {  
    val weather = Await.result(weatherOf(location), 10.seconds)  
    s"\nThe weather in $location is: $weather\n"  
  }  
  
  println(showWeatherOf("Berlin"))  
}
```

# 4. Scala REPL

*.../example03*

The **console** command starts a Scala REPL from the sbt prompt with the project dependencies and classes on the classpath.

The **initialCommands** setting in *build.sbt* initializes the REPL session - typically used for the imports you need in every session of this project.

The **console** command starts a Scala REPL from the sbt prompt with the project dependencies and classes on the classpath.

The **initialCommands** setting in *build.sbt* initializes the REPL session - typically used for the imports you need in every session of this project.

```
ThisBuild / initialCommands :=  
    """  
    |import scala.concurrent._  
    |import scala.concurrent.duration._  
    |import scala.concurrent.ExecutionContext.Implicits.global  
    |import weather.WeatherLib._  
    |""".stripMargin
```

The **console** command starts a Scala REPL from the sbt prompt with the project dependencies and classes on the classpath.

The **initialCommands** setting in *build.sbt* initializes the REPL session - typically used for the imports you need in every session of this project.

```
ThisBuild / initialCommands :=  
    """  
    |import scala.concurrent._  
    |import scala.concurrent.duration._  
    |import scala.concurrent.ExecutionContext.Implicits.global  
    |import weather.WeatherLib._  
    |""".stripMargin
```

```
sbt:Example03> console  
[info] Starting scala interpreter...  
Welcome to Scala 2.12.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_212).  
Type in expressions for evaluation. Or try :help.  
  
import scala.concurrent._  
import scala.concurrent.duration._  
import scala.concurrent.ExecutionContext.Implicits.global  
import weather.WeatherLib._  
  
scala> weatherOf("Berlin")  
res0: scala.concurrent.Future[String] = Future(<not completed>)  
  
scala> Await.result(res0, 10.seconds)  
res1: String = showers
```

# 5. Aliases

*.../example03*

# .sbtrc

- Project specific aliases are defined in `[project_baseDirectory]/.sbtrc`.
- User specific aliases (valid in all projects of this user) are defined in `~/.sbtrc`.

# .sbtrc

- Project specific aliases are defined in **[project\_baseDirectory]/.sbtrc**.
- User specific aliases (valid in all projects of this user) are defined in **~/.sbtrc**.

```
alias a = alias
alias r = reload
alias c = compile
alias dl = dependencyList
```

```
sbt:Example03> a
  a = alias
  r = reload
  c = compile
  dl = dependencyList
```

```
sbt:Example03> r
[info] Loading ...
[info] Set current project to Example03 (in build file:.../example03/)
```

```
sbt:Example03> c
[info] Compiling 2 Scala sources to .../example03/target/scala-2.12/classes ...
[info] Done compiling.
[success] Total time: ...
```

```
sbt:Example03> dl
[info] Updating ProjectRef(uri("file:.../example03/"), "root")...
[info] Done updating.
[warn] There may be incompatibilities among your library dependencies; run 'evict'
[info] com.eed3si9n:gigahorse-core_2.12:0.5.0
[info] com.eed3si9n:gigahorse-okhttp_2.12:0.5.0
[info] com.example:example03_2.12:0.1.0
[info] com.fasterxml.jackson.core:jackson-annotations:2.9.8
[info] com.fasterxml.jackson.core:jackson-core:2.9.8
...
...
```

# 6. Plugins

*.../example03*

*<https://www.scala-sbt.org/release/docs/Using-Plugins.html>*

# Preinstalled Plugins

Some plugins are already installed and enabled in sbt.

The **plugins** command lists the currently installed plugins.

# Preinstalled Plugins

Some plugins are already installed and enabled in sbt.

The **plugins** command lists the currently installed plugins.

```
sbt:Example03> plugins
In file:.../example03/
  sbt.plugins.IvyPlugin: enabled in root
  sbt.plugins.JvmPlugin: enabled in root
  sbt.plugins.CorePlugin: enabled in root
  sbt.ScriptedPlugin
  sbt.plugins.SbtPlugin
  sbt.plugins.JUnitXmlReportPlugin: enabled in root
  sbt.plugins.Giter8TemplatePlugin: enabled in root
```

# User-installed Plugins

- Project specific plugins are installed in `project/plugins.sbt`.

# User-installed Plugins

- Project specific plugins are installed in **project/plugins.sbt**.
- User specific plugins (valid in all projects of this user) are installed in **~/.sbt/1.0/plugins/plugins.sbt**.

# Plugin: *sbt-updates*

shows possible library dependency updates.

(This possibly saves you from looking up the latest versions of libraries on mvnrepository.com.)

Source code: <https://github.com/rtimush/sbt-updates>

# Plugin: *sbt-updates*

shows possible library dependency updates.

(This possibly saves you from looking up the latest versions of libraries on mvnrepository.com.)

Source code: <https://github.com/rtimush/sbt-updates>

## plugins.sbt

```
addSbtPlugin("com.timushev.sbt" % "sbt-updates" % "0.4.1")
```

```
sbt:Example03> dependencyUpdates
[info] Found 3 dependency updates for example03
[info]   com.typesafe.play:play-json : 2.7.3  -> 2.7.4
[info]   org.scala-lang:scala-library : 2.12.8          -> 2.13.0
[info]   org.scalatest:scalatest:test : 3.0.5  -> 3.0.8
```

# Plugin: *sbt-dependency-graph*

shows a tree of library dependencies.

Source code: <https://github.com/jrudolph/sbt-dependency-graph>

# Plugin: *sbt-dependency-graph*

shows a tree of library dependencies.

Source code: <https://github.com/jrudolph/sbt-dependency-graph>

## plugins.sbt

```
addSbtPlugin("net.virtual-void" % "sbt-dependency-graph" % "0.9.2")
```

```
sbt:Example03> dependencyTree
[info] com.example:example03_2.12:0.1.0 [S]
[info]   +-com.eed3si9n:gigahorse-okhttp_2.12:0.5.0 [S]
[info]     +-com.eed3si9n:gigahorse-core_2.12:0.5.0 [S]
[info]       +-com.typesafe:ssl-config-core_2.12:0.4.0 [S]
[info]         +-com.typesafe:config:1.3.3
[info]           +-org.scala-lang.modules:scala-parser-combinators_2.12:1.1.2 [S]
[info]             +-org.reactivestreams:reactive-streams:1.0.2
[info]             +-org.slf4j:slf4j-api:1.7.26
[info]             +-com.squareup.okhttp3:okhttp:3.14.2
[info]               +-com.squareup.okio:okio:1.17.2
[info] ...
[info] ... // more dependencies
```

## Plugin: *sbt-sh*

enables shell commands from the sbt prompt.

Source code: <https://github.com/melezov/sbt-sh>

# Plugin: *sbt-sh*

enables shell commands from the sbt prompt.

Source code: <https://github.com/melezov/sbt-sh>

## plugins.sbt

```
addSbtPlugin("com.oradian.sbt" % "sbt-sh" % "0.3.0")
```

```
sbt:Example03> sh ls -l
total 8
-rw-r--r-- 1 hermann staff 367 1 Jul 15:55 build.sbt
drwxr-xr-x 6 hermann staff 192 2 Jul 12:22 project
drwxr-xr-x 4 hermann staff 128 30 Jun 23:04 src
drwxr-xr-x 6 hermann staff 192 2 Jul 12:25 target
```

# Plugin: *sbt-git*

enables git commands from the sbt prompt.

Source code: <https://github.com/sbt/sbt-git>

# Plugin: *sbt-git*

enables git commands from the sbt prompt.

Source code: <https://github.com/sbt/sbt-git>

## plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-git" % "1.0.0")
```

# Plugin: *sbt-git*

enables git commands from the sbt prompt.

Source code: <https://github.com/sbt/sbt-git>

## plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-git" % "1.0.0")
```

```
sbt:Example03> git status
[info] Auf Branch master
[info] Ihr Branch ist auf demselben Stand wie 'origin/master'.
[info] Änderungen, die nicht zum Commit vorgemerkt sind:
[info]   (benutzen Sie "git add/rm <Datei>...", um die Änderungen zum Commit vorzu
[info]   (benutzen Sie "git checkout -- <Datei>...", um die Änderungen im Arbeitsve
[info]     gelöscht:      src/main/scala/example/WeatherLib.scala
[info]     gelöscht:      src/main/scala/example/WeatherApp.scala
[info]     gelöscht:      src/test/scala/example/WeatherSpec.scala
[info] Unversionierte Dateien:
[info]   (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumer
[info]     src/test/scala/weather/
[info] keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "gi
```

# Plugin: *sbt-coursier*

alternative for the ivy dependency manager built into sbt.  
Fetches dependency jars in parallel.

As of sbt 1.3.0 the need to install this plugin disappears.  
Coursier will become the default dependency manager already built into sbt.

Web site: <https://get-coursier.io/docs/sbt-coursier>  
Source code: <https://github.com/coursier/coursier>

# Plugin: *sbt-coursier*

alternative for the ivy dependency manager built into sbt.  
Fetches dependency jars in parallel.

As of sbt 1.3.0 the need to install this plugin disappears.  
Coursier will become the default dependency manager already built into sbt.

Web site: <https://get-coursier.io/docs/sbt-coursier>  
Source code: <https://github.com/coursier/coursier>

## plugins.sbt

```
addSbtPlugin("io.get-coursier" % "sbt-coursier" % "1.1.0-M14-4")
```

# Plugin: *sbt-native-packager*

enables Java/Scala app packaging in different package formats: zip, dmg (macOS), msi (Windows), deb and rpm (Linux) and docker etc.

Sie Chapter 11: App Packaging

Web site: <https://sbt-native-packager.readthedocs.io/en/stable/>

Source code: <https://github.com/sbt/sbt-native-packager>

# Plugin: *sbt-native-packager*

enables Java/Scala app packaging in different package formats: zip, dmg (macOS), msi (Windows), deb and rpm (Linux) and docker etc.

Sie Chapter 11: App Packaging

Web site: <https://sbt-native-packager.readthedocs.io/en/stable/>

Source code: <https://github.com/sbt/sbt-native-packager>

## project/plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.3.24")
```

## build.sbt

```
enablePlugin(JavaAppPackaging)
```

Unlike many other plugins this one must be enabled explicitly in *build.sbt*.

