Spark IN ACTION

Petar Zečević
Marko Bonaći

SAMPLE CHAPTER

MANNING

Spark in Action

by Petar Zečević
and Marko Bonači

Chapter 3

brief contents

3 Writing Spark applications

This chapter covers

- Generating a new Spark project in Eclipse
- Loading a sample dataset from the GitHub archive
- Writing an application that analyzes GitHub logs
- Working with DataFrames in Spark
- Submitting your application to be executed

In this chapter, you'll learn to write Spark applications. Most Spark programmers use an integrated development environment (IDE), such as IntelliJ or Eclipse. There are readily available resources online that describe how to use IntelliJ IDEA with Spark, whereas Eclipse resources are still hard to come by. That is why, in this chapter, you'll learn how to use Eclipse for writing Spark programs. Nevertheless, if you choose to stick to IntelliJ, you'll still be able to follow along. After all, these two IDEs have similar sets of features.

You'll start by downloading and configuring Eclipse and then installing Eclipse plug-ins that are necessary for working with Scala. You'll use Apache Maven (a software project-management tool) to configure Spark application projects in this chapter. The Spark project itself is configured using Maven. We prepared a Maven Archetype (a template for quickly bootstrapping Maven projects) in the book's GitHub repository at https://github.com/spark-in-action, which will help you bootstrap your new Spark application project in just a few clicks.

Throughout this chapter, you'll be developing an application that counts GitHub push events (code commits to GitHub) made by your company's employees. You'll use an exciting new construct, the `DataFrame`, which saw the light of day in Spark 1.3.0.

Lots of content awaits you. Ready?

3.1 Generating a new Spark project in Eclipse

This section describes how to create a Spark project in Eclipse. We trust you know how to install Eclipse (you can follow the online instructions at http://wiki.eclipse.org/Eclipse/Installation), so we won't go into details. You can install it onto your development machine or in the spark-in-action VM. The decision is all yours, because it won't significantly affect how you build your Spark project. We installed it in the VM, in the /home/spark/eclipse folder, and used /home/spark/workspace as the workspace folder. To view the Eclipse GUI started from the VM, you'll need to set up an X Window system (on Windows, you can use Xming: https://sourceforge.net/projects/xming) and set the `DISPLAY` variable in your VM Linux shell to point to the IP address of your running X Window system.

If you wish to use some other IDE, such as IntelliJ, you can skip this section and start from section 3.2. If you wish to continue using Eclipse, you also need to install these two plug-ins:

- Scala IDE plug-in
- Scala Maven integration for Eclipse (`m2eclipse-scala`)

To install the Scala IDE plug-in, follow these steps:

1. Go to Help[1] > Install new Software, and click Add in the upper-right corner.
2. When the Add Repository window appears, enter `scala-ide` in the Name field.
3. Enter `http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site` in the Location field
4. Confirm by clicking OK.
5. Eclipse looks up the URL you entered and displays the available software it found there. Select only the Scala IDE for Eclipse entry and all its subentries.
6. Confirm the selection on the next screen, and accept the license on the one after that. Restart Eclipse when prompted.

[1] Readers new to Ubuntu may not know that the toolbar of the currently active window is always located at the top of the screen, and it's revealed when you hover the cursor near the top.

To install the Scala Maven integration for Eclipse plug-in, follow the same steps as for the Scala IDE plug-in, only enter `http://alchim31.free.fr/m2e-scala/update-site` in the Location field and `m2eclipse-scala` in the Name field.

Once you have all these set up, you're ready to start a new Eclipse project that will host your application. To simplify setting up new projects (either for examples in this book or for your future Spark projects), we have prepared an Archetype called `scala-archetype-sparkinaction` (available from our GitHub repository), which is used to create a starter Spark project in which versions and dependencies have already been taken care of.

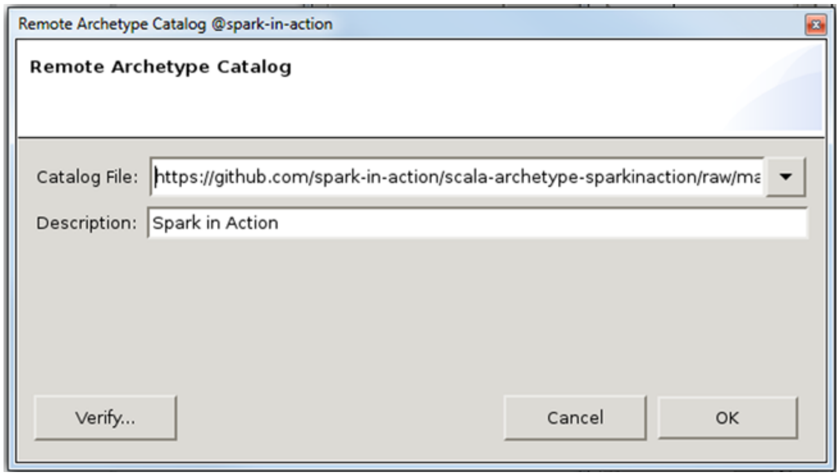
To create a project in Eclipse, on your toolbar menu select File > New > Project > Maven > Maven Project. Don't make any changes on the first screen of the New Maven Project Wizard, but click Next.

On the second screen, click Configure (which opens the Maven > Archetypes section of the Eclipse Preferences). Click the Add Remote Catalog button, and fill in the following values in the dialog that pops up (see figure 3.1):

- *Catalog File:* `https://github.com/spark-in-action/scala-archetype-sparkinaction/raw/master/archetype-catalog.xml`
- *Description:* `Spark in Action`

Click OK, and then close the Preferences window. You should see a progress bar in the lower-right corner, which is how Eclipse notifies you that it went to look up the catalog you just added.

You're now back in the New Maven Project wizard. Select Spark in Action in the Catalog drop-down field and scala-archetype-sparkinaction artifact (see figure 3.2).

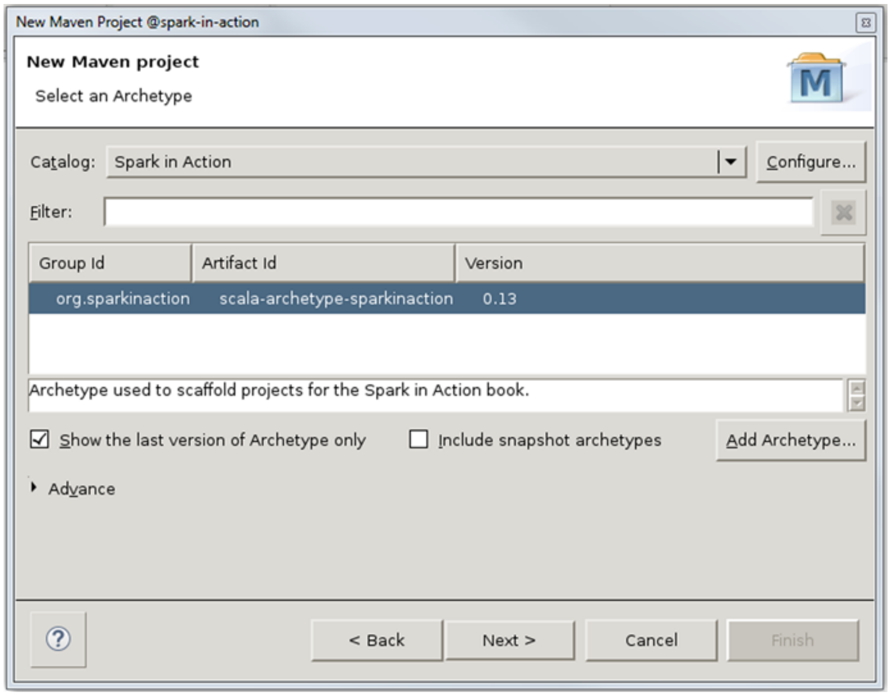

Remote Archetype Catalog @spark-in-action

Remote Archetype Catalog

Catalog File: https://github.com/spark-in-action/scala-archetype-sparkinaction/raw/ma

Description: Spark in Action

Figure 3.1 **Adding the *Spark in Action* Maven Remote Archetype Catalog to your Eclipse Preferences**



New Maven Project @spark-in-action

New Maven project

Select an Archetype

Catalog: Spark in Action

Filter:

Group Id	Artifact Id	Version
org.sparkinaction	scala-archetype-sparkinaction	0.13

Archetype used to scaffold projects for the Spark in Action book.

