Covers Play, Akka, and Reactive Streams

Reactive Web Applications

Manuel Bernhardt

Foreword by James Roper

SAMPLE CHAPTER

MANNING

Reactive Web Applications

by Manuel Bernhardt

Chapter 2

brief contents

2 Your first reactive web application

This chapter covers

- Creating a new Play project
- Streaming data from a remote server and broadcasting it to clients
- Dealing with failure

In the previous chapter, we talked about the key benefits of adopting a reactive approach to web application design and operation, and you saw that the Play Framework is a good technology for this. Now it's time to get your hands dirty and build a reactive web application. We'll build a simple application that connects to the Twitter API to retrieve a stream of tweets and send them to clients using WebSockets.

2.1 Creating and running a new project

An easy way to start a new Play project is to use the Lightbend Activator, which is a thin wrapper around Scala's sbt build tool that provides templates for creating new projects. The following instructions assume that you have the Activator

installed on your computer. If you don't, appendix A provides detailed instructions for installing it.

Let's get started by creating a new project called "twitter-stream" in the workspace directory, using the `play-scala-v24` template:

```
~/workspace » activator new twitter-stream play-scala-2.4
```

This will start the process of creating a new project with Activator, using the template as a scaffold:

```
Fetching the latest list of templates...

OK, application "twitter-stream" is being created using the "play-scala-2.4"
➥ template.

To run "twitter-stream" from the command line, "cd twitter-stream" then:
/Users/mb/workspace/twitter-stream/activator run

To run the test for "twitter-stream" from the command line,
➥ "cd twitter-stream" then:
/Users/mb/workspace/twitter-stream/activator test

To run the Activator UI for "twitter-stream" from the command line,
➥ "cd twitter-stream" then:
/Users/mb/workspace/twitter-stream/activator ui
```

You can now run this application from the project directory:

```
~/workspace » cd twitter-stream
~/workspace/twitter-stream » activator run
```

If you point your browser to http://localhost:9000, you'll see the standard welcome page for a Play project. At any time when running a Play project, you can access the documentation at http://localhost:9000/@documentation.

> **PLAY RUNTIME MODES** Play has a number of runtime modes. In *dev mode* (triggered with the `run` command), the sources are constantly watched for changes, and the project is reloaded with any new changes for rapid development. *Production mode*, as its name indicates, is used for the production operation of a Play application. Finally, *test mode* is active when running tests, and it's useful for retrieving specific configuration settings for the test environment.

Besides running the application directly with the `activator run` command, it's possible to use an interactive console. You can stop the running application by hitting Ctrl-C and start the console simply by running `activator`:

```
~/workspace/twitter-stream » activator
```

That will start the console, as follows:

```
[info] Loading project definition from
          /Users/mb/workspace/twitter-stream/project
[info] Set current project to twitter-stream
          (in build file:/Users/mb/workspace/twitter-stream/)
[twitter-stream] $
```

Once you're in the console, you can run commands such as `run`, `clean`, `compile`, and so on. Note that this console is not Play-specific, but common to all sbt projects. Play adds a few commands to it and makes it more suited to web application development.

Table 2.1 lists some useful commands:

Table 2.1 Useful sbt console commands for working with Play

Command	Description
`run`	Runs the Play project in dev mode
`start`	Starts the Play project in production mode
`clean`	Cleans all compiled classes and generated sources
`compile`	Compiles the project
`test`	Runs the tests
`dependencies`	Shows all the library dependencies of the project, including transitive ones
`reload`	Reloads the project settings if they have been changed

When you start the application in the console with `run`, you can stop it and return to the console by pressing Ctrl-D.

> **AUTO-RELOADING** By prepending a command with `~`, such as `~ run` or `~ compile`, you can instruct sbt to listen to changes in the source files. In this way, every time a source file is saved, the project is automatically recompiled or reloaded.

Now that you're all set to go, let's start building a simple reactive application, which, as you may have guessed from the name of the empty project we've created, has something to do with Twitter.

What we'll build is an application that will connect to one of Twitter's streaming APIs, transform the stream asynchronously, and broadcast the transformed stream to clients using WebSocket, as illustrated in figure 2.1. We'll start by building a small Twitter client to stream the data, and then build the transformation pipeline that we'll plug into a broadcasting mechanism.