See: [11. Packaging](#)

Having installed all these plugins our plugin list becomes somewhat longer:

```
sbt:Example03> plugins
In file:.../example03/
  sbt.plugins.IvyPlugin: enabled in root
  sbt.plugins.JvmPlugin: enabled in root
  sbt.plugins.CorePlugin: enabled in root
  sbt.ScriptedPlugin
  sbt.plugins.SbtPlugin
  sbt.plugins.JUnitXmlReportPlugin: enabled in root
  sbt.plugins.Giter8TemplatePlugin: enabled in root
  coursier.sbtcoursier.CoursierPlugin: enabled in root
  coursier.sbtcoursiershared.SbtCoursierShared: enabled in root
  net.vonbuchholtz.sbt.dependencycheck.DependencyCheckPlugin: enabled in root
  com.typesafe.sbt.GitBranchPrompt
  com.typesafe.sbt.GitPlugin: enabled in root
  com.typesafe.sbt.GitVersioning
  com.timushev.sbt.updates.UpdatesPlugin: enabled in root
  com.oradian.sbt.SbtShPlugin: enabled in root
  net.virtualvoid.sbt.graph.DependencyGraphPlugin: enabled in root
```

This list is without *sbt-native-packager*.

This plugin bundle adds a lot more plugins, one for each package format.

# Some other useful plugins

## sbt-assembly

- create fat JARs
- <https://github.com/sbt/sbt-assembly>

## sbt-scalariform

- code formatting using Scalariform
- <https://github.com/sbt/sbt-scalariform>

## new-sbt-scalafmt

- code formatting using Scalafmt
- <https://github.com/lucidsoftware/neo-sbt-scalafmt>

# Some other useful plugins

## sbt-assembly

- create fat JARs
- <https://github.com/sbt/sbt-assembly>

## sbt-scalariform

- code formatting using Scalariform
- <https://github.com/sbt/sbt-scalariform>

## new-sbt-scalafmt

- code formatting using Scalafmt
- <https://github.com/lucidsoftware/neo-sbt-scalafmt>

Many more to find at:

<https://www.scala-sbt.org/1.x/docs/Community-Plugins.html>

# 7. Multi-project Builds

*.../example04*

*<https://www.scala-sbt.org/release/docs/Multi-Project.html>*

# Multi-project Build

In *example03* we kept all source code in one sbt project, the "root" project. The *src* directory was located directly under the root directory of the build.

Now we split the code into subprojects - "WeatherLib" and "WeatherApp" - with their own project directories "lib" and "app". The third subproject "root" does not have and does not need a "src" folder of its own. It is the over-arching project which controls the other two.

# Multi-project Build

In *example03* we kept all source code in one sbt project, the "root" project. The *src* directory was located directly under the root directory of the build.

Now we split the code into subprojects - "WeatherLib" and "WeatherApp" - with their own project directories "lib" and "app". The third subproject "root" does not have and does not need a "src" folder of its own. It is the over-arching project which controls the other two.

Subprojects are (if not specified otherwise) completely independent of each other, i.e: tasks can be executed in parallel.

The subprojects have completely unrelated settings. If a setting is undefined in a subproject, the default setting from *ThisBuild* is used.

## Directory tree

```
example04 $ tree
.
├── app
│   └── src
│       └── main
│           └── scala
│               └── app
│                   └── WeatherApp.scala
└── lib
    └── src
        ├── main
        │   └── scala
        │       └── libWeather
        │           └── Weather.scala
        └── test
            └── scala
                └── libWeather
                    └── WeatherSpec.scala
└── build.sbt
└── project
    └── Dependencies.scala
```

## build.sbt (with three subprojects: root, app, lib)

```
import Dependencies._

ThisBuild / scalaVersion      := "2.12.8"
ThisBuild / version          := "0.1.0"
ThisBuild / organization     := "com.example"
ThisBuild / organizationName := "example"

lazy val root = (project in file("."))
  .aggregate(app, lib)
  .settings(
    name := "Example04",
  )

lazy val app = (project in file("app"))
  .dependsOn(lib)
  .settings(
    name := "WeatherApp",
    initialCommands := """ ... // imports """.stripMargin,
  )

lazy val lib = (project in file("lib"))
  .settings(
    name := "WeatherLib",
    libraryDependencies += Seq(okHttp, playJson, scalaTest % Test)
  )
```

## .aggregate(...)

*root.aggregate(app, lib) :*

- Every command issued in "root" is propagated to "app" and "lib".
- e.g.: *compile* in "root" triggers *app/compile* and "*lib/compile*"
- *app/compile* and "*lib/compile*" could be executed in parallel if the subprojects do not depend on each other.

## .aggregate(...)

*root.aggregate(app, lib) :*

- Every command issued in "root" is propagated to "app" and "lib".
- e.g.: *compile* in "root" triggers *app/compile* and "lib/compile"
- *app/compile* and "lib/compile" could be executed in parallel if the subprojects do not depend on each other.

## .dependsOn(...)

*app.dependsOn(lib) :*

- "lib" is part of the classpath of "app".
- "lib must be updated and successfully compiled before "app" can be compiled (prevents parallel compilation).

## .aggregate(...)

*root.aggregate(app, lib) :*

- Every command issued in "root" is propagated to "app" and "lib".
- e.g.: *compile* in "root" triggers *app/compile* and *"lib/compile"*
- *app/compile* and *"lib/compile"* could be executed in parallel if the subprojects do not depend on each other.

## .dependsOn(...)

*app.dependsOn(lib) :*

- "lib" is part of the classpath of "app".
- "lib must be updated and successfully compiled before "app" can be compiled (prevents parallel compilation).

## project specific sbt commands:

- **projects** lists the projects in the current build.
- **project [name]** switches the current project.

```
sbt:Example04> projects  
[info] In file:.../example04/  
[info]      app  
[info]      lib  
[info] * root
```

```
sbt:Example04> project lib  
[info] Set current project to WeatherLib (in build file:.../example04/)
```

```
sbt:WeatherLib> projects  
[info] In file:.../example04/  
[info]      app  
[info] * lib  
[info]      root
```

```
sbt:WeatherLib> compile  
[info] Compiling 1 Scala source to .../example04/lib/target/scala-2.12/classes ...  
[info] Done compiling.
```

```
sbt:WeatherLib> project root  
[info] Set current project to Example04 (in build file:.../example04/)
```

```
sbt:Example04> compile  
[info] Compiling 1 Scala source to .../example04/app/target/scala-2.12/classes ...  
[info] Done compiling.
```

# 8. Cross Builds

*.../example05*

*<https://www.scala-sbt.org/release/docs/Cross-Build.html>*

# Using cross-built libraries

We already used cross-built libraries when specifying the *libraryDependencies* with %% between group id and artefact id.

# Using cross-built libraries

We already used cross-built libraries when specifying the *libraryDependencies* with %% between group id and artifact id.

```
libraryDependencies += "com.typesafe.play" %% "play-json" % "2.7.4"
```

This automatically appends the current *scalaVersion* to the artifact id.  
E.g.: *play-json* becomes *play-json\_2.13*.

# Creating cross-built libraries

The following `example05` shows how to create cross-built libraries.

Cross-building is controlled by the setting `crossScalaVersions`.

```
import Dependencies._

val scala213 = "2.13.0"
val scala212 = "2.12.8"
val supportedScalaVersions = List(scala213, scala212)

ThisBuild / scalaVersion      := scala213
ThisBuild / version          := "0.1.0"
ThisBuild / organization     := "com.example"
ThisBuild / organizationName := "example"

lazy val root = (project in file("."))
  .aggregate(app, lib)
  .settings(
    name := "Example05",
    publish / skip := true, // nothing to publish
    crossScalaVersions := Nil, // set to Nil on the aggregating project
  )

lazy val app = (project in file("app"))
  .dependsOn(lib)
  .settings(
    name := "WeatherApp",
    publish / skip := true, // only libraries must be published
    crossScalaVersions := supportedScalaVersions,
    initialCommands := """ ... // imports """.stripMargin,
  )

lazy val lib = (project in file("lib"))
  .settings(
    name := "WeatherLib",
    crossScalaVersions := supportedScalaVersions,
```

# `crossScalaVersions`

- `crossScalaVersions` is the setting to achieve cross-building our source code to different binary Scala versions.
- Its value is a Seq of Strings, each containing a valid Scala version like "2.12.8" or "2.13.0".
- Skip cross-building in the overarching "root" project which has no source code. (Set `crossScalaVersions` to Nil.)

# `crossScalaVersions`

- `crossScalaVersions` is the setting to achieve cross-building our source code to different binary Scala versions.
- Its value is a Seq of Strings, each containing a valid Scala version like "2.12.8" or "2.13.0".
- Skip cross-building in the overarching "root" project which has no source code. (Set `crossScalaVersions` to Nil.)

