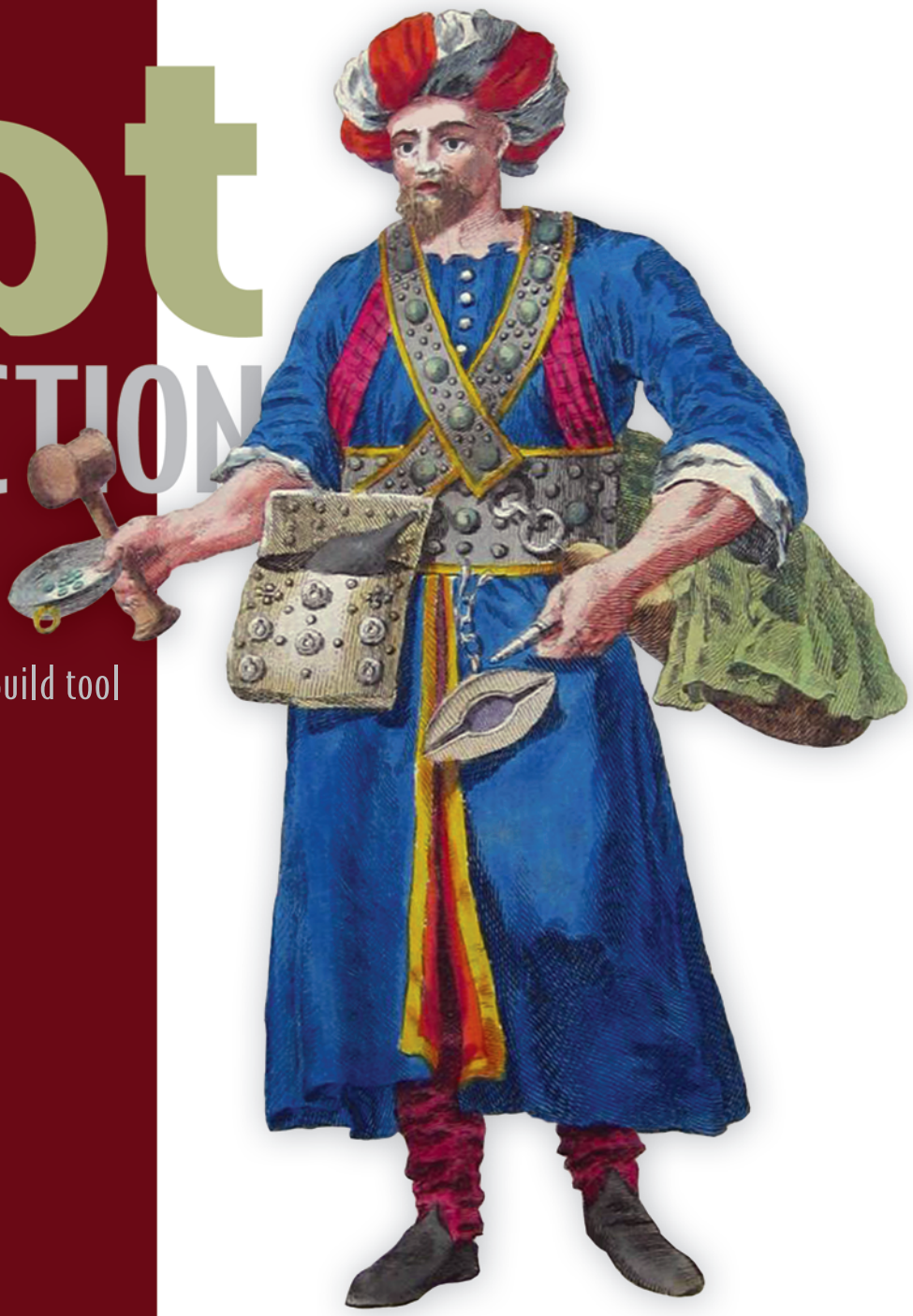
SAMPLE CHAPTER

sbt IN ACTION



The simple Scala build tool

Joshua Suereth
Matthew Farwell

MANNING

sbt in Action

by Joshua Suereth
Matthew Farwell

Chapter 5

brief contents

PART 1 WHY SBT? 1

1 ■ Why sbt? 3
2 ■ Getting started 24

PART 2 UNDERSTANDING SBT'S CORE CONCEPTS 41

3 ■ Core concepts 43
4 ■ The default build 66

PART 3 WORKING WITH SBT 85

5 ■ Testing 87
6 ■ The IO and Process libraries 110
7 ■ Accepting user input 128
8 ■ Using plugins and external libraries 146
9 ■ Debugging your build 162

PART 4 EXTENDING SBT 181

10 ■ Automating workflows with commands 183
11 ■ Defining a plugin 198

PART 5 DEPLOYING YOUR PROJECTS 223

12 ■ Distributing your projects 225

5 Testing

This chapter covers

- Configuring your build to use specs2
- Learning how to run JUnit tests
- Incorporating external libraries and code into your testing
- Using the ScalaCheck library to improve your testing experience
- Incorporating Selenium HTML tests using the ScalaTest Selenium DSL

In the previous chapters, you set up your `preowned-kittens` project and learned about the default build. Obviously, you're quality conscious, so you want to do as much testing as possible. In this chapter you'll learn how to use the power and interactivity of sbt to make the development and testing cycle a pleasant experience. You're also eclectic in your choice of testing frameworks, so we'll show you how to set up four: specs2, ScalaCheck, JUnit, and ScalaTest. Each has its strengths and weaknesses; thus you can use each of these frameworks for testing different aspects of the project. We'll show you how to configure each framework separately.

You also want to demonstrate to potential customers that you'll take good care of their kittens and that they can trust you. How can you show this? Well, you'll

include on your site a report of the tests that you've run as part of the development cycle. This will help persuade potential clients. To generate the reports, you'll have to fork your tests to run them in a different JVM from sbt. Because there isn't a built-in solution for reporting with JUnit, you'll incorporate our custom code into the build to get the reports you want.