☑ Show the last version of Archetype only ☐ Include snapshot archetypes

Figure 3.2 **Choosing the Maven Archetype that you want to use as the new project's template. Select scala-archetype-sparkinaction.**

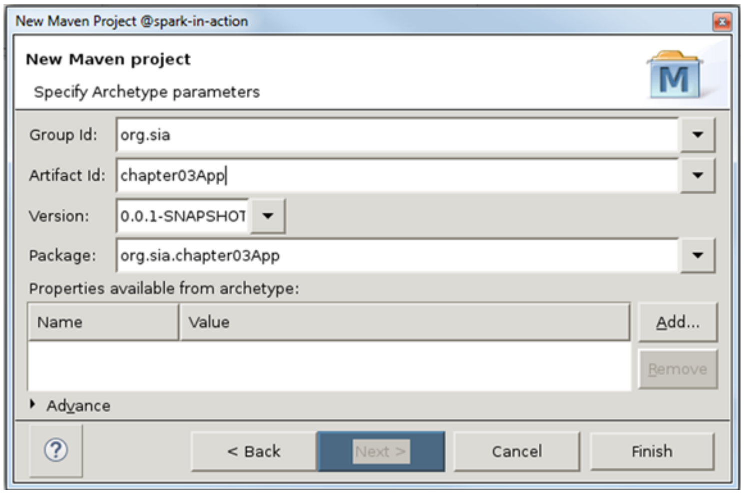
The next screen prompts you to enter your project's parameters. Notice (in figure 3.3) that your root package consists of `groupId` and `artifactId`.

groupId and artifactId

If you're new to Maven, it may be easier for you to look at `artifactId` as your project name and `groupId` as its fully qualified organization name. For example, Spark has `groupId` `org.apache` and `artifactId` `spark`. The Play framework has `com.typesafe` as its `groupId` and `play` as its `artifactId`.

You can specify whichever values you like for `groupId` and `artifactId`, but it may be easier for you to follow along if you choose the same values as we did (see figure 3.3). Confirm by clicking Finish.

Let's examine the structure of the generated project. Looking from the top, the first (root) entry is the project's main folder, always named the same as the project. We call this folder the project's *root* (or *project root*, interchangeably).



New Maven Project @spark-in-action

New Maven project

Specify Archetype parameters

Group Id: org.sia

Artifact Id: chapter03App

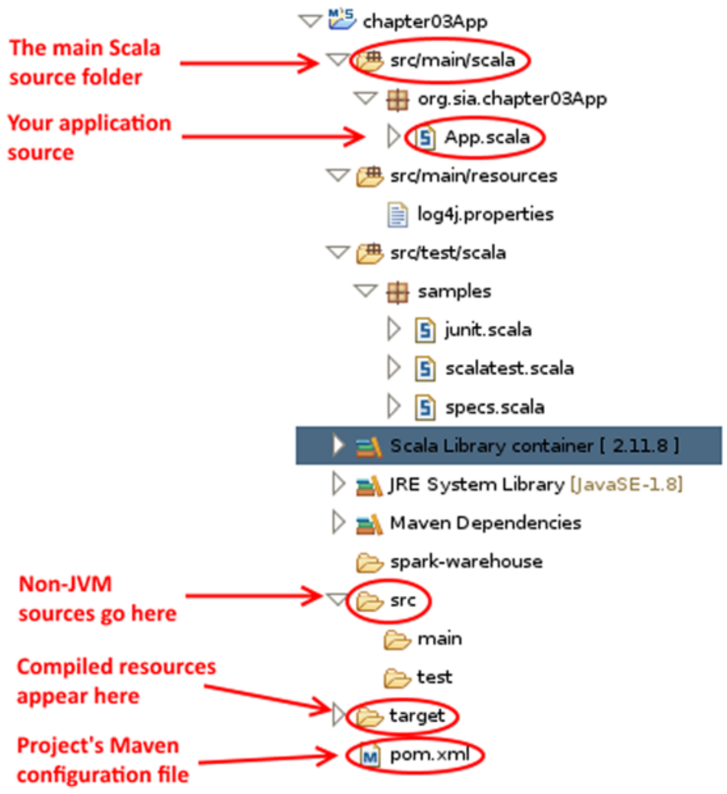
Version: 0.0.1-SNAPSHOT

Package: org.sia.chapter03App

Properties available from archetype:

Name	Value

Figure 3.3 **Creating a Maven project: specifying project parameters**



```
chapter03App
  src/main/scala
    org.sia.chapter03App
      App.scala
  src/main/resources
    log4j.properties
  src/test/scala
    samples
      junit.scala
      scalatest.scala
      specs.scala
  Scala Library container [ 2.11.8 ]
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  spark-warehouse
  src
    main
    test
  target
  pom.xml
```

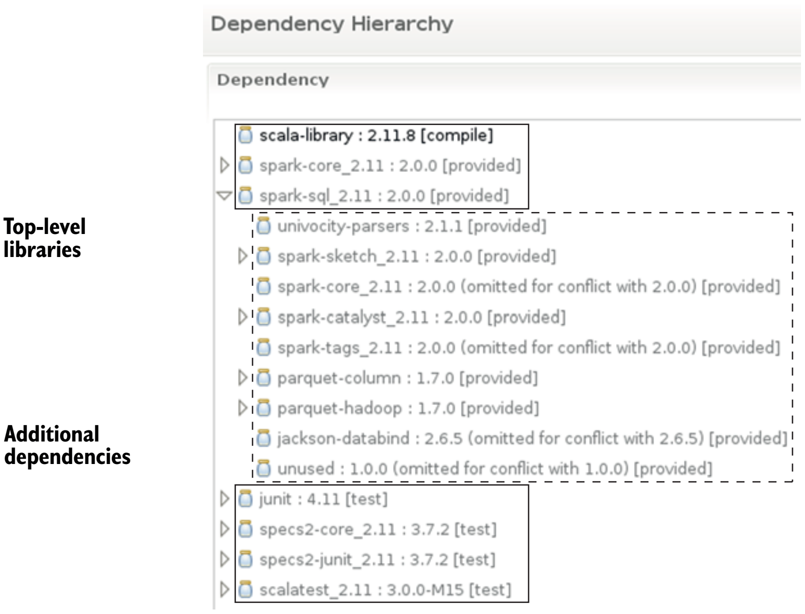
The main Scala source folder
Your application source
Non-JVM sources go here
Compiled resources appear here
Project's Maven configuration file

Figure 3.4 **The newly generated project in Eclipse's Package Explorer window**

src/main/scala is your main Scala source folder (if you were to add Java code to your project, you would create an src/main/java source folder). In the scala folder, you have the root package,[2] `org.sia.chapter03App`, with a single Scala source file in it: App.scala. This is the Scala file where you'll start writing your application.