## `+ and ++ commands`

- The Scala version can be selected at the sbt prompt with the `++` command, e.g. `++2.12.8`
- If the `++` command is followed by a `!` you can also switch to a Scala version not listed in `build.sbt`, e.g. `++2.11.12!`
- To build against all versions listed in `crossScalaVersions`, prefix the action to run with `+`, e.g. `+compile` or `+test`.

```
sbt:Example05> scalaVersion  
[info] app / scalaVersion  
[info] 2.13.0  
[info] lib / scalaVersion  
[info] 2.13.0  
[info] scalaVersion  
[info] 2.13.0
```

```
sbt:Example05> compile  
[info] Compiling 1 Scala source to .../example05/lib/target/scala-2.13/classes ...  
[info] Done compiling.  
[info] Compiling 1 Scala source to .../example05/app/target/scala-2.13/classes ...  
[info] Done compiling.  
[success] Total time: ...
```

```
sbt:Example05> ++2.12.8 -v  
[info] Setting Scala version to 2.12.8 on 2 projects.  
[info] Switching Scala version on:  
[info]   lib (2.12.8, 2.13.0)  
[info]   app (2.12.8, 2.13.0)  
[info] Excluding projects:  
[info]   * root()  
[info] Reapplying settings...  
[info] Set current project to Example05 (in build file:.../example05/)
```

```
sbt:Example05> compile
[info] Compiling 1 Scala source to .../example05/lib/target/scala-2.12/classes ...
[info] Done compiling.
[info] Compiling 1 Scala source to .../example05/app/target/scala-2.12/classes ...
[info] Done compiling.
```

```
sbt:Example05> clean
```

```
sbt:Example05> +compile
[info] Setting Scala version to 2.13.0 on 2 projects.
[info] ...
[info] Compiling 1 Scala source to .../example05/lib/target/scala-2.13/classes ...
[info] Done compiling.
[info] Compiling 1 Scala source to .../example05/app/target/scala-2.13/classes ...
[info] Done compiling.

...
[info] Setting Scala version to 2.12.8 on 2 projects.
[info] ...
[info] Compiling 1 Scala source to .../example05/lib/target/scala-2.12/classes ...
[info] Done compiling.
[info] Compiling 1 Scala source to .../example05/app/target/scala-2.12/classes ...
[info] Done compiling.

...
```

```
sbt:Example05> project lib  
[info] Set current project to WeatherLib (in build file:.../example05/)
```

```
sbt:WeatherLib> ++2.11.12! compile  
[info] Forcing Scala version to 2.11.12 on all projects.  
[info] Reapplying settings...  
[info] Set current project to WeatherLib (in build file:.../example05/)  
[info] Compiling 1 Scala source to .../example05/lib/target/scala-2.11/classes ...  
[info] Done compiling.  
[success] Total time: ...
```

```
sbt:Example05> project lib  
[info] Set current project to WeatherLib (in build file:.../example05/)
```

```
sbt:WeatherLib> ++2.11.12! compile  
[info] Forcing Scala version to 2.11.12 on all projects.  
[info] Reapplying settings...  
[info] Set current project to WeatherLib (in build file:.../example05/)  
[info] Compiling 1 Scala source to .../example05/lib/target/scala-2.11/classes ...  
[info] Done compiling.  
[success] Total time: ...
```

The class files are compiled to version specific subdirectories of the *target* directories.

```
example05 $ tree lib/target/scala-*  
lib/target/scala-2.11  
└── classes  
    └── libWeather  
        └── *.class  
lib/target/scala-2.12  
└── classes  
    └── libWeather  
        └── *.class  
lib/target/scala-2.13  
└── classes  
    └── libWeather
```

# 9. Version specific source code

*.../example06*

# Version specific code locations

If you have source files for different Scala versions, place them into different versions specific source directories, e.g.:

- *src/main/scala* for code common to all Scala versions
- *src/main/scala-2.12* for code specific to Scala 2.12.x
- *src/main/scala-2.13* for code specific to Scala 2.13.x

# Version specific code locations

If you have source files for different Scala versions, place them into different versions specific source directories, e.g.:

- *src/main/scala* for code common to all Scala versions
- *src/main/scala-2.12* for code specific to Scala 2.12.x
- *src/main/scala-2.13* for code specific to Scala 2.13.x

```
example06 $ tree app/src
app/src
└── main
    ├── scala
    │   └── app
    │       └── WeatherBase.scala
    ├── scala-2.12
    │   └── app
    │       └── WeatherApp.scala
    └── scala-2.13
        └── app
            └── WeatherApp.scala
```

*src/main/scala/app/WeatherBase.scala*

```
class WeatherBase {
    import libWeather.Weather._
    implicit val ec: ExecutionContext = ExecutionContext.global

    def showWeatherOf(location: String): String = {
        val scalaVersion = util.Properties.versionString
        val weather = Await.result(weatherOf(location), 10.seconds)
        s"\nScala $scalaVersion:\nThe weather in $location is: $weather\n"
    }
}
```

*src/main/scala-2.12/app/WeatherApp.scala*

```
object WeatherApp extends WeatherBase with App {
    println(showWeatherOf("Berlin"))
}
```

*src/main/scala-2.13/app/WeatherApp.scala*

```
object WeatherApp extends WeatherBase with App {
    import scala.util.chaining._
    showWeatherOf("Berlin") tap println
}
```

# 10. Library Dependencies

*.../example07*

*<https://www.scala-sbt.org/release/docs/Library-Dependencies.html>*

# Unmanaged dependencies

Create a directory named ***lib*** under your subproject's root and put the jars you need into it. sbt will add these jars to your subproject's classpath.

This way of adding library dependencies is very uncommon, but may be useful in some situations.

# Managed dependencies

This is the usual way to define dependencies. We used it already in the first example.

You have to specify the setting **libraryDependencies** which takes a *Seq[ModuleId]*. Each *ModuleId* consists of a groupId, artifactId and revision joined with the %-operator and optionally followed by a configuration such as "test" or *Test* (the type-safe way).

When using the %%-operator between groupId and artifactId sbt automatically detects the right binary Scala version of the library from the current setting of **scalaVersion**.

# Managed dependencies

This is the usual way to define dependencies. We used it already in the first example.

You have to specify the setting **libraryDependencies** which takes a *Seq[ModuleId]*. Each *ModuleId* consists of a groupId, artifactId and revision joined with the %-operator and optionally followed by a configuration such as "test" or Test (the type-safe way).

When using the %%-operator between groupId and artifactId sbt automatically detects the right binary Scala version of the library from the current setting of **scalaVersion**.

```
// gigahorse-okhttp for binary Scala version 2.12.x
libraryDependencies += "com.eed3si9n" % "gigahorse-okhttp_2.12" % "0.5.0"
```

```
// gigahorse-okhttp using the current scalaVersion setting
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0"
```

# Flexible Ivy revisions

Ending the revision number with a + selects the latest available subrevision of a library.

```
// gigahorse-okhttp using the latest 0.5.x revision for the current scalaVersion  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.+"
```

# Flexible Ivy revisions

Ending the revision number with a + selects the latest available subrevision of a library.

```
// gigahorse-okhttp using the latest 0.5.x revision for the current scalaVersion  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.+"
```

You may also specify *"latest.release"*, *"latest.milestone"* or *"latest.integration"* instead of a revision number.

```
// depend on the latest available release of the gigahorse-okhttp module  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "latest.release"
```

```
// depend on the latest available integration (release or milestone) of the gigahorse-okhttp module  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "latest.integration"
```

# Flexible Ivy revisions

Ending the revision number with a + selects the latest available subrevision of a library.

```
// gigahorse-okhttp using the latest 0.5.x revision for the current scalaVersion  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.+"
```

You may also specify "*latest.release*", "*latest.milestone*" or "*latest.integration*" instead of a revision number.

```
// depend on the latest available release of the gigahorse-okhttp module  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "latest.release"
```

```
// depend on the latest available integration (release or milestone) of the gigahorse-okhttp module  
libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "latest.integration"
```

See also:

<https://ant.apache.org/ivy/history/2.3.0/ivyfile/dependency.html#revision>

# Resolvers

Resolvers is a Seq of (local or remote) locations looked up by sbt when resolving library dependencies.

# Resolvers

Resolvers is a Seq of (local or remote) locations looked up by sbt when resolving library dependencies.

## Default Resolvers

Some resolvers are predefined by default. (Hence we don't need to define resolvers in many cases.)

- the Maven2 repository at <https://repo1.maven.org/maven2/>
- the local sbt cache at `~/.sbt/preloaded`
- jars published locally to Ivy at `~/.ivy/local`
- the project resolver which resolves inter-project dependencies, e.g. the dependency between the `app` and `lib` subprojects

The default resolvers can be listed in a quite weird format with  
**show externalResolvers**

To change or remove the default resolvers, you would need to override  
**externalResolvers**.

# User-defined Resolvers

To add your own resolvers to the default ones, use the setting **resolvers**.

To add an additional repository, use

```
resolvers += "name" at "location"
```

# User-defined Resolvers

To add your own resolvers to the default ones, use the setting **resolvers**.

To add an additional repository, use

```
resolvers += "name" at "location"
```

## Examples

```
resolvers += "Local Maven Repository" at "file://" + Path.userHome.getAbsolutePath() + ".m2/repository"
```

```
resolvers += "Sonatype OSS Public" at "https://oss.sonatype.org/content/repositories/public"
```

```
// using a predefined shortcut for the "Sonatype public" repo  
resolvers += Resolver.sonatypeRepo("public")
```

```
resolvers += "Typesafe Releases" at "https://repo.typesafe.com/typesafe/releases"
```

```
// using a predefined shortcut for the "Typesafe releases" repo  
resolvers += Resolver.typesafeRepo("releases")
```

# 11. Packaging

*.../example07*

# Commands for package generation

sbt comes with some commands for the generation of packages (jar files).

- *packageBin* Produces a binary jar.
- *packageSrc* Produces a jar containing sources and resources.
- *packageDoc* Produces a jar containing API documentation.
- *package* Is an alias for *packageBin*.