5.1 Configuring specs2 with sbt

You'll recall from chapter 2 that you have a set of specs2 specifications for your testing:

```
object LogicSpec extends Specification {
  "The 'matchLikelihood' method" should {
    "be 100% when all attributes match" in {
      val tabby = Kitten(1, Set("male", "tabby"))
      val prefs = BuyerPreferences(Set("male", "tabby"))
      Logic.matchLikelihood(tabby, prefs) must beGreaterThan(0.999)
    }
    "be 0% when no attributes match" in {
      val tabby = Kitten(1, Set("male", "tabby"))
      val prefs = BuyerPreferences(Set("female", "calico"))
      val result = Logic.matchLikelihood(tabby, prefs)
      result must beLessThan(0.001)
    }
  }
}
```

All that you need for sbt to compile and run these tests is the following in build.sbt:

```
libraryDependencies += "org.specs2" %% "specs2" % "1.14" % "test"
```

If you run `sbt test`, you'll get the following output:

```
[info] LogicSpec
[info]
[info] The 'matchLikelihood' method should
[info] + be 100% when all attributes match
[info] + be 0% when no attributes match
[info]
[info]
[info] Total for specification LogicSpec
[info] Finished in 11 ms
[info] 3 examples, 0 failure, 0 error
[info]
[info] Passed: Total 3, Failed 0, Errors 0, Passed 4
[success] Total time: 1 s, completed 11-Jan-2015 21:26:40
```

Already you're doing well as far as the build itself is concerned. But sbt isn't just a build tool; it can help with the development process as well. Let's start at the beginning by looking in depth at three tasks: `test`, `testOnly`, and `testQuick`. From the command line, you can run any one of these tasks:

```
$ sbt test
$ sbt testOnly org.preownedkittens.LogicSpec
$ sbt testQuick
```

`test` runs all of the tests that sbt can find. `testOnly` runs only those tests specified on the command line. You can have wildcards in there; for instance:

```
$ sbt testOnly *Logic*
```

This will run all of the tests that contain the string "Logic." Finally, `testQuick` runs all of the tests that (1) failed in the previous run, (2) haven't yet been run, or (3) depend on code that has changed.

You can use sbt to improve the development experience by writing your tests, running just those tests from the command line, and then changing the code so that the tests pass, rerunning the tests after every save.

This works well, but you can do even better. If you prefix an sbt command with ~, sbt will wait in a loop, looking for changed files. If it detects a changed file, it will rerun the task, along with all of its dependencies. This works through the command line as well as the console. Let's see. Start the sbt console and then type `~test`, as shown in the following listing.

Listing 5.1 Running the `~test` task

```
$ sbt
[info] Loading project definition from .../chapter5/project
[info] Set current project to preowned-kittens (in build ...)
> ~test
[info] LogicSpec                                  <-- Runs tests continuously
[info]
[info] The 'matchLikelihood' method should
[info] + be 100% when all attributes match
[info] + be 0% when no attributes match           <-- Results of tests
[info]
[info] Total for specification LogicSpec
[info] Finished in 15 ms
[info] 2 examples, 0 failure, 0 error
[info]                                                  Waiting for
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2    file changes
[success] Total time: 2 s, completed 15-May-2015 09:45:35
1. Waiting for source changes... (press enter to interrupt)
```

You can see that it has run the tests, and then, on the last line, it's waiting for you to change the source code. If you save one of the source files again, it will rerun all of the tests. This is extremely useful when you're in the middle of a code/test cycle, because you get immediate feedback from your tests without having to run them each time.

Now, running all of the tests may take some time, so you may wish to run only a subset. You can do this with `testOnly`, in exactly the same manner, as shown in the next listing.

Listing 5.2 Running the `~testOnly` task

```
> ~testOnly *Logic*          <-- Run only *Logic* tests
[info] LogicSpec
[info]
```

```
[info] The 'matchLikelihood' method should
[info] + be 100% when all attributes match
[info] + be 0% when no attributes match
[info]
[info] Total for specification LogicSpec
[info] Finished in 14 ms
[info] 2 examples, 0 failure, 0 error
[info]
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 2 s, completed 15-May-2015 09:48:42
1. Waiting for source changes... (press enter to interrupt)
```

In addition to using wildcards, you can autocomplete the test names by using the Tab key. If you enter

```
> ~testOnly org<TAB>
```

then sbt will autocomplete the test name to the full name of the test:

```
> ~testOnly org.preownedkittens.LogicSpec
```

Testing tasks

There are three main testing tasks: `test`, `testOnly`, and `testQuick`.

`test` runs all of the tests sbt can find in your project.

`testOnly <testname1>` runs only the test(s) specified on the command line. You can use wildcards, (*) and if you're running in the sbt console, you can use autocomplete by pressing <TAB>.

`testQuick` runs the tests that have previously failed, that haven't already been run, or that depend on code that has been recompiled. You can also specify a test.

5.1.1 Reports and forking tests

Now, as part of your site, you want to incorporate your test reports so that the kitten owners will have more confidence in the site.

specs2 can generate an HTML report of the tests you've run, so you'll use that as a basis of your reports. First you need to generate the HTML; this is done by specifying the `html` option to specs2 in build.sbt:

```
testOptions in Test += Tests.Argument("html")
```

specs2 uses pegdown, an HTML generator library, to generate the HTML, so for this to work you need to add another dependency, `"org.pegdown" % "pegdown" % "1.0.2"`. Note that you're adding a test configuration:

```
libraryDependencies += "org.specs2" %% "specs2" % "1.14" % "test"

libraryDependencies += "org.pegdown" % "pegdown" % "1.0.2" % "test"
```

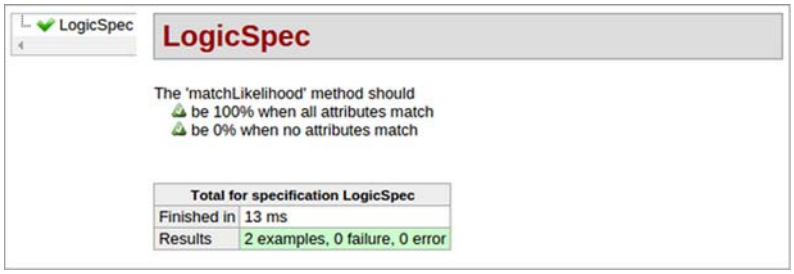


Figure 5.1 The HTML for the target/specs2-reports/LogicSpec.html report

When you reload and run the tests again, this produces a nice HTML report in target/specs2-reports/LogicSpec.html, along with the images and CSS required; see figure 5.1.

> **Remember to reload when changing build definition**
>
> Whenever you change the build.sbt file (or indeed any of the other files that contribute to the sbt build definition), you need to reload to get the changes recognized by sbt:
>
> ```
> > reload
> [info] Loading project definition from /home/mfarwell/code/sbt/sbt-in-
>             action-examples/chapter5/project
> [info] Set current project to preowned-kittens (in build file:/home/
> mfarwell/code/sbt/sbt-in-action-examples/chapter5/)
> ```

These files will now be generated each time you run the tests because by default the tests are run as part of the build. In real life, you'd publish these somewhere as part of your build.

