Simple Build Tool

This lesson covers SBT! Specific topics include:

- creating an sbt project
- basic commands
- the sbt console
- continuous command execution
- customizing your project
- custom commands
- quick tour of sbt source (if time)

About SBT

SBT is a modern build tool. While it is written in Scala and provides many Scala conveniences, it is a general purpose build tool.

Why SBT?

- Sane(ish) dependency management
  - Ivy for dependency management
  - Only-update-on-request model
- Full Scala language support for creating tasks
- Continuous command execution
- Launch REPL in project context

Getting Started

- Download the jar
- Create an sbt shell script that calls the jar, e.g.

```
java -Xmx512M -jar sbt-launch.jar "$@"
```

- make sure it’s executable and in your path
- run sbt to create your project

```
[local ~/projects]$ sbt
Project does not exist, create new project? (y/N/s) y
Name: sample
Organization: com.twitter
Version [1.0]: 1.0-SNAPSHOT
Scala version [2.7.7]: 2.8.1
sbt version [0.7.4]:
Getting Scala 2.7.7 ...
:: retrieving :: org.scala-tools.sbt#boot-scala
confs: [default]
2 artifacts copied, 0 already retrieved (9911kB/221ms)
Getting org.scala-tools.sbt sbt_2.7.7 0.7.4 ...
:: retrieving :: org.scala-tools.sbt#boot-app
confs: [default]
15 artifacts copied, 0 already retrieved (4096kB/167ms)
[success] Successfully initialized directory structure.
Getting Scala 2.8.1 ...
:: retrieving :: org.scala-tools.sbt#boot-scala
confs: [default]
2 artifacts copied, 0 already retrieved (15118kB/386ms)
[info] Building project sample 1.0-SNAPSHOT against Scala 2.8.1
[info]    using sbt.DefaultProject with sbt 0.7.4 and Scala 2.7.7
>
```

Note that it’s good form to start out with a SNAPSHOT version of your project.

Project Layout

- `project` – project definition files
  - `project/build/` *yourproject* `.scala` – the main project definition file
  - `project/build.properties` – project, sbt and scala version definitions
- `src/main` – your app code goes here, in a subdirectory indicating the code's language (e.g. `src/main/scala`, `src/main/java`)
- `src/main/resources` – static files you want added to your jar (e.g. logging config)
- `src/test` – like `src/main`, but for tests
- `lib_managed` – the jar files your project depends on. Populated by sbt update
- `target` – the destination for generated stuff (e.g. generated thrift code, class files, jars)

Adding Some Code

We'll be creating a simple JSON parser for simple tweets. Add the following code to `src/main/scala/com/twitter/sample/SimpleParser.scala`

```
package com.twitter.sample

case class SimpleParsed(id: Long, text: String)

class SimpleParser {

  val tweetRegex = "\"id\":(.*),\"text\":\"(.*)\"".r

  def parse(str: String) = {
    tweetRegex.findFirstMatchIn(str) match {
      case Some(m) => {
        val id = str.substring(m.start(1), m.end(1)).toInt
        val text = str.substring(m.start(2), m.end(2))
        Some(SimpleParsed(id, text))
      }
      case _ => None
    }
  }
}
```

This is ugly and buggy, but should compile.

Testing in the Console

SBT can be used both as a command line script and as a build console. We'll be primarily using it as a build console, but most commands can be run standalone by passing the command as an argument to SBT, e.g.

```
sbt test
```

Note that if a command takes arguments, you need to quote the entire argument path, e.g.

`sbt 'test-only com.twitter.sample.SampleSpec'`

It's weird that way.

Anyway. To start working with our code, launch sbt

```
[local ~/projects/sbt-sample]$ sbt
[info] Building project sample 1.0-SNAPSHOT against Scala 2.8.1
[info]    using sbt.DefaultProject with sbt 0.7.4 and Scala 2.7.7
>
```

SBT allows you to start a Scala REPL with all your project dependencies loaded. It compiles your project source before launching the console, providing us a quick way to bench test our parser.

```
> console
[info]
[info] == compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
```

```
[info] Compiling main sources...
[info] Nothing to compile.
[info]   Post-analysis: 3 classes.
[info] == compile ==
[info]
[info] == copy-test-resources ==
[info] == copy-test-resources ==
[info]
[info] == test-compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling test sources...
[info] Nothing to compile.
[info]   Post-analysis: 0 classes.
[info] == test-compile ==
[info]
[info] == copy-resources ==
[info] == copy-resources ==
[info]
[info] == console ==
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_22).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Our code has compiled, and we're provide the typical Scala prompt. We'll create a new parser, an exemplar tweet, and ensure it "works"