These commands generate jar files under  
**[subproject\_root]/target/scala-[scalaVersion]**

In a cross build project all commands can be prefixed with + in order to package jars for all supported scala versions of the build.

```
sbt:Example07> project lib  
[info] Set current project to WeatherLib (in build file:.../example07/)
```

```
sbt:WeatherLib> packageBin  
[info] Packaging .../example07/lib/target/scala-2.13/weatherlib_2.13-0.1.0.jar ...  
[info] Done packaging.
```

```
sbt:WeatherLib> packageSrc  
[info] Packaging .../example07/lib/target/scala-2.13/weatherlib_2.13-0.1.0-sources  
[info] Done packaging.
```

```
sbt:WeatherLib> packageDoc  
[info] Main Scala API documentation to .../example07/lib/target/scala-2.13/api...  
model contains 3 documentable templates  
[info] Main Scala API documentation successful.  
[info] Packaging .../example07/lib/target/scala-2.13/weatherlib_2.13-0.1.0-javadoc  
[info] Done packaging.
```

```
example07 $ ls -1 lib/target/scala-2.13/*.jar  
lib/target/scala-2.13/weatherlib_2.13-0.1.0-javadoc.jar  
lib/target/scala-2.13/weatherlib_2.13-0.1.0-sources.jar  
lib/target/scala-2.13/weatherlib_2.13-0.1.0.jar
```

# 12. Publishing

*.../example07*

*<https://www.scala-sbt.org/1.x/docs/Publishing.html>*

# Local Publishing

- Using the command **publishLocal** you can publish a jar (typically a library jar) to the local Ivy repository under `~/.ivy2/local`.
- Using the command **publishM2** you can publish a jar (typically a library jar) to the local Maven2 repository under `~/.m2/repository`.
- **publishMavenStyle** is *true* (default) if the jars are published in Maven style (with a pom file) or *false* if the Ivy style is used.

```
sbt:Example07> project lib  
[info] Set current project to WeatherLib (in build file:.../example07/)
```

```
sbt:WeatherLib> publishMavenStyle  
[info] true
```

```
sbt:WeatherLib> publishLocal  
[info] Writing Ivy file .../example07/lib/target/scala-2.13/resolution-cache/com.e  
[info] Wrote .../example07/lib/target/scala-2.13/weatherlib_2.13-0.1.0.pom  
[info] :: delivering :: com.example#weatherlib_2.13;0.1.0 :: 0.1.0 :: release :: F  
[info]     delivering ivy file to .../example07/lib/target/scala-2.13/ivy-0.1.0.xml  
[info]     published weatherlib_2.13 to ~/.ivy2/local/com.example/weatherlib_2.13/  
[info]     published ivy to ~/.ivy2/local/com.example/weatherlib_2.13/0.1.0/ivys/i
```

# Publishing to a resolver (ie.e to a remote repo)

To publish a jar to a remote repo (in Maven style or in Ivy style) ...

- define the resolver (the destination to publish to) with the **publishTo** setting
- add the credentials for that resolver to the **credentials** setting
- (or alternatively and preferably - load the credentials from a properties file)
- run the **publish** command

# What to publish?

Publishing makes only sense for libraries, not for applications.

To prevent a subproject from being published add the setting:

**publish / skip := true**

# What to publish?

Publishing makes only sense for libraries, not for applications.

To prevent a subproject from being published add the setting:  
**publish / skip := true**

```
...
lazy val root = (project in file("."))
  .aggregate(app, lib)
  .settings(
    name := "Example05",
    publish / skip := true, // no code to publish
    ...
  )
lazy val app = (project in file("app"))
  .dependsOn(lib)
  .settings(
    name := "WeatherApp",
    publish / skip := true, // no need to publish the app
    ...
  )
lazy val lib = (project in file("lib"))
  .settings(
    name := "WeatherLib",
    // the library project should be published
    ...
  )
```

# 13. Installable Packages with *sbt-native-packager*

*.../example08*

*<https://sbt-native-packager.readthedocs.io/>*

# Library or App?

If you write a library you have to publish it to a public repo so that others can use it as a library dependency.

This can be done with sbt without additional plugin.

# Library or App?

If you write a library you have to publish it to a public repo so that others can use it as a library dependency.

This can be done with sbt without additional plugin.

If you write an application you have to generate a installable package in the format of the intended destination system, e.g. a Debian package for Debian or Ubuntu Linux or an MSI package for Windows or a Docker image or some other installable format.

The **sbt-native-packager** plugin allows you to create a software distribution of an application and supports many formats.

# Enable JavaAppPackaging

## project/plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.3.24")
```

## build.sbt

```
lazy val app = (project in file("app"))
  .dependsOn(lib)
  .enablePlugins(JavaAppPackaging)
  .settings( ... )
```

The plugin must be enabled for the project you want to package.

# Staging

The **stage** command creates a local directory ( *.../target/universal/stage* ) with all the files laid out as required in the final distribution.

# Staging

The **stage** command creates a local directory (*.../target/universal/stage*) with all the files laid out as required in the final distribution.

```
sbt:Example08> project app  
[info] Set current project to WeatherApp (in build file:.../example08/)
```

```
sbt:WeatherApp> stage  
[info] Packaging ...
```

# Staging

The **stage** command creates a local directory (*.../target/universal/stage*) with all the files laid out as required in the final distribution.

```
sbt:Example08> project app  
[info] Set current project to WeatherApp (in build file:.../example08/)
```

```
sbt:WeatherApp> stage  
[info] Packaging ...
```

```
example08 $ tree app/target/universal/stage  
app/target/universal/stage  
├── bin  
│   ├── weatherapp  
│   └── weatherapp.bat  
└── lib  
    ├── com.eed3si9n.gigahorse-core_2.13-0.5.0.jar  
    ├── com.eed3si9n.gigahorse-okhttp_2.13-0.5.0.jar  
    ├── com.example.weatherapp-0.1.0.jar  
    ├── com.example.weatherlib-0.1.0.jar  
    ├── com.fasterxml.jackson.core.jackson-annotations-2.9.8.jar  
    └── ...
```

# Start Scripts

*bin/weatherapp* can start the application on Mac and Linux.

*bin/weatherapp.bat* can start the application on Windows.

```
example08 $ app/target/universal/stage/bin/weatherapp
```

```
Scala version 2.13.0:
```

```
The weather in Berlin is: light rain
```

```
example08 $
```

# Start Scripts

*bin/weatherapp* can start the application on Mac and Linux.

*bin/weatherapp.bat* can start the application on Windows.

```
example08 $ app/target/universal/stage/bin/weatherapp  
Scala version 2.13.0:  
The weather in Berlin is: light rain  
example08 $
```

# Setting: maintainer

The following packaging commands require a **maintainer** setting in the build. Otherwise sbt would issue a warning: "The maintainer is empty"

```
maintainer      := "functional.hacker@example.com",
```

# Zip distribution

The **universal:packageBin** command creates a zip file containing all staged files. The **dist** command achieves the same.

# Zip distribution

The **universal:packageBin** command creates a zip file containing all staged files. The **dist** command achieves the same.

```
sbt:WeatherApp> universal:packageBin  
[info] ...  
[info] Your package is ready in .../example08/app/target/universal/weatherapp-0.1.
```

# Zip distribution

The **universal:packageBin** command creates a zip file containing all staged files. The **dist** command achieves the same.

```
sbt:WeatherApp> universal:packageBin  
[info] ...  
[info] Your package is ready in .../example08/app/target/universal/weatherapp-0.1.
```

```
example08 $ unzip -l app/target/universal/weatherapp-0.1.0.zip  
Archive: app/target/universal/weatherapp-0.1.0.zip  
      Length      Date    Time     Name  
-----  
      5007  07-06-2019 00:16  weatherapp-0.1.0/lib/com.example.weatherapp-0.1.0.ja  
     6357  07-05-2019 23:22  weatherapp-0.1.0/lib/com.example.weatherlib-0.1.0.ja  
   33391  12-16-2018 00:06  weatherapp-0.1.0/lib/com.fasterxml.jackson.datatype.  
   42704  06-19-2019 02:15  weatherapp-0.1.0/lib/com.eed3si9n.gigahorse-okhttp_2  
 160144  06-19-2019 02:15  weatherapp-0.1.0/lib/com.eed3si9n.gigahorse-core_2.1  
 425763  05-19-2019 19:38  weatherapp-0.1.0/lib/com.squareup.okhttp3.okhttp-3.1  
...  
 726698  06-12-2019 12:45  weatherapp-0.1.0/lib/com.typesafe.play.play-json_2.1  
 10567  07-06-2019 00:33  weatherapp-0.1.0/bin/weatherapp  
   6317  07-06-2019 00:33  weatherapp-0.1.0/bin/weatherapp.bat  
-----  
 14304396  
                  23 files
```

# Gzipped tarball distribution

The `universal:packageZipTarball` command creates a tgz file containing all the files staged.

# Gzipped tarball distribution

The **universal:packageZipTarball** command creates a tgz file containing all the files staged.

```
sbt:WeatherApp> universal:packageZipTarball
[info] ...
Making /var/folders/.../weatherapp-0.1.0/bin/weatherapp executable
Making /var/folders/.../weatherapp-0.1.0/bin/weatherapp.bat executable
Running with tar -pcvf /var/folders/zg/1y5syxvj0mb5v3v0p0m8mhnc0000gn/T/sbt_68ba81
a weatherapp-0.1.0
a weatherapp-0.1.0/bin
a weatherapp-0.1.0/lib
a weatherapp-0.1.0/lib/com.squareup.okio.okio-1.17.2.jar
...
a weatherapp-0.1.0/bin/weatherapp
a weatherapp-0.1.0/bin/weatherapp.bat
[success] Total time: ...
```

# Gzipped tarball distribution

The `universal:packageZipTarball` command creates a tgz file containing all the files staged.

```
sbt:WeatherApp> universal:packageZipTarball
[info] ...
Making /var/folders/.../weatherapp-0.1.0/bin/weatherapp executable
Making /var/folders/.../weatherapp-0.1.0/bin/weatherapp.bat executable
Running with tar -pcvf /var/folders/zg/1y5syxvj0mb5v3v0p0m8mhnc0000gn/T/sbt_68ba81
a weatherapp-0.1.0
a weatherapp-0.1.0/bin
a weatherapp-0.1.0/lib
a weatherapp-0.1.0/lib/com.squareup.okio.okio-1.17.2.jar
...
a weatherapp-0.1.0/bin/weatherapp
a weatherapp-0.1.0/bin/weatherapp.bat
[success] Total time: ...
```

```
example08 $ unzip -l app/target/universal/weatherapp-0.1.0.zip
HermannMBP:example08 hermann$ tar -tzf app/target/universal/weatherapp-0.1.0.tgz
weatherapp-0.1.0/
weatherapp-0.1.0/bin/
weatherapp-0.1.0/lib/
weatherapp-0.1.0/lib/com.squareup.okio.okio-1.17.2.jar
...
weatherapp-0.1.0/bin/weatherapp
weatherapp-0.1.0/bin/weatherapp.bat
```

# OSX DMG distribution

The `universal:packageOsxDmg` command creates a dmg file for macOS.

# OSX DMG distribution

The `universal:packageOsxDmg` command creates a dmg file for macOS.