To include them as part of your website, you want to change the output directory so that you can pick them up later when you build the final site to deploy. To change the output directory of specs2, you can specify a Java system property

```
-Dspecs2.outDir=<directory>
```

and it will create the files in there. You could achieve this by adding a custom task called `System.setProperty("specs2.outDir", "/something")`, which is executed before the test, and then a `System.clearProperty("specs2.outDir")` after. This solution is fairly complex and changes the running sbt environment. Although this doesn't matter in this use case, it's still not very nice. For a better solution, you should run the tests in a different instance of the JVM, and then you can specify parameters to that JVM. In sbt, this is called *forking* the JVM. You can do this for various tasks, such as compiling and testing. And obviously running the application forks the JVM. In sbt, you can use the `javaOptions` setting to specify the options to the new JVM:

```
javaOptions in Test += "-Dspecs2.outDir=target/generated/test-reports"
```

This will create a specific directory, named generated, to put your generated files in so that everything is in one place. You'll use this again later for other generated files. Note that you're putting the generated files in the target directory so that it will be cleaned up when you execute a `clean`. `javaOptions` can be applied to the `test` and `run` tasks, and you can specify anything you're able to on the Java command line; for instance:

```
javaOptions in run += "-Xmx2048m" // we need lots of heap space
```

If you specify the `run` scope, then this will apply to both the `run` and `run-main` tasks (`run-main` allows you to select the class to run, whereas `run` selects the class for you). But `javaOptions` won't work without forking:

```
fork in Test := true
```

> **Forking your JVM**
>
> When you specify `javaOptions`, you must set `fork` to be `true` for it to be taken into account. If you don't, you'll be pulling your hair out when it doesn't work. `fork` doesn't always need to be `true` for all tasks. In the previous case you're setting it only for the testing tasks. One task that you don't need to fork for is compilation. sbt does the necessary steps for these to work correctly, and you don't need to fork to compile.

Again, the `fork` setting can apply to the `run`, `run-main`, and `test` tasks, with `run` and `run-main` sharing the same setting (`run`).

There's one more thing you need to watch out for here: you shouldn't be hard-coding `target` in your `-D` string; this is bad practice, because this value can change. `target` is the directory where sbt puts all of its work, but this isn't fixed; it's actually a setting. If you hard-coded `target`, someone could change it like this:

```
target := file(baseDirectory.value / "foobar")
```

Then `sbt clean` wouldn't clean up your generated files. To make sure you always use the correct value, use the value of the `target` setting:

```
javaOptions in Test += "-Dspecs2.outDir=" + (target.value / "generated/test-
     reports").getAbsolutePath
```

The `target.value` actually returns a file (not a `String`). A file has a method called `/`, which appends the argument to the name of the file and produces another file. Because the current working directory isn't necessarily the base directory, either, call `getAbsolutePath` to avoid problems with relative paths.

What have you done? You wanted to generate HTML reports for your tests, so you configured specs2 to do that. The default output directory wasn't good enough, so you changed it to target/generated/test-reports by forking your JVM for the tests and setting the `system` property for the output directory. Now, for each subproject using this setting, its target directory will have the generated specs2 tests.

Forking processes

Most of the time you don't need to fork, but there are four main reasons why you may want to fork:

- *New JVM requires different parameters*—If you want to change the memory used by the new JVM or change the JVM itself, you'll need to fork. A common use case is to add a `-D` option, as you've done.
- `System.exit()`—If your code calls `System.exit()`, this normally shuts down the JVM. Most of the time sbt copes with this, but there are certain situations where it doesn't.
- *Threads*—If your code creates a lot of new threads, and these threads are not tidied before the main method returns, then this can cause problems. For instance, a GUI using Swing creates a number of threads. In general, these don't terminate until the JVM itself terminates.
- *Class loading*—If you're using class loaders, or if you're deserializing for any reason, this can cause issues. Note that it may not be you who is doing the class loading but a library that you're using, such as scalate. Scalate is a template engine that creates HTML from a template. It does this by creating Scala files and then compiling and loading the classes. This can cause problems with PermGen in some JVMs.

5.1.2 Digging deeper: other options for forking

sbt provides other options when you're forking your processes. Note that for these options to work, you need to have `fork := true`.

Changing the JVM

You can also specify a Java installation by using the `javaHome` setting. This is the directory in which the Java installation is found:

```
javaHome := file("/path/to/jre")
```

This doesn't change sbt itself. You can also specify the configuration for the `run` and/or `test` tasks.

Changing the working directory

When a task forks, you can set the working directory for the forked JVM:

```
baseDirectory := file("/working/directory")
baseDirectory in (Compile,run) := file("/working/directory")
```

Note, again, that this doesn't affect sbt itself, just the new JVM. Again, you can specify the configuration for the `run` and `test` tasks for different configurations.

Input and output

When you fork a process, you can change where the output goes to and where input is read from. You do this using the `outputStrategy` setting:

```
outputStrategy := Some(CustomOutput(new java.io.FileOutputStream("/tmp/
     run.log")))
```

By default, all standard output is logged to the sbt console at the Info level, and all standard errors are logged at the Error level. There are any number of options for output. Here's how to send all output (standard out and error) to sbt standard out (not the logger):

```
outputStrategy := Some(StdoutOutput)
```

Finally, if you want your task to wire the standard input of the new process into the standard input for sbt—for example, if you want to ask a question of the user—you can use `connectInput`:

```
connectInput in run := true
```

5.2 JUnit and using custom code

The preowned-kittens.com website wasn't actually a new project when you inherited it. Initially it was written in Java, but the process was taking so long that the original kittens became cats. But as a leftover from the first version, you inherited a number of legacy tests that were written in Java, using the JUnit testing framework. So as not to waste that effort, you decided to keep these tests around and run them against the new Scala code.

You'll recall from chapter 2 that sbt can compile Java, so all you have to do is include the files in the correct places in the source tree, which are src/main/java and src/test/java. You need to link these tests into your build. You do this by adding two dependencies into your build. The first is JUnit itself:

```
libraryDependencies += "junit" % "junit" % "4.11" % "test"
```

As an example test, use the file shown in the following listing.

Listing 5.3 Failing JUnit test

```
package org.preownedkittens;

import org.junit.*;
import scala.collection.immutable.*;

public class LogicJavaTest {
    @Test
    public void testKitten() {
        Kitten kitten = new Kitten(1, new HashSet());
        Assert.assertEquals(1,
     kitten.attributes().size());                    <-- (1) This fails!
    }
}
```

Add this in, run `sbt test`, and all of your tests will pass. Which they shouldn't, because you have a failing test (1). In fact, sbt isn't even running the Java test. Why? As you've seen, sbt "knows" how to run certain test frameworks out of the box. But how does it

know this? sbt defines a `test-interface`, which allows sbt (1) to find the list of classes to run as tests, and (2) to run those tests. JUnit doesn't know about this interface.