```
scala> import com.twitter.sample._
import com.twitter.sample._

scala> val tweet = """{"id":1,"text":"foo"}"""
tweet: java.lang.String = {"id":1,"text":"foo"}

scala> val parser = new SimpleParser
parser: com.twitter.sample.SimpleParser = com.twitter.sample.SimpleParser@71060c3e

scala> parser.parse(tweet)
res0: Option[com.twitter.sample.SimpleParsed] = Some(SimpleParsed(1,"foo"}))

scala>
```

Adding Dependencies

Our simple parser works for this very small set of inputs, but we want to add tests and break it. The first step is adding the specs test library and a real JSON parser to our project. To do this we have to go beyond the default SBT project layout and create a project.

SBT considers Scala files in the project/build directory to be project definitions. Add the following to project/build/SampleProject.scala

```
import sbt._

class SampleProject(info: ProjectInfo) extends DefaultProject(info) {
  val jackson = "org.codehaus.jackson" % "jackson-core-asl" % "1.6.1"
  val specs = "org.scala-tools.testing" % "specs_2.8.0" % "1.6.5" % "test"
}
```

A project definition is an SBT class. In our case we extend SBT's DefaultProject.

You declare dependencies by specifying a val that is a dependency. SBT uses reflection to scan all the dependency vals in your project and build up a dependency tree at build time. The syntax here may be new, but this is equivalent to the maven dependency

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-core-asl</artifactId>
  <version>1.6.1</version>
</dependency>
<dependency>
  <groupId>org.scala-tools.testing</groupId>
  <artifactId>specs_2.8.0</artifactId>
  <version>1.6.5</version>
  <scope>test</scope>
</dependency>
```

Now we can pull down dependencies for our project. From the command line (not the sbt console), run sbt update

```
[local ~/projects/sbt-sample]$ sbt update
[info] Building project sample 1.0-SNAPSHOT against Scala 2.8.1
[info]    using SampleProject with sbt 0.7.4 and Scala 2.7.7
[info]
[info] == update ==
[info] :: retrieving :: com.twitter#sample_2.8.1 [sync]
[info] confs: [compile, runtime, test, provided, system, optional, sources, javadoc]
[info] 1 artifacts copied, 0 already retrieved (2785kB/71ms)
[info] == update ==
[success] Successful.
[info]
[info] Total time: 1 s, completed Nov 24, 2010 8:47:26 AM
[info]
[info] Total session time: 2 s, completed Nov 24, 2010 8:47:26 AM
[success] Build completed successfully.
```

You'll see that sbt retrieved the specs library. You'll now also have a lib_managed directory, and lib_managed/scala_2.8.1/test will have specs_2.8.0-1.6.5.jar

Adding Tests

Now that we have a test library added, put the following code in src/test/scala/com/twitter/sample/SimpleParserSpec.scala

```
package com.twitter.sample

import org.specs._

object SimpleParserSpec extends Specification {
  "SimpleParser" should {
    val parser = new SimpleParser()
    "work with basic tweet" in {
      val tweet = """{"id":1,"text":"foo"}"""
      parser.parse(tweet) match {
        case Some(parsed) => {
          parsed.text must be_==("foo")
          parsed.id must be_==(1)
        }
        case _ => fail("didn't parse tweet")
      }
    }
  }
}
```

In the sbt console, run test

```
> test
[info]
[info] == compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling main sources...
[info] Nothing to compile.
[info]   Post-analysis: 3 classes.
[info] == compile ==
[info]
[info] == test-compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling test sources...
[info] Nothing to compile.
[info]   Post-analysis: 10 classes.
[info] == test-compile ==
[info]
[info] == copy-test-resources ==
[info] == copy-test-resources ==
[info]
[info] == copy-resources ==
[info] == copy-resources ==
[info]
[info] == test-start ==
[info] == test-start ==
[info]
[info] == com.twitter.sample.SimpleParserSpec ==
[info] SimpleParserSpec
[info] SimpleParser should
[info]   + work with basic tweet
[info] == com.twitter.sample.SimpleParserSpec ==
[info]
[info] == test-complete ==
[info] == test-complete ==
[info]
[info] == test-finish ==
[info] Passed: : Total 1, Failed 0, Errors 0, Passed 1, Skipped 0
[info]
[info] All tests PASSED.
[info] == test-finish ==
[info]
[info] == test-cleanup ==
[info] == test-cleanup ==
[info]
[info] == test ==
[info] == test ==
[success] Successful.
[info]
[info] Total time: 0 s, completed Nov 24, 2010 8:54:45 AM
>
```