```
sbt:WeatherApp> universal:packageOsxDmg
[info] ...
created: ....example08/app/target/universal/weatherapp-0.1.0.dmg
/dev/disk3          GUID_partition_scheme
/dev/disk3s1        Apple_HFS
"disk3" ejected.
[success] Total time: ...
```

# OSX DMG distribution

The `universal:packageOsxDmg` command creates a dmg file for macOS.

```
sbt:WeatherApp> universal:packageOsxDmg
[info] ...
created: .../example08/app/target/universal/weatherapp-0.1.0.dmg
/dev/disk3          GUID_partition_scheme
/dev/disk3s1        Apple_HFS
.../example08/app/targe
"disk3" ejected.
[success] Total time: ...
```

```
example08 $ hdiutil mount app/target/universal/weatherapp-0.1.0.dmg
/dev/disk3          GUID_partition_scheme
/dev/disk3s1        Apple_HFS
.../Volumes/weatherapp-0.1
```

```
example08 $ tree /Volumes/weatherapp-0.1.0
/Volumes/weatherapp-0.1.0
├── bin
│   ├── weatherapp
│   └── weatherapp.bat
└── lib
    ├── com.eed3si9n.gigahorse-okhttp_2.13-0.5.0.jar
    ├── com.example.weatherapp-0.1.0.jar
    └── com.example.weatherlib-0.1.0.jar
```

# Dockerize the App

The command **docker:publishLocal** creates a docker image with the app.

# Dockerize the App

The command **docker:publishLocal** creates a docker image with the app.

```
sbt:WeatherApp> docker:publishLocal
...
[info] Sending build context to Docker daemon 14.33MB
[info] Step 1/15 : FROM openjdk:8 as stage0
[info]    ----> b8d3f94869bb
...
[info] Step 15/15 : CMD []
[info]    ----> Running in a1e6517b25a4
[info] Removing intermediate container a1e6517b25a4
[info]    ----> 96d21665e894
[info] Successfully built 96d21665e894
[info] Successfully tagged weatherapp:0.1.0
[info] Built image weatherapp with tags [0.1.0]
[success] Total time: ...
```

# Dockerize the App

The command **docker:publishLocal** creates a docker image with the app.

```
sbt:WeatherApp> docker:publishLocal
...
[info] Sending build context to Docker daemon 14.33MB
[info] Step 1/15 : FROM openjdk:8 as stage0
[info]    ----> b8d3f94869bb
...
[info] Step 15/15 : CMD []
[info]    ----> Running in a1e6517b25a4
[info] Removing intermediate container a1e6517b25a4
[info]    ----> 96d21665e894
[info] Successfully built 96d21665e894
[info] Successfully tagged weatherapp:0.1.0
[info] Built image weatherapp with tags [0.1.0]
[success] Total time: ...
```

```
example08 $ docker run weatherapp:0.1.0
Scala version 2.13.0:
The weather in Berlin is: light rain
```

# Archetypes

sbt-native-packager supports many more archetypes and formats.

- These examples used the *JavaAppPackaging* archetype.
- For Java server apps you shold use the *JavaServerAppPackaging* archetype.

# Archetypes

sbt-native-packager supports many more archetypes and formats.

- These examples used the *JavaAppPackaging* archetype.
- For Java server apps you shold use the *JavaServerAppPackaging* archetype.

# General idea

The general idea is "native packaging". I.e. create a Windows package on a Windows system, create a Debian package on a Debian based Linux system (Debian or Ubuntu), an RPM package on Redhat or SuSE Linux and a DMG package on a Mac.

# 14. Predefined Settings and Tasks

*.../example08*

*<https://www.scala-sbt.org/release/docs/Basic-Def.html>*

# Keys

There are three flavours of keys reflected in three different key types:

- **SettingKey[T]**: a key for a value computed once (the value is computed when loading the subproject and kept around until *reload*).
- **TaskKey[T]**: a key for a value, called a task, that has to be recomputed each time, potentially with side effects.
- **InputKey[T]**: a key for a task that has command line arguments as input. (We will ignore *InputKeys* here and explore them later.)

# Keys

There are three flavours of keys reflected in three different key types:

- **SettingKey[T]**: a key for a value computed once (the value is computed when loading the subproject and kept around until *reload*).
- **TaskKey[T]**: a key for a value, called a task, that has to be recomputed each time, potentially with side effects.
- **InputKey[T]**: a key for a task that has command line arguments as input. (We will ignore *InputKeys* here and explore them later.)

sbt provides a bunch of predefined keys (defined in *sbt.Keys*):

[https://www.scala-sbt.org/release/api/sbt/Keys\\$.html](https://www.scala-sbt.org/release/api/sbt/Keys$.html)

## Examples:

- *name* is a *SettingKey[String]*.
- *libraryDependencies* is a *SettingKey[Seq[String]]*.
- *compile* is a *TaskKey[CompileAnalysis]*.
- *test* is a *TaskKey[Unit]*.
- *packageBin* is a *TaskKey[File]*.

# Display predefined Settings

A setting can be displayed by just invoking the setting's name at the sbt prompt. (You can also precede the settings name with the **show** command.)

# Display predefined Settings

A setting can be displayed by just invoking the setting's name at the sbt prompt. (You can also precede the settings name with the **show** command.)

```
sbt:Example08> project lib  
[info] Set current project to WeatherLib (in build file:.../example08/)
```

```
sbt:WeatherLib> show name  
[info] WeatherLib
```

```
sbt:WeatherLib> libraryDependencies  
[info] * org.scala-lang:scala-library:2.13.0  
[info] * com.eed3si9n:gigahorse-okhttp:0.5.+  
[info] * com.typesafe.play:play-json:2.7.+  
[info] * org.scalatest:scalatest:3.0.+:test
```

# Manipulating the value of a setting

Values of settings are typically defined in the build file (*build.sbt*). They can also be set at the command prompt with the **set** command.

# Manipulating the value of a setting

Values of settings are typically defined in the build file (*build.sbt*). They can also be set at the command prompt with the **set** command.

```
sbt:WeatherLib> set libraryDependencies += "org.typelevel" %% "cats-core" % "1.6.1"
[info] Defining libraryDependencies
[info] The new value will be used by allDependencies, dependencyPositions, depende
[info] Reapplying settings...
[info] Set current project to WeatherLib (in build file:.../example08/)
```

```
sbt:WeatherLib> libraryDependencies
[info] * org.scala-lang:scala-library:2.13.0
[info] * com.eed3si9n:gigahorse-okhttp:0.5.+
[info] * com.typesafe.play:play-json:2.7.+
[info] * org.scalatest:scalatest:3.0.+:test
[info] * org.typelevel:cats-core:1.6.1
```

# List available settings

- **settings:** shows a list of the most common predefined settings
- **settings -v:** extends that list
- **settings -vv:** extends that list further
- **settings -vvv:** extends that list even more
- **settings -V:** displays all available settings

# Predefined Tasks

Tasks (like *compile*, *test* or *packageBin*) are mostly predefined in sbt. But they can be redefined in the build file (*build.sbt*).

A task contains executable code. To display its result you have to run it.

Just running a task does not show its result. To run it and to see the result invoke it with **show**.

# Predefined Tasks

Tasks (like *compile*, *test* or *packageBin*) are mostly predefined in sbt. But they can be redefined in the build file (*build.sbt*).

A task contains executable code. To display its result you have to run it.

Just running a task does not show its result. To run it and to see the result invoke it with **show**.

```
sbt:WeatherLib> compile  
[success] Total time: ...
```

```
sbt:WeatherLib> show compile  
[info] Analysis: 1 Scala source, 3 classes, 5 binary dependencies  
[success] Total time: ...
```

# List available tasks

- **tasks:** shows a list of the most common predefined tasks
- **tasks -v:** extends that list
- **tasks -vv:** extends that list further
- **tasks -vvv:** extends that list even more
- **tasks -V:** displays all available tasks

# 15. Custom Settings and Tasks

*.../example09*

*<https://www.scala-sbt.org/release/docs/Custom-Settings.html>*

# Custom Settings

Define your own setting with `settingKey[T]`.

```
val sampleStringSetting: SettingKey[String] = settingKey[String]("A sample string")
val sampleIntSetting: SettingKey[Int] = settingKey[Int]("A sample int setting.")

ThisBuild / organization := "com.example"
ThisBuild / version     := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.13.0"

lazy val root = (project in file("."))
  .settings(
    name := "Example9a",

    sampleStringSetting := {
      println(">>> Executing sampleStringSetting ...")
      System.getProperty("java.home")
    },

    sampleIntSetting := {
      println(">>> Executing sampleIntSetting ...")
      val sum = 1 + 2
      println("sum: " + sum)
      sum
    },
  )
```

```
example09aCustomSettings $ sbt
...
[success] Total time: ...
[info] Loading settings for project root from build.sbt ...
>>> Executing sampleStringSetting ...
>>> Executing sampleIntSetting ...
sum: 3
[info] Set current project to Example9a (in build file:.../ex09aCustomSettings/)
```

```
sbt:Example9a> sampleStringSetting
[info] /Library/Java/JavaVirtualMachines/jdk1.8.0_212.jdk/Contents/Home/jre
```

```
sbt:Example9a> sampleIntSetting
[info] 3
```

# Custom Tasks

Define your own task with `taskKey[T]`.

```
val sampleStringTask: TaskKey[String] = taskKey[String]("A sample string task.")
val sampleIntTask: TaskKey[Int] = taskKey[Int]("A sample int task.")