Next comes the main test folder with prepared samples for various test types, followed by the container for your Scala library. You're using the Scala that came with the Scala IDE Eclipse plug-in. Next is Maven Dependencies, described a bit later. Below Maven Dependencies is JDK (Eclipse refers to JRE and JDK in the same way, as JRE System Library). Further down in the Package Explorer window is again the src folder, but this time in the role of an ordinary folder (non-source or, more precisely, non-jvm-source folder), in case you want to add other types of resources to your project, such as images, JavaScript, HTML, CSS, anything you don't want to be processed by the Eclipse JVM tooling. Then there is the target folder, which is where compiled resources go (like .class or .jar files).

Finally, there is the all-encompassing pom.xml, which is the project's Maven specification. If you open pom.xml from the project root and switch to the Dependency Hierarchy tab, you'll have a much better view of the project's dependencies and their causality (see figure 3.5).



```
scala-library : 2.11.8 [compile]
▷ spark-core_2.11 : 2.0.0 [provided]
▽ spark-sql_2.11 : 2.0.0 [provided]
    univocity-parsers : 2.1.1 [provided]
  ▷ spark-sketch_2.11 : 2.0.0 [provided]
    spark-core_2.11 : 2.0.0 (omitted for conflict with 2.0.0) [provided]
  ▷ spark-catalyst_2.11 : 2.0.0 [provided]
    spark-tags_2.11 : 2.0.0 (omitted for conflict with 2.0.0) [provided]
  ▷ parquet-column : 1.7.0 [provided]
  ▷ parquet-hadoop : 1.7.0 [provided]
    jackson-databind : 2.6.5 (omitted for conflict with 2.6.5) [provided]
    unused : 1.0.0 (omitted for conflict with 1.0.0) [provided]
▷ junit : 4.11 [test]
▷ specs2-core_2.11 : 3.7.2 [test]
▷ specs2-junit_2.11 : 3.7.2 [test]
▷ scalatest_2.11 : 3.0.0-M15 [test]
```

Figure 3.5 Project's libraries dependency hierarchy (in pom.xml)

[2] On the filesystem level, `org.sia.chapter03App` consists of three folders. In other words, this is the path to the App.scala file: chapter03App/src/main/scala/org/sia/chapter03App/App.scala.

There are only six libraries at the top level (all explicitly listed in pom.xml). Each of those libraries brought with it its own dependencies; and many of those dependencies, in turn, have their own dependencies, and so on

In the next section, you'll see a plausible real-world example of developing a Spark application. You'll need Eclipse, so don't close it just yet.

3.2 Developing the application

Say you need to create a daily report that lists all employees of your company and the number of *pushes*[3] they've made to GitHub. You can implement this using the GitHub archive site (www.githubarchive.org/), put together by Ilya Grigorik from Google (with the help of the GitHub folks), where you can download GitHub archives for arbitrary time periods. You can download a single day from the archive and use it as a sample for building the daily report.

3.2.1 Preparing the GitHub archive dataset

Open your VM's terminal, and type in the following commands:

```
$ mkdir -p $HOME/sia/github-archive
$ cd $HOME/sia/github-archive
$ wget http://data.githubarchive.org/2015-03-01-{0..23}.json.gz
```

This downloads all public GitHub activity from March 1, 2015, in the form of 24 JSON files, one for each hour of the day:

```
2015-03-01-0.json.gz
2015-03-01-1.json.gz
...
2015-03-01-23.json.gz
```

To decompress the files, you can run

```
$ gunzip *
```

After a few seconds, you're left with 24 JSON files. You'll notice that the files are rather large (around 1 GB in total, decompressed), so you can use just the first hour of the day (44 MB) as a sample during development, instead of the entire day.

But the extracted files aren't valid JSON. Each file is a set of valid JSON strings separated with newlines, where each line is a single JSON object—that is, a GitHub event (push, branch, create repo, and so on).[4] You can preview the first JSON object with `head` (which is used to list `n` lines from the top of a file), like this:

[3] To *push changes*, in distributed source control management systems (such as Git), means to transfer content from your local repository to a remote repository. In earlier SCM systems, that was known as *commit*.

[4] GitHub API documentation on types of events: https://developer.github.com/v3/activity/events/types/.

```
$ head -n 1 2015-03-01-0.json
{"id":"2614896652","type":"CreateEvent","actor":{"id":739622,"login":
➥ "treydock","gravatar_id":"","url":"https://api.github.com/users/
➥ treydock","avatar_url":"https://avatars.githubusercontent.com/u/
➥ 739622?"},"repo":{"id":23934080,"name":"Early-Modern-OCR/emop-
➥ dashboard","url":"https://api.github.com/repos/Early-Modern-OCR/emop-
➥ dashboard"},"payload":{"ref":"development","ref_type":"branch","master_
➥ branch":"master","description":"","pusher_type":"user"},"public":true,
➥ "created_at":"2015-03-01T00:00:00Z","org":{"id":10965476,"login":
➥ "Early-Modern-OCR","gravatar_id":"","url":"https://api.github.com/
➥ orgs/Early-Modern-OCR","avatar_url":
➥ "https://avatars.githubusercontent.com/u/10965476?"}}
```

Uh, that's tough to read. Fortunately, an excellent program called `jq` (http://stedolan.github.io/jq) makes working with JSON from the command line much easier. Among other things, it's great for pretty-printing and color highlighting JSON. You can download it from http://stedolan.github.io/jq/download. If you're working in the spark-in-action VM, it's already installed for you.

To try it out, you can pipe a JSON line to `jq`:

```
$ head -n 1 2015-03-01-0.json | jq '.'
{
  "id": "2614896652",
  "type": "CreateEvent",
  "actor": {
    "id": 739622,
    "login": "treydock",
    "gravatar_id": "",
    "url": "https://api.githb.com/users/treydock",
    "avatar_url": "https://avatars.githubusercontent.com/u/739622?"
  },
  "repo": {
    "id": 23934080,
    "name": "Early-Modern-OCR/emop-dashboard",
    "url": "https://api.github.com/repos/Early-Modern-OCR/emop-dashboard"
  },
  "payload": {
    "ref": "development",
    "ref_type": "branch",
    "master-branch": "master",
    "description": "",
    "pusher_type": "user",
  },
  "public": true,
  "created_at": "2015-03-01T00:00:00Z",
  "org": {
    "id": 10965476,
    "login": "Early-Modern-OCR",
    "gravatar_id": "",
    "url": "https://api.github.com/orgs/Early-Modern-OCR",
    "avatar_url": "https://avatars.githubusercontent.com/u/10965476?"
  }
}
```

Wow! That certainly is pretty. Now you can easily see that this first log entry from the file has `"CreateEvent"` type and that its `payload.ref_type` is `"branch"`. So someone named `"treydock"` (`actor.login`) created a repository branch called `"development"` (`payload.ref`) in the first second of March 1, 2015 (`created_at`). This is how you'll distinguish types of events, because all you need to count are push events. Looking at the GitHub API (https://developer.github.com/v3/activity/events/types/#pushevent), you find out that the push event's `type` is, unsurprisingly, `"PushEvent"`.

OK, you obtained the files needed to develop a prototype, and you know how to prettify its contents so you can understand the structure of GitHub log files. Next, you can start looking into the problem of ingesting a JSON-like structured file into Spark.

3.2.2 Loading JSON

Spark SQL and its `DataFrame` facility, which was introduced in Spark v.1.3.0, provide a means for ingesting JSON data into Spark. At the time of this announcement, everyone was talking about `DataFrames` and the benefits they were going to bring to computation speed and data interchange between Spark components (Spark Streaming, MLlib, and so on). In Spark 1.6.0 `DataSets` were introduced as generalized and improved `DataFrames`.