> **The test-interface of sbt**
>
> sbt supports, by default, ScalaTest, ScalaCheck, and specs2. This is because all of those test frameworks include in their jars a class that implements the sbt `test-interface` classes. JUnit does not, because it's not a Scala testing framework; it's a Java one.

In order to run your JUnit tests, you need to define an sbt `test-interface` for JUnit. Fortunately, someone has already done it for you, and all you need to do is add it to the dependencies for your project. It's called `junit-interface`:

```
libraryDependencies += "junit" % "junit" % "4.11" % "test" // already added
libraryDependencies += "com.novocode" % "junit-interface" % "0.11" % "test"
```

Now when you run your tests, they fail as expected:

```
$ sbt "testOnly org.preownedkittens.LogicJavaTest"
[info] Loading project definition from ...
[info] Set current project to preowned-kittens ...
[info] Test run started
[info] Test org.preownedkittens.LogicJavaTest.testKitten started
[error] Test org.preownedkittens.LogicJavaTest.testKitten failed:
     expected:<1> but was:<0>, took 0.045 sec
[info] Test run finished: 1 failed, 0 ignored, 1 total, 0.053s
[error] Failed: Total 1, Failed 1, Errors 0, Passed 0
[error] Failed tests:
[error]     org.preownedkittens.LogicJavaTest
[error] (analytics/test:testOnly) sbt.TestsFailedException: Tests
     unsuccessful
[error] Total time: 1 s, completed 14-Jan-2015 23:32:02
```

Correct the test, and everybody is happy. You've now incorporated your JUnit tests into your build.

5.2.1 Report generation with JUnit

You've generated HTML reports with specs2, but can you do this with JUnit? There isn't an easy way to have your reports, like in specs2. But the previous project owners produced reports from their JUnit tests, using a `RunListener` class that they defined. A `RunListener` is a JUnit-defined class with a defined set of methods that are called when tests start, finish, or fail. It looks like this:

```
public class RunListener {
    public void testRunStarted(Description description) throws Exception { }
    public void testRunFinished(Result result) throws Exception { }
    public void testStarted(Description description) throws Exception { }
    public void testFinished(Description description) throws Exception { }
    public void testFailure(Failure failure) throws Exception { }
```

```
    public void testAssumptionFailure(Failure failure) { }
    public void testIgnored(Description description) throws Exception { }
}
```

Each of these methods is called on a specific event. For instance, `testStarted` is called before each JUnit test method, and `testRunFinished` is called once, at the end of all tests.

This is the Java class that was defined, which gives you a basic HTML report:

```
package com.preownedkittens.sbt;

import org.junit.*;
import java.io.*;
import org.junit.runner.*;
import org.junit.runner.notification.*;

public class JUnitListener extends RunListener {
    private PrintWriter pw;
    private boolean testFailed;
    private String outputFile = System.getProperty("junit.output.file");

    public void testRunStarted(Description description) throws Exception {
        pw = new PrintWriter(new FileWriter(outputFile));
        pw.println("<html><head><title>JUnit report</title></head><body>");
    }
    public void testRunFinished(Result result) throws Exception {
        pw.println("</body></html>");
        pw.close();
    }
    public void testStarted(Description description) throws Exception {
        pw.print("<p> Test " + description.getDisplayName() + " ");
        testFailed = false;

    }
    public void testFinished(Description description) throws Exception {
        if (!testFailed) {
            pw.print("OK");
        }
        pw.println("</p>");
    }
    public void testFailure(Failure failure) throws Exception {
        testFailed = true;
        pw.print("FAILED!");
    }
    public void testAssumptionFailure(Failure failure) {
        pw.print("ASSUMPTION FAILURE");
    }
    public void testIgnored(Description description) throws Exception {
        pw.print("IGNORED");
    }
}
```

This isn't going to win any prizes for prettiness, but it does the job. It produces an HTML page like the one shown in figure 5.2.

Test testKitten(org.usedkittens.LogicJavaTest) FAILED! **Figure 5.2 JUnit HTML report**

Let's find a place to store this output. You need to be able to specify this `RunListener` to `junit-interface`. Most of the implementation details of the previous class are irrelevant here, but we need to cover two things. The first is how to tell the `RunListener` which file to output to, which is done through a system property:

```
private String outputFile = System.getProperty("junit.output.file");
```

As with specs2, this means that you need to add a `"-Djunit.output.file= " + (target .value / "generated/junit.html")` to your build.sbt and fork the tests as before:

```
javaOptions in Test += "-Djunit.output.file=" + (target.value / "generated/
     junit.html").getAbsolutePath
fork in Test := true
```

In your build.sbt, the `fork` setting is already `true` for your build, so you don't need to do it twice.

sbt settings are immutable

When we're talking about settings, the most recently defined setting wins. If you specify the same setting twice, the value that was defined last will be the one that's used. This can be confusing for those expecting a more line-by-line flow, where line *x* is executed, and then that value used in line *x* + 1; for example:

```
name := "preowned-kittens"
organization := name.value + " Inc"
name := "This is the one"
```

The final value of the `organization` setting is "This is the one Inc". Note that we aren't recommending that you specify a setting twice. As you can see, it can get very confusing. It's good practice to specify a setting only once.

Additionally, you need to specify the `RunListener` class to `junit-interface`. This needs to be done through adding entries to `testOptions`, the same as before when you were using specs2:

```
testOptions += Tests.Argument("--run-
     listener=com.preownedkittens.sbt.JUnitListener")
```

Now everything should work. But, as you may have spotted, there's a problem with using `testOptions` again. You now have these two lines in the build.sbt file:

```
testOptions += Tests.Argument("html")
testOptions += Tests.Argument("--run-
     listener=com.preownedkittens.sbt.JUnitListener")
```

The problem here is that you're adding two parameters to `testOptions`, but they're for two different testing libraries! You need to be able to differentiate between the testing libraries, sending only the specs2 options to specs2, and the `junit-interface` parameters to JUnit. Fortunately, sbt provides a way to do this—a delimiter for testing library options, which it calls a test framework:

```
testOptions += Tests.Argument(TestFrameworks.Specs2, "html")
testOptions += Tests.Argument(TestFrameworks.JUnit, "--run-
     listener=com.preownedkittens.sbt.JUnitListener")
```

sbt test frameworks

sbt defines five `TestFramework`s:

```
val ScalaCheck = new
        TestFramework("org.scalacheck.ScalaCheckFramework")
val ScalaTest = new
        TestFramework("org.scalatest.tools.ScalaTestFramework")
val Specs = new TestFramework("org.specs.runner.SpecsFramework")
val Specs2 = new TestFramework("org.specs2.runner.SpecsFramework")
val JUnit = new TestFramework("com.novocode.junit.JUnitFramework")
```

These are defined by sbt and can be used out of the box. But if you use a test framework that isn't defined here, you can define and create your own.