ThisBuild / organization := "com.example"
ThisBuild / version     := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.8"

lazy val root = (project in file("."))
  .settings(
    name := "Example9b",

    sampleStringTask := {
      println(">>> Executing sampleStringTask ...")
      System.getProperty("java.home")
    },

    sampleIntTask := {
      println(">>> Executing sampleIntTask ...")
      val sum = 1 + 2
      println("sum: " + sum)
      sum
    },
  )
```

```
example09bCustomTasks $ sbt  
...  
[success] Total time: ...  
[info] Loading settings for project root from build.sbt ...  
[info] Set current project to Example9b (in build file:.../example09bCustomTasks/)
```

```
sbt:Example9b> sampleStringTask  
>>> Executing sampleStringTask ...  
[success] Total time: ...
```

```
sbt:Example9b> show sampleStringTask  
>>> Executing sampleStringTask ...  
[info] /Library/Java/JavaVirtualMachines/jdk1.8.0_212.jdk/Contents/Home/jre  
[success] Total time: ...
```

```
sbt:Example9b> sampleIntTask  
>>> Executing sampleIntTask ...  
sum: 3  
[success] Total time: ...
```

```
sbt:Example9b> show sampleIntTask  
>>> Executing sampleIntTask ...  
sum: 3  
[info] 3  
[success] Total time: ...
```

# Execution semantics of tasks

If you do **not** invoke `.value` on another task all lines inside the task implementation will be invoked sequentially, e.g. in the `sampleIntTask` from the previous example.

```
sampleIntTask := {  
    println(">>> Executing sampleIntTask ...")  
    val sum = 1 + 2  
    println("sum: " + sum)  
    sum  
}
```

# Execution semantics of tasks

If you do **not** invoke `.value` on another task all lines inside the task implementation will be invoked sequentially, e.g. in the `sampleIntTask` from the previous example.

```
sampleIntTask := {  
    println(">>> Executing sampleIntTask ...")  
    val sum = 1 + 2  
    println("sum: " + sum)  
    sum  
}
```

This is no longer true if you invoke `.value` on another task.

- `.value` expresses a **task dependency**.
- `x := y.value` means: Task `x` depends on task (or setting) `y`.
- If Task `A` depends on the Tasks `B` and `C`, `B` and `C` are executed **in parallel**, but Task `A` is executed when `B` and `C` have finished.
- Dependent tasks are **deduplicated**.

# Rules

- A task may depend on a task or a setting.
- A setting may depend on another setting.
- A setting never depends on a task.

```
val startServer = taskKey[Unit]("start server")
val stopServer = taskKey[Unit]("stop server")
val sampleIntTask = taskKey[Int]("A sample int task.")
val sampleStringTask = taskKey[String]("A sample string task.")

...
lazy val root = (project in file("."))
  .settings(
    name := "Example9c",
    startServer := {
      println("starting...")
      Thread.sleep(500)
    },
    stopServer := {
      println("stopping...")
      Thread.sleep(500)
    },
    sampleIntTask := {
      startServer.value // This line is invoked BEFORE the other statements
      val sum = 1 + 2
      println("sum: " + sum)
      stopServer.value // !!! This line is invoked BEFORE the other statements
      sum
    },
    sampleStringTask := {
      startServer.value // This line is invoked BEFORE the other statements
      val s = sampleIntTask.value.toString // This line is invoked BEFORE the other
      println("s: " + s)
      s
    }
  )

```

```
example09cExecutionSemanticsOfTasks $ sbt  
...  
[info] Set current project to Example9c (in build file:.../example09cExecutionSema
```

```
sbt:Example9c> sampleIntTask  
stopping...  
starting...  
sum: 3  
[success] Total time: ...
```

```
sbt:Example9c> sampleStringTask  
stopping...  
starting...  
sum: 3  
s: 3  
[success] Total time: ...
```

```
example09cExecutionSemanticsOfTasks $ sbt  
...  
[info] Set current project to Example9c (in build file:.../example09cExecutionSema
```

```
sbt:Example9c> sampleIntTask  
stopping...  
starting...  
sum: 3  
[success] Total time: ...
```

```
sbt:Example9c> sampleStringTask  
stopping...  
starting...  
sum: 3  
s: 3  
[success] Total time: ...
```

- *startServer* and *stopServer* are invoked before any other statement of *sampleIntTask*, because *sampleIntTask* depends on those other tasks.
- *startServer* and *stopServer* are executed in parallel as these tasks do not depend on each other. Hence you might see the output of *startServer* before the output *stopServer* or vice versa.
- *sampleStringTask* depends 2 x on *startServer* (once directly and once via *simpleIntTask*). But *startServer* is executed only once. sbt deduplicates

# Visualize dependencies with *inspect*

```
sbt:Example9c> inspect sampleIntTask
[info] Task: Int
...
[info] Dependencies:
[info]   stopServer
[info]   startServer
...
```

```
sbt:Example9c> inspect sampleStringTask
[info] Task: java.lang.String
...
[info] Dependencies:
[info]   sampleIntTask
[info]   startServer
...
```

```
sbt:Example9c> inspect tree sampleStringTask
[info] sampleStringTask = Task[java.lang.String]
[info]   +-sampleIntTask = Task[Int]
[info]     | +-startServer = Task[Unit]
[info]     | +-stopServer = Task[Unit]
[info]     |
[info]     +-startServer = Task[Unit]
```

# Task Graph

All tasks of a build form a **task graph** that doesn't allow cycles - a directed acyclic graph (DAG) of tasks where the edges denote a **happens-before** relationship .

More examples illustrate this in:

<https://www.scala-sbt.org/release/docs/Task-Graph.html>

# 16. Global Settings

<https://www.scala-sbt.org/release/docs/Global-Settings.html>

## `build.sbt` (or any other file suffixed with `.sbt`)

- Local settings go into `./build.sbt` in the project's base directory.
- Global settings used in all projects are located in `~/.sbt/1.0/global.sbt`.
- `build.sbt` and `global.sbt` are just naming conventions. Any other name suffixed with `.sbt` is allowed.

## `build.sbt` (or any other file suffixed with `.sbt`)

- Local settings go into `./build.sbt` in the project's base directory.
- Global settings used in all projects are located in `~/.sbt/1.0/global.sbt`.
- `build.sbt` and `global.sbt` are just naming conventions. Any other name suffixed with `.sbt` is allowed.

## Plugins

- Local plugins are defined in `./project/plugins.sbt`.
- Global plugins are defined in `~/.sbt/1.0/plugins/plugins.sbt`.

## `build.sbt` (or any other file suffixed with `.sbt`)

- Local settings go into `./build.sbt` in the project's base directory.
- Global settings used in all projects are located in `~/.sbt/1.0/global.sbt`.
- `build.sbt` and `global.sbt` are just naming conventions. Any other name suffixed with `.sbt` is allowed.

## Plugins

- Local plugins are defined in `./project/plugins.sbt`.
- Global plugins are defined in `~/.sbt/1.0/plugins/plugins.sbt`.

## Startup commands (like alias definitions)

- Local startup commands go into `.sbtrc` in the project's base directory.
- Global startup commands go into `~/.sbtrc`.

# Unifying sbt history

- With the `!:` command you can display sbt's command history.
- By default every subproject (*root*, *app*, *lib*) has it's own history (saved in it's own history file).

# Unifying sbt history

- With the `!:` command you can display sbt's command history.
- By default every subproject (*root*, *app*, *lib*) has it's own history (saved in it's own history file).
- To unify the history of all files let **historyPath** point to the same file.
- To have this feature available in all projects we do this globally in ...

# Unifying sbt history

- With the `!:` command you can display sbt's command history.
  - By default every subproject (*root*, *app*, *lib*) has it's own history (saved in it's own history file).
- 
- To unify the history of all files let **historyPath** point to the same file.
  - To have this feature available in all projects we do this globally in ...

`~/.sbt/1.0/global.sbt`

```
historyPath := Some( (target in LocalRootProject).value / ".history")
cleanKeepFiles -= (target in LocalRootProject).value / ".history"
```

# 17. Ammonite REPL

*<http://ammonite.io/#SBTIntegration>*

Ammonite is a very powerful Scala REPL.

Let's make it available in sbt (like the Scala console) in all our projects...

Ammonite is a very powerful Scala REPL.

Let's make it available in sbt (like the Scala console) in all our projects...

`~/.sbt/1.0/global.sbt`

```
libraryDependencies += "com.lihaoyi" % "ammonite" % "1.6.9" % "test" cross CrossVersion.full

Test / sourceGenerators += Def.task {
  val file = (Test / sourceManaged).value / "amm.scala"
  IO.write(file, """object amm extends App { ammonite.Main.main(args) }""")
  Seq(file)
}.taskValue

addCommandAlias("amm", "Test/runMain amm")
```

Ammonite is a very powerful Scala REPL.

Let's make it available in sbt (like the Scala console) in all our projects...

`~/.sbt/1.0/global.sbt`

```
libraryDependencies += "com.lihaoyi" % "ammonite" % "1.6.9" % "test" cross CrossVersion.full

Test / sourceGenerators += Def.task {
  val file = (Test / sourceManaged).value / "amm.scala"
  IO.write(file, """object amm extends App { ammonite.Main.main(args) }""")
  Seq(file)
}.taskValue

addCommandAlias("amm", "Test/runMain amm")
```

```
sbt:Example08> amm
[info] Running amm
Loading...
Compiling (synthetic)/ammonite/predef/interpBridge.sc
Compiling (synthetic)/ammonite/predef/replBridge.sc
Compiling (synthetic)/ammonite/predef/sourceBridge.sc
Compiling (synthetic)/ammonite/predef/frontEndBridge.sc
Compiling (synthetic)/ammonite/predef/DefaultPredef.sc
Welcome to the Ammonite Repl 1.6.9
(Scala 2.13.0 Java 1.8.0_212)
If you like Ammonite, please support our development at www.patreon.com/lihaoyi
@ val x = 2 + 3
x: Int = 5
```

# 18. Input Tasks

*example10a, example10b*

*<https://www.scala-sbt.org/release/docs/Input-Tasks.html>*

# InputTask

Input Tasks take command line arguments as input. They parse user input and produce a task to run.

# InputTask

Input Tasks take command line arguments as input. They parse user input and produce a task to run.

## Overview of Keys

There are three flavours of keys reflected in three key types:

- **SettingKey[T]**: a key for a value computed once (the value is computed when loading the subproject and kept around until reload).
- **TaskKey[T]**: a key for a value, called a task, that has to be recomputed each time, potentially with side effects.
- **InputKey[T]**: a key for a task that has command line arguments as input.