DataFrame API

A `DataFrame` is an RDD that has a schema. You can think of it as a relational database table, in that each column has a name and a known type. The power of `DataFrames` comes from the fact that, when you create a `DataFrame` from a structured dataset (in this case, JSON), Spark is able to infer a schema by making a pass over the entire JSON dataset that's being loaded. Then, when calculating the execution plan, Spark can use the schema and do substantially better computation optimizations. Note that `DataFrame` was called `SchemaRDD` before Spark v1.3.0.

SQL is so ubiquitous that the new `DataFrame` API was quickly met with acclamation by the wider Spark community. It allows you to attack a problem from a higher vantage point when compared to Spark Core transformations. The SQL-like syntax lets you express your intent in a more declarative fashion: you describe what you want to achieve with a dataset, whereas with the Spark Core API you basically specify how to transform the data (to reshape it so you can come to a useful conclusion).

You may therefore think of Spark Core as a set of fundamental building blocks on which all other facilities are built. The code you write using the `DataFrame` API gets translated to a series of Spark Core transformations under the hood.

We'll talk about `DataFrames` extensively in chapter 5. For now, let's focus on features relevant to the task at hand.

SQLContext is the main interface to Spark SQL (analogous to what SparkContext is to Spark Core). Since Spark 2.0, both contexts are merged into a single class: SparkSession. Its read method gives access to the DataFrameReader object, which you use for getting various data. DataFrameReader's json method is for reading the JSON data. Here's what the scaladocs (http://mng.bz/amo7) say:

```
def json(paths: String*): DataFrame
Loads a JSON file (one object per line) and returns the result as a
➥ [[DataFrame]].
```

One object per line: that's exactly how the GitHub archive files are structured.

Bring your Eclipse forward, and switch to the Scala perspective (Window > Open Perspective > Other > Scala). Then open the App.scala file (to quickly locate the file, you can use Ctrl-Shift-R and then type the first few letters of the filename in the dialog that pops up), clean it up, and leave only the SparkContext initialization, like this:

```
import org.apache.spark.sql.SparkSession

object App {
  def main(args : Array[String]) {
    val spark = SparkSession.builder()
        .appName("GitHub push counter")
        .master("local[*]")
        .getOrCreate()

    val sc = spark.sparkContext
  }
}
```

To load the first JSON file, add the following snippet. Because neither a tilde (~)[5] nor $HOME can be used directly in the path, you end up first retrieving the HOME environment variable so you can use it to compose the JSON file path:

```
val homeDir = System.getenv("HOME")
val inputPath = homeDir + "/sia/github-archive/2015-03-01-0.json"
val ghLog = spark.read.json(inputPath)
```

The json method returns a DataFrame, which has many of the standard RDD methods you used before, like filter, map, flatMap, collect, count, and so on.

The next task to tackle is filtering the log entries so you're left only with push events. By taking a peek at DataFrame (since Spark 2.0, DataFrame is a special case of DataSet; it's a DataSet containing Row objects) in the scaladocs (http://mng.bz/3EQc), you can quickly find out that the filter function is overloaded and that one version takes a condition expression in the form of a String and another one takes a Column.

[5] The tilde is equivalent to $HOME. You can use them interchangeably.

The `String` argument will be parsed as SQL, so you can write the following line below the line that loads JSON into `ghLog`:

```
val pushes = ghLog.filter("type = 'PushEvent'")
```

It's time to see whether the code you have so far works. Will the application compile and start successfully? How will the loading go? Will the schema be inferred successfully? Have you specified the filter expression correctly?

3.2.3 Running the application from Eclipse

For the purpose of finding out answers to these questions, add the following code below `filter`:

```
pushes.printSchema
println("all events: " + ghLog.count)
println("only pushes: " + pushes.count)
pushes.show(5)
```

Pretty-prints the schema of the pushes DataFrame

Prints the first 5 rows (defaults to 20 if you call it with no arguments) from a DataFrame in a tabular format

Then, build the project by right-clicking the main project folder and choosing Run As > Maven Install. Once the process finishes, right-click App.scala in the Package Explorer, and choose Run As > Scala Application.

If there is no such option, you will need to create a new Run Configuration. To do this, click Run As > Run Configurations ..., then choose `Scala Application` and press the `New` button. Enter *Chapter03App* in the `Name` field and *org.sia.chapter03App.App* in the `Main class` field and hit `Run`.

> **Keyboard shortcut for running a Scala application**
> You may have noticed the keyboard shortcut located next to Scala Application (in Package Explorer > Run As). It says Alt-Shift-X S. This means you first need to press together on Alt, Shift, and X, then release all the keys, and then press S by itself.

From now on, we won't always explicitly tell you when to run the application. Follow the code (always inserting it at the bottom of the existing code), and every time you see the output, run the application using either of the two methods described here.

Hopefully, you can now see the output in Eclipse's console window, where the `printSchema` method outputs the inferred schema (if the output is buried in many `INFO` and `WARN` messages, you probably skipped the logging-configuration step in section 2.2.1). The inferred schema consists of the union of all JSON keys, where each key has been assigned a type and the `nullable` attribute (which is always `true` in inferred schemas because, understandably, Spark leaves that decision to you):

```
root
 |-- actor: struct (nullable = true)
 |    |-- avatar_url: string (nullable = true)
 |    |-- gravatar_id: string (nullable = true)
 |    |-- id: long (nullable = true)
 |    |-- login: string (nullable = true)
 |    |-- url: string (nullable = true)
 |-- created_at: string (nullable = true)
 |-- id: string (nullable = true)
 |-- org: struct (nullable = true)
 |    |-- avatar_url: string (nullable = true)
 |    |-- gravatar_id: string (nullable = true)
 |    |-- id: long (nullable = true)
 |    |-- login: string (nullable = true)
 |    |-- url: string (nullable = true)
 |-- payload: struct (nullable = true)
 |    |-- action: string (nullable = true)
 |    |-- before: string (nullable = true)
 |    |-- comment: struct (nullable = true)
 |    |    |-- _links: struct (nullable = true)
 |    |    |    |-- html: struct (nullable = true)
 |    |    |    |    |-- href: string (nullable = true)
 ...
```

You can see that the inferred schema fits your previous findings regarding the GitHub username. You'll have to use the `actor` object and its property `login`, thus `actor.login`. You'll need that information soon, to count the number of pushes per employee. Scroll down to find the first count (`all events`) and then a bit more to find the second one (`only pushes`):

```
...
all events: 17786
...
only pushes: 8793
...
```

At the bottom of the output, you can see the first five rows (we removed four columns—`id`, `org`, `payload`, and `public`—from the middle so the output can fit on a single line):

```
+--------------------+--------------------+<->+--------------------+---------+
|               actor|          created_at|<->|                repo|     type|
+--------------------+--------------------+<->+--------------------+---------+
|[https://avatars...|2015-03-01T00:00:00Z|<->|[31481156,bezerra...|PushEvent|
|[https://avatars...|2015-03-01T00:00:00Z|<->|[31475673,demianb...|PushEvent|
|[https://avatars...|2015-03-01T00:00:00Z|<->|[31481269,ricardo...|PushEvent|
|[https://avatars...|2015-03-01T00:00:00Z|<->|[24902852,actorap...|PushEvent|
|[https://avatars...|2015-03-01T00:00:00Z|<->|[24292601,komasui...|PushEvent|
+--------------------+--------------------+<->+--------------------+---------+
```

To summarize, there were 17,786 events, out of which 8,793 were push events, in the first hour of March 1, 2015. So, it's all working as expected.