Our test works! Now we can add more. One of the nice things SBT provides is a way to run triggered actions. Prefacing an action with a tilde starts up a loop that runs the action any time source files change. Lets run ~test and see what happens.

```
[info] == test ==
[success] Successful.
[info]
[info] Total time: 0 s, completed Nov 24, 2010 8:55:50 AM
1. Waiting for source changes... (press enter to interrupt)
```

Now let’s add the following test cases

```
    "reject a non-JSON tweet" in {
      val tweet = """"id":1,"text":"foo""""
      parser.parse(tweet) match {
        case Some(parsed) => fail("didn't reject a non-JSON tweet")
        case e => e must be_==(None)
      }
    }

    "ignore nested content" in {
      val tweet = """{"id":1,"text":"foo","nested":{"id":2}}"""
      parser.parse(tweet) match {
        case Some(parsed) => {
          parsed.text must be_==("foo")
          parsed.id must be_==(1)
        }
        case _ => fail("didn't parse tweet")
      }
    }

    "fail on partial content" in {
      val tweet = """{"id":1}"""
      parser.parse(tweet) match {
        case Some(parsed) => fail("didn't reject a partial tweet")
        case e => e must be_==(None)
      }
    }
```

After we save our file, SBT detects our changes, runs tests, and informs us our parser is lame

```
[info] == com.twitter.sample.SimpleParserSpec ==
[info] SimpleParserSpec
[info] SimpleParser should
[info]   + work with basic tweet
[info]   x reject a non-JSON tweet
[info]     didn't reject a non-JSON tweet (Specification.scala:43)
[info]   x ignore nested content
[info]     'foo","nested":{"id' is not equal to 'foo' (SimpleParserSpec.scala:31)
[info]   + fail on partial content
```

So let's rework our JSON parser to be real

```
package com.twitter.sample

import org.codehaus.jackson._
import org.codehaus.jackson.JsonToken._

case class SimpleParsed(id: Long, text: String)

class SimpleParser {

  val parserFactory = new JsonFactory()

  def parse(str: String) = {
    val parser = parserFactory.createJsonParser(str)
    if (parser.nextToken() == START_OBJECT) {
      var token = parser.nextToken()
      var textOpt:Option[String] = None
      var idOpt:Option[Long] = None
      while(token != null) {
        if (token == FIELD_NAME) {
          parser.getCurrentName() match {
            case "text" => {
              parser.nextToken()
              textOpt = Some(parser.getText())
            }
            case "id" => {
              parser.nextToken()
              idOpt = Some(parser.getLongValue())
            }
            case _ => // noop
          }
        }
        token = parser.nextToken()
      }
      if (textOpt.isDefined && idOpt.isDefined) {
        Some(SimpleParsed(idOpt.get, textOpt.get))
      } else {
        None
      }
    } else {
      None
    }
  }
}
```

This is a simple Jackson parser. When we save, SBT recompiles our code and reruns our tests. Getting better!

```
info] SimpleParser should
[info]   + work with basic tweet
[info]   + reject a non-JSON tweet
[info]   x ignore nested content
[info]     '2' is not equal to '1' (SimpleParserSpec.scala:32)
[info]   + fail on partial content
[info] == com.twitter.sample.SimpleParserSpec ==
```

Uh oh. We need to check for nested objects. Let's add some ugly
guards to our token reading loop.

```
  def parse(str: String) = {
    val parser = parserFactory.createJsonParser(str)
```

```
      var nested = 0
      if (parser.nextToken() == START_OBJECT) {
        var token = parser.nextToken()
        var textOpt:Option[String] = None
        var idOpt:Option[Long] = None
        while(token != null) {
          if (token == FIELD_NAME && nested == 0) {
            parser.getCurrentName() match {
              case "text" => {
                parser.nextToken()
                textOpt = Some(parser.getText())
              }
              case "id" => {
                parser.nextToken()
                idOpt = Some(parser.getLongValue())
              }
              case _ => // noop
            }
          } else if (token == START_OBJECT) {
            nested += 1
          } else if (token == END_OBJECT) {
            nested -= 1
          }
          token = parser.nextToken()
        }
        if (textOpt.isDefined && idOpt.isDefined) {
          Some(SimpleParsed(idOpt.get, textOpt.get))
        } else {
          None
        }
      } else {
        None
      }
    }
```

And… it works!

Packaging and Publishing

At this point we can run the package command to generate a jar file. However we may want to share our jar with other teams. To do this we'll build on StandardProject, which gives us a big head start.