Now rerun your tests, and you'll get the target/generated/junit.html generated along with the target/generated/test-reports/* that you had for specs2.

5.3 ScalaCheck

ScalaCheck is a test framework that's designed for property-based testing. The main difference between a more traditional unit-testing framework and a property-based framework is that with a traditional framework, you have to provide the data with which to test your classes. With a property-based framework, it provides the data. You tell it what sort of data you want, and then it generates a set of data and runs the tests. You need to provide some code that asserts that a combination of data is correct. Let's look at an example. In chapter 2, you created a specs2 test to test the buyer-kitten-matching algorithm. It looked like this:

```
object LogicSpec extends Specification {
  "The 'matchLikelihood' method" should {
    "be 100% when all attributes match" in {
      val tabby = Kitten(1, Set("male", "tabby"))
      val prefs = BuyerPreferences(Set("male", "tabby"))
      Logic.matchLikelihood(tabby, prefs) must beGreaterThan(0.999)
    }
    ... // elided to save space
  }
}
```

These are good tests, but using only four test cases to cover the logic in this test seems a bit light. To up your confidence in the algorithm a bit, you can add tests for the same method using ScalaCheck, as shown in the next listing.

Listing 5.4 ScalaCheck property testers

```
package org.preownedkittens

import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck._

object LogicSpecification extends Properties("Logic") {
  val allAttributes = Array("Harlequin","Tortoiseshell","Siamese",
                "Alien","Rough","Tom","Sad","Overweight")

  val genKitten: Gen[Kitten] = for {
    attributes <- Gen.containerOf[Set,String](Gen.oneOf(allAttributes))
  } yield Kitten(1, attributes)

  val genBuyerPreferences: Gen[BuyerPreferences] = (for {
    attributes <- Gen.containerOf[Set,String](Gen.oneOf(allAttributes))
  } yield BuyerPreferences(attributes))

  def matches(x: String, a: Kitten) =
        if (a.attributes.contains(x)) 1.0 else 0.0

  property("matchLikelihood") = forAll(genKitten, genBuyerPreferences)
➥((a: Kitten, b: BuyerPreferences) => {
    if (b.attributes.size == 0) true
    else {
      val num = b.attributes.map{matches(_, a)}.sum
      num / b.attributes.size - Logic.matchLikelihood(a, b) < 0.001
    }
  })
}
```

1 Generator for kittens

2 Generator for buyer preferences

3 Property tester

The methods `genKitten` 1 and `genBuyerPreferences` 2 are the data generators. The work is done by 3, the property tester. This takes the two generators for the case classes you created and produces a partial function. This is called 100 times with 100 generated values. These are random values. The assertion in the `property` function is `true` if the implemented version agrees with the test version. If the generated values don't meet the assertion, the test fails.

If the test does fail, you'd normally take the failing instance and put it into another test format, such as specs2 or JUnit. This cycle is an example of how you could use the interactivity of sbt to help you pass the tests. Each time you run a ScalaCheck test, the data generated is different, so you could have the following development cycle:

- Add the property-based tests for a method.
- In sbt interactive mode, run `test` and look for failing tests.
- For each failing test, add the failing data to a specs2 test. Fix the bug and rerun the ScalaCheck test.
- If there aren't any failing tests, rerun ScalaCheck a couple of times to make sure.
- Stop when bored.

To achieve this, you can use the `~test` feature of sbt. This executes all of the tests each time one of the source files is changed. Try it:

```
$ sbt
> ~test
[info] > ARG_0: Kitten(1,List(Rough, Rough, Overweight))
[info] > ARG_1: BuyerPreferences(List(Alien, Tortoiseshell, Sad, Rough))
[info] ! Logic.matchLikelihood: Falsified after 15 passed tests.
[info] Test run finished: 1 failed, 0 ignored, 1 total, 0.0s
….
[error] Failed: : Total 9, Failed 5, Errors 0, Passed 4, Skipped 0
[error] Failed tests:
[error]         org.preownedkittens.LogicSpecification
[error] (test:test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 3 s, completed 24 mars 2013 17:56:29
1. Waiting for source changes... (press enter to interrupt)
```

We cut down the output here to make things readable. This is showing you that the method `matchLikelihood` is failing with the following data:

```
[info] > ARG_0: Kitten(id,List(Rough, Rough, Overweight))
[info] > ARG_1: BuyerPreferences(List(Alien, Tortoiseshell, Sad, Rough))
```

You've told ScalaCheck to use your attribute strings. This shows that you have a `Kitten` with three attributes and `BuyerPreferences` with four attributes. There's one immediate problem with the data that you're passing in: there are two `Rough` attributes. It doesn't make sense to have duplicated attributes in a `Kitten` (or indeed `BuyerPreferences`). This means that the data model is probably wrong. This shouldn't be a `List[String]` but instead a `Set[String]`. You have to fix your model, changing the `List` into a `Set`. Add a new test to your `LogicSpec` that gives multiple duplicate attributes to the `Kitten` and to the `BuyerPreference` class to make sure it doesn't happen again:

```
"be 100% when all attributes match (with duplicates)" in {
  val tabby = Kitten(1, Set("male", "tabby", "male"))
  val prefs = BuyerPreferences(Set("male", "tabby", "tabby"))
  Logic.matchLikelihood(tabby, prefs) must beGreaterThan(0.999)
}
```

Once the new model.scala and logic.scala are saved, the tests get rerun automatically because you're doing `~test`. Here's the output, again cut down:

```
> test
[info] ! Logic.matchLikelihood: Falsified after 7 passed tests.
[info] > ARG_0: Kitten(id,Set(Rough))
[info] > ARG_1: BuyerPreferences(Set(Harlequin, Rough))
[error] Failed: : Total 9, Failed 5, Errors 0, Passed 4, Skipped 0
[error] Failed tests:
[error]         org.preownedkittens.LogicSpecification
[error] (test:test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 3 s, completed 24 mars 2013 19:36:39
```

There's still a problem. There are no duplicates, but the algorithm is wrong somewhere. The problem is actually in the definition of `Logic.matchLikelihood`:

```
object Logic {
  /** Determines the match likelihood and returns % match. */
  def matchLikelihood(kitten: Kitten, buyer: BuyerPreferences): Double = {
    val matches = buyer.attributes.toList map { attribute =>
      kitten.attributes contains attribute
    }
    val nums = matches map { b => if(b) 1 else 0 } // (a)
    nums.sum / nums.size    // (b)
  }
}
```