3.2.4 Aggregating the data

You managed to filter out every type of event but PushEvent, and that's a good start. Next, you need to group the push events by username and, in the process, count the number of pushes in each group (rows for each username):

```
val grouped = pushes.groupBy("actor.login").count
grouped.show(5)
```

This groups all rows by actor.login column and, similar to regular SQL, performs count as the aggregate operation during grouping. Think about it: as multiple rows (with the same value in the actor.login column) get collapsed down to a single row (that's what *grouping* means), what happens with values in other columns? count[6] tells Spark to ignore the values in those columns and count the number of rows that get collapsed for each unique login. The result answers the question of how many push events each unique login has. In addition to count, the API lists other aggregation functions including min, max, avg, and sum.

The first five rows of the resulting DataFrame, grouped, are displayed in Eclipse Console View:

```
+----------+-----+
|     login|count|
+----------+-----+
|    gfgtdf|    1|
|   mdorman|    1|
|quinngrier|    1|
| aelveborn|    1|
|  jwallesh|    3|
+----------+-----+
```

That looks good, but it's impossible to see who had the highest number of pushes. The only thing left to do is to sort the dataset by the count column:

```
val ordered = grouped.orderBy(grouped("count").desc)
ordered.show(5)
```

This orders the grouped DataFrame by the value in the count column and names the new, sorted DataFrame ordered. The expression grouped("count") returns the count column from the grouped DataFrame (it implicitly calls DataFrame.apply[7]), on which you call desc to order the output by count in descending order (the default ordering is asc):

[6] For other operations (such as sum, avg, max, and so on), consult the GroupedData scaladoc: http://mng.bz/X8lA.
[7] See the apply method and the examples at the top of the DataFrame API for more info: http://mng.bz/X8lA.

```
+------------------+-----+
|             login|count|
+------------------+-----+
|      greatfirebot|  192|
|diversify-exp-user|  146|
|     KenanSulayman|   72|
|        manuelrp07|   45|
|    mirror-updates|   42|
+------------------+-----+
```

It works! That's great—but this is the list of all users pushing to GitHub, not only employees of your company. So, you need to exclude non-employees from the list.

3.2.5 Excluding non-employees

We prepared a file on our GitHub repository containing GitHub usernames for employees of your imaginary company. It should already be downloaded in your home directory at first-edition/ch03/ghEmployees.txt.

To use the employee list, load it into some type of Scala collection. Because `Set` has faster random lookup than `Seq` (sequential collection types, such as `Array` and `List`), let's use `Set`. To load the entire file[8] into a new `Set`, you can do the following (this is only a piece of the complete application, which will be given in the next section):

```
import scala.io.Source.fromFile

val empPath = homeDir + "/first-edition/ch03/ghEmployees.txt"
val employees = Set() ++ (
  for {
    line <- fromFile(empPath).getLines
  } yield line.trim
)
```

The for expression[9] reads each line from the file and stores it into the line variable.

Set() creates an empty, immutable set. The method named ++ adds multiple elements to the set.

yield also operates on every cycle of the for loop, adding a value to a hidden collection that will be returned (and destroyed) as the result of the entire for expression, once the loop ends.

This code snippet works like the following pseudocode:

```
In each iteration of the for loop:
    Read the next line from the file
    Initialize a new line variable so that it contains the current line
        as its value
    Take the value from the line variable, trim it, and add it to the
        temporary collection
Once the last iteration finishes:
    Return the temporary, hidden collection as the result of for
    Add the result of for to an empty set
    Assign the set to the employees variable
```

[8] Loading an entire file isn't considered good practice, but because you know ghEmployees.txt should never go over 1 MB, in this case it's fine (with 208 employees, it weighs only 2 KB).

[9] More about `for` expressions in Scala: http://mng.bz/k8q2.

This for expression probably looks complicated if you're new to Scala, but give it a couple of tries; Scala's for comprehensions are a powerful and succinct way of dealing with iteration. There are no index variables unless you need them. We encourage you to try a few for-related examples from that Scala book of yours (or other Scala resource you choose to use).

You now have the set of employee usernames, but how do you compare it with your ordered DataFrame that contains usernames and the corresponding counts? Well, you already used DataFrame's filter method. Its scaladocs (http://mng.bz/3EQc) say

```
def filter(conditionExpr: String): DataSet
```

filters rows using the given SQL expression:

```
val oldPeopleDf = peopleDf.filter("age > 15")
```

But this is only a simple example that compares values of the peopleDf DataFrame's age with the literal number 15. If a row's age is greater than 15, the row is included in oldPeopleDf.

You, on the other hand, need to compare the login column *against the set* of employees and filter out each row whose login value isn't in that set. You could use DataSet's filter function to specify filtering criteria based on the contents of individual Row objects (DataFrames are just DataSets containing Row objects), but you couldn't use that method in DataFrame SQL expressions. That's where user-defined functions (UDFs) come into play.

The SparkSession (http://mng.bz/j9As) class contains the udf method, which is used for registering UDFs. Because you need to check whether each login is in the set of your company's employees, the first thing you need to do is write a general filtering function that checks whether a string is in a set:

```
val isEmp: (String => Boolean) = (arg: String) => employees.contains(arg)
```

You explicitly define isEmp as a function that takes a String and returns a Boolean (often said as "function from String to Boolean"), but thanks to Scala's type inference, you could have made it more terse:

```
val isEmp = user => employees.contains(user)
```

Because employees is a Set of Strings, Scala knows that the isEmp function should take a String.

In Scala, the return value of a function is the value of its last statement, so it isn't difficult to infer that the function should return whatever the method contains returns, which is a Boolean.

Next you register isEmp as a UDF:

```
val isEmployee = spark.udf.register("isEmpUdf", isEmp)
```

Now, when Spark goes to execute the UDF, it will take all of its dependencies (only the employees set, in this case) and send them along with *each and every task*, to be executed on a cluster.

3.2.6 Broadcast variables

We'll talk more about tasks and Spark execution in general in the coming chapters; for now, to explain broadcast variables, we'll tell you that if you were to leave the program like this, you'd be sending the employees set some 200 times over the network (the approximate number of tasks your application will generate for the purpose of excluding non-employees). You don't need to imagine this, because you'll soon see it in your program's output.

Broadcast variables are used for this purpose, because they allow you to send a variable *exactly once* to each node in a cluster. Moreover, the variable is automatically cached in memory on the cluster nodes, ready to be used during the execution of the program.

Spark uses a peer-to-peer protocol, similar to BitTorrent, to distribute broadcast variables, so that, in the case of large clusters, the master doesn't get clogged while broadcasting a potentially large variable to all nodes. This way, worker nodes know how to exchange the variable among themselves, so it spreads out through the cluster organically, like a virus or gossip. In fact, you'll often hear this type of communication between nodes referred to as a *gossip protocol*, further explained at http://en.wikipedia.org/wiki/Gossip_protocol.

The good thing about broadcast variables is that they're simple to use. All you have to do to "fix" your program is to turn your regular employees variable into a broadcast variable, which you then use in place of employees.

You need to add an additional line, just above the isEmp function's definition:

```
val bcEmployees = sc.broadcast(employees)
```

Then change how you refer to this variable, because broadcast variables are accessed using their value method (you need to change this line and not add it; the complete source code of the program is given in listing 3.1):

```
val isEmp = user => bcEmployees.value.contains(user)
```

That's all—everything else stays unchanged. The last thing to do here is to finally filter the ordered DataFrame using your newly created isEmployee UDF function:

```
import sqlContext.implicits._
val filtered = ordered.filter(isEmployee($"login"))
filtered.show()
```

By writing the previous line, you basically tell filter to apply the isEmployee UDF on the login column. If isEmployee returns true, the row gets included in the filtered DataFrame.