The first step is include StandardProject as an SBT plugin. Plugins are a way to introduce dependencies to your build, rather than your project. These dependencies are defined in project/plugins/Plugins.scala. Add the following to the Plugins.scala file.

```
import sbt._

class Plugins(info: ProjectInfo) extends PluginDefinition(info) {
  val twitterMaven = "twitter.com" at "http://maven.twttr.com/"
  val defaultProject = "com.twitter" % "standard-project" % "0.7.14"
}
```

Note that we've specified a maven repository as well as a dependency. That's because the standard project library is hosted by us, which isn't one of the default repos sbt checks.

We'll also update our project definition to extend StandardProject, include an SVN publishing trait, and define the repository we wish to publish to. Alter SampleProject.scala to the following

```
import sbt._
import com.twitter.sbt._

class SampleProject(info: ProjectInfo) extends StandardProject(info) with SubversionPublisher {
  val jackson = "org.codehaus.jackson" % "jackson-core-asl" % "1.6.1"
  val specs = "org.scala-tools.testing" % "specs_2.8.0" % "1.6.5" % "test"

  override def subversionRepository = Some("http://svn.local.twitter.com/maven/")
}
```

Now if we run the publish action we'll see the following

```
[info] == deliver ==
IvySvn Build-Version: null
IvySvn Build-DateTime: null
[info] :: delivering :: com.twitter#sample;1.0-SNAPSHOT :: 1.0-SNAPSHOT :: release :: Wed Nov 24 10:26:45 PST 2010
[info] delivering ivy file to /Users/mmcbride/projects/sbt-sample/target/ivy-1.0-SNAPSHOT.xml
[info] == deliver ==
[info]
[info] == make-pom ==
[info] Wrote /Users/mmcbride/projects/sbt-sample/target/sample-1.0-SNAPSHOT.pom
[info] == make-pom ==
[info]
[info] == publish ==
[info] :: publishing :: com.twitter#sample
[info] Scheduling publish to http://svn.local.twitter.com/maven/com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.jar
[info] published sample to com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.jar
[info] Scheduling publish to http://svn.local.twitter.com/maven/com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.pom
[info] published sample to com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.pom
[info] Scheduling publish to http://svn.local.twitter.com/maven/com/twitter/sample/1.0-SNAPSHOT/ivy-1.0-SNAPSHOT.xml
[info] published ivy to com/twitter/sample/1.0-SNAPSHOT/ivy-1.0-SNAPSHOT.xml
[info] Binary diff deleting com/twitter/sample/1.0-SNAPSHOT
[info] Commit finished r977 by 'mmcbride' at Wed Nov 24 10:26:47 PST 2010
[info] Copying from com/twitter/sample/.upload to com/twitter/sample/1.0-SNAPSHOT
[info] Binary diff finished : r978 by 'mmcbride' at Wed Nov 24 10:26:47 PST 2010
[info] == publish ==
[success] Successful.
[info]
[info] Total time: 4 s, completed Nov 24, 2010 10:26:47 AM
```

And (after some time), we can go to binaries.local.twitter.com to see our published jar.

Adding Tasks

Tasks are Scala functions. The simplest way to add a task is to include a val in your project definition using the task method, e.g.

```
lazy val print = task {log.info("a test action"); None}
```

If you want dependencies and a description you can add them like this

```
lazy val print = task {log.info("a test action"); None}.dependsOn(compile) describedAs("prints a line after compile")
```

If we reload our project and run the print action we'll see the following

```
> print
[info]
[info] == print ==
[info] a test action
[info] == print ==
[success] Successful.
[info]
[info] Total time: 0 s, completed Nov 24, 2010 11:05:12 AM
>
```

So it works. If you're defining a task in a single project this works just fine. However if you're defining this in a plugin it's fairly inflexible. I may want to

```
lazy val print = printAction
def printAction = printTask.dependsOn(compile) describedAs("prints a line after compile")
def printTask = task {log.info("a test action"); None}
```

This allows consumers to override the task itself, the dependencies and/or description of the task, or the action. Most built in SBT actions follow this pattern. As an example, we can modify the builtin package task to print the current timestamp by doing the following

```
lazy val printTimestamp = task { log.info("current time is " + System.currentTimeMillis); None}
override def packageAction = super.packageAction.dependsOn(printTimestamp)
```

There are many examples in StandardProject of tweaking SBT defaults and adding custom tasks.

Quick Reference

Common Commands

- actions – show actions available for this project
- update – downloads dependencies
- compile – compiles source
- test – runs tests
- package – creates a publishable jar file
- publish-local – installs the built jar in your local ivy cache
- publish – pushes your jar to a remote repo (if configured)

Moar Commands

- test-failed – run any specs that failed
- test-quick – run any specs that failed and/or had dependencies updated
- clean-cache – remove all sorts of sbt cached stuff. Like clean for sbt
- clean-lib – remove everything in lib_managed

Project Layout

TBD

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;