```
// build.sbt

...
val demo = inputKey[Unit]("A demo input task.")

demo := {
    import complete.DefaultParsers._

    println(">>> Current Scala version:")
    println(scalaVersion.value) // access a setting

    val args: Seq[String] = spaceDelimited("<arg>").parsed // get parsed result
    println(">>> Arguments to demo:")
    args foreach println
}
```

```
// build.sbt

...
val demo = inputKey[Unit]("A demo input task.")

demo := {
    import complete.DefaultParsers._

    println(">>> Current Scala version:")
    println(scalaVersion.value) // access a setting

    val args: Seq[String] = spaceDelimited("<arg>").parsed // get parsed result
    println(">>> Arguments to demo:")
    args foreach println
}
```

```
sbt:Example10a> demo these are the args !
>>> Current Scala version:
2.12.8
>>> Arguments to demo:
these
are
the
args
!
```

# Shell commands implemented with InputTask

```
// build.sbt
...
lazy val shellTask = inputKey[Int]("Execute a shell command")

shellTask := {
    import complete.DefaultParsers._
    import scala.language.postfixOps
    import scala.sys.process._

    val args: Seq[String] = spaceDelimited("<arg>").parsed
    val commandLine = args.mkString(" ")
    Process(Seq("/bin/sh", "-c", commandLine)) !
}

addCommandAlias("sh", "shellTask")
```

# Shell commands implemented with InputTask

```
// build.sbt
...
lazy val shellTask = inputKey[Int]("Execute a shell command")

shellTask := {
    import complete.DefaultParsers._
    import scala.language.postfixOps
    import scala.sys.process._

    val args: Seq[String] = spaceDelimited("<arg>").parsed
    val commandLine = args.mkString(" ")
    Process(Seq("/bin/sh", "-c", commandLine)) !
}

addCommandAlias("sh", "shellTask")
```

```
sbt:Example10b> sh find . -name *.class | wc -l
14
```

# 19. Commands

*example11a, example11b*

*<https://www.scala-sbt.org/release/docs/Commands.html>*

# What is a “command”?

The docs say:

A “command” looks similar to a task: it’s a named operation that can be executed from the sbt console.

However, a command’s implementation takes as its parameter the entire state of the build (represented by *State*) and computes a new state. This means that a command can look at or modify other sbt settings, for example. Typically, you would resort to a command when you need to do something that’s impossible in a regular task.

# What is a “command”?

The docs say:

A “command” looks similar to a task: it’s a named operation that can be executed from the sbt console.

However, a command’s implementation takes as its parameter the entire state of the build (represented by *State*) and computes a new state. This means that a command can look at or modify other sbt settings, for example. Typically, you would resort to a command when you need to do something that’s impossible in a regular task.

The most general command takes ...

- a name: the name with which the command is to be invoked
- a parser: used to parse the user input after the command name
- an action to be executed which takes the current build state and turns it to the next state:  $(State, T) \Rightarrow State$

```
// project/DemoCommand.scala
import sbt._, Keys._

object DemoCommand {
    // A simple, multiple-argument command that prints "Hi, " followed
    // by the arguments. It leaves the current state unchanged.
    // The args are already parsed into a Seq[String].
    def demoCommand = Command.args("demo", "<args>") { (state, args) =>
        println("Hi, " + args.mkString(" "))
        state
    }
    // This is just one of several ways to construct a command.
}
```

```
// build.sbt
...
import DemoCommand._

lazy val root = (project in file("."))
  .settings(
    name := "Example11a",
    commands += demoCommand,
  )
```

```
// project/DemoCommand.scala
import sbt._, Keys._

object DemoCommand {
    // A simple, multiple-argument command that prints "Hi, " followed
    // by the arguments. It leaves the current state unchanged.
    // The args are already parsed into a Seq[String].
    def demoCommand = Command.args("demo", "<args>") { (state, args) =>
        println("Hi, " + args.mkString(" "))
        state
    }
    // This is just one of several ways to construct a command.
}
```

```
// build.sbt
...
import DemoCommand._

lazy val root = (project in file("."))
  .settings(
    name := "Example11a",
    commands += demoCommand,
  )
```

```
sbt:Example11a> demo here comes the sun!
Hi, here comes the sun!
```

# Shell command implemented as sbt command

```
// project/ShellCommand.scala
import sbt._
import scala.sys.process._
import scala.language.postfixOps

object ShellCommand {
  def shellCommand = Command.single("sh") { (state, commandLine) =>
    Process(Seq("/bin/sh", "-c", commandLine)) !;
    state
  }
}
```

```
// build.sbt
...
lazy val root = (project in file("."))
  .settings(
    name := "Example11b",
    commands += ShellCommand.shellCommand,
  )
```

# Shell command implemented as sbt command

```
// project/ShellCommand.scala
import sbt._
import scala.sys.process._
import scala.language.postfixOps

object ShellCommand {
  def shellCommand = Command.single("sh") { (state, commandLine) =>
    Process(Seq("/bin/sh", "-c", commandLine)) !;
    state
  }
}
```

```
// build.sbt
...
lazy val root = (project in file("."))
  .settings(
    name := "Example11b",
    commands += ShellCommand.shellCommand,
  )
```

```
sbt:Example11b> sh find . -name *.class | wc -l
14
```

The *DemoCommand* and the *ShellCommand* examples were quite simple.

*DemoCommand* created the command with the factory method *Command.args*, where the user input was already parsed into *Seq[String]* - a sequence of white space delimited words.

*ShellCommand* created the command with the factory method *Command.single*, where the user input was parsed into a *String* - i.e the user input was unchanged (which was sufficient for our shell use case).

In both cases the action function  $(State, T) \Rightarrow State$  passed the state back unchanged.

The *DemoCommand* and the *ShellCommand* examples were quite simple.

*DemoCommand* created the command with the factory method *Command.args*, where the user input was already parsed into *Seq[String]* - a sequence of white space delimited words.

*ShellCommand* created the command with the factory method *Command.single*, where the user input was parsed into a *String* - i.e the user input was unchanged (which was sufficient for our shell use case).

In both cases the action function  $(State, T) \Rightarrow State$  passed the state back unchanged.

For more complex parsing use cases, where parsing also supports tab completion, see:

<https://www.scala-sbt.org/release/docs/Parsing-Input.html>

For State changes inside the action function of a command see:

<https://www.scala-sbt.org/release/docs/Commands.html>

# Command vs. InputTask

- Both can parse user input (and support tab completion).
- A *Command* can change the build state, but *InputTask* cannot.
- If you don't need to change the build state, prefer *InputTask*.
- A description can be defined for settings, tasks and input tasks, but not for commands.
- In the next chapter we will implement a ShellPlugin, once with a *Command*, once with an *InputTask* and compare both solutions.

# 20. Plugin Development

*example12a, example12b, example12c, example12d, example12e*

*<https://www.scala-sbt.org/release/docs/Plugins.html>*

# AutoPlugin

An AutoPlugin defines a group of settings and the conditions where the settings are automatically added to a build (called "activation").

# AutoPlugin

An AutoPlugin defines a group of settings and the conditions where the settings are automatically added to a build (called "activation").

In your meta-project (i.e. in directory *project*) ...

- create a Scala object extending *sbt.AutoPlugin*
- override def *requires* = ...  
to define the plugins this plugin is depending on
- override def *trigger* = *noTrigger* | *allRequirements*  
to specify whether to enable the plugin manually or automatically
- override lazy val *buildSettings* = ...  
to change settings at the build level as needed (or use *projectSettings* or *globalSettings* to change settings at project or global level as needed)

# AutoPlugin

An AutoPlugin defines a group of settings and the conditions where the settings are automatically added to a build (called "activation").

In your meta-project (i.e. in directory *project*) ...

- create a Scala object extending *sbt.AutoPlugin*
- override def *requires* = ...  
to define the plugins this plugin is depending on
- override def *trigger* = *noTrigger* | *allRequirements*  
to specify whether to enable the plugin manually or automatically
- override lazy val *buildSettings* = ...  
to change settings at the build level as needed (or use *projectSettings* or *globalSettings* to change settings at project or global level as needed)

In *build.sbt*

- enable your plugin with *enablePlugins*, if *trigger* was *noTrigger*
- Nothing to do, if *trigger* was *allRequirements*

# Plugin enabled manually

```
// project/src/main/scala/shell/ShellCommand.scala
package shell

import sbt._, Keys._
import scala.language.postfixOps
import scala.sys.process._

object ShellPlugin extends AutoPlugin {

    override def requires: Plugins = empty // other plugins this plugin depends on
    override def trigger: PluginTrigger = noTrigger // enable manually in build.sbt
    override lazy val buildSettings = Seq(commands += shellCommand) // add build level settings

    lazy val shellCommand: Command = Command.single("sh") { (state, commandLine) =>
        Process(Seq("/bin/sh", "-c", commandLine)) !;
        state
    }
}
```

```
// build.sbt
...
lazy val root = (project in file("."))
    .enablePlugins(ShellPlugin)
    .settings(
        name := "Example12a",
    )
```

```
sbt:Example12a> sh find . -name *.class | wc -l  
14
```

```
sbt:Example12a> plugins  
In file:.../example12a/  
  sbt.plugins.IvyPlugin: enabled in root  
  sbt.plugins.JvmPlugin: enabled in root  
  sbt.plugins.CorePlugin: enabled in root  
  sbt.ScriptedPlugin  
  sbt.plugins.SbtPlugin  
  ...  
  shell.ShellPlugin: enabled in root
```

# Plugin enabled automatically

```
// project/src/main/scala/shell/ShellCommand.scala
package shell

import sbt._, Keys._
import scala.language.postfixOps
import scala.sys.process._

object ShellPlugin extends AutoPlugin {

    override def requires: Plugins = empty // other plugins this plugin depends on
    override def trigger: PluginTrigger = allRequirements // enable automatically
    override lazy val buildSettings = Seq(commands += shellCommand) // add build level settings

    lazy val shellCommand: Command = Command.single("sh") { (state, commandLine) =>
        Process(Seq("/bin/sh", "-c", commandLine)) !;
        state
    }
}
```