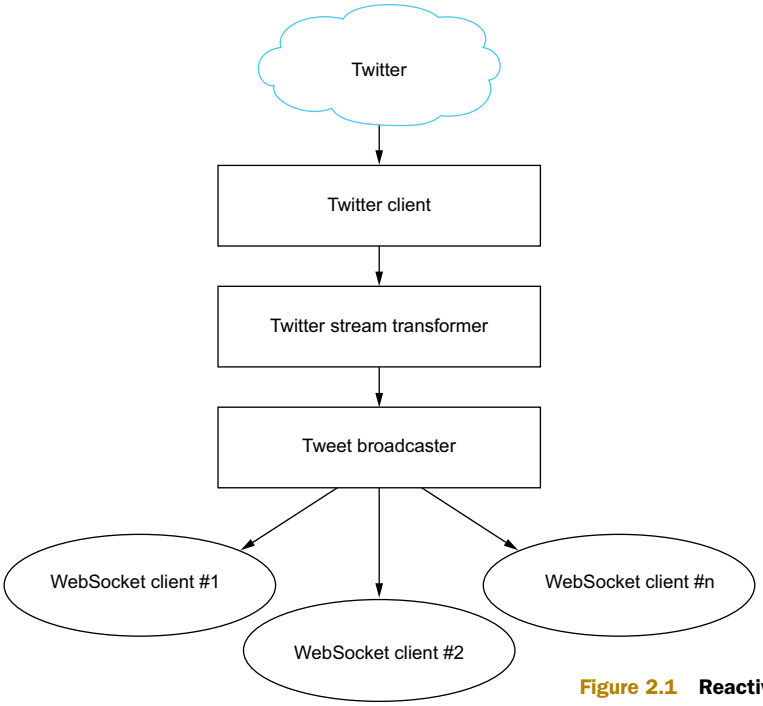


Figure 2.1 Reactive Twitter broadcaster

2.2 Connecting to Twitter's streaming API

To get started, we'll connect to the Twitter filter API.[1] At this point, we'll just focus on getting data from Twitter and displaying it on the console—we'll deal with sending it to clients connecting to our application at a later stage.

Start by opening the project in your favorite IDE. Most modern IDEs have extensions to support Play projects nowadays, and you can find resources on the topic in the Play documentation (www.playframework.com/documentation), so we won't look into setting up various flavors of IDEs here.

2.2.1 Getting the connection credentials to the Twitter API

Twitter uses the OAuth authentication mechanism to secure its API. To use the API, you need a Twitter account and OAuth consumer key and tokens. Register with Twitter (if you haven't already), and then you can go to https://apps.twitter.com where you can request access to the API for an application. This way, you'll get an API key and an API secret, which together represent the consumer key. In addition to these keys, you'll need to generate request tokens (in the Details tab of the Twitter Apps web application). At the end of this process, you should have access to four values:

[1] The Twitter API documentation can be found at https://dev.twitter.com/streaming/reference/post/statuses/filter.

- The API key
- The API secret
- An access token
- An access token secret

Once you have all the necessary keys, you'll need to add them to the application configuration in conf/application.conf. This way, you'll be able to retrieve them easily from the application later on. Add the keys at the end of the file as follows:

```
# Twitter
twitter.apiKey="<your api key>"
twitter.apiSecret="<your api secret>"
twitter.token="<your access token>"
twitter.tokenSecret="<your access token secret>"
```

2.2.2 Working around a bug with OAuth authentication

As a technical book author, I want my examples to flow and my code to look simple, beautiful, and elegant. Unfortunately the reality of software development is that bugs can be anywhere, even in projects with a very high code quality, which the Play Framework definitely is. One of those bugs has its origins in the async-http-client library that Play uses, and it plagues the 2.4.x series of the Play Framework. It can't be easily addressed without breaking binary compatibility, which is why it will likely not be fixed within the 2.4.x series.[2]

More specifically, this bug breaks the OAuth authentication mechanism when a request contains characters that need to be encoded (such as the `@` or `#` characters). As a result, we have to use a workaround in all chapters making use of the Twitter API. Open the build.sbt file at the root of the project, and add the following line:

```
libraryDependencies += "com.ning" % "async-http-client" % "1.9.29"
```

2.2.3 Streaming data from the Twitter API

The first thing we'll do now is add some functionality to the existing `Application` controller in app/controllers/Application.scala. When you open the file, it should look rather empty, like this:

```
class Application extends Controller {

  def index = Action {
    Ok(views.html.index("Your new application is ready."))
  }

}
```

The `index` method defines a means for obtaining a new `Action`. Actions are the mechanism Play uses to deal with incoming HTTP requests, and you'll learn a lot more about them in chapter 4.

