sbt Reference Manual

Contents

Preface

Guía de Inicio de sbt

sbt usa un número pequeño de conceptos para soportar *build definitions* (definiciones de construcción de software) flexibles y poderosas. No hay tantos conceptos, pero sbt no es exactamente como cualquier otro sistema de construcción de sofware y hay detalles con los que *usted tropezará* si no ha leído la documentación.

La Guía de inicio cubre los conceptos que usted necesita para crear y mantnere una *build definition*.

¡Es *altamente recomendable* leer la Guía de inicio!

Si usted tiene mucha prisa, los conceptos más importantes pueden encontrarse en sbt build definition, scopes, y más sobre settings. Pero no prometemos que sea una buena idea dejar de leer las otras páginas de la guía.

Es mejor leer en orden, ya que las páginas posteriores de la Guía de inicio utilizan conceptos introducidos con anterioridad.

Gracias por utilizar sbt *¡Diviértase!*

Instalando sbt

Para crear un proyecto de sbt, necesitará dar los siguientes pasos:

- Instalar sbt y crear un script para iniciarlo.
- Configurar un proyecto simple hola mundo.
- Crear un directorio llamado project con archivos de código fuente en él.
- Crear su *build definition* (definición de construcción del proyecto).
- Continuar con ejecución para aprender a ejecutar sbt.
- Enseguida continuar con .sbt build definition para aprender más sobre las *build definitions*.

Ultimately, the installation of sbt boils down to a launcher JAR and a shell script, but depending on your platform, we provide several ways to make the process less tedious. Head over to the installation steps for Mac, Windows, or Linux.

Tips y notas

Si tiene algún problema ejecutando sbt, vea Setup Notes en las codificaciones de la terminal, HTTP proxies, y opciones de la JVM.

Installing sbt on Mac

Installing from a third-party package

> **Note:** Los paquetes de terceros pueden no proporcionar la última versión disponible. Por favor asegúrese de reportar cualquier problema con estos paquetes a los mantenedores respectivos.

Homebrew

```
$ brew install sbt@1
```

Macports

```
$ port install sbt
```

Installing sbt on Windows

Windows installer

Download msi installer and install it.

Installing from a universal package

Download ZIP or TGZ package and expand it.

Installing manually

See instruction to install manually.

Installing sbt on Linux

Installing from a universal package

Download ZIP or TGZ package and expand it.

RPM and DEB

The following packages are also officially supported:

- RPM package
- DEB package

> **Note:** Por favor reporte cualquier problema que se tenga con los paquetes arriba mencionados al projecto sbt-launcher-package.

Gentoo

En el árbol oficial no hay ebuild para sbt. Pero existen ebuilds para hacer un *merge* de sbt a partir de los binarios. Para hacer un merge de sbt a partir de estos ebuilds, puede hacer lo siguiente:

```
$ mkdir -p /usr/local/portage && cd /usr/local/portage
$ git clone git://github.com/whiter4bbit/overlays.git
$ echo "PORTDIR_OVERLAY=$PORTDIR_OVERLAY /usr/local/portage/overlays" >> /etc/make.conf
$ emerge sbt-bin
```

> **Note:** Por favor reporte cualquier problema con el ebuild aquí.

Instalación manual

See instruction to install manually.

Installing sbt manually

La instalación manual requiere la descarga de sbt-launch.jar y la creación de un script para ejecutarlo.

Unix

Ponga sbt-launch.jar en `~/bin`.

Cree un script para ejecutar el jar, mediante la creación de `~/bin/sbt` con el siguiente contenido:

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled
java $SBT_OPTS -jar `dirname $0`/sbt-launch.jar "$@"
```

Haga el script ejecutable con:

```
$ chmod u+x ~/bin/sbt
```

Windows

La instalación manual para Windows varía según el tipo de terminal y dependiendo de si Cygwin es usado o no. En todos los casos, ponga el archivo batch o el script en el *path* de modo que pueda iniciar `sbt` en cualquier directorio mediante teclear `sbt` en la línea de comandos. También, ajuste los settings de la JVM de acuerdo con su máquina si es necesario.

Non-Cygwin Para usuarios que no utilizan Cygwin, pero que usan la terminal standard de Windows, cree un archivo batch `sbt.bat`:

```
set SCRIPT_DIR=%~dp0
java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -jar "%SCRIPT_DIR%sbt-launch.jar'
```

y ponga el sbt-launch.jar que descargó en el mismo directorio que archivo batch.

Cygwin con la terminal standard de Windows Si utiliza Cygwin con la terminal standard de Windows, cree un script de bash `~/bin/sbt`:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled
java $SBT_OPTS -jar sbt-launch.jar "$@"
```

Reemplace `sbt-launch.jar` con la ruta hasta el sbt-launch.jar que descargó y recuerde utilizar `cygpath` si es necesario. Haga el scrip ejecutable con:

```
$ chmod u+x ~/bin/sbt
```