```
// build.sbt
...
lazy val root = (project in file("."))
    // no need to enable ShellPlugin
    .settings(
        name := "Example12b",
    )
```

```
sbt:Example12b> sh find . -name *.class | wc -l  
14
```

```
sbt:Example12b> plugins  
In file:.../example12b/  
  sbt.plugins.IvyPlugin: enabled in root  
  sbt.plugins.JvmPlugin: enabled in root  
  sbt.plugins.CorePlugin: enabled in root  
  sbt.ScriptedPlugin  
  sbt.plugins.SbtPlugin  
  ...  
  shell.ShellPlugin: enabled in root
```

# Settings and Tasks in Plugins

```
// project/HelloPlugin
import sbt._
import Keys._

object HelloPlugin extends AutoPlugin {

    object autoImport {
        val greeting = settingKey[String]("greeting")
        val hello = taskKey[Unit]("say hello")
    }

    import autoImport._
    override def trigger = allRequirements
    override lazy val buildSettings = Seq(greeting := "Hi!", hello := helloTask.value)
    lazy val helloTask = Def.task {
        println(greeting.value)
    }
}
```

```
// build.sbt - UNCHANGED!!! *HelloPlugin* is enabled automatically!
```

# Settings and Tasks in Plugins

```
// project/HelloPlugin
import sbt._
import Keys._

object HelloPlugin extends AutoPlugin {

    object autoImport {
        val greeting = settingKey[String]("greeting")
        val hello = taskKey[Unit]("say hello")
    }

    import autoImport._
    override def trigger = allRequirements
    override lazy val buildSettings = Seq(greeting := "Hi!", hello := helloTask.value)
    lazy val helloTask = Def.task {
        println(greeting.value)
    }
}
```

```
// build.sbt - UNCHANGED!!! *HelloPlugin* is enabled automatically!
```

```
sbt:Example12c> hello
Hi!
```

# ShellPlugin implemented with InputTask

```
// project/src/main/scala/shell/ShellPlugin
package shell

import sbt._, Keys._
import complete.DefaultParsers._
import scala.sys.process.Process
import scala.language.postfixOps

object ShellPlugin extends AutoPlugin {

    object autoImport {
        lazy val sh = inputKey[Int]("Execute a shell command")
    }

    import autoImport._

    override def requires: Plugins = empty // other plugins this plugin depends on
    override def trigger: PluginTrigger = allRequirements // enable automatically
    override lazy val buildSettings = Seq(sh := shellTask.evaluated)

    lazy val shellTask = Def.inputTask {

        val args: Seq[String] = spaceDelimited("<arg>").parsed
        val commandLine = args.mkString(" ")
        Process(Seq("/bin/sh", "-c", commandLine)) !
    }
}
```

```
// build.sbt - doesn't know about the automatically enabled plugins
...
lazy val root = (project in file("."))
  .settings(
    name := "Example12d",
  )
```

```
// build.sbt - doesn't know about the automatically enabled plugins
...
lazy val root = (project in file("."))
  .settings(
    name := "Example12d",
  )
```

```
sbt:Example12d> sh find . -name *.class | wc -l
15
```

# Publishing the ShellPlugin

Before we publish our new sbt-ShellPlugin to the local Ivy2 repository we have to set sensible values to the setting **name** and **organisation** in the meta-project.

- **reload plugins** brings us to the meta-project, i.e. the project defined in the *project* directory.
- **reload return** brings us back again.

```
sbt:Example12d> reload plugins  
...
```

```
sbt:project> organization  
[info] default
```

```
sbt:project> name  
[info] project
```

```
sbt:project> set organization := "com.example.myplugins"  
[info] Defining organization  
...  
[success] Total time: ...
```

```
sbt:project> set name := "sbt-myshell"  
[info] Defining name  
...  
[success] Total time: ...
```

```
sbt:sbt-myshell> organization  
[info] com.example.myplugins
```

```
sbt:sbt-myshell> name  
[info] sbt-myshell
```

```
sbt:sbt-myshell> publishLocal
...
[info]      published sbt-myshell to ~/.ivy2/local/com.example.myplugins/sbt-myshell
[info]      published ivy to ~/.ivy2/local/com.example.myplugins/sbt-myshell/scala_
[success] Total time: ...
```

```
sbt:sbt-myshell> publishLocal
...
[info]     published sbt-myshell to ~/.ivy2/local/com.example.myplugins/sbt-myshell
[info]     published ivy to ~/.ivy2/local/com.example.myplugins/sbt-myshell/scala_
[success] Total time: ...
```

Let's save these settings permanently.

- **session save** writes these settings permanently to *project/build.sbt*.

```
sbt:sbt-myshell> publishLocal
...
[info]     published sbt-myshell to ~/.ivy2/local/com.example.myplugins/sbt-myshell
[info]     published ivy to ~/.ivy2/local/com.example.myplugins/sbt-myshell/scala_
[success] Total time: ...
```

Let's save these settings permanently.

- **session save** writes these settings permanently to *project/build.sbt*.

```
sbt:sbt-myshell> session save
[info] Reapplying settings...
...
```

```
sbt:sbt-myshell> reload return
...
[info] Set current project to Example12d (in build file:.../example12d/)
```

```
sbt:Example12d> sh cat project/build.sbt
organization := "com.example.myplugins"

name := "sbt-myshell"
```

```
sbt:Example12d> sh tree ~/.ivy2/local/*
~/.ivy2/local/com.example.myplugins
└── sbt-myshell
    ├── scala_2.12
    │   └── sbt_1.0
    │       └── 0.1.0-SNAPSHOT
    │           ├── docs
    │           │   ├── sbt-myshell-javadoc.jar
    │           │   ├── sbt-myshell-javadoc.jar.md5
    │           │   └── sbt-myshell-javadoc.jar.sha1
    │           ├── ivys
    │           │   ├── ivy.xml
    │           │   ├── ivy.xml.md5
    │           │   └── ivy.xml.sha1
    │           ├── jars
    │           │   ├── sbt-myshell.jar
    │           │   ├── sbt-myshell.jar.md5
    │           │   └── sbt-myshell.jar.sha1
    │           ├── poms
    │           │   ├── sbt-myshell.pom
    │           │   ├── sbt-myshell.pom.md5
    │           │   └── sbt-myshell.pom.sha1
    │           └── srcts
    │               └── sbt-myshell-sources.jar
```

# Using the locally published Plugin

In any other sbt project we can use the plugin we just published in the previous section.

As we do with any other plugin, we import it into some sbt file in the meta-project, usually in *project/plugins.sbt*.

# Using the locally published Plugin

In any other sbt project we can use the plugin we just published in the previous section.

As we do with any other plugin, we import it into some sbt file in the meta-project, usually in *project/plugins.sbt*.

```
addSbtPlugin("com.example.myplugins" % "sbt-myshell" % "0.1.0-SNAPSHOT")
```

```
sbt:Example12e> plugins
In file:.../example12e/
  sbt.plugins.IvyPlugin: enabled in root
  sbt.plugins.JvmPlugin: enabled in root
  sbt.plugins.CorePlugin: enabled in root
  ...
  shell.ShellPlugin: enabled in root
```

```
sbt:Example12e> sh find . -name *.class | wc -l
17
```

```
sbt:Example12e> sh cat project/plugins.sbt
addSbtPlugin("com.example.myplugins" % "sbt-myshell" % "0.1.0-SNAPSHOT")
```

```
sbt:Example12e> inspect sh
[info] Input task: Int
[info] Description:
[info]   Execute a shell command
[info] Provided by:
[info]   {file:.../example12e/} / sh
[info] Defined at:
[info]   (shell.ShellPlugin.buildSettings) ShellPlugin.scala:39
...
```

# Publish Plugin to Bintray

To make the plugin available to the Scala community, publish it to Bintray.

# Publish Plugin to Bintray

To make the plugin availabe to the Scala community, publish it to Bintray.

First go to <https://bintray.com/signup/oss> to create an Open Source Distribution Bintray Account.

The detailed procedure how to publish is described in:

- <https://www.scala-sbt.org/release/docs/Bintray-For-Plugins.html>
- <https://www.scala-sbt.org/release/docs/Publishing.html>

# Publish Plugin to Bintray

To make the plugin available to the Scala community, publish it to Bintray.

First go to <https://bintray.com/signup/oss> to create an Open Source Distribution Bintray Account.

The detailed procedure how to publish is described in:

- <https://www.scala-sbt.org/release/docs/Bintray-For-Plugins.html>
- <https://www.scala-sbt.org/release/docs/Publishing.html>

Plugins best practices can be found here:

- <https://www.scala-sbt.org/release/docs/Plugins-Best-Practices.html>

# Publish Plugin to Bintray

To make the plugin available to the Scala community, publish it to Bintray.

First go to <https://bintray.com/signup/oss> to create an Open Source Distribution Bintray Account.

The detailed procedure how to publish is described in:

- <https://www.scala-sbt.org/release/docs/Bintray-For-Plugins.html>
- <https://www.scala-sbt.org/release/docs/Publishing.html>

Plugins best practices can be found here:

- <https://www.scala-sbt.org/release/docs/Plugins-Best-Practices.html>

We already mentioned one of them:

- Use settings and tasks. Avoid commands if possible.

# 21. References

# References

- Code and Slides of this Talk:  
<https://github.com/hermannhueck/pragmatic-sbt>
- sbt Reference Manual  
<https://www.scala-sbt.org/1.x/docs/index.html>
- sbt Native Packager Docs  
<https://www.scala-sbt.org/1.x/docs/index.html>
- gitter8 templates for sbt  
<https://github.com/foundweekends/giter8/wiki/giter8-templates>
- sbt Community Plugins  
<https://www.scala-sbt.org/release/docs/Community-Plugins.html>
- sbt core concepts  
Talk by Eugene Yokota at Scala Days Lausanne 2019  
<https://www.youtube.com/watch?v=-shamsTC7rQ>  
<https://portal.klewel.com/watch/webcast/scala-days-2019/talk/15/>
- sbt-native-packager - package all the things  
Talk by Nepomuk Seiler at Scala Days Berlin 2018

# Thank You

## Q & A

<https://github.com/hermannhueck/pragmatic-sbt>