Listing 3.1 Complete source code of the program

```
package org.sia.chapter03App
import org.apache.spark.sql.SparkSession
import scala.io.Source.fromFile

object App {

  def main(args : Array[String]) {
    // TODO expose appName and master as app. params
    val spark = SparkSession.builder()
        .appName("GitHub push counter")
        .master("local[*]")
        .getOrCreate()

        val sc = spark.sparkContext

    // TODO expose inputPath as app. param
    val homeDir = System.getenv("HOME")
    val inputPath = homeDir + "/sia/github-archive/2015-03-01-0.json"
    val ghLog = spark.read.json(inputPath)

    val pushes = ghLog.filter("type = 'PushEvent'")
    val grouped = pushes.groupBy("actor.login").count
    val ordered = grouped.orderBy(grouped("count").desc)

    // TODO expose empPath as app. param
    val empPath = homeDir + "/first-edition/ch03/ghEmployees.txt"
    val employees = Set() ++ (
      for {
        line <- fromFile(empPath).getLines
      } yield line.trim
    )
    val bcEmployees = sc.broadcast(employees)        <- Broadcasts the employees set


    import spark.implicits._
    val isEmp = user => bcEmployees.value.contains(user)
    val isEmployee = spark.udf.register("SetContainsUdf", isEmp)
    val filtered = ordered.filter(isEmployee($"login"))
    filtered.show()
  }
}
```

If you were writing this application for real, you would parameterize `appName`, `appMaster`, `inputPath`, and `empPath` before sending the application to be tested in a production-simulation environment. Who knows which Spark cluster the application is ultimately going to run on? Even if you knew all the parameters in advance, it would still be prudent to specify those parameters from the outside. It makes the application more flexible. One obvious consequence of hard-coding the parameters in the application code is that every time a parameter changes, the application has to be recompiled.

To run the application, follow these steps:

1. In the top menu, go to Run > Run Configurations.
2. At left in the dialog, select Scala Application > App$.
3. Click Run.

The top 20 commit counts are displayed at the bottom of your Eclipse output:

```
+--------------+-----+
|         login|count|
+--------------+-----+
|  KenanSulayman|   72|
|     manuelrp07|   45|
|        Somasis|   26|
|direwolf-github|   24|
|EmanueleMinotto|   22|
|        hansliu|   21|
|         digipl|   20|
|      liangyali|   19|
|       fbennett|   18|
|         shryme|   18|
|      jmarkkula|   18|
|        chapuni|   18|
|         qeremy|   16|
|     martagalaz|   16|
|     MichaelCTH|   15|
|        mfonken|   15|
|         tywins|   14|
|         lukeis|   12|
|     jschnurrer|   12|
|    eventuserum|   12|
+--------------+-----+
```

That's it! You have an application that's working on a one-hour subset of data.

It's obvious what you need to tackle next. The task was to run the report daily, which means you need to include all 24 JSON files in the calculation.

3.2.7 Using the entire dataset

First, let's create a new Scala source file, to avoid messing with the working one-hour example. In the Package Explorer, right-click the App.scala file, select Copy, right-click App.scala again, and choose Paste. In the dialog that appears, enter `GitHubDay.scala` as the object's name, and click OK. When you double-click the new file to open it, notice the red around the filename.

In the GitHubDay.scala file, you can see the reason for the error: the object's name[10] is still `App`. Piece of cake: rename the object `App` to `GitHubDay`, and—nothing happens. Still red. Press Ctrl-S to save the file. Now the red is gone.

Next, open the `SparkSession` scaladoc (http://mng.bz/j9As) to see whether there is a way to create a `DataFrame` by ingesting multiple files with a single command.

[10] Objects are Scala's singletons. For more details, see http://mng.bz/y2ja.

Although the `read` method in the scaladoc says nothing about ingesting multiple files, let's try it.

Change the line starting with `val inputPath` to include all JSON files in the github-archive folder, like this:

```
val inputPath = homeDir + "/sia/github-archive/*.json"
```

Run the application. In about 90 seconds, depending on your machine, you get the result for March 1, 2015:

```
+---------------+-----+
|          login|count|
+---------------+-----+
|  KenanSulayman| 1727|
|direwolf-github|  561|
|         lukeis|  288|
|           keum|  192|
|        chapuni|  184|
|     manuelrp07|  104|
|         shryme|  101|
|            uqs|   90|
|    jefflevesque|   79|
|        BitKiwi|   68|
|         qeremy|   66|
|        Somasis|   59|
|         jvodan|   57|
|     BhawanVirk|   55|
|       Valicek1|   53|
|      evelynluu|   49|
|  TheRingMaster|   47|
|    larperdoodle|   42|
|         digipl|   42|
|      jmarkkula|   39|
+---------------+-----+
```

Yup, this really works. There is just one more thing to do: you need to assure yourself that all the dependencies are available when the application is run, so it can be used from anywhere and run on any Spark cluster.

3.3 Submitting the application

The application will be run on your in-house Spark cluster. When an application is run, it must have access to all the libraries it depends on; and because your application will be shipped off to be executed on a Spark cluster, you know it'll have access to all Spark libraries and their dependencies (Spark is always installed on all nodes of a cluster).

To visualize the application's dependencies, open pom.xml from the root of your project and switch to the Dependency Hierarchy tab. You can see that your application depends only on libraries that are available in your cluster (`scala`, `spark-core`, and `spark-sql`).

After you finish packaging the application, you'll probably first send it to the testing team, so it can be tested before the word *production* is even spoken out loud. But there is a potential problem: you can't be 100% sure that the Spark operations team (with their custom Spark builds) include `spark-sql` when they prepare the testing environment. It may well happen that you send your app to be tested, only to get an angry email.

3.3.1 Building the uberjar

You have two options for including additional JAR files in Spark programs that are going to be run in production (there are others ways, but only these two are production-grade).

- Use the `--jars` parameter of the `spark-submit` script, which will transfer all the listed JARs to the executors.
- Build a so-called *uberjar*: a JAR that contains all needed dependencies.

To avoid fiddling with libraries manually[11] on multiple clusters, let's build an uberjar.

To illustrate this concept, you'll introduce another dependency into the application: another library you need to include and distribute along with it. Let's say you would like to include the `commons-email` library for simplified email sending, provided by Apache Commons (although you won't be using it in the code of the current example). The uberjar will need to contain only the code you wrote, the `commons-email` library, and all the libraries that `commons-email` depends on.