The problem is the integer division at (b), which will always be either 1 or 0. Between chapter 2 and chapter 4 someone has introduced a regression at (a). These should be doubles, not integers. You must add a test to `LogicSpec`, using the test case provided by ScalaCheck:

```
"be 66% when two from three attributes match" in {
  val tabby = Kitten(1, Set("female", "calico", "overweight"))
  val prefs = BuyerPreferences(Set("female", "calico", "thin"))
  val result = Logic.matchLikelihood(tabby, prefs)
  result must beBetween(0.666, 0.667)
}
```

When the test is run, it fails:

```
[info] Compiling 1 Scala source to C:\code\sbt\sbt-in-action-
     examples\chapter5\target\scala-2.10\test-classes...
…
[error] Failed tests:
[error]         org.preownedkittens.LogicSpecification
[error]         org.preownedkittens.LogicSpec
[error] (test:test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 5 s, completed 24 mars 2013 20:03:15
```

And now you can fix the problem:

```
val nums = matches map { b => if(b) 1.0 else 0.0 } // (b)
```

This demonstrates some of the power of sbt as a development environment, in conjunction with the right kind of tests. Note that your tests are still not complete, but this gives you more confidence that you're going in the right direction. One more thing that could be added to your ScalaCheck tests is to use real attributes rather than autogenerated ones. This would increase the chances of finding problems with your logic with realistic data.

Another thing you can do is to augment the number of times that the property is tested. By default, ScalaCheck uses 100 different combinations. Because ScalaCheck isn't guaranteed to find all of your problems, it's probably a good idea to up the number and see if anything breaks. You can do this in one of two ways: through `testOptions` or on the command line. First, `testOptions`:

```
testOptions += Tests.Argument(TestFrameworks.ScalaCheck, "-s", "5000")
```

The `-s` is the minimum number of successful tests needed to have a passing test. This will run the test with 5000 sets of data.

Alternatively, if you're using the `test` or `testOnly` task, you can specify this on the command line:

```
> ~testOnly org.preownedkittens.LogicSpecification -- -s 5000
```

The `--` means that this is the end of the tests to run, and you're starting the options to pass to the test framework.

5.4 Integration testing

In this section you'll add integration tests to your build, which will be run at a different time than the unit tests that were written in the previous sections. We'll use the ScalaTest Selenium DSL to illustrate this, and, as we've done with the others, we'll incorporate the ScalaTest HTML reports.

5.4.1 ScalaTest and Selenium

Another commonly used Scala testing framework is ScalaTest. ScalaTest implements a number of different styles of testing, including specification-style testing like specs2, unit testing like JUnit, and even behavior-driven development-style testing. Which style you use depends on what you want to test and what stage of your project that you're at.

One of the recently added features of ScalaTest is the Selenium DSL (domain-specific language). Selenium is a tool that aids the testing of websites. It's available for a number of languages, including Java/Scala, Ruby, Python, and the .NET languages. Selenium works by starting a browser via what it calls a web driver and interacting with it, telling it to click this button or enter some text into this or that field. It can drive Microsoft Internet Explorer, Mozilla Firefox, and Google Chrome browsers, among others. One of the advantages of using Selenium is that you're interacting with the system as a user would interact; you're performing the same actions as an end user. The disadvantage is that you need to test the full stack, from the browser to the database. You need all of the pieces. For this reason, Selenium tests are generally considered to be integration tests.

You'll use the `FlatSpec` classes of ScalaTest. ScalaTest integrates Selenium through an internal DSL, so you're actually writing Scala code, but it turns out to be much more readable than normal Scala. This is mostly easier to write, and it can be useful when you're explaining the tests to a third party who doesn't know the code intimately. Let's see an example:

```
"Home page" should "redirect to kitten list" in {
    go to "http://localhost:9000"
    currentUrl should startWith ("http://localhost:9000/kittens")
}
```

The goal of the DSL is to make the code more readable and understandable, and this is a simple test, but there's quite a bit going on in this example. You can read the aim of the test by reading the first line. If the user goes to the bare URL (without the `/kittens`), then the user is redirected to the page http://localhost:9000/kittens. The first thing to note is that the example code is pure Scala, so you can do anything you normally would be able to in code; it just looks a bit more readable. Line 1 is pretty much plain text, which aids the description of the test. Line 2 opens the bare URL http://localhost:9000, the default page for the site, and then, when that action has been completed by the browser, it checks that the current URL is actually /kittens (line 3), so the website has redirected the user to this page. When you run this test, you get output like the following from sbt:

```
[info] SeleniumSpec:
[info] Home page
[info] - should redirect to kitten list
```

An aside about your site: it has three pages. The initial page looks like figure 5.3, which is a list of all kittens that you currently have on your books along with a form that allows the user to select three attributes that they want from a kitten.

When visitors click the Find Me A Kitten button, another page shows all of the kittens that match their selected attributes, and they can click the Purchase button, which adds the kitten to the basket.[1] See figure 5.4.

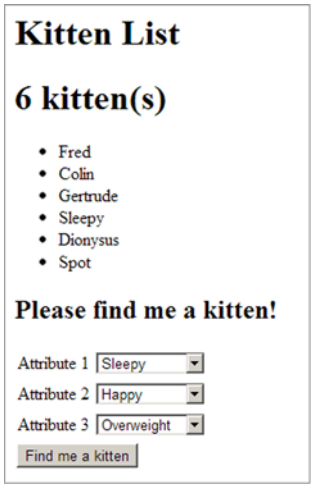


Figure 5.3 The initial page for your preowned-kittens site

Figure 5.4 Visitors to your site make their selections for purchase.

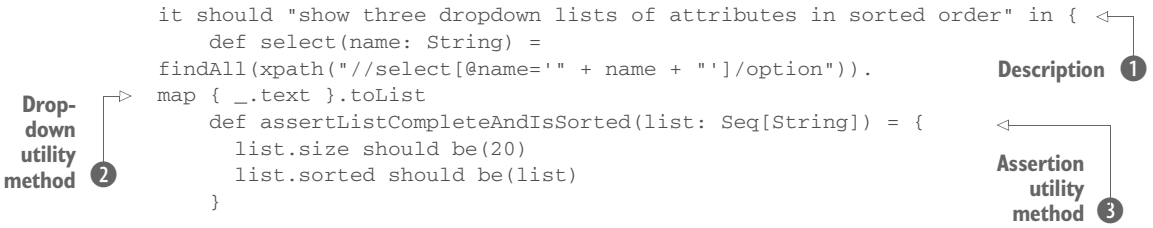
[1] Shopping basket, not cat basket.