Cygwin con una terminal Ansi Si utiliza Cygwin con una terminal Ansi (que soporte secuentas de escape Ansi y que sea configurable mediante `stty`), cree un script `~/bin/sbt`:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled
stty -icanon min 1 -echo > /dev/null 2>&1
java -Djline.terminal=jline.UnixTerminal -Dsbt.cygwin=true $SBT_OPTS -jar sbt-launch.jar "$@"
stty icanon echo > /dev/null 2>&1
```

Reemplace `sbt-launch.jar` con la ruta hasta el sbt-launch.jar que descargó y recuerde utilizar `cygpath` si es necesario. Entonces, haga que el script sea ejecutable con:

```
$ chmod u+x ~/bin/sbt
```

Para que la tecla *backspace* funcione correctamente en la consola de scala, necesita asegurarse de que dicha tecla esté enviando el caracter de borrado, de acuerdo a la configuración de `stty`. Para la terminal por default de cygwin (mintty) puede encontrar una configuración en Options -> Keys "Backspace sends ^H" que necesitará estar palomeada si su tecla de borrado envía el caracter por default de cygwin `^H`.

> **Note:** Otras configuraciones no están actualmente soportadas. Por favor envíe pull requests implementando o describiendo dicho soporte.

Hello, World

Esta página asume que usted ha instalado sbt.

Cree un directorio project con código fuente

Un proyecto válido de sbt puede ser un directorio que contenga un único archivo de código fuente. Intente crear un directorio `hello` con un archivo `hw.scala`, que contenga lo siguiente:

```
object Hola {
  def main(args: Array[String]) = println("¡Hola!")
}
```

Después, desde el directorio `hello`, inicie sbt y teclee `run` en la consola interactiva de sbt. En Linux u OS X los comandos tal vez se vean de la siguiente manera:

```
$ mkdir hello
$ cd hello
$ echo 'object Hola { def main(args: Array[String]) = println("¡Hola!") }' > hw.scala
$ sbt
...
> run
...
Hola!
```

En este caso, sbt funciona simplemente por convención. sbt encontrará lo siguiente de manera automática:

- Código fuente en el directorio base.
- Código fuente en `src/main/scala` o `src/main/java`.
- Pruebas en `src/test/scala` o `src/test/java`
- Archivos de datos en `src/main/resources` o `src/test/resources`
- jars en `lib`

Por default, sbt construirá proyectos con la misma versión de Scala utilizada para ejecutar sbt en sí mismo.

Usted puede ejecutar el proyecto con `sbt run` o ingresar a la REPL de Scala con `sbt console`. `sbt console` configura el classpath de su proyecto para que pueda probar ejemplos de Scala basados en el código de su proyecto.

Build definition (Definición de la construcción)

La mayoría de los proyectos necesitarán algo de configuración manual. La configuración básica de la construcción va en un archivo llamado `build.sbt`, localizado en el directorio base del proyecto.

Por ejemplo, si su proyecto está en el directorio `hello`, en `hello/build.sbt` usted puede escribir:

Note la línea en blanco entre cada ítem. Esto no es simplemente porque sí; se requieren las líneas en blanco para separar cada ítem. En .sbt build definition usted aprenderá más sobre cómo escribir un archivo `build.sbt`.

Si usted planea empaquetar su proyecto en un jar, tal vez desee configurar al menos el nombre y la versión en un archivo `build.sbt`.

Configurando la versión de sbt

Usted puede forzar una versión partivular de sbt al crear un archivo `hello/project/build.properties`. En este archivo, escriba:

```
sbt.version=1.1.5
```

para forzar el uso de sbt 1.1.5. sbt es 99% compatible (con respecto al código fuente) de una *release* a otra. Sin embargo, configurar la versión de sbt en `project/build.properties` evita cualquier confusión potencial.

Estructura de directorios

Esta página asume que usted ha instalado sbt y ha visto el ejemplo Hello, World.

Directorio base

En la terminología de sbt, el "directorio base" es el directorio que contiene al proyecto. De modo que si usted creó el proyecto `hello` que contiene `hello/build.sbt` y `hello/hw.scala` como se indicó en el ejemplo Hello, World, `hello` es su directorio base.

Código fuente

El código fu8ente puede ponerse en el directorio base del proyecto como en el caso de `hello/hw.scala`. Sin embargo, la mayoría de las personas no hacen esto para proyectos reales; se traduce en mucho desorden.

sbt utiliza la misma estructura de directorios que Maven para el código fuente por default (todos las rutas son relativas al directorio base):

```
src/
  main/
    resources/
       <archivos que se incluyen en el jar principal van aquí>
    scala/
       <código fuente de Scala de main>
    java/
       <código fuente de Java de main>
  test/
    resources
       <archivos que se incluyen en el jar de test van aquí>
    scala/
       <código fuente de Scala para test>
    java/
       <código fuente de Java para test>
```

Otros directorios en `src/` serán ignorados. Adicionalmente, todos los directorios ocultos serán ignorados.

Archivos de definición de la construcción de sbt (sbt build definition files)

Ya ha visto `build.sbt` en el directorio base del proyecto. Otros archivos sbt aparecen en el subdirectorio `project`.

El subdirectorio `project` puede contener archivos `.scala`, que se combinan con los archivos `.sbt` para formar la definición completa de la construcción.

Vea .scala build definition para más información.

