Getting Started with sbt

Contents

Preface

sbt uses a small number of concepts to support flexible and powerful build definitions. There are not that many concepts, but sbt is not exactly like other build systems and there are details you *will* stumble on if you haven't read the documentation.

The Getting Started Guide covers the concepts you need to know to create and maintain an sbt build definition.

It is *highly recommended* to read the Getting Started Guide!

If you are in a huge hurry, the most important conceptual background can be found in .sbt build definition, scopes, and more kinds of setting. But we don't promise that it's a good idea to skip the other pages in the guide.

It's best to read in order, as later pages in the Getting Started Guide build on concepts introduced earlier.

Thanks for trying out sbt and *have fun*!

Installing sbt

To create an sbt project, you'll need to take these steps:

- Install sbt and create a script to launch it.
- Setup a simple hello world project
  - Create a project directory with source files in it.
  - Create your build definition.
- Move on to running to learn how to run sbt.
- Then move on to .sbt build definition to learn more about 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, Linux, or manual installation.

Tips and Notes

If you have any trouble running sbt, see Setup Notes on terminal encodings, HTTP proxies, and JVM options.

Installing sbt on Mac

Installing from a third-party package

> **Note:** Third-party packages may not provide the latest version. Please make sure to report any issues with these packages to the relevant maintainers.

Macports

```
$ port install sbt
```

Homebrew

```
$ brew install sbt
```

Installing from a universal package

Download ZIP or TGZ package, and expand it.

Installaing manually

See instruction to install manually.

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.

Installaing 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:** Please report any issues with these to the sbt-launcher-package project.

Gentoo

In the official tree there is no ebuild for sbt. But there are ebuilds to merge sbt from binaries. To merge sbt from this ebuilds you can do:

```
$ 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:** Please report any issues with the ebuild here.

Installaing manually

See instruction to install manually.

Installing sbt manually

Manual installation requires downloading sbt-launch.jar and creating a script to start it.

Unix

Put sbt-launch.jar in `~/bin`.

Create a script to run the jar, by creating `~/bin/sbt` with these contents:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
java $SBT_OPTS -jar `dirname $0`/sbt-launch.jar "$@"
```

Make the script executable:

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

Windows

Manual installation for Windows varies by terminal type and whether Cygwin is used. In all cases, put the batch file or script on the path so that you can launch sbt in any directory by typing `sbt` at the command prompt. Also, adjust JVM settings according to your machine if necessary.

Non-Cygwin For non-Cygwin users using the standard Windows terminal, create a batch file `sbt.bat`:

```
set SCRIPT_DIR=%~dp0
java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M -jar "%SCRIP
```

and put the downloaded sbt-launch.jar in the same directory as the batch file.

Cygwin with the standard Windows termnial If using Cygwin with the standard Windows terminal, create a bash script `~/bin/sbt`:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
java $SBT_OPTS -jar sbt-launch.jar "$@"
```

Replace sbt-launch.jar with the path to your downloaded sbt-launch.jar and remember to use cygpath if necessary. Make the script executable:

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

Cygwin with an Ansi terminal Cygwin with an Ansi terminal (supports Ansi escape sequences and is configurable via stty), create a bash script `~/bin/sbt`:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
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
```

Replace sbt-launch.jar with the path to your downloaded sbt-launch.jar and remember to use cygpath if necessary. Then, make the script executable:

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

In order for backspace to work correctly in the scala console, you need to make sure your backspace key is sending the erase character as configured by stty. For the default cygwin terminal (mintty) you can find a setting under Options -> Keys “Backspace sends ˆH” which will need to be checked if your erase key is the cygwin default of ˆH.

> **Note:** Other configurations are currently unsupported. Please submit a pull request implementing or describing that support.

Hello, World

This page assumes you’ve installed sbt.

Create a project directory with source code

A valid sbt project can be a directory containing a single source file. Try creating a directory `hello` with a file `hw.scala`, containing the following:

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

Now from inside the `hello` directory, start sbt and type `run` at the sbt interactive console. On Linux or OS X the commands might look like this:

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

```
$ sbt
...
> run
...
Hi!
```

In this case, sbt works purely by convention. sbt will find the following automatically:

- Sources in the base directory
- Sources in `src/main/scala` or `src/main/java`
- Tests in `src/test/scala` or `src/test/java`
- Data files in `src/main/resources` or `src/test/resources`
- jars in `lib`

By default, sbt will build projects with the same version of Scala used to run sbt itself.

You can run the project with `sbt run` or enter the Scala REPL with `sbt console`. `sbt console` sets up your project's classpath so you can try out live Scala examples based on your project's code.

Build definition

Most projects will need some manual setup. Basic build settings go in a file called `build.sbt`, located in the project's base directory.

For example, if your project is in the directory `hello`, in `hello/build.sbt` you might write:

```
name := "hello"

version := "1.0"

scalaVersion := "2.10.3"
```

Notice the blank line between every item. This isn't just for show; they're actually required in order to separate each item. In .sbt build definition you'll learn more about how to write a `build.sbt` file.

If you plan to package your project in a jar, you will want to set at least the name and version in a `build.sbt`.

Setting the sbt version

You can force a particular version of sbt by creating a file `hello/project/build.properties`. In this file, write:

```
sbt.version=0.13.5
```

to force the use of sbt 0.13.5. sbt is 99% source compatible from release to release. Still, setting the sbt version in `project/build.properties` avoids any potential confusion.

Directory structure

This page assumes you've installed sbt and seen the Hello, World example.

Base directory

In sbt's terminology, the "base directory" is the directory containing the project. So if you created a project `hello` containing `hello/build.sbt` and `hello/hw.scala` as in the Hello, World example, `hello` is your base directory.