[2] https://github.com/playframework/playframework/pull/4826

Start by adding a new `tweets` action to the controller.

Listing 2.1 Defining a new `tweets` action

```
import play.api.mvc._

class Application extends Controller {
  def tweets = Action {
    Ok
  }
}
```

This action won't do anything other than return a 200 Ok response when accessed. To access it, we first need to make it accessible in Play's routes. Open the conf/routes file and add a new route to the newly created action, so you get the following result.

Listing 2.2 Route to the newly created `tweets` action

```
# Routes
# This file defines all application routes
# (Higher priority routes first)
# ~~~~

# Home page
GET     /               controllers.Application.index
GET     /tweets         controllers.Application.tweets

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file    controllers.Assets.at(path="/public", file)
```

Now when you run the application and access the /tweets file, you should get an empty page in your browser. This is great, but not very useful. Let's go one step further by retrieving the credentials from the configuration file.

Go back to the app/controllers/Application.scala controller and extend the `tweets` action as follows.

Listing 2.3 Retrieving the configuration

Uses Action.async to return a Future of a result for the next step

Retrieves the Twitter credentials from application.conf

```
import play.api.libs.oauth.{ConsumerKey, RequestToken}
import play.api.Play.current
import scala.concurrent.Future
import play.api.libs.concurrent.Execution.Implicits._

def tweets = Action.async {
  val credentials: Option[(ConsumerKey, RequestToken)] = for {
    apiKey <- Play.configuration.getString("twitter.apiKey")
    apiSecret <- Play.configuration.getString("twitter.apiSecret")
    token <- Play.configuration.getString("twitter.token")
    tokenSecret <- Play.configuration.getString("twitter.tokenSecret")
```

```
    } yield (
        ConsumerKey(apiKey, apiSecret),
        RequestToken(token, tokenSecret)
      )

    credentials.map { case (consumerKey, requestToken) =>
      Future.successful {
        Ok
      }
    } getOrElse {
      Future.successful {
        InternalServerError("Twitter credentials missing")
      }
    }
  }
```

Wraps the result in a successful Future block until the next step

Returns a 500 Internal Server Error if no credentials are available

Wraps the result in a successful Future block to comply with the return type

Now that we have access to our Twitter API credentials, we'll see whether we can get anything back from Twitter. Replace the simple `Ok` result in app/controllers/Application.scala with the following bit of code to connect to Twitter.

Listing 2.4 First attempt at connecting to the Twitter API

```
// ...
import play.api.libs.ws._

def tweets = Action.async {
  credentials.map { case (consumerKey, requestToken) =>
    WS
      .url("https://stream.twitter.com/1.1/statuses/filter.json")
      .sign(OAuthCalculator(consumerKey, requestToken))
      .withQueryString("track" -> "reactive")
      .get()
      .map { response =>
        Ok(response.body)
      }
    } getOrElse {
      Future.successful {
        InternalServerError("Twitter credentials missing")
      }
    }
}

def credentials: Option[(ConsumerKey, RequestToken)] = for {
  apiKey <- Play.configuration.getString("twitter.apiKey")
  apiSecret <- Play.configuration.getString("twitter.apiSecret")
  token <- Play.configuration.getString("twitter.token")
  tokenSecret <- Play.configuration.getString("twitter.tokenSecret")
} yield (
  ConsumerKey(apiKey, apiSecret),
  RequestToken(token, tokenSecret)
)
```

The API URL

OAuth signature of the request

Specifies a query string parameter

Executes an HTTP GET request

Play's WS library lets you easily access the API by signing the request appropriately following the OAuth standard. You're currently tracking all the tweets that contain the word "reactive," and for the moment you only log the status of the response from Twitter to see if you can connect with these credentials. This may look fine at first sight, but there's a catch: if you were to execute the preceding code, you wouldn't get any useful results. The streaming API, as its name indicates, returns a (possibly infinite) stream of tweets, which means that the request would never end. The WS library would time out after a few seconds, and you'd get an exception in the console.

What you need to do, therefore, is consume the stream of data you get. Let's rewrite the previous call to `WS` and use an *iteratee* (discussed in a moment) to simply print the results you get back.