```
build.sbt
project/
  Build.scala
```

Tal vez pueda ver archivos `.sbt` dentro de `project/` pero no son equivalentes a archivos `.sbt` en el directorio base del proyecto. La explicación de esto viene después, dado que necesitará algo de antecedentes primero.

Productos de la construcción

Los archivos generados (clases compiladas, paquetes en jars, archivos gestionados (*managed files*), caches, y documentación) será escrita al directorio `target` por default.

Configurando el sistema de control de versiones

Su archivo `.gitignore` (o el equivalente para otro sistema de control de versiones) debe contener:

```
target/
```

Note que el texto anterior tiene una `/` de forma deliberada (para que únicamente los directorios sean seleccionados) y de manera deliberada no tiene una `/` al inicio (para que el directorio `project/target/` también sea seleccionado, además de simplemente el directorio `target/`).

Ejecución

Esta página describe cómo utilizar `sbt` una vez que usted a configurado su proyecto. Se asume que usted ha instalado sbt y que ha creado un proyecto Hello, World u otro proyecto.

Modo interactivo

Ejecute sbt en el directorio de su proyecto sin argumentos:

```
$ sbt
```

Ejecutar sbt sin ningún argumento en la línea de comandos, inicia sbt en modo interactivo. El modo interactivo tiene una línea de comandos (¡con *tab completion* e historia!).

Por ejemplo, usted puede teclear `compile` en el prompt de sbt:

```
> compile
```

Para key:`compile` de nuevo, presione la tecla "arriba" y entonces enter.

Para ejecutar su programa nuevamente, teclee `run`.

Para dejar el modo interactivo, teclee `exit` o utilice Ctrl+D (Unix) o Ctrl+Z (Windows).

Modo Batch (por lotes)

También puede ejecutar sbt en modo batch, especificando una lista separada por espacios de comandos de sbt como argumentos. Para comandos de sbt que toman argumentos, pase el comando y los argumentos como uno solo a `sbt` mediante encerrarlos entre comillas. Por ejemplo:

```
$ sbt clean compile "testOnly TestA TestB"
```

En este ejemplo, la *key* `testOnly` tiene argumentos, `TestA` y `TestB`. Los comandos se ejecutarán en sequencia (`clean`, `compile`, y entonces `testOnly`).

Construcción y test continuos

Para acelerar el ciclo de edición-compilación-prueba, puede pedir a sbt que recompile automáticamente o que ejecute los tests siempre que se guarde un archivo de código fuente.

Puede conseguir que un comando se ejecute siempre que uno o más archivos de código fuente cambien al agregar como prefijo `~`. Por ejemplo, en modo interactivo, intente:

```
> ~ compile
```

Presione enter para dejar de observar sus cambios.

Usted puede usar el prefijo `~` ya sea en modo interactivo o en modo *batch*.

Vea Triggered Execution para más detalles.

Comandos comunes

Aquí encontrará algunos de los comandos de sbt más comunes. Para una lista más completa, vea Command Line Reference.

clean

Borra todos los archivos generados (en el directorio target).

compile

Compila los archivos de código fuente de main (en los directorios src/main/scala y src/main/java).

test

Compila y ejecuta todos los tests.

console

Inicia el interprete de Scala con un classpath que incluye el código fuente compilado y todas las dependencias. Para regresar a sbt, teclee :quit, Ctrl+D (Unix), o Ctrl+Z (Windows).

run <argument>*

Ejecuta la clase principal para el proyecto en la misma máquina virtual que sbt.

package

crea un archivo jar que contiene los archivos en src/main/resources y las clases compiladas de src/main/scala y src/main/java.

help <command>

Despliega ayuda detallada para el comando especificado. Si no se proporciona ningún comando, despliega una breve descripción de todos los comandos.

reload

Recarga la definición de la construcción (los archivos build.sbt, project/*.scala, project/*.sbt). Este comando es necario si cambia la definición de la construcción.

Tab completion

El modo interactivo tiene *tab completion*, incluyendo el caso cuando se tiene un prompt vacio. Una convención especial de sbt es que presionar tab una vez puede mostrar únicamente un subconjunto de *completions* más probables, mientras que presionarlo más veces muestra opciones más verbosas.

Comandos de historia

El modo interactivo recuerda la historia, incluso si usted sale de sbt y lo reinicia. La manera más simple de acceder a la historia es con la tecla "arriba". También se soportan los siguientes comandos:

!

Muestra la ayuda para los comandos de historia.

!!

Ejecuta el comando previo de nuevo.

!:

Muestra todos los comandos previos.

!:n

Muestra los n comandos previos.

!n

Ejecuta el comando con índice n, como se indica con el comando !:.

!-n

Ejecuta el comando n-th previo a este.

!cadena

Ejecuta el comando más reciente que comienza con 'cadena'.

!?cadena

Ejecuta el comando más reciente que contenga 'cadena'.

`.sbt` build definition

Esta página describe las *build definitions*, incluyendo algo de "teoría" y la sintaxis de `build.sbt`. Se asume que usted sabe como usar sbt y que ha leído las páginas previas en la Guía de inicio.

`.sbt` vs `.scala` Build Definition

Una *build definition* para sbt puede contener archivos con terminación `.sbt`, localizados en el directorio base de un proyecto, y archivos con extensión `.scala`, localizados en el subdirectorio `project/` del directorio base.

Esta página trata sobre los archivos `.sbt`, que son apropiados para la mayoría de los casos. Los archivos `.scala` se usan típicamente para compartir código entre archivos `.sbt` y para *build definitions* más complicadas.

Vea .scala build definition (más adelante en la *Guía de inicio*) para más información sobre los archivos `.scala`.

¿Qué es una *Build Definition*?

Después de examinar un proyecto y procesar los archivos para la definición de la construcción del proyecto, sbt termina con un mapa inmutable (un conjunto de pares llave-valor) describiendo la construcción.

Por ejemplo, una llave es `name` y se mapea a un valor de tipo cadena (`String`), el nombre de su proyecto.

Los archivos de definición de la construcción no afectan el mapa de sbt directamente.

En lugar de esto, la definición de la construcción crea una lista enorme de objectos con el tipo `Setting[T]` donde `T` es el tipo del valor en el mapa. Un `Setting` describe una *transformación del mapa*, tal como añadir un nuevo valor llave-valor o agregar a un valor existente. (En el espíritu de la programación funcional con estructuras de datos y valores inmutables, una transformación regresa un nuevo mapa – no se actualiza el viejo mapa en sí mismo).

En `build.sbt`, usted puede crear un `Setting[String]` para el nombre de su proyecto como se indica a continuación:

```
name := "hello"
```

Este `Setting[String]` transforma el mapa al añadir (o reemplazar) la llave `name`, dándole el valor

`"hello"`. El mapa transformado se convierte en el nuevo mapa de sbt.

Para crear el mapa, sbt primero ordena la lista de *settings* (configuraciones) de modo que todos los cambios al mismo se realicen juntos, y los valores que dependen de otras llaves se procesan después de las llaves de las que dependen. Entonces sbt visita la lista ordenada de `Settings`s y aplica cada uno al mapa a la vez.

Resumen: Una definición de construcción define una lista de `Setting[T]`, donde un `Setting[T]` es una transformación que afecta el mapa de pares de llaves-valores de sbt y `T` es el tipo de cada valor.

De qué manera `build.sbt` define la configuración

`build.sbt` define una `Seq[Setting[_]]`; se trata de una lista de expresiones de Scala, separada por líneas en blanco, donde cada una se convierte en un elemento de la secuencia. Si usted colocara `Seq(` antes del contenido de un archivo `.sbt` y `)` al final y reemplazara las líneas blancas con comas, entonces estaría observando el código `.scala` equivalente.

A continuación se muestra un ejemplo:

```
name := "hello"

version := "1.0"

scalaVersion := "2.10.3"
```

Cada `Setting` se define con una expresión de Scala. Las expresiones en `build.sbt` son independientes la una de la otra, y son expresiones, más bien que sentencias completas de Scala. Estas expresiones pueden estar entremezcladas con `val`s, `lazy val`s, y `def`s. No se permiten `object`s ni `class`es en `build.sbt`. Estos deben ir en el directorio `project/` como archivos de código fuente completos.

Por la izquierda, `name`, `version`, y `scalaVersion` son *keys* (llaves). Una *key* es una instancia de `SettingKey[T]`, `TaskKey[T]`, o `InputKey[T]` donde `T` es el valor esperado para el tipo. La clase de *keys* se explican abajo.

Las *keys* tienen un método llamado `:=`, que regresa un `Setting[T]`. Usted podría usar una sintáxis similar a la de Java para invocar al método:

```
name.:=("hello")
```

Pero Scala permite usar `name := "hello"` en lugar de lo anterior (en Scala, un método con un único parámetro puede utilizar cualquiera de las dos sintaxis).

El método `:=` en la *key* `name` regresa un `Setting`, específicamente un `Setting[String]`. `String` también aparece en el tipo de `name` en sí misma, el cuál es `SettingKey[String]`. En este caso, el valor `Setting[String]` regresado es una transformación para agregar o reemplazar la *key* `name` en el mapa de sbt, dándole el valor `"hello"`.

Si usted usa el tipo de valor equivocado, la definición de la construcción no compilará:

```
name := 42  // no compila
```

Las *settings* (configuraciones) deben estar separadas por líneas en blanco

No es posible escribir un `build.sbt` como el siguiente:

```
// NO compila, pues no hay líneas en blanco
name := "hello"
version := "1.0"
scalaVersion := "2.10.3"
```

sbt necesita un tipo de delimitador para indicar donde termina una expresión y comienza la siguiente.

Los archivos `.sbt` contienen una lista de expresiones de Scala, no un único programa de Scala. Estas expresiones tienen que separarse y pasarse al compilador de manera individual.

Keys

Tipos Existen tres tipos de llaves:

- `SettingKey[T]`: una *key* para un valor que se calcula una sola vez (el valor es calculado cuando se carga el proyecto, y se mantiene).
- `TaskKey[T]`: una *key* para un valor, llamado una *task* (tarea), que tiene que ser recalculada cada vez, potencialmente con efectos laterales.
- `InputKey[T]`: una *key* para una *task* que tiene argumentos para la línea de comandos como entrada. Vea /Extending/Input-Tasks para más detalles.

Built-in Keys (Llaves ya incluídas) Las llaves ya incluídas son simplemente campos de un objeto llamado Keys. Un archivo `build.sbt` tiene implícitamente un `import sbt.Keys._`, de modo que `sbt.Keys.name` puede ser referido como `name`.

Custom Keys (llaves personalizadas) Las llaves personalizadas pueden definirse con sus métodos de creación respectivos: `settingKey`, `taskKey`, e `inputKey`. Cada método espera el tipo del valor asociado con la llave así como una descripción. El nombre de la llave se toma del `val` al que se le asignó la llave. Por ejemplo, para definir una llave para una nueva tarea llamado `hello`, :

```
lazy val hello = taskKey[Unit]("An example task")
```

Aquí se usó el hecho de que un archivo `.sbt` puede contener `val`s y `def`s además de *settings* (configuraciones). Todas estas definiciones son evaluadas antes que las configuraciones sin importar donde se definan en el archivo. `val`s y `def`s deben estar separadas de las *settings* mediante líneas blancas.

> **Note**: Típicamente, se utilizan lazy vals en lugar de vals para evitar problemas de inicialización.

Task vs. Setting keys (Llaves para *Tasks* vs. Llaves para *Settings*) Se dice que una `TaskKey[T]` define una *task*. Las *tasks* son operaciones tales como `compile` o `package`. Pueden regresar `Unit` (`Unit` es el tipo de Scala análogo a `void`), o pueden regresar un valor relacionado con la tarea, por ejemplo, `package` es una `TaskKey[File]` y su valor es el archivo jar que este crea.

Cada vez que inicia una tarea de ejecución, por ejemplo mediante teclear `compile` en el prompt interactivo de sbt, sbt volverá a ejecutar cualquier *task* envuelta exactamente una vez.

El mapa de sbt que describe el proyecto puede mantener una cadena fija para un *setting* tal como `name`, pero tiene que haber algo de código ejecutable para una tarea como `compile` – incluso si dicho código ejecutable eventualmente regresa una cadena, tiene que ejecutarse cada vez.

Una key dada siempre se refiere ya sea a una task o a un setting. Es decir, "taskiness" (si debe ejecutarse cada vez) es una propiedad de la *key*, no del valor.

Definiendo tasks y settings

Usando `:=`, usted puede asignar un valor a un *setting* y un cómputo a una *task*. En el caso de un *setting*, el valor será calculado una sola vez al momento de cargar el proyecto. Para una tarea, el cómputo se realizará cada vez que se ejecute la tarea.

Por ejemplo, para implementar la tarea `hello` de la sección anterior, :

```
hello := { println("Hello!") }
```

Ya vimos un ejemplo de definición de un *setting* para el nombre del proyecto, :