Source code

Source code can be placed in the project's base directory as with `hello/hw.scala`. However, most people don't do this for real projects; too much clutter.

sbt uses the same directory structure as Maven for source files by default (all paths are relative to the base directory):

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    java/
       <test Java sources>
```

Other directories in `src/` will be ignored. Additionally, all hidden directories will be ignored.

sbt build definition files

You've already seen `build.sbt` in the project's base directory. Other sbt files appear in a `project` subdirectory.

`project` can contain `.scala` files, which are combined with `.sbt` files to form the complete build definition. See .scala build definition for more.

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

You may see `.sbt` files inside `project/` but they are not equivalent to `.sbt` files in the project's base directory. Explaining this will come later, since you'll need some background information first.

Build products

Generated files (compiled classes, packaged jars, managed files, caches, and documentation) will be written to the `target` directory by default.

Configuring version control

Your `.gitignore` (or equivalent for other version control systems) should contain:

```
target/
```

Note that this deliberately has a trailing `/` (to match only directories) and it deliberately has no leading `/` (to match `project/target/` in addition to plain `target/`).

Running

This page describes how to use sbt once you have set up your project. It assumes you've installed sbt and created a Hello, World or other project.

Interactive mode

Run sbt in your project directory with no arguments:

```
$ sbt
```

Running sbt with no command line arguments starts it in interactive mode. Interactive mode has a command prompt (with tab completion and history!).

For example, you could type `compile` at the sbt prompt:

```
> compile
```

To `compile` again, press up arrow and then enter.

To run your program, type `run`.

To leave interactive mode, type `exit` or use Ctrl+D (Unix) or Ctrl+Z (Windows).

Batch mode

You can also run sbt in batch mode, specifying a space-separated list of sbt commands as arguments. For sbt commands that take arguments, pass the command and arguments as one argument to sbt by enclosing them in quotes. For example,

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

In this example, `testOnly` has arguments, `TestA` and `TestB`. The commands will be run in sequence (`clean`, `compile`, then `testOnly`).

Continuous build and test

To speed up your edit-compile-test cycle, you can ask sbt to automatically recompile or run tests whenever you save a source file.

Make a command run when one or more source files change by prefixing the command with `~`. For example, in interactive mode try:

```
> ~ compile
```

Press enter to stop watching for changes.

You can use the `~` prefix with either interactive mode or batch mode.

See Triggered Execution for more details.

Common commands

Here are some of the most common sbt commands. For a more complete list, see Command Line Reference.

clean

Deletes all generated files (in the target directory).

compile

Compiles the main sources (in src/main/scala and src/main/java directories).

test

Compiles and runs all tests.

console

Starts the Scala interpreter with a classpath including the compiled sources and all dependencies. To return to sbt, type :quit, Ctrl+D (Unix), or Ctrl+Z (Windows).

run <argument>*

Runs the main class for the project in the same virtual machine as sbt.

package

Creates a jar file containing the files in src/main/resources and the classes compiled from src/main/scala and src/main/java.

help <command>

Displays detailed help for the specified command. If no command is provided, displays brief descriptions of all commands.

reload

Reloads the build definition (build.sbt, project/*.scala, project/*.sbt files). Needed if you change the build definition.

Tab completion

Interactive mode has tab completion, including at an empty prompt. A special sbt convention is that pressing tab once may show only a subset of most likely completions, while pressing it more times shows more verbose choices.

History Commands

Interactive mode remembers history, even if you exit sbt and restart it. The simplest way to access history is with the up arrow key. The following commands are also supported:

!

Show history command help.

!!

Execute the previous command again.

!:

Show all previous commands.

!:n

Show the last n commands.

!n

Execute the command with index n, as shown by the !: command.

!-n

Execute the nth command before this one.

!string

Execute the most recent command starting with 'string.'

!?string

Execute the most recent command containing 'string.'

.sbt build definition

This page describes sbt build definitions, including some "theory" and the syntax of `build.sbt`. It assumes you know how to use sbt and have read the previous pages in the Getting Started Guide.

`.sbt` vs `.scala` Build Definition

An sbt build definition can contain files ending in `.sbt`, located in the base directory of a project, and files ending in `.scala`, located in the `project/` subdirectory of the base directory.

This page discusses `.sbt` files, which are suitable for most cases. The `.scala` files are typically used for sharing code across `.sbt` files and for more complex build definitions. See .scala build definition (later in Getting Started) for more on `.scala` files.

What is a Build Definition?

After examining a project and processing build definition files, sbt ends up with an immutable map (set of key-value pairs) describing the build.

For example, one key is `name` and it maps to a string value, the name of your project.

Build definition files do not affect sbt's map directly.

Instead, the build definition creates a huge list of objects with type `Setting[T]` where `T` is the type of the value in the map. A `Setting` describes a *transformation to the map*, such as adding a new key-value pair or appending to an existing value. (In the spirit of functional programming with immutable data structures and values, a transformation returns a new map – it does not update the old map in-place.)

In `build.sbt`, you might create a `Setting[String]` for the name of your project like this:

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

This `Setting[String]` transforms the map by adding (or replacing) the name key, giving it the value `"hello"`. The transformed map becomes sbt's new map.

To create the map, sbt first sorts the list of settings so that all changes to the same key are made together, and values that depend on other keys are processed after the keys they depend on. Then sbt walks over the sorted list of `Settings` and applies each one to the map in turn.

Summary: A build definition defines a list of `Setting[T]`, where a `Setting[T]` is a transformation affecting sbt's map of key-value pairs and `T` is the type of each value.

How build.sbt defines settings

`build.sbt` defines a `Seq[Setting[_]]`; it's a list of Scala expressions, separated by blank lines, where each one becomes one element in the sequence. If you put `Seq(` in front of the `.sbt` file and `)` at the end and replace the blank lines with commas, you'd be looking at the equivalent `.scala` code.

Here's an example:

```
name := "hello"

version := "1.0"

scalaVersion := "2.10.3"
```

Each `Setting` is defined with a Scala expression. The expressions in `build.sbt` are independent of one another, and they are expressions, rather than complete Scala statements. These expressions may be interspersed with `val`s, `lazy val`s, and `def`s. Top-level `object`s and `class`es are not allowed in `build.sbt`. Those should go in the `project/` directory as full Scala source files.

On the left, `name`, `version`, and `scalaVersion` are *keys*. A key is an instance of `SettingKey[T]`, `TaskKey[T]`, or `InputKey[T]` where `T` is the expected value type. The kinds of key are explained below.

Keys have a method called `:=`, which returns a `Setting[T]`. You could use a Java-like syntax to call the method:

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

But Scala allows `name := "hello"` instead (in Scala, a single-parameter method can use either syntax).

The `:=` method on key `name` returns a `Setting`, specifically a `Setting[String]`. `String` also appears in the type of `name` itself, which is `SettingKey[String]`. In this case, the returned Setting[String] is a transformation to add or replace the `name` key in sbt's map, giving it the value `"hello"`.

If you use the wrong value type, the build definition will not compile:

```
name := 42  // will not compile
```

Settings must be separated by blank lines

You can't write a build.sbt like this:

```
// will NOT compile, no blank lines
name := "hello"
version := "1.0"
scalaVersion := "2.10.3"
```

sbt needs some kind of delimiter to tell where one expression stops and the next begins.

`.sbt` files contain a list of Scala expressions, not a single Scala program. These expressions have to be split up and passed to the compiler individually.

Keys

Types There are three flavors of key:

- `SettingKey[T]`: a key for a value computed once (the value is computed when loading the project, and kept around).
- `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. Check out Input Tasks for more details.

Built-in Keys The built-in keys are just fields in an object called Keys. A `build.sbt` implicitly has an `import sbt.Keys._`, so `sbt.Keys.name` can be referred to as `name`.

Custom Keys Custom keys may be defined with their respective creation methods: `settingKey`, `taskKey`, and `inputKey`. Each method expects the type of the value associated with the key as well as a description. The name of the key is taken from the `val` the key is assigned to. For example, to define a key for a new task called `hello`,

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

Here we have used the fact that an `.sbt` file can contain `val`s and `def`s in addition to settings. All such definitions are evaluated before settings regardless of where they are defined in the file. `val`s and `def`s must be separated from settings by blank lines.

> **Note:** Typically, lazy vals are used instead of vals to avoid initialization order problems.

Task vs Setting keys A `TaskKey[T]` is said to define a *task*. Tasks are operations such as `compile` or `package`. They may return `Unit` (`Unit` is Scala for `void`), or they may return a value related to the task, for example `package` is a `TaskKey[File]` and its value is the jar file it creates.

Each time you start a task execution, for example by typing `compile` at the interactive sbt prompt, sbt will re-run any tasks involved exactly once.

sbt's map describing the project can keep around a fixed string value for a setting such as name, but it has to keep around some executable code for a task such as `compile` – even if that executable code eventually returns a string, it has to be re-run every time.

A given key always refers to either a task or a plain setting. That is, "taskiness" (whether to re-run each time) is a property of the key, not the value.

Defining tasks and settings

Using `:=`, you can assign a value to a setting and a computation to a task. For a setting, the value will be computed once at project load time. For a task, the computation will be re-run each time the task is executed.

For example, to implement the `hello` task from the previous section, :

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

We already saw an example of defining settings when we defined the project's name,

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

Types for tasks and settings From a type-system perspective, the `Setting` created from a task key is slightly different from the one created from a setting key. `taskKey := 42` results in a `Setting[Task[T]]` while `settingKey := 42` results in a `Setting[T]`. For most purposes this makes no difference; the task key still creates a value of type `T` when the task executes.

The `T` vs. `Task[T]` type difference has this implication: a setting can't depend on a task, because a setting is evaluated only once on project load and is not re-run. More on this in more kinds of setting, coming up soon.

Keys in sbt interactive mode

In sbt's interactive mode, you can type the name of any task to execute that task. This is why typing `compile` runs the `compile` task. `compile` is a task key.

If you type the name of a setting key rather than a task key, the value of the setting key will be displayed. Typing a task key name executes the task but doesn't display the resulting value; to see a task's result, use `show <task name>` rather than plain `<task name>`. The convention for keys names is to use `camelCase` so that the command line name and the Scala identifiers are the same.

To learn more about any key, type `inspect <keyname>` at the sbt interactive prompt. Some of the information `inspect` displays won't make sense yet, but at the top it shows you the setting's value type and a brief description of the setting.

Imports in build.sbt

You can place import statements at the top of `build.sbt`; they need not be separated by blank lines.

There are some implied default imports, as follows:

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

(In addition, if you have .scala files, the contents of any `Build` or `Plugin` objects in those files will be imported. More on that when we get to .scala build definition.)

Adding library dependencies

To depend on third-party libraries, there are two options. The first is to drop jars in `lib/` (unmanaged dependencies) and the other is to add managed dependencies, which will look like this in `build.sbt`:

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

This is how you add a managed dependency on the Apache Derby library, version 10.4.1.3.

The `libraryDependencies` key involves two complexities: `+=` rather than `:=`, and the `%` method. `+=` appends to the key's old value rather than replacing it, this is explained in more kinds of setting. The `%` method is used to construct an Ivy module ID from strings, explained in Library dependencies.

We'll skip over the details of library dependencies until later in the Getting Started Guide. There's a whole page covering it later on.

Scopes

This page describes scopes. It assumes you've read and understood the previous page, .sbt build definition.

The whole story about keys

Previously we pretended that a key like `name` corresponded to one entry in sbt's map of key-value pairs. This was a simplification.

In truth, each key can have an associated value in more than one context, called a "scope."

Some concrete examples:

- if you have multiple projects in your build definition, a key can have a different value in each project.
- the `compile` key may have a different value for your main sources and your test sources, if you want to compile them differently.
- the `packageOptions` key (which contains options for creating jar packages) may have different values when packaging class files (`packageBin`) or packaging source code (`packageSrc`).

There is no single value for a given key `name`, because the value may differ according to scope.

However, there is a single value for a given *scoped* key.

If you think about sbt processing a list of settings to generate a key-value map describing the project, as discussed earlier, the keys in that key-value map are *scoped* keys. Each setting defined in the build definition (for example in `build.sbt`) applies to a scoped key as well.

Often the scope is implied or has a default, but if the defaults are wrong, you'll need to mention the desired scope in `build.sbt`.

Scope axes

A *scope axis* is a type, where each instance of the type can define its own scope (that is, each instance can have its own unique values for keys).

There are three scope axes:

- Projects
- Configurations
- Tasks

Scoping by project axis If you put multiple projects in a single build, each project needs its own settings. That is, keys can be scoped according to the project.

The project axis can also be set to "entire build", so a setting applies to the entire build rather than a single project. Build-level settings are often used as a fallback when a project doesn't define a project-specific setting.

Scoping by configuration axis A *configuration* defines a flavor of build, potentially with its own classpath, sources, generated packages, etc. The configuration concept comes from Ivy, which sbt uses for managed dependencies Library Dependencies, and from MavenScopes.

Some configurations you'll see in sbt:

- `Compile` which defines the main build (`src/main/scala`).
- `Test` which defines how to build tests (`src/test/scala`).
- `Runtime` which defines the classpath for the `run` task.

By default, all the keys associated with compiling, packaging, and running are scoped to a configuration and therefore may work differently in each configuration. The most obvious examples are the task keys `compile`, `package`, and `run`; but all the keys which *affect* those keys (such as `sourceDirectories` or `scalacOptions` or `fullClasspath`) are also scoped to the configuration.

Scoping by task axis Settings can affect how a task works. For example, the `packageSrc` task is affected by the `packageOptions` setting.

To support this, a task key (such as `packageSrc`) can be a scope for another key (such as `packageOptions`).

The various tasks that build a package (`packageSrc`, `packageBin`, `packageDoc`) can share keys related to packaging, such as `artifactName` and `packageOptions`. Those keys can have distinct values for each packaging task.

Global scope

Each scope axis can be filled in with an instance of the axis type (for example the task axis can be filled in with a task), or the axis can be filled in with the special value `Global`.

`Global` means what you would expect: the setting's value applies to all instances of that axis. For example if the task axis is Global, then the setting would apply to all 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).

More kinds of setting

This page explains other ways to create a Setting, beyond the basic `:=` method. It assumes you've read .sbt build definition and scopes.

Refresher: Settings

Remember, a build definition creates a list of `Setting`, which is then used to transform sbt's description of the build (which is a map of key-value pairs). A Setting is a transformation with sbt's earlier map as input and a new map as output. The new map becomes sbt's new state.

Different settings transform the map in different ways. Earlier, you read about the `:=` method.

The `Setting` which `:=` creates puts a fixed, constant value in the new, transformed map. For example, if you transform a map with the setting `name := "hello"` the new map has the string `"hello"` stored under the key name.

Settings must end up in the master list of settings to do any good (all lines in a `build.sbt` automatically end up in the list, but in a .scala file you can get it wrong by creating a `Setting` without putting it where sbt will find it).

Appending to previous values: `+=` and `++=`

Assignment with `:=` is the simplest transformation, but keys have other methods as well. If the `T` in `SettingKey[T]` is a sequence, i.e. the key's value type is a sequence, you can append to the sequence rather than replacing it.

- `+=` will append a single element to the sequence.
- `++=` will concatenate another sequence.

For example, the key `sourceDirectories in Compile` has a `Seq[File]` as its value. By default this key's value would include `src/main/scala`. If you wanted to also compile source code in a directory called source (since you just have to be nonstandard), you could add that directory:

```
sourceDirectories in Compile += new File("source")
```

Or, using the `file()` function from the sbt package for convenience:

```
sourceDirectories in Compile += file("source")
```

(`file()` just creates a new `File`.)

You could use `++=` to add more than one directory at a time:

```
sourceDirectories in Compile ++= Seq(file("sources1"), file("sources2"))
```

Where `Seq(a, b, c, ...)` is standard Scala syntax to construct a sequence.

To replace the default source directories entirely, you use `:=` of course:

```
sourceDirectories in Compile := Seq(file("sources1"), file("sources2"))
```

Computing a value based on other keys' values

Reference the value of another task or setting by calling value on the key for the task or setting. The value method is special and may only be called in the argument to `:=`, `+=`, or `++=`.

As a first example, consider defining the project organization to be the same as the project name.

```
// name our organization after our project (both are SettingKey[String])
organization := name.value
```

Or, set the name to the name of the project's directory:

```
// name is a Key[String], baseDirectory is a Key[File]
// name the project after the directory it's inside
name := baseDirectory.value.getName
```

This transforms the value of `baseDirectory` using the standard `getName` method of `java.io.File`.

Using multiple inputs is similar. For example,

```
name := "project " + name.value + " from " + organization.value + " version " + version.valu
```

This sets the name in terms of its previous value as well as the organization and version settings.

Settings with dependencies In the setting `name := baseDirectory.value.getName`, `name` will have a *dependency* on `baseDirectory`. If you place the above in `build.sbt` and run the sbt interactive console, then type `inspect name`, you should see (in part):

```
[info] Dependencies:
[info]  *:baseDirectory
```

This is how sbt knows which settings depend on which other settings. Remember that some settings describe tasks, so this approach also creates dependencies between tasks.

For example, if you `inspect compile` you'll see it depends on another key `compileInputs`, and if you `inspect compileInputs` it in turn depends on other keys. Keep following the dependency chains and magic happens. When you type `compile` sbt automatically performs an `update`, for example. It Just Works because the values required as inputs to the `compile` computation require sbt to do the `update` computation first.

In this way, all build dependencies in sbt are *automatic* rather than explicitly declared. If you use a key's value in another computation, then the computation depends on that key. It just works!

When settings are undefined Whenever a setting uses `:=`, `+=`, or `++=` to create a dependency on itself or another key's value, the value it depends on must exist. If it does not, sbt will complain. It might say *"Reference to undefined setting"*, for example. When this happens, be sure you're using the key in the scope that defines it.

It's possible to create cycles, which is an error; sbt will tell you if you do this.

Tasks based on other keys' values You can compute values of some tasks or settings to define or append value for another task. It's done by using `Def.task` and `taskValue`, as argument to `:=`, `+=` or `++=`.

As a first example, consider appending a source generator using the project base directory and compilation classpath.

```
sourceGenerators in Compile += Def.task {
  myGenerator(baseDirectory.value, (managedClasspath in Compile).value)
}.taskValue
```

Tasks with dependencies As noted in .sbt build definition, task keys create a `Setting[Task[T]]` rather than a `Setting[T]` when you build a setting with `:=`, etc. Tasks can use settings as inputs, but settings cannot use tasks as inputs.

Take these two keys (from Keys):

```
val scalacOptions = taskKey[Seq[String]]("Options for the Scala compiler.")
val checksums = settingKey[Seq[String]]("The list of checksums to generate and to verify for
```

(`scalacOptions` and `checksums` have nothing to do with each other, they are just two keys with the same value type, where one is a task.)

It is possible to compile a `build.sbt` that aliases `scalacOptions` to `checksums`, but not the other way. For example, this is allowed:

```
// The scalacOptions task may be defined in terms of the checksums setting
scalacOptions := checksums.value
```

There is no way to go the *other* direction. That is, a setting key can't depend on a task key. That's because a setting key is only computed once on project load, so the task would not be re-run every time, and tasks expect to re-run every time.

```
// The checksums setting may not be defined in terms of the scalacOptions task
checksums := scalacOptions.value
```

Appending with dependencies: `+=` and `++=`

Other keys can be used when appending to an existing setting or task, just like they can for assigning with `:=`.

For example, say you have a coverage report named after the project, and you want to add it to the files removed by clean:

```
cleanFiles += file("coverage-report-" + name.value + ".txt")
```

Library dependencies

This page assumes you've already read the earlier Getting Started pages, in particular .sbt build definition, scopes, and more kinds of setting.

Library dependencies can be added in two ways:

- *unmanaged dependencies* are jars dropped into the `lib` directory
- *managed dependencies* are configured in the build definition and downloaded automatically from repositories

Unmanaged dependencies

Most people use managed dependencies instead of unmanaged. But unmanaged can be simpler when starting out.

Unmanaged dependencies work like this: add jars to `lib` and they will be placed on the project classpath. Not much else to it!

You can place test jars such as ScalaCheck, Specs2, and ScalaTest in `lib` as well.

Dependencies in `lib` go on all the classpaths (for `compile`, `test`, `run`, and `console`). If you wanted to change the classpath for just one of those, you would adjust `dependencyClasspath in Compile` or `dependencyClasspath in Runtime` for example.

There's nothing to add to `build.sbt` to use unmanaged dependencies, though you could change the `unmanagedBase` key if you'd like to use a different directory rather than `lib`.

To use `custom_lib` instead of `lib`:

```
unmanagedBase := baseDirectory.value / "custom_lib"
```

`baseDirectory` is the project's root directory, so here you're changing `unmanagedBase` depending on `baseDirectory` using the special `value` method as explained in more kinds of setting.

There's also an `unmanagedJars` task which lists the jars from the `unmanagedBase` directory. If you wanted to use multiple directories or do something else complex, you might need to replace the whole `unmanagedJars` task with one that does something else, e.g. empty the list for `Compile` configuration regardless of the files in `lib` directory:

```
unmanagedJars in Compile := Seq.empty[sbt.Attributed[java.io.File]]
```

Managed Dependencies

sbt uses Apache Ivy to implement managed dependencies, so if you're familiar with Ivy or Maven, you won't have much trouble.

The `libraryDependencies` key Most of the time, you can simply list your dependencies in the setting `libraryDependencies`. It's also possible to write a Maven POM file or Ivy configuration file to externally configure your dependencies, and have sbt use those external configuration files. You can learn more about that here.

Declaring a dependency looks like this, where `groupId`, `artifactId`, and `revision` are strings:

```
libraryDependencies += groupID % artifactID % revision
```

or like this, where `configuration` can be a string or Configuration val:

```
libraryDependencies += groupID % artifactID % revision % configuration
```

`libraryDependencies` is declared in Keys like this:

```
val libraryDependencies = settingKey[Seq[ModuleID]]("Declares managed dependencies.")
```

The `%` methods create `ModuleID` objects from strings, then you add those `ModuleID` to `libraryDependencies`.

Of course, sbt (via Ivy) has to know where to download the module. If your module is in one of the default repositories sbt comes with, this will just work. For example, Apache Derby is in the standard Maven2 repository:

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

If you type that in `build.sbt` and then `update`, sbt should download Derby to `~/.ivy2/cache/org.apache.derby/`. (By the way, `update` is a dependency of `compile` so there's no need to manually type `update` most of the time.)

Of course, you can also use `++=` to add a list of dependencies all at once:

```
libraryDependencies ++= Seq(
  groupID % artifactID % revision,
  groupID % otherID % otherRevision
)
```

In rare cases you might find reasons to use `:=` with `libraryDependencies` as well.

Getting the right Scala version with %% If you use groupID %% artifactID % revision rather than groupID % artifactID % revision (the difference is the double %% after the groupID), sbt will add your project's Scala version to the artifact name. This is just a shortcut. You could write this without the %%:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.11.1" % "0.3"
```

Assuming the scalaVersion for your build is 2.11.1, the following is identical (note the double %% after "org.scala-tools"):

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

The idea is that many dependencies are compiled for multiple Scala versions, and you'd like to get the one that matches your project to ensure binary compatibility.

The complexity in practice is that often a dependency will work with a slightly different Scala version; but %% is not smart about that. So if the dependency is available for 2.10.1 but you're using scalaVersion := "2.10.4", you won't be able to use %% even though the 2.10.1 dependency likely works. If %% stops working, just go see which versions the dependency is really built for, and hardcode the one you think will work (assuming there is one).

See Cross Building for some more detail on this.

Ivy revisions The revision in groupID % artifactID % revision does not have to be a single fixed version. Ivy can select the latest revision of a module according to constraints you specify. Instead of a fixed revision like "1.6.1", you specify "latest.integration", "2.9.+", or "[1.0,)". See the Ivy revisions documentation for details.

Resolvers Not all packages live on the same server; sbt uses the standard Maven2 repository by default. If your dependency isn't on one of the default repositories, you'll have to add a *resolver* to help Ivy find it.

To add an additional repository, use

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

with the special at between two strings.

For example:

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

The `resolvers` key is defined in Keys like this:

```
val resolvers = settingKey[Seq[Resolver]]("The user-defined additional resolvers for automat
```

The `at` method creates a `Resolver` object from two strings.

sbt can search your local Maven repository if you add it as a repository:

```
resolvers += "Local Maven Repository" at "file://"+Path.userHome.absolutePath+"/.m2/reposito
```

or, for convenience:

```
resolvers += Resolver.mavenLocal
```

See Resolvers for details on defining other types of repositories.

Overriding default resolvers `resolvers` does not contain the default resolvers; only additional ones added by your build definition.

sbt combines `resolvers` with some default repositories to form `externalResolvers`.

Therefore, to change or remove the default resolvers, you would need to override `externalResolvers` instead of `resolvers`.

Per-configuration dependencies Often a dependency is used by your test code (in `src/test/scala`, which is compiled by the `Test` configuration) but not your main code.

If you want a dependency to show up in the classpath only for the `Test` configuration and not the `Compile` configuration, add `% "test"` like this:

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

You may also use the type-safe version of `Test` configuration as follows:

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

Now, if you type `show compile:dependencyClasspath` at the sbt interactive prompt, you should not see the derby jar. But if you type `show test:dependencyClasspath`, you should see the derby jar in the list.

Typically, test-related dependencies such as ScalaCheck, Specs2, and ScalaTest would be defined with `% "test"`.

There are more details and tips-and-tricks related to library dependencies on this page.

Multi-project builds

This page introduces multiple projects in a single build.

Please read the earlier pages in the Getting Started Guide first, in particular you need to understand build.sbt before reading this page.

Multiple projects

It can be useful to keep multiple related projects in a single build, especially if they depend on one another and you tend to modify them together.

Each sub-project in a build has its own source directories, generates its own jar file when you run package, and in general works like any other project.

A project is defined by declaring a lazy val of type Project. For example, :

```
lazy val util = project

lazy val core = project
```

The name of the val is used as the project's ID and base directory name. The ID is used to refer to the project at the command line. The base directory may be changed from the default using the in method. For example, the following is a more explicit way to write the previous example:

```
lazy val util = project.in(file("util"))

lazy val core = project in file("core")
```

Dependencies

Projects in the build can be completely independent of one another, but usually they will be related to one another by some kind of dependency. There are two types of dependencies: aggregate and classpath.

Aggregation Aggregation means that running a task on the aggregate project will also run it on the aggregated projects. For example,

```
lazy val root = (project in file(".")).
  aggregate(util, core)

lazy val util = project

lazy val core = project
```

In the above example, the root project aggregates `util` and `core`. Start up sbt with two subprojects as in the example, and try compile. You should see that all three projects are compiled.