Listing 2.5 Printing out the stream of data from Twitter

```
// ...
import play.api.libs.iteratee._
import play.api.Logger

def tweets = Action.async {

  val loggingIteratee = Iteratee.foreach[Array[Byte]] { array =>
    Logger.info(array.map(_.toChar).mkString)
  }

  credentials.map { case (consumerKey, requestToken) =>
    WS
      .url("https://stream.twitter.com/1.1/statuses/filter.json")
      .sign(OAuthCalculator(consumerKey, requestToken))
      .withQueryString("track" -> "reactive")
      .get { response =>
        Logger.info("Status: " + response.status)
        loggingIteratee
      }.map { _ =>
        Ok("Stream closed")
      }
}

def credentials = ...
```

Defines a logging iteratee that consumes a stream asynchronously and logs the contents when data is available

Sends a GET request to the server and retrieves the response as a (possibly infinite) stream

Feeds the stream directly into the consuming loggingIteratee; the contents aren't loaded in memory first but are directly passed to the iteratee

Returns a 200 Ok result when the stream is entirely consumed or closed

QUICK INTRODUCTION TO ITERATEES

An iteratee is a construct that allows you to consume streams of data asynchronously; it's one of the cornerstones of the Play Framework. Iteratees are typed with input and output types: an `Iteratee[E, A]` consumes chunks of `E` to produce one or more `A`'s.

In the case of the `loggingIteratee` in listing 2.5, the input is an `Array[Byte]` (because you retrieve a raw stream of data from Twitter), and the output is of type

`Unit`, which means you don't produce any result other than the data logged out on the console.

The counterpart of an iteratee is an *enumerator*. Just as the iteratee is an asynchronous consumer of data, the enumerator is an asynchronous producer of data: an `Enumerator[E]` produces chunks of `E`.

Finally, there's another piece of machinery that lets you transform streaming data on the fly, called an *enumeratee*. An `Enumeratee[From, To]` takes chunks of type `From` from an enumerator and transforms them into chunks of type `To`.

On a conceptual level, you can think of an enumerator as being a faucet, an enumeratee as being a filter, and an iteratee as being a glass, as in figure 2.2.

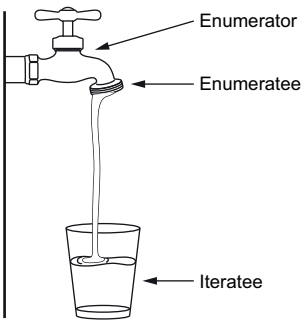


Figure 2.2 Enumerators, enumeratees, and iteratees

Let's go back to our `loggingIteratee` for a second, defined as follows:

```
val loggingIteratee = Iteratee.foreach[Array[Byte]] { array =>
    Logger.info(array.map(_.toChar).mkString)
}
```

The `Iteratee.foreach[E]` method creates a new iteratee that consumes each input it receives by performing a side-effecting action (of result type `Unit`). It's important to understand here that `foreach` isn't a method of an iteratee, but rather a method of the Iteratee library used to create a "foreach" iteratee. The Iteratee library offers many other methods for building iteratees, and we'll look at some of them later on.

At this point, you may wonder how this is any different from using other streaming mechanisms, such as `java.io.InputStream` and `java.io.OutputStream`. As mentioned earlier, iteratees let you manipulate streams of data asynchronously. In practice, this means that these streams won't hold on to a thread in the absence of new data. Instead, the thread that they use will be freed for use by other tasks, and only when there's a signal that new data is arriving will the streaming continue. In contrast, a `java.io.OutputStream` blocks the thread it's using until new data is available.

> **THE FUTURE OF ITERATEES IN PLAY** At the time of writing, Play is largely built on top of iteratees, enumerators, and enumeratees. *Reactive Streams* is a new standard for nonblocking stream manipulation with backward pressure on the JVM that we'll talk about in chapter 9. Although we use iteratees in this chapter and later in the book, the roadmap for the next major release of Play is to gradually replace iteratees with *Akka Streams*, which implement the Reactive Streams standard. Chapter 9 will cover this toolset as well as how to convert from iteratees to Akka Streams and vice versa.