```
name := "hello"
```

Tipos para las tareas y los settings Desde la perspectiva del sistema de tipos, el `Setting` creado a partir de una *task key* es ligeramente distinta de la creada a partir de una *setting key*. `taskKey := 42` resulta en una `Setting[Task[T]]` mientras que `settingKey := 42` resulta en una `Setting[T]`. Para la mayoría de los propósitos no hay diferencia, la *task key* todavía crea un valor de tipo `T` cuando la tarea se ejecuta.

La diferencia entre los tipos `T` y `Task[T]` tiene la siguiente implicación: un *setting* no puede depender de una *task*, poque un *setting* es evaluado únicamente una vez al momento de cargar el proyecto y no se vuelve a ejecutar. Se escribirá más sobre este asunto pronto en more kinds of setting.

Keys en modo sbt interactivo

En el modo interactivo de sbt, usted puede teclear el nombre de cualquier tarea para ejecutar dicha tarea. Es por esto que al teclear `compile` se ejecuta la *task* de compilación. La *key* `compile` es una llave para una *task*.

Si usted teclea el nombre de una *key* para *setting* más bien que una para *task*, entonces el valor de la *key* para *setting* será mostrado. Al teclear el nombre de una *task* se ejecuta dicha *task*, pero no se despliega el valor resultante; para

ver el resultado de la *task*, use `show <nombre de la tarea>` más bien que simplemente `<nombre de la tarea`. La convención para los nombres de las llaves es usar `estiloDeCamello` de modo que el nombre utilizado en la línea de comandos y el identificador de Scala sean idénticos.

Para aprender más sobre cualquier *key*, teclee `inspect <nombre de la key>` en el prompt interactivo de sbt. Algo de la información que `inspect` despliega no tendrá sentido todavía, pero en la parte superior le mostrará el tipo del valor para el *setting* y una breve descripción del tal.

Imports en `build.sbt`

Puede poner sentencias import en la parte superior de `build.sbt`; no necesitan estar separadas por líneas en blanco.

Hay algunos imports por default, como se indica a continuación:

```
import sbt._
import Keys._
```

(Además, si usted tiene archivos .scala, el contenido de cualquier objeto `Build` o `Plugin` en estos archivos será importado. Más sobre este asunto cuando se llegue a definiciones de construccion .scala.)

Añadiendo dependencias (librerías)

Para agregar dependencias de librerías de terceros, hay dos opciones. La primera es añadir jars en el directorio `lib/` (para *unmanaged dependencies*) y la otra es agregar *managed dependencies*, que se verán como se muestra a continuación en `build.sbt`:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

Así es como se agrega una *managed dependency* sobre la librería Apache Derby, versión 10.4.1.3.

La key `libraryDependencies` envuelve dos complejidades: `+=` más bien que `:=`, y el método `%`. `+=` agrega algo al valor anterior de la *key* más bien que reemplazarlo; esto se explica en más sobre los settings. El método `%` se usa para construir un ID para un módulo de Ivy a partir de cadenas, como se explica en library dependencies.

Por lo pronto, omitiremos los detalles del manejo de las dependencias (librerías) hasta más tarde en la Guía de inicio. Hay una página completa que cubre el tema más tarde.

Scopes

Esta página describe los *scopes*. Se asume que usted ha leído y comprendido la página previa, .sbt build definition.

La historia completa sobre las *keys*

Previamente supusimos que una *key* como `name` correspondía a una entrada en el mapa de sbt de pares llave-valor (key-value). Esto fue una simplificación.

En verdad, cada llave puede tener un valor asociado en más de un contexto, llamado un "scope".

Algunos ejemplos concretos:

- Si usted tiene múltiples proyectos en la definición de la construcción, una *key* puede tener un valor diferente en cada proyecto.
- La *key* `compile` puede tener un valor diferente para sus archivos de código fuente de main comparado con el correspondiente valor para el código fuente de test, si usted desea que se compilen de manera distinta.
- La *key* `packageOpitons` (que contiene opciones para crear paquetes jar) puede tener diferentes valores para el empaquetado de archivos class (packageBin) o para el empaquetado de código fuente (packageSrc).

No hay un único valor para una key dada, porque el valor puede variar de acuerdo con el *scope*.

Sin embargo, existe un único valor para una *scoped key* (llaves con un contexto).

Si usted se imagina que sbt está procesando una lista de *settings* para generar un mapa de llave-valor (*key-value*) que describe al proyecto, como se discutió anteriormente, las *keys* en dicho mapa son *scoped keys*. Cada *setting* definido en la definición de la construcción del proyecto (por ejemplo en `build.sbt`) aplica a una *scoped key* también.

Con frecuencia el *scope* es implícito o tiene un valor por default, pero si dichos valores son incorrectos, entonces tendrá que indicar el *scope* deseado en `build.sbt`.

Ejes del Scope

Un *eje del scope* es un tipo, donde cada instancia del tipo puede definir su propio *scope* (esto es, cada instancia puede tener sus propios valores únicos para las *keys*).

Hay tres ejes del scope:

- Projects
- Configurations
- Tasks

Scoping mediante del eje del proyecto

Si usted coloca múltiples proyectos en una construcción única, cada proyecto necesita sus propios *settings*. Es decir, las *keys* pueden estar en *scope* de acuerdo al proyecto.

Los ejes del proyecto también pueden configurarse para la "entera construcción", de modo que un *setting* aplique a la construcción completa más bien que a un solo proyecto. Los *settings* de *nivel de construcción* con frecuencia se usan como un plan de reserva cuando un proyecto no define un *setting* específico para un proyecto.

Scoping mendiante el eje de configuración Una configuración define el tipo de construcción, potencialmente con su propio classpath, código fuente, paquetes generados, etc. El concepto de configuración viene de Iviy, que sbt usa para managed dependencies, y para MavenScopes.

Algunas configuraciones que verá en sbt:

- `Compile` que define la construcción principal (*main*) (`src/main/scala`).
- `Test` que define cómo construir tests (`src/test/scala`).
- `Runtime` que define el classpath para la *task* `run`.

Por default, todas las llaves asociadas con la compilación, empaquetamiento y la ejecución tienen un scope de configuración y por lo tanto pueden funcionar de manera diferente en cada configuración. Los ejemplos más obvios son las *keys* para *tasks* `compile`, `package`, y `run`; pero todas las llaves que *afectan* dichas *keys* (tales como `sourceDirectories` o `scalacOptions` o `fullClasspath`) también tienen scope de configuración.

Scoping mediante el eje task Los *settings* pueden afectar cómo funcionan las *tasks*. Por ejemplo, la *key* de *setting* `packageOptions` afecta a la *key* `packageSrc` de *task*.

Para soportar esto, una *key de task* (tal como `packageSrc`) puede ser el scopde para otra *key* (tal como `packageOptions`).

Las diferentes *tasks* que construyen un paquete (`packageSrc`, `packageBin`, `packageDoc`) pueden compartir *keys* relacionadas al empaquetamiento, tales como `artifactName` y `packageOptions`. Dichas *keys* pueden tener distintos valores para cada *task* de empaquetamiento.

Scope global

Cada eje de scope puede llenarse con una instancia del tipo de eje (por ejemplo el eje de *task* puede llevarse con una *task*), o el eje puede llenarse con el valor especial `Global`.

`Global` significa lo que usted espera: el valor del *setting* aplica a todas las instancias de ese eje. Por ejemplo, si el eje de la *task* es `Global`, entonces dicho *setting* aplicaría a todas las *tasks*.

Delegation

A scoped key may be undefined, if it has no value associated with it in its scope.

For each scope, sbt has a fallback search path made up of other scopes. Typically, if a key has no associated value in a more-specific scope, sbt will try to get a value from a more general scope, such as the `Global` scope or the entire-build scope.

This feature allows you to set a value once in a more general scope, allowing multiple more-specific scopes to inherit the value.

You can see the fallback search path or "delegates" for a key using the `inspect` command, as described below. Read on.

Referring to scoped keys when running sbt

On the command line and in interactive mode, sbt displays (and parses) scoped keys like this:

```
{<build-uri>}<project-id>/config:intask::key
```

- `{<build-uri>}/<project-id>` identifies the project axis. The `<project-id>` part will be missing if the project axis has "entire build" scope.
- `config` identifies the configuration axis.
- `intask` identifies the task axis.
- `key` identifies the key being scoped.

`*` can appear for each axis, referring to the `Global` scope.

If you omit part of the scoped key, it will be inferred as follows:

- the current project will be used if you omit the project.
- a key-dependent configuration will be auto-detected if you omit the configuration or task.

For more details, see Interacting with the Configuration System.

Examples of scoped key notation

- `fullClasspath` specifies just a key, so the default scopes are used: current project, a key-dependent configuration, and global task scope.
- `test:fullClasspath` specifies the configuration, so this is `fullClasspath` in the `test` configuration, with defaults for the other two scope axes.
- `*:fullClasspath` specifies `Global` for the configuration, rather than the default configuration.
- `doc::fullClasspath` specifies the `fullClasspath` key scoped to the `doc` task, with the defaults for the project and configuration axes.
- `{file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath` specifies a project, `{file:/home/hp/checkout/hello/}default-aea33a`, where the project is identified with the build `{file:/home/hp/checkout/hello/}` and then a project id inside that build `default-aea33a`. Also specifies configuration `test`, but leaves the default task axis.
- `{file:/home/hp/checkout/hello/}/test:fullClasspath` sets the project axis to "entire build" where the build is `{file:/home/hp/checkout/hello/}`.
- `{.}/test:fullClasspath` sets the project axis to "entire build" where the build is `{.}`. `{.}` can be written `ThisBuild` in Scala code.
- `{file:/home/hp/checkout/hello/}/compile:doc::fullClasspath` sets all three scope axes.

Inspecting scopes

In sbt's interactive mode, you can use the inspect command to understand keys and their scopes. Try `inspect test:fullClasspath`:

```
$ sbt
> inspect test:fullClasspath
[info] Task: scala.collection.Seq[sbt.Attributed[java.io.File]]
[info] Description:
[info]  The exported classpath, consisting of build products and unmanaged and managed, internal
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath
[info] Dependencies:
[info]  test:exportedProducts
[info]  test:dependencyClasspath
[info] Reverse dependencies:
[info]  test:runMain
[info]  test:run
[info]  test:testLoader
[info]  test:console
[info] Delegates:
[info]  test:fullClasspath
```