Add the following dependency to pom.xml (right below the Spark SQL dependency):

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-email</artifactId>
  <version>1.3.1</version>
  <scope>compile</scope>
</dependency>
```

Looking again at the dependency hierarchy in pom.xml, you can see that `commons-email` depends on the `mail` and `activation` libraries. Some of those libraries may, in turn, have their own dependencies. For instance, `mail` also depends on `activation`. This dependency tree can grow arbitrarily long and branchy.

You would probably start worrying, if you didn't know about `maven-shade-plugin`. Yes, Maven comes to the rescue yet again: `maven-shade-plugin` is used to build uberjars. We've also included a `maven-shade-plugin` configuration in pom.xml.

Because you wish to include the `commons-email` library in the uberjar, its *scope* needs to be set to `compile`. Maven uses the `scope` property of each dependency to determine the phase during which a dependency is required. `compile`, `test`, `package`, and `provided` are some of the possible values for `scope`.

If `scope` is set to `provided`, the library and all its dependencies won't be included in uberjar. If you omit the scope, Maven defaults to `compile`, which means the library is needed during application compilation and at runtime.

[11] For example, in the test environment, you can provide libraries locally and let the driver (a machine from which you connect to a cluster) expose the libraries over a provisional HTTP server to all other nodes (see "Advanced Dependency Management" at http://spark.apache.org/docs/latest/submitting-applications.html).

As always, after you change pom.xml, update the project by right-clicking its root and selecting Maven > Update Project; then click OK without changing the defaults. Depending on the type of changes made in pom.xml, updating the project may or may not be necessary. If a project update is needed and you haven't yet done so, Eclipse draws your attention to the fact by displaying an error in the Problems view and putting a red marker on the project root.

3.3.2 *Adapting the application*

To adapt your application to be run using the `spark-submit` script, you need to make some modifications. First remove the assignment of the application name and Spark master parameters from `SparkConf` (because those will be provided as arguments to `spark-submit`) and instead provide an empty `SparkConf` object when creating `SparkContext`. The final result is shown in the following listing.

Listing 3.2 Final version of the adapted, parameterized application

```
package org.sia.chapter03App

import scala.io.Source.fromFile
import org.apache.spark.sql.SparkSession

object GitHubDay {
  def main(args : Array[String]) {
    val spark = SparkSession.builder().getOrCreate()

    val sc = spark.sparkContext

    val ghLog = spark.read.json(args(0))

    val pushes = ghLog.filter("type = 'PushEvent'")
    val grouped = pushes.groupBy("actor.login").count
    val ordered = grouped.orderBy(grouped("count").desc)

    val employees = Set() ++ (
      for {
        line <- fromFile(args(1)).getLines
      } yield line.trim
    )
    val bcEmployees = sc.broadcast(employees)

    import spark.implicits._
    val isEmp = user => bcEmployees.value.contains(user)
    val sqlFunc = spark.udf.register("SetContainsUdf", isEmp)
    val filtered = ordered.filter(sqlFunc($"login"))

    filtered.write.format(args(3)).save(args(2))
  }
}
```

The last line saves the result to an output file, but you do it in such a way that the person who's invoking `spark-submit` decides on the path and format of the written output (currently available built-in formats: JSON, Parquet,[12] and JDBC).

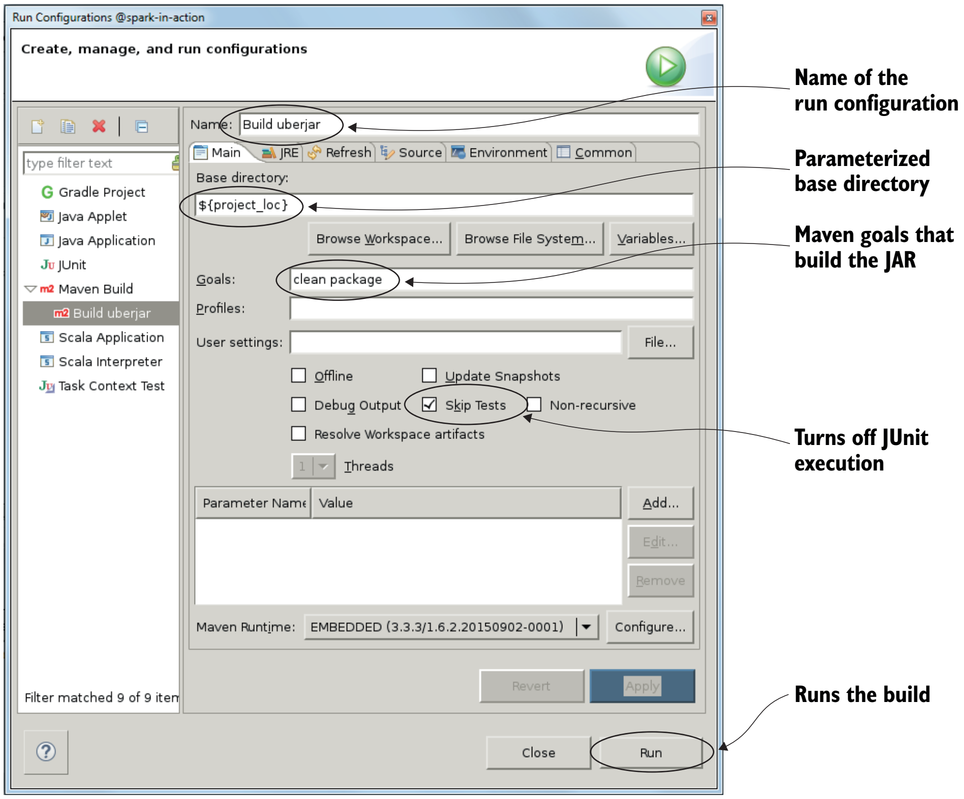
[12] Parquet is a fast columnar file format that contains a schema: http://parquet.apache.org/.

So, the application will take four arguments:

- Path to the input JSON files
- Path to the employees file
- Path to the output file
- Output format

To build the uberjar, select the project root in the Package Explorer, and then choose Run > Run Configurations. Choose Maven Build, and click New Launch Configuration. In the dialog that appears (see figure 3.6), enter `Build uberjar` in the Name field, click the Variables button, and choose project_loc in the dialog (which fills the Base Directory field with the `${project_loc}` value).[13]

In the Goals field, enter `clean package`. Select the Skip Tests check box, save the configuration by clicking Apply, and trigger the build by clicking Run.


Run Configurations @spark-in-action

Create, manage, and run configurations

Name: Build uberjar

Base directory: ${project_loc}

Goals: clean package

Profiles:

User settings:

☐ Offline ☐ Update Snapshots
☐ Debug Output ☑ Skip Tests ☐ Non-recursive
☐ Resolve Workspace artifacts
1 Threads

Parameter Name	Value

Maven Runtime: EMBEDDED (3.3.3/1.6.2.20150902-0001)

Filter matched 9 of 9 items

Figure 3.6 Specifying the run configuration for uberjar packaging

[13] You should use a variable so that any future project can also use this run configuration (which wouldn't be possible if you hard-coded the project's path with, for example, a Browse Workspace button). This way, the build will be triggered against the project that is currently selected in the Package Explorer.

After running the build, you should see a result similar to the following (truncated for a cleaner view):

```
[INFO] --- maven-shade-plugin:2.4.2:shade (default) @ chapter03App ---
[INFO] Including org.scala-lang:scala-library:jar:2.10.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing /home/spark/workspace/chapter03App/target/chapter03App...
[INFO] Dependency-reduced POM written at: /home/spark/workspace/chapter0...
[INFO] Dependency-reduced POM written at: /home/spark/workspace/chapter0...
     [INFO] Dependency-reduced POM written at: /home/spark/workspace/
     chapter0...
[INFO] --------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] --------------------------------------------------------------------
[INFO] Total time: 23.670 s
[INFO] Finished at: 2016-04-23T10:55:06+00:00
[INFO] Final Memory: 20M/60M
[INFO] --------------------------------------------------------------------
```

You should now have a file named chapter03App-0.0.1-SNAPSHOT.jar in your project's target folder. Because the file isn't visible in Eclipse, check the filesystem by right-clicking the target folder in the Package Explorer and selecting Show In > System Explorer (or open a terminal and navigate to that folder manually).

Let's do a quick test on a local Spark installation. That should flush out most of the potential errors.

3.3.3 Using spark-submit

The Spark documentation on submitting applications (http://mng.bz/WY2Y) gives docs and examples of using the `spark-submit` shell script:

```
spark-submit \
    --class <main-class> \
    --master <master-url> \
    --deploy-mode <deploy-mode> \
    --conf <key>=<value> \
    ... # other options
    <application-jar> \
    [application-arguments]
```

`spark-submit` is a helper script that's used to submit applications to be executed on a Spark cluster. It's located in the bin subfolder of your Spark installation.

Before submitting the application, open another terminal to display the application log as it runs. Recall in chapter 2 that you changed the default log4j configuration so that the complete log is written in /usr/local/spark/logs/info.log. You can still see the log in real time by using the `tail` command, which displays content from the end of a file. It's similar to `head`, which you used to get the first line of that JSON file in section 3.3.2.

If you supply the `-f` parameter, `tail` waits until the content is appended to the file; as soon as that happens, `tail` outputs it in the terminal. Issue the following command in your second terminal:

```
$ tail -f /usr/local/spark/logs/info.log
```

Bring the first terminal back to the front, and enter the following:

```
$ spark-submit --class org.sia.chapter03App.GitHubDay --master local[*]
➥ --name "Daily GitHub Push Counter" chapter03App-0.0.1-SNAPSHOT.jar
➥ "$HOME/sia/github-archive/*.json"
➥ "$HOME/first-edition/ch03/ghEmployees.txt"
➥ "$HOME/sia/emp-gh-push-output" "json"
```

Pasting blocks of code into the Spark Scala shell

When submitting a Python application, you specify a Python file name instead of the application JAR file. You also skip the `--class` argument. For the GitHubDay example:

```
$ spark-submit --master local[*] --name "Daily GitHub Push Counter"
➥ GitHubDay.py "$HOME/sia/github-archive/*.json"
➥ "$HOME/sia/ghEmployees. txt" "$HOME/sia/emp-gh-push-output" "json"
```

For more information, see the Python version of the GitHubDay application in our online repository.

One or two minutes later, depending on your machine, the command will end without any errors. List the contents of the output folder:

```
$ cd $HOME/sia/emp-gh-push-output
$ ls -la
```

You see as many as 42 files (filenames are shortened here for nicer output):

```
-rw-r--r-- 1 spark spark  720 Apr 23 09:40 part-r-00000-b24f792c-...
-rw-rw-r-- 1 spark spark   16 Apr 23 09:40 .part-r-00000-b24f792c-....crc
-rw-r--r-- 1 spark spark  529 Apr 23 09:40 part-r-00001-b24f792c-...
-rw-rw-r-- 1 spark spark   16 Apr 23 09:40 .part-r-00001-b24f792c-....crc
-rw-r--r-- 1 spark spark  328 Apr 23 09:40 part-r-00002-b24f792c-...
-rw-rw-r-- 1 spark spark   12 Apr 23 09:40 .part-r-00002-b24f792c-....crc
-rw-r--r-- 1 spark spark  170 Apr 23 09:40 part-r-00003-b24f792c-...
-rw-rw-r-- 1 spark spark   12 Apr 23 09:40 .part-r-00003-b24f792c-....crc
-rw-r--r-- 1 spark spark    0 Apr 23 09:40 part-r-00004-b24f792c-...
-rw-rw-r-- 1 spark spark    8 Apr 23 09:40 .part-r-00004-b24f792c-....crc
...
-rw-r--r-- 1 spark spark    0 Apr 22 19:20 _SUCCESS
-rw-rw-r-- 1 spark spark    8 Apr 22 19:20 ._SUCCESS.crc
```

The presence of the _SUCCESS file signifies that the job finished successfully. The crc files are used to verify file validity by calculating the cyclic redundancy check (CRC)[14] code for each data file. The file named ._SUCCESS.crc signifies that the CRC calculation for all the files was successful.

To see the contents of the first data file, you can use the `cat` command, which sends the contents of the file to the standard output (terminal):

```
$ cat $HOME/sia/emp-gh-push-output/part-r-00000-b24f792c-c0d0-425b-85db-
➥ 3322aab8f3e0
{"login":"KenanSulayman","count":1727}
{"login":"direwolf-github","count":561}
{"login":"lukeis","count":288}
{"login":"keum","count":192}
{"login":"chapuni","count":184}
{"login":"manuelrp07","count":104}
{"login":"shryme","count":101}
{"login":"uqs","count":90}
{"login":"jefflevesque","count":79}
{"login":"BitKiwi","count":68}
{"login":"qeremy","count":66}
{"login":"Somasis","count":59}
{"login":"jvodan","count":57}
{"login":"BhawanVirk","count":55}
{"login":"Valicek1","count":53}
{"login":"evelynluu","count":49}
{"login":"TheRingMaster","count":47}
{"login":"larperdoodle","count":42}
{"login":"digipl","count":42}
{"login":"jmarkkula","count":39}
```

3.4 Summary

- Online resources describe how to use IntelliJ IDEA with Spark, but Eclipse resources are still hard to come by. That's why we chose Eclipse for writing the Spark programs in this chapter.
- We've prepared an Archetype called `scala-archetype-sparkinaction` (available from our GitHub repository), which is used to create a starter Spark project in which the versions and dependencies are taken care of.
- The GitHub archive site (https://www.githubarchive.org/) provides GitHub archives for arbitrary time periods.
- Spark SQL and `DataFrame` (which is a `DataSet` containing `Row` objects) provide a means for ingesting JSON data into Spark.
- `SparkSession`'s `jsonFile` method provides a means for ingesting JSON data. Each line in an input file needs to be a complete JSON object.

[14] A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

- `DataSet`'s `filter` method can parse SQL expressions and return a subset of the data.
- You can run a Spark application directly from Eclipse.
- The `SparkSession` class contains the `udf` method, which is used to register user-defined functions.
- Broadcast variables are used to send a variable exactly once to each node in a cluster.
- `maven-shade-plugin` is used to build an uberjar containing all of your application's dependencies.
- You can run a Spark application on a cluster using the `spark-submit` script.

DATA/PROGRAMMING

Spark IN ACTION

Zečević • Bonaći

Big data systems distribute datasets across clusters of machines, making it a challenge to efficiently query, stream, and interpret them. Spark can help. It is a processing system designed specifically for distributed data. It provides easy-to-use interfaces, along with the performance you need for production-quality analytics and machine learning. And Spark 2 adds improved programming APIs, better performance, and countless other upgrades.

Spark in Action teaches you the theory and skills you need to effectively handle batch and streaming data using Spark. You'll get comfortable with the Spark CLI as you work through a few introductory examples. Then, you'll start programming Spark using its core APIs. Along the way, you'll work with structured data using Spark SQL, process near-real-time streaming data, apply machine learning algorithms, and munge graph data using Spark GraphX. For a zero-effort startup, you can download the preconfigured virtual machine ready for you to try the book's code.

What's Inside

- Updated for Spark 2.0
- Real-life case studies
- Spark DevOps with Docker
- Examples in Scala, and online in Java and Python

Written for experienced programmers with some background in big data or machine learning.

Petar Zečević and **Marko Bonaći** are seasoned developers heavily involved in the Spark community.

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

MANNING

$49.99 / Can $57.99 [INCLUDING eBOOK]

Free eBook
SEE INSERT

"Dig in and get your hands dirty with one of the hottest data processing engines today. A great guide."
—Jonathan Sharley
Pandora Media

"Must-have! Speed up your learning of Spark as a distributed computing framework."
—Robert Ormandi, Yahoo!

"An ambitiously comprehensive overview of Spark and its diverse ecosystem."
—Jonathan Miller, Optensity

"An easy-to-follow, step-by-step guide."
—Gaurav Bhardwaj
3Pillar Global

ISBN-13: 978-1-61729-260-6
ISBN-10: 1-61729-260-5
54999
9 781617 292606