In the project doing the aggregating, the root project in this case, you can control aggregation per-task. For example, to avoid aggregating the `update` task:

```
lazy val root = (project in file(".")).
  aggregate(util, core).
  settings(
    aggregate in update := false
  )

[...]
```

`aggregate in update` is the aggregate key scoped to the `update` task. (See scopes.)

Note: aggregation will run the aggregated tasks in parallel and with no defined ordering between them.

Classpath dependencies A project may depend on code in another project. This is done by adding a `dependsOn` method call. For example, if core needed util on its classpath, you would define core as:

```
lazy val core = project.dependsOn(util)
```

Now code in `core` can use classes from `util`. This also creates an ordering between the projects when compiling them; `util` must be updated and compiled before core can be compiled.

To depend on multiple projects, use multiple arguments to `dependsOn`, like `dependsOn(bar, baz)`.

Per-configuration classpath dependencies `foo dependsOn(bar)` means that the `compile` configuration in `foo` depends on the `compile` configuration in `bar`. You could write this explicitly as `dependsOn(bar % "compile->compile")`.

The `->` in `"compile->compile"` means "depends on" so `"test->compile"` means the `test` configuration in `foo` would depend on the `compile` configuration in `bar`.

Omitting the `->config` part implies `->compile`, so `dependsOn(bar % "test")` means that the `test` configuration in `foo` depends on the `Compile` configuration in `bar`.

A useful declaration is `"test->test"` which means `test` depends on `test`. This allows you to put utility code for testing in `bar/src/test/scala` and then use that code in `foo/src/test/scala`, for example.

You can have multiple configurations for a dependency, separated by semicolons. For example, `dependsOn(bar % "test->test;compile->compile")`.

Default root project

If a project is not defined for the root directory in the build, sbt creates a default one that aggregates all other projects in the build.

Because project `hello-foo` is defined with `base = file("foo")`, it will be contained in the subdirectory foo. Its sources could be directly under `foo`, like `foo/Foo.scala`, or in `foo/src/main/scala`. The usual sbt directory structure applies underneath `foo` with the exception of build definition files.

Any `.sbt` files in `foo`, say `foo/build.sbt`, will be merged with the build definition for the entire build, but scoped to the `hello-foo` project.

If your whole project is in hello, try defining a different version (`version := "0.6"`) in `hello/build.sbt`, `hello/foo/build.sbt`, and `hello/bar/build.sbt`. Now `show version` at the sbt interactive prompt. You should get something like this (with whatever versions you defined):

```
> show version
[info] hello-foo/*:version
[info]  0.7
[info] hello-bar/*:version
[info]  0.9
[info] hello/*:version
[info]  0.5
```

`hello-foo/*:version` was defined in `hello/foo/build.sbt`, `hello-bar/*:version` was defined in `hello/bar/build.sbt`, and `hello/*:version` was defined in `hello/build.sbt`. Remember the syntax for scoped keys. Each `version` key is scoped to a project, based on the location of the `build.sbt`. But all three `build.sbt` are part of the same build definition.

Each project's settings can go in `.sbt` files in the base directory of that project, while the `.scala` file can be as simple as the one shown above, listing the projects and base directories. *There is no need to put settings in the `.scala` file.*

You may find it cleaner to put everything including settings in `.scala` files in order to keep all build definition under a single project directory, however. It's up to you.

You cannot have a project subdirectory or `project/*.scala` files in the sub-projects. `foo/project/Build.scala` would be ignored.

Navigating projects interactively

At the sbt interactive prompt, type `projects` to list your projects and `project <projectname>` to select a current project. When you run a task like `compile`, it runs on the current project. So you don't necessarily have to compile the root project, you could compile only a subproject.

You can run a task in another project by explicitly specifying the project ID, such as `subProjectID/compile`.

Common code

The definitions in `.sbt` files are not visible in other `.sbt` files. In order to share code between `.sbt` files, define one or more Scala files in the `project/` directory of the build root. This directory is also an sbt project, but for your build.

For example:

`<root>/project/Common.scala`:

```
import sbt._
import Keys._

object Common {
  def text = "org.example"
}
```

`<root>/build.sbt`:

```
organization := Common.text
```

See .scala Build Definition for details.

Using plugins

Please read the earlier pages in the Getting Started Guide first, in particular you need to understand build.sbt and library dependencies, before reading this page.

What is a plugin?

A plugin extends the build definition, most commonly by adding new settings. The new settings could be new tasks. For example, a plugin could add a `codeCoverage` task which would generate a test coverage report.

Declaring a plugin

If your project is in directory `hello`, and you're adding sbt-site plugin to the build definition, create `hello/project/site.sbt` and declare the plugin dependency by passing the plugin's Ivy module ID to `addSbtPlugin`:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-site" % "0.7.0")
```

If you're adding sbt-assembly, create `hello/project/assembly.sbt` with the following:

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

Not every plugin is located on one of the default repositories and a plugin's documentation may instruct you to also add the repository where it can be found:

```
resolvers += Resolver.sonatypeRepo("public")
```

Plugins usually provide settings that get added to a project to enable the plugin's functionality. This is described in the next section.

Enabling and disabling auto plugins

A plugin can declare that its settings be automatically added to the build definition, in which case you don't have to do anything to add them.

As of sbt 0.13.5, there is a new auto plugins feature that enables plugins to automatically, and safely, ensure their settings and dependencies are on a project. Many auto plugins should have their default settings automatically, however some may require explicit enablement.

If you're using an auto plugin that requires explicit enablement, then you you have to add the following to your `build.sbt`:

```
lazy val util = (project in file("util")).
  enablePlugins(FooPlugin, BarPlugin).
  settings(
    name := "hello-util"
  )
```

The `enablePlugins` method allows projects to explicitly define the auto plugins they wish to consume.

Projects can also exclude plugins using the `disablePlugins` method. For example, if we wish to remove the `IvyPlugin` settings from `util`, we modify our `build.sbt` as follows:

```
lazy val util = (project in file("util")).
  enablePlugins(FooPlugin, BarPlugin).
  disablePlugins(plugins.IvyPlugin).
  settings(
    name := "hello-util"
  )
```

Auto plugins should document whether they need to explicitly enabled. If you're curious which auto plugins are enabled for a given project, just run the `plugins` command on the sbt console.

For example:

```
> plugins
In file:/home/jsuereth/projects/sbt/test-ivy-issues/
        sbt.plugins.IvyPlugin: enabled in scala-sbt-org
        sbt.plugins.JvmPlugin: enabled in scala-sbt-org
        sbt.plugins.CorePlugin: enabled in scala-sbt-org
        sbt.plugins.JUnitXmlReportPlugin: enabled in scala-sbt-org
```

Here, the `plugins` output is showing that the sbt default plugins are all enabled. sbt's default settings are provided via three plugins:

1. `CorePlugin`: Provides the core parallelism controls for tasks
2. `IvyPlugin`: Provides the mechanisms to publish/resolve modules.
3. `JvmPlugin`: Provides the mechanisms to compile/test/run/package Java/Scala projects.

In addition, `JUnitXmlReportPlugin` provides an experimental support for generating junit-xml.

Older non-auto plugins often require settings to be added explictly, so that multi-project build could have different types of projects. The plugin documentation will indicate how to configure it, but typically for older plugins this involves adding the base settings for the plugin and customizing as necessary.

For example, for the sbt-site plugin, create `site.sbt` with the following content

```
site.settings
```

to enable it for that project.

If the build defines multiple projects, instead add it directly to the project:

```
// don't use the site plugin for the `util` project
lazy val util = (project in file("util"))

// enable the site plugin for the `core` project
lazy val core = (project in file("core")).
  settings(site.settings : _*)
```

Global plugins

Plugins can be installed for all your projects at once by declaring them in `~/.sbt/0.13/plugins/`. `~/.sbt/0.13/plugins/` is an sbt project whose classpath is exported to all sbt build definition projects. Roughly speaking, any `.sbt` or `.scala` files in `~/.sbt/0.13/plugins/` behave as if they were in the `project/` directory for all projects.

You can create `~/.sbt/0.13/plugins//build.sbt` and put `addSbtPlugin()` expressions in there to add plugins to all your projects at once. Because doing so would increase the dependency on the machine environment, this feature should be used sparingly. See Best Practices.

Available Plugins

There's a list of available plugins.

Some especially popular plugins are:

- those for IDEs (to import an sbt project into your IDE)
- those supporting web frameworks, such as xsbt-web-plugin.

For more details, including ways of developing plugins, see Plugins. For best practices, see Plugins-Best-Practices.

Custom settings and tasks

This page gets you started creating your own settings and tasks.

To understand this page, be sure you've read earlier pages in the Getting Started Guide, especially .build.sbt and more kinds of setting.

Defining a key

Keys is packed with examples illustrating how to define keys. Most of the keys are implemented in Defaults.

Keys have one of three types. SettingKey and TaskKey are described in .sbt build definition. Read about InputKey on the Input Tasks page.

Some examples from Keys:

```
val scalaVersion = settingKey[String]("The version of Scala used for building.")
val clean = taskKey[Unit]("Deletes files produced by the build, such as generated sources, c
```

The key constructors have two string parameters: the name of the key ("scalaVersion") and a documentation string ("The version of scala used for building.").

Remember from .sbt build definition that the type parameter T in SettingKey[T] indicates the type of value a setting has. T in TaskKey[T] indicates the type of the task's result. Also remember from .sbt build definition that a setting has a fixed value until project reload, while a task is re-computed for every "task execution" (every time someone types a command at the sbt interactive prompt or in batch mode).

Keys may be defined in a .sbt file, .scala file, or in a plugin. Any val found in a Build object in your .scala build definition files or any val found in a Plugin object from a plugin will be imported automatically into your .sbt files.

Implementing a task

Once you've defined a key for your task, you'll need to complete it with a task definition. You could be defining your own task, or you could be planning to redefine an existing task. Either way looks the same; use := to associate some code with the task key:

```
val sampleStringTask = taskKey[String]("A sample string task.")

val sampleIntTask = taskKey[Int]("A sample int task.")

sampleStringTask := System.getProperty("user.home")

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

If the task has dependencies, you'd reference their value using value, as discussed in more kinds of setting.

The hardest part about implementing tasks is often not sbt-specific; tasks are just Scala code. The hard part could be writing the "meat" of your task that does whatever you're trying to do. For example, maybe you're trying to format HTML in which case you might want to use an HTML library (you would add a library dependency to your build definition and write code based on the HTML library, perhaps).

sbt has some utility libraries and convenience functions, in particular you can often use the convenient APIs in IO to manipulate files and directories.

Use plugins!

If you find you have a lot of custom code, consider moving it to a plugin for re-use across multiple builds.

It's very easy to create a plugin, as teased earlier and discussed at more length here.

This page has been a quick taste; there's much much more about custom tasks on the Tasks page.

.scala build definition

This page assumes you've read previous pages in the Getting Started Guide, *especially* .sbt build definition and more kinds of setting.

sbt is recursive

`build.sbt` is so simple, it conceals how sbt really works. sbt builds are defined with Scala code. That code, itself, has to be built. What better way than with sbt?

The `project` directory *is another project inside your project* which knows how to build your project. The project inside `project` can (in theory) do anything any other project can do. *Your build definition is an sbt project.*

And the turtles go all the way down. If you like, you can tweak the build definition of the build definition project, by creating a `project/project/` directory.

Here's an illustration.

```
hello/                  # your project's base directory

    Hello.scala        # a source file in your project (could be in
                        #   src/main/scala too)

    build.sbt         # build.sbt is part of the source code for the
                        #   build definition project inside project/
```

```
   project/         # base directory of the build definition project

        Build.scala     # a source file in the project/ project,
                  #   that is, a source file in the build definition

      build.sbt     # this is part of a build definition for a project
                  #   in project/project ; build definition's build
                        #   definition


      project/      # base directory of the build definition project
                        #   for the build definition

          Build.scala # source file in the project/project/ project
```

Don't worry! Most of the time you are not going to need all that. But understanding the principle can be helpful.

By the way: any time files ending in `.scala` or `.sbt` are used, naming them `build.sbt` and `Build.scala` are conventions only. This also means that multiple files are allowed.

`.scala` source files in the build definition project

`.sbt` files are merged into their sibling project directory. Looking back at the project layout:

```
hello/                   # your project's base directory

    build.sbt         # build.sbt is part of the source code for the
                       #   build definition project inside project/

   project/         # base directory of the build definition project

        Build.scala     # a source file in the project/ project,
                  #   that is, a source file in the build definition
```

The Scala expressions in build.sbt are compiled alongside and merged with `Build.scala` (or any other `.scala` files in the `project/` directory).

`*.sbt` files in the base directory for a project become part of the project build definition project also located in that base directory.

The `.sbt` file format is a convenient shorthand for adding settings to the build definition project.

Relating build.sbt to Build.scala

To mix `.sbt` and `.scala` files in your build definition, you need to understand how they relate.