Let's now get back to our application. Our approach to turning the `Array[Byte]` into a `String` is very crude (and, as you'll see later, problematic), but if someone were to

tweet about "reactive," we'd be able to see something. If you want to check that things are going well, you can write a tweet yourself, as I just did:

```
[info] application - Status: 200
[info] application - {"created_at":"Fri Sep 19 15:08:07 +0000 2014","id
":512981466662592512,"id_str":"512981466662592512","text":"Writing the
second chapter of my book about #reactive web-applications with #PlayFr
amework. I need a tweet with \"reactive\" for an example.","source":"<a
 href=\"http:\/\/itunes.apple.com\/us\/app\/twitter\/id409789998?mt=12\
" rel=\"nofollow\">Twitter for Mac<\/a>","truncated":false,"in_reply_to
_status_id":null,"in_reply_to_status_id_str":null,"in_reply_to_user_id"
:null,"in_reply_to_user_id_str":null,"in_reply_to_screen_name":null,"us
er":{"id":12876952,"id_str":"12876952","name":"Manuel Bernhardt","scree
n_name":"elmanu","location":"Vienna" ...
```

GETTING MORE TWEETS For all the advantages of reactive applications, the keyword "reactive" is slightly less popular than more common topics on Twitter, so you may want to use another term to get faster-paced data. (One keyword that always works well, and not only on Twitter, is "cat.")

2.2.4 Asynchronously transforming the Twitter stream

Great, you just managed to connect to the Twitter streaming API and display some results! But to do something a bit more advanced with the data, you'll need to parse the JSON representation to manipulate it more easily, as shown in figure 2.3.

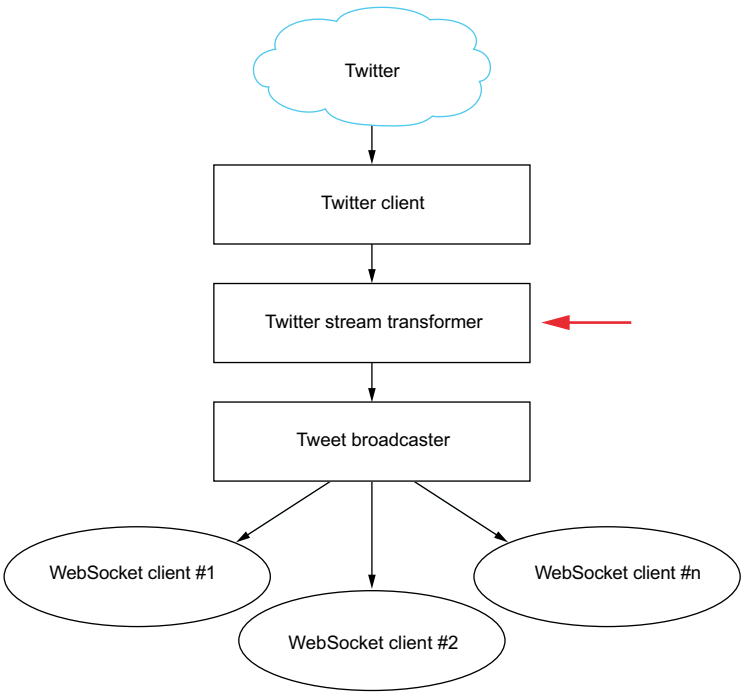


Figure 2.3 Twitter stream transformation step

Play has a built-in JSON library that can take care of parsing textual JSON files into a structured representation that can easily be manipulated. But you first need to pay a little more attention to the data you're receiving, because there are a few things that can go wrong:

- Tweets are encoded in UTF-8, so you need to decode them appropriately, taking into account variable-length encoding.
- In some cases, a tweet is split over several chunks of `Array[Byte]`, so you can't just assume that each chunk can be parsed right away.

These issues are rather complex to solve, and they may take quite some time to get right. Instead of doing it ourselves, let's use the play-extra-iteratees library. Add the following lines to the build.sbt file.

Listing 2.6 Including the play-extra-iteratees library in the project

```
resolvers += "Typesafe private" at
  "https://private-repo.typesafe.com/typesafe/maven-releases"

libraryDependencies +=
  "com.typesafe.play.extras" %% "iteratees-extras" % "1.5.0"
```

To make the changes visible to the project in the console, you need to run the `reload` command (or exit and restart, but `reload` is faster).