You're using the Play Framework for your site, and you've added a `main` method to run it,[2] so you can use `sbt run` to make the site available:

```
$ sbt run
[info] Loading project definition from ...
[info] Set current project to preowned-kittens ...
[info] Running Global
[info] play - database [default] connected at jdbc:h2:mem:play
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
```

For now, you can run the site in one window and the tests from another. You'll have a better solution to this in chapter 6.

You can add another more in-depth test:




```
it should "show three dropdown lists of attributes in sorted order" in {
    def select(name: String) =
findAll(xpath("//select[@name='" + name + "']/option")).
map { _.text }.toList
    def assertListCompleteAndIsSorted(list: Seq[String]) = {
      list.size should be(20)
      list.sorted should be(list)
    }

    go to homePage + "/kittens"

    assertListCompleteAndIsSorted(select("select1"))
    assertListCompleteAndIsSorted(select("select2"))
    assertListCompleteAndIsSorted(select("select3"))
  }
```

We don't need to go into too much detail here, but you can see that line 1 contains the description ❶, except that you don't have to say `Home page` again. You can just say `it`. You define two utility methods, one to select the text from the drop-down lists ❷, and one to make some assertions that they contain the default value and that they are sorted ❸. The test itself goes to the `/kittens` page and then asserts that all of the lists are correct and present. The output now looks like this:

```
[info] SeleniumSpec:
[info] Home page
[info] - should redirect to kitten list
[info] - should show three dropdown lists of attributes in sorted order
```

5.4.2 Challenges of integration testing

What challenges does this present to the build? The first, most obvious one is that in order to run your tests, you need a website that's up and running. This means that before you run your tests, you'll have to actually build the site: you'll have to package the website in some manner, and you'll have to start the website so that you can interact with it. Normally, integration tests use realistic data, so you may need a database and also may need to clean up the database before each test run.

[2] You can download the code from the GitHub repository.

Figure 5.5 The Selenium architecture

This implies that the integration tests can't be run at the same time as all of your other tests. Usually, when you're developing, you want feedback as quickly as possible, so it's worth running the unit tests before you start worrying about starting servers or any other expensive operations. It's also conceptually a good idea to separate the two sets of tests because you'll end up with a cleaner build. We'll talk about starting the server and the packaging in chapters 6 and 8.

Another thing you need to take into account is how Selenium works. It's not quite as simple as starting a browser and forgetting it. The architecture actually looks more like figure 5.5.

The ScalaTest Selenium DSL drives the Selenium server. The Selenium server in turn drives the browser. There's a specific driver for each browser. The browser interacts directly with the site under test.

Most of the time it's easier to use Firefox as a target, because Selenium has a built-in server for Firefox. But we'll use Google Chrome in our examples, so you'll need to tell Selenium where your Chrome driver is and, importantly, start up the Chrome driver server before the tests begin and shut it down after the tests have finished. Because the startup and shutdown of the Chrome server is expensive, you can't do it for each test, so it's better that you do it only once, before all the tests, and then shut it down at the end. Again, more about this in chapter 6.

Finally (as if all this wasn't enough), one of the advantages of Selenium is its multi-browser capability; it can handle Firefox, Chrome, Internet Explorer, and other browsers. Therefore, you want to write your tests in such a way that you can easily switch the browser that you're testing against. And you'll want to run tests against all browsers each night. But this will be a pain if you have to do this every time you build locally, so you'll want an option in your build to run the full suite of browser tests on the continuous integration server only at night.

That's quite a list. Let's get started.

5.4.3 Adding integration tests to sbt

sbt has a built-in configuration for integration tests. To use it, you add the integration test settings to your top-level build, changing the `PreownedKittenProject` method:

```
def PreownedKittenProject(name: String): Project = (
  Project(name, file(name))
  .settings( Defaults.itSettings : _*)                    (1) Adds integration test settings
  .settings(
```

```
    libraryDependencies += "org.specs2" %% "specs2" % "1.14" % "test",
    javacOptions in Compile ++= Seq("-target", "1.6", "-source", "1.6"),
    resolvers ++= Seq(
      "Typesafe Repository" at "http://repo.typesafe.com/typesafe/releases/",
      "teamon.eu Repo" at "http://repo.teamon.eu/"
    )
  )
  .configs(IntegrationTest)                          ❷ Adds integration
)                                                      test configuration
```

This adds the predefined integration test configuration ❷ to your project. From now on, you can use the name `it` to refer to this configuration; for instance, when you want to add a dependency to your build. ❶ adds the compilation, testing, and packaging tasks, as well as the settings that apply to these tasks, to the `IntegrationTest` configuration. You can also use this in your build.sbt file.

By default, this configuration uses the directory src/it, so it will look for Scala sources in src/it/scala, and resources in src/it/resources. It compiles classes to target/it-classes.

Now you can add the relevant dependencies to your build[3] in build.sbt:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.0" % "it"
libraryDependencies += "org.seleniumhq.selenium" % "selenium-java" % "2.31.0"
     % "it"
```

Note that you're using the `it` configuration here, not `test`. If you wanted to use ScalaTest for both the `test` and `it` configurations, you could say

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.0" % "it,test"
```

Next add your tests into src/it/scala/SeleniumSpec.scala:

```
class SeleniumSpec extends FlatSpec with ShouldMatchers with BeforeAndAfter
     with BeforeAndAfterAll with HtmlUnit {
  val homePage: String = "http://localhost:9000"

  "Home page" should "redirect to kitten list" in {
    go to "http://localhost:9000"
    currentUrl should startWith ("http://localhost:9000/kittens")
  }

  it should "show three dropdown lists of attributes in sorted order" in {
    def select(name: String) = findAll(xpath("//select[@name='" + name + "']/
     option")).
map { _.text }.toList
    def assertListCompleteAndIsSorted(list: Seq[String]) = {
      list.size should be(20)
      list.sorted should be(list)
    }

    go to homePage + "/kittens"
```

[3] Be careful about using the correct version of Selenium for your browser. You may need to change the Selenium dependency version here.

```
    assertListCompleteAndIsSorted(select("select1"))
    assertListCompleteAndIsSorted(select("select2"))
    assertListCompleteAndIsSorted(select("select3"))
  }
}
```

Now you can run it. As I said before, you need to tell Selenium where to find the web driver for Chrome, which you do with a `System` property. Currently this is stored in src/it/resources and is part of your source tree. In build.sbt, you add a `-D` for `chromedriver`:

```
javaOptions in IntegrationTest += "-Dwebdriver.chrome.driver=" +
(baseDirectory.value / "src/it/resources/chromedriver.exe").getAbsolutePath
```

Of course, this works only on Windows systems. To enable people who work with Linux-based systems, including Mac OS, to run your tests as well, you can add a method that uses a Java system property to detect if you're running Windows and run the correct server accordingly:

```
def chromeDriver = if (System.getProperty("os.name").startsWith("Windows"))
"chromedriver.exe" else "chromedriver"

javaOptions in IntegrationTest += "-Dwebdriver.chrome.driver="
+ (baseDirectory.value / "src/it/resources" / chromeDriver).getAbsolutePath
```

Those running on other systems will have to find their own versions of the `chromedriver` and modify their builds accordingly.[4] Note that you're using `baseDirectory` similarly to how you used `target` before. You'll also need to ensure that Chrome is installed on your local machine.