The following two files illustrate. First, if your project is in hello, create `hello/project/Build.scala` as follows:

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

object HelloBuild extends Build {
  val sampleKeyA = settingKey[String]("demo key A")
  val sampleKeyB = settingKey[String]("demo key B")
  val sampleKeyC = settingKey[String]("demo key C")
  val sampleKeyD = settingKey[String]("demo key D")

  override lazy val settings = super.settings ++
    Seq(
      sampleKeyA := "A: in Build.settings in Build.scala",
      resolvers := Seq()
    )

  lazy val root = Project(id = "hello",
    base = file("."),
    settings = Seq(
      sampleKeyB := "B: in the root project settings in Build.scala"
    ))
}
```

Now, create `hello/build.sbt` as follows:

```scala
sampleKeyC in ThisBuild := "C: in build.sbt scoped to ThisBuild"

sampleKeyD := "D: in build.sbt"
```

Start up the sbt interactive prompt. Type `inspect sampleKeyA` and you should see (among other things):

```
[info] Setting: java.lang.String = A: in Build.settings in Build.scala
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}/*:sampleKeyA
```

and then `inspect sampleKeyC` and you should see:

```
[info] Setting: java.lang.String = C: in build.sbt scoped to ThisBuild
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}/*:sampleKeyC
```

Note that the "Provided by" shows the same scope for the two values. That is, `sampleKeyC in ThisBuild` in a `.sbt` file is equivalent to placing a setting in the `Build.settings` list in a `.scala` file. sbt takes build-scoped settings from both places to create the build definition.

Now, `inspect sampleKeyB`:

```
[info] Setting: java.lang.String = B: in the root project settings in Build.scala
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}hello/*:sampleKeyB
```

Note that `sampleKeyB` is scoped to the project (`{file:/home/hp/checkout/hello/}hello`) rather than the entire build (`{file:/home/hp/checkout/hello/}`).

As you've probably guessed, `inspect sampleKeyD` matches `sampleKeyB`:

```
[info] Setting: java.lang.String = D: in build.sbt
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}hello/*:sampleKeyD
```

sbt *appends* the settings from `.sbt` files to the settings from `Build.settings` and `Project.settings` which means `.sbt` settings take precedence. Try changing `Build.scala` so it sets key `sampleC` or `sampleD`, which are also set in `build.sbt`. The setting in `build.sbt` should "win" over the one in `Build.scala`.

One other thing you may have noticed: `sampleKeyC` and `sampleKeyD` were available inside `build.sbt`. That's because sbt imports the contents of your `Build` object into your `.sbt` files. In this case `import HelloBuild._` was implicitly done for the `build.sbt` file.

In summary:

- In `.scala` files, you can add settings to `Build.settings` for sbt to find, and they are automatically build-scoped.
- In `.scala` files, you can add settings to `Project.settings` for sbt to find, and they are automatically project-scoped.
- Any `Build` object you write in a `.scala` file will have its contents imported and available to `.sbt` files.
- The settings in `.sbt` files are *appended* to the settings in `.scala` files.
- The settings in `.sbt` files are project-scoped unless you explicitly specify another scope.

When to use .scala files

In .scala files, you can write any Scala code, including top-level classes and objects. Also, there are no restrictions on blank lines, since they are standard .scala files.

The recommended approach is to define most configuration in .sbt files, using .scala files for task implementations or to share values, such as keys, across .sbt files.

The build definition project in interactive mode

You can switch the sbt interactive prompt to have the build definition project in project/ as the current project. To do so, type reload plugins.

```
> reload plugins
[info] Set current project to default-a0e8e4 (in build file:/home/hp/checkout/hello/project/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/project/Build.scala)
> reload return
[info] Loading project definition from /home/hp/checkout/hello/project
[info] Set current project to hello (in build file:/home/hp/checkout/hello/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/hw.scala)
>
```

As shown above, you use reload return to leave the build definition project and return to your regular project.

Reminder: it's all immutable

It would be wrong to think that the settings in build.sbt are added to the settings fields in Build and Project objects. Instead, the settings list from Build and Project, and the settings from build.sbt, are concatenated into another immutable list which is then used by sbt. The Build and Project objects are "immutable configuration" forming only part of the complete build definition.

In fact, there are other sources of settings as well. They are appended in this order:

- Settings from Build.settings and Project.settings in your .scala files.
- Your user-global settings; for example in ~/.sbt/0.13/global.sbt you can define settings affecting *all* your projects.

- Settings injected by plugins, see using plugins coming up next.
- Settings from `.sbt` files in the project.
- Build definition projects (i.e. projects inside `project`) have settings from global plugins (`~/.sbt/0.13/plugins/`) added. Using plugins explains this more.

Later settings override earlier ones. The entire list of settings forms the build definition.

Getting Started summary

This page wraps up the Getting Started Guide.

To use sbt, there are a small number of concepts you must understand. These have some learning curve, but on the positive side, there isn't much to sbt *except* these concepts. sbt uses a small core of powerful concepts to do everything it does.

If you've read the whole Getting Started series, now you know what you need to know.

sbt: The Core Concepts

- the basics of Scala. It's undeniably helpful to be familiar with Scala syntax. Programming in Scala written by the creator of Scala is a great introduction.
- .sbt build definition
- your build definition is one big list of `Setting` objects, where a `Setting` transforms the set of key-value pairs sbt uses to perform tasks.
- to create a `Setting`, call one of a few methods on a key: `:=`, `+=`, or `++=`.
- there is no mutable state, only transformation; for example, a `Setting` transforms sbt's collection of key-value pairs into a new collection. It doesn't change anything in-place.
- each setting has a value of a particular type, determined by the key.
- *tasks* are special settings where the computation to produce the key's value will be re-run each time you kick off a task. Non-tasks compute the value once, when first loading the build definition.
- Scopes
- each key may have multiple values, in distinct scopes.
- scoping may use three axes: configuration, project, and task.
- scoping allows you to have different behaviors per-project, per-task, or per-configuration.
- a configuration is a kind of build, such as the main one (`Compile`) or the test one (`Test`).

- the per-project axis also supports "entire build" scope.
- scopes fall back to or *delegate* to more general scopes.
- .sbt vs. .scala build definition
- put most of your configuration in `build.sbt`, but use `.scala` build definition files for defining classes and larger task implementations.
- the build definition is an sbt project in its own right, rooted in the project directory.
- Plugins are extensions to the build definition
- add plugins with the `addSbtPlugin` method in `project/plugins.sbt` (NOT `build.sbt` in the project's base directory).

If any of this leaves you wondering rather than nodding, please ask for help, go back and re-read, or try some experiments in sbt's interactive mode.

Good luck!

Advanced Notes

Since sbt is open source, don't forget you can check out the source code too!