```
[info]  runtime:fullClasspath
[info]  compile:fullClasspath
[info]  *:fullClasspath
[info]  {.}/test:fullClasspath
[info]  {.}/runtime:fullClasspath
[info]  {.}/compile:fullClasspath
[info]  {.}/*:fullClasspath
[info]  */test:fullClasspath
[info]  */runtime:fullClasspath
[info]  */compile:fullClasspath
[info]  */*:fullClasspath
[info] Related:
[info]  compile:fullClasspath
[info]  compile:fullClasspath(for doc)
[info]  test:fullClasspath(for doc)
[info]  runtime:fullClasspath
```

On the first line, you can see this is a task (as opposed to a setting, as explained in .sbt build definition). The value resulting from the task will have type `scala.collection.Seq[sbt.Attributed[java.io.File]]`.

"Provided by" points you to the scoped key that defines the value, in this case `{file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath` (which is the `fullClasspath` key scoped to the `test` configuration and the `{file:/home/hp/checkout/hello/}default-aea33a` project).

"Dependencies" may not make sense yet; stay tuned for the next page.

You can also see the delegates; if the value were not defined, sbt would search through:

- two other configurations (`runtime:fullClasspath`, `compile:fullClasspath`). In these scoped keys, the project is unspecified meaning "current project" and the task is unspecified meaning `Global`
- configuration set to `Global` (`*:fullClasspath`), since project is still unspecified it's "current project" and task is still unspecified so `Global`
- project set to `{.}` or `ThisBuild` (meaning the entire build, no specific project)
- project axis set to `Global` (`*/test:fullClasspath`) (remember, an unspecified project means current, so searching `Global` here is new; i.e. `*` and "no project shown" are different for the project axis; i.e. `*/test:fullClasspath` is not the same as `test:fullClasspath`)
- both project and configuration set to `Global` (`*/*:fullClasspath`) (remember that unspecified task means `Global` already, so `*/*:fullClasspath` uses `Global` for all three axes)

Try `inspect fullClasspath` (as opposed to the above example, inspect `test:fullClasspath`) to get a sense of the difference. Because the configuration is omitted, it is autodetected as `compile`. `inspect compile:fullClasspath` should therefore look the same as `inspect fullClasspath`.

Try `inspect *:fullClasspath` for another contrast. `fullClasspath` is not defined in the `Global` configuration by default.

Again, for more details, see Interacting with the Configuration System.

Referring to scopes in a build definition

If you create a setting in `build.sbt` with a bare key, it will be scoped to the current project, configuration `Global` and task `Global`:

```
name := "hello"
```

Run sbt and `inspect name` to see that it's provided by `{file:/home/hp/checkout/hello/}default-aea33a/*` that is, the project is `{file:/home/hp/checkout/hello/}default-aea33a`, the configuration is `*` (meaning global), and the task is not shown (which also means global).

`build.sbt` always defines settings for a single project, so the "current project" is the project you're defining in that particular `build.sbt`. (For multi-project builds, each project has its own `build.sbt`.)

Keys have an overloaded method called in used to set the scope. The argument to in can be an instance of any of the scope axes. So for example, though there's no real reason to do this, you could set the `name` scoped to the `Compile` configuration:

```
name in Compile := "hello"
```

or you could set the name scoped to the `packageBin` task (pointless! just an example):

```
name in packageBin := "hello"
```

or you could set the `name` with multiple scope axes, for example in the `packageBin` task in the `Compile` configuration:

```
name in (Compile, packageBin) := "hello"
```

or you could use `Global` for all axes:

```
name in Global := "hello"
```

(`name in Global` implicitly converts the scope axis `Global` to a scope with all axes set to `Global`; the task and configuration are already `Global` by default, so here the effect is to make the project `Global`, that is, define `*/*:name` rather than `{file:/home/hp/checkout/hello/}default-aea33a/*:name`)

If you aren't used to Scala, a reminder: it's important to understand that in and `:=` are just methods, not magic. Scala lets you write them in a nicer way, but you could also use the Java style:

```
name.in(Compile).:=("hello")
```

There's no reason to use this ugly syntax, but it illustrates that these are in fact methods.

When to specify a scope

You need to specify the scope if the key in question is normally scoped. For example, the `compile` task, by default, is scoped to `Compile` and `Test` configurations, and does not exist outside of those scopes.

To change the value associated with the `compile` key, you need to write `compile in Compile` or `compile in Test`. Using plain `compile` would define a new compile task scoped to the current project, rather than overriding the standard compile tasks which are scoped to a configuration.

If you get an error like *"Reference to undefined setting"*, often you've failed to specify a scope, or you've specified the wrong scope. The key you're using may be defined in some other scope. sbt will try to suggest what you meant as part of the error message; look for "Did you mean compile:compile?"

One way to think of it is that a name is only *part* of a key. In reality, all keys consist of both a name, and a scope (where the scope has three axes). The entire expression `packageOptions in (Compile, packageBin)` is a key name, in other words. Simply `packageOptions` is also a key name, but a different one (for keys with no in, a scope is implicitly assumed: current project, global config, global task).