You can now run the tests. For the integration test configuration, the `test` and `testOnly` tasks are available but have to be prefixed by `it`:

```
> it:test
[info] SeleniumSpec:
[info] Home page
[info] - redirect to kitten list
[info] - show three dropdown lists of attributes in sorted order
[info] Passed: : Total 2, Failed 0, Errors 0, Passed 2, Skipped 0
[success] Total time: 32 s, completed 7 avr. 2013 22:23:23
```

You can also use the autorun feature of sbt as usual: `~it:test`.

As you can see, this is quite a lot of setup for the tests that you have, but if you had more tests, it would become more worth it. If you were playing along at home, you would have noticed a few things when you executed the tests.

The first, most obvious, is that the tests are quite slow to execute, at least compared to the normal unit test cycle. This becomes especially painful if you're using the add test/develop cycle that you used in the previous section.

---

[4] We've put the relevant drivers into the GitHub repository, or you can download them yourself by searching the internet for Selenium Chrome driver.

Second, you can actually interact with the browser started by the tests during the tests. You can close it. This, not surprisingly, causes the tests to fail.

Third, for the Windows users among you, when you run this on a Windows machine, you get a UAC warning from Windows every time you start the `chromeserver`. UAC stands for User Account Control. Basically, it's warning you that you're about to do something that requires administrator privileges. It's generally a bad idea to disable these kinds of security checks, but they do interrupt the workflow because you have to click on the warning to allow the tests to continue.

This means that these tests as they're written aren't necessarily a good fit for your interactive test/code cycle. It also slows down the development of these tests.

But there is a partial solution to these problems. As I said before, Selenium supports the Google Chrome, Mozilla Firefox, and Microsoft Internet Explorer browsers, but it also supports another "browser" called Apache HtmlUnit, which acts like a browser but isn't interactive. It does all the things that a browser does: downloads the HTML pages, downloads the JavaScript, and even executes the JavaScript, but it doesn't require any external process. It's much quicker and simpler to use, but you have the disadvantage of not testing directly through a browser, so you could develop your tests using the HtmlUnit interface and run the full set of tests in your build during the night using the full browser.

I find that using the HtmlUnit interface allows me to develop integration tests more quickly and easily than using the full browser tests. But for JavaScript-heavy applications, this doesn't work as well.

Finally, you need a nice HTML solution for your reports. Fortunately, ScalaTest has a very nice solution; all you need to do is add an `-h` option to your `testOptions` in build.sbt:

```
testOptions += Tests.Argument(TestFrameworks.ScalaTest, "-h",
(target.value / "html-test-report").getAbsolutePath)
```

You use the `test` framework delimiter to ensure that these options apply only to ScalaTest. Note that ScalaTest uses pegdown as well, so you need to add pegdown to the dependencies for the integration tests. You can do this in one of two ways. If the versions were the same, you could add the `it` configuration to the existing dependency that you put in for specs2:

```
libraryDependencies += "org.pegdown" % "pegdown" % "1.0.2" % "test,it"
```

But in this case you want to use a different version (1.1.0), so you can add a new line with the `it` configuration, and this won't cause any problems or clashes:

```
libraryDependencies += "org.pegdown" % "pegdown" % "1.1.0" % "it"
```

Now running the tests produces the HTML output for ScalaTest that looks like figure 5.6.

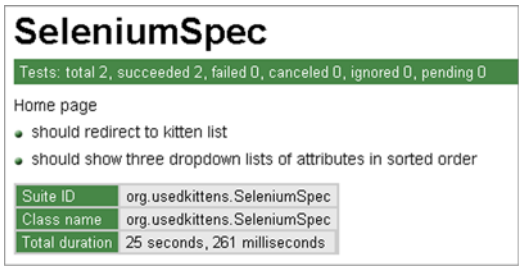


Figure 5.6 The HTML output after running your tests

5.5 Summary

In sbt, you can easily incorporate multiple styles of testing. Most Scala testing frameworks are supported natively through the sbt `test-interface`. You can also easily add options specific to one testing framework. You can add relatively complex tasks to your build fairly easily, and you can generate some nice reports on your tests for inclusion in your site.

Finally, through use of ~ you can have a better development experience because you get immediate feedback through your tests.

I think your kittens will be impressed.

But the job isn't finished yet. You still need to package the generated HTML files for inclusion, and you still need a mechanism for running your Selenium integration tests on the build server. You'll need to add some specific tasks to your build for that. We'll look at this in chapter 6.

SCALA/PROGRAMMING

sbt IN ACTION

Suereth • Farwell

Free eBook
SEE INSERT

Sbt is a build tool native to Scala that can transform any build scenario into a streamlined, automated, and repeatable process. Its interactive shell lets you customize your builds on the fly, and with sbt's unique incremental compilation feature, you can update only the parts of your project that change, without having to rebuild everything. Mastering sbt, along with the right patterns and best practices, is guaranteed to save you time and trouble on every project.

sbt in Action, first and foremost, teaches you how to build Scala projects effectively. It introduces the sbt tool with a simple project that establishes the fundamentals of running commands and tasks. Next, it shows you how to use the peripheral libraries in sbt to make common tasks simpler. Along the way, you'll work through real projects that demonstrate how to build and deploy your projects regardless of development methodology or process.

What's Inside

- Master sbt's loosely coupled libraries
- Effectively manage dependencies
- Automate and simplify your builds
- Customize builds and tasks

Readers should be comfortable reading Scala code. No experience with sbt required.

Josh Suereth is an engineer at Typesafe and the author of *Scala in Depth*. **Matthew Farwell** is a senior developer and the author of the Scalastyle style checker.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/sbt-in-action

"Offers a fast introduction to the basics of building software with sbt, and complements it with detailed coverage of sbt's more advanced topics."
—Dusan Kysel, Cyber Brain

"Covers core concepts using the Scala build tool. A useful read for beginners and advanced users alike."
—Iain Campbell, Tango Telecom

"Despite being a Java guy, I liked how the book tried to fool me into learning Scala. As I worked through the examples, I couldn't help but start piecing some of it together."
—Eric Weinberg, Emprata

ISBN 13: 978-1-617291-27-2
ISBN 10: 1-617291-27-7
54499
9 781617 291272

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]