Armed with this library, you now have the necessary tools to handle this stream of JSON objects properly:

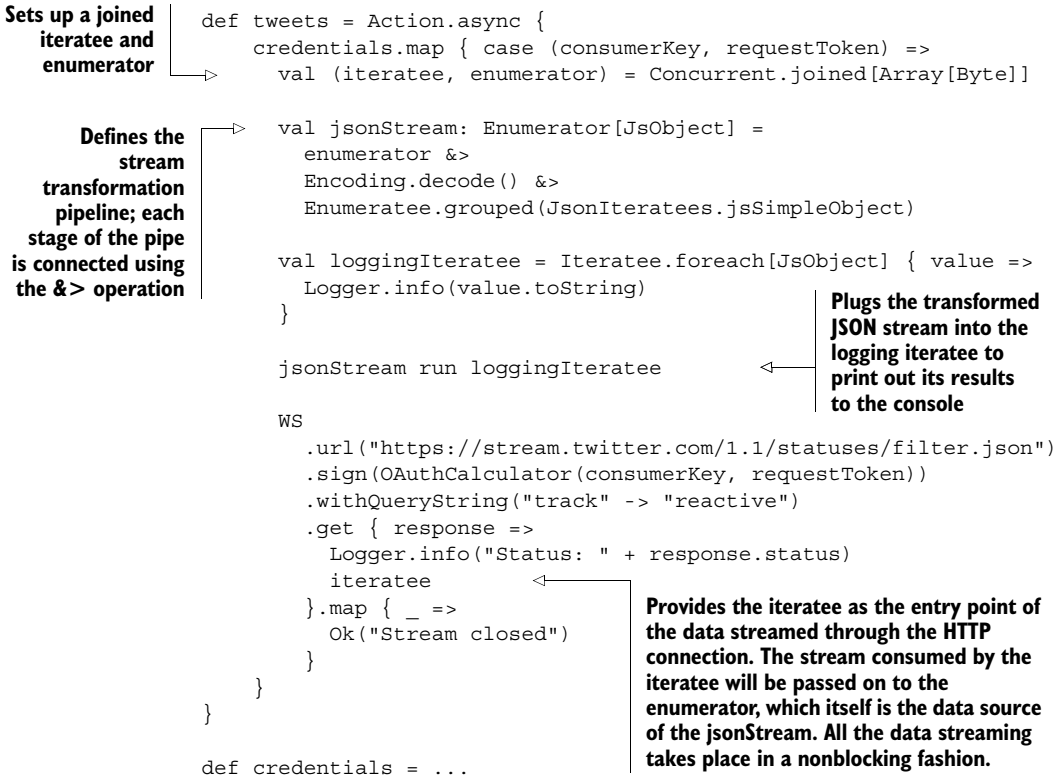
- `play.extras.iteratees.Encoding.decode` will decode the stream of bytes as a UTF-8 string.
- `play.extras.iteratees.JsonIteratees.jsSimpleObject` will parse a single JSON object.
- `play.api.libs.iteratee.Enumeratee.grouped` will apply the `jsSimpleObject` iteratee over and over again until the stream is finished.

We'll start with a stream of `Array[Byte]`, decode it into a stream of `CharString`, and finally parse it into JSON objects of kind `play.api.libs.JsObject` by continuously parsing one JSON object out of the incoming stream of `CharString`. `Enumeratee.grouped` continuously applies the same iteratee over and over to the stream until it's finished.

You can set up the necessary plumbing by evolving your code in app/controllers/Application.conf as follows.

Listing 2.7 Reactive plumbing for the data from Twitter

```
// ...
import play.api.libs.json._
import play.extras.iteratees._
```




```
def tweets = Action.async {
    credentials.map { case (consumerKey, requestToken) =>
      val (iteratee, enumerator) = Concurrent.joined[Array[Byte]]

      val jsonStream: Enumerator[JsObject] =
        enumerator &>
        Encoding.decode() &>
        Enumeratee.grouped(JsonIteratees.jsSimpleObject)

      val loggingIteratee = Iteratee.foreach[JsObject] { value =>
        Logger.info(value.toString)
      }

      jsonStream run loggingIteratee

      WS
        .url("https://stream.twitter.com/1.1/statuses/filter.json")
        .sign(OAuthCalculator(consumerKey, requestToken))
        .withQueryString("track" -> "reactive")
        .get { response =>
          Logger.info("Status: " + response.status)
          iteratee
        }.map { _ =>
          Ok("Stream closed")
        }
    }
}

def credentials = ...
```

The first thing you have to do in this setup is get an enumerator to work with. Iteratees are used to consume streams, whereas enumeratees produce them, and you need a producing pipe so you can add adapters to it. The `Concurrent.joined` method provides you with a connected pair of iteratee and enumerator: whatever data is consumed by the iteratee will be immediately available to the enumerator.

Next, you want to turn the raw `Array[Byte]` into a proper stream of parsed `JsObject` objects. To this end, start off with your enumerator and pipe the results to two transforming enumeratees:

- `Encoding.decode()` to turn the `Array[Byte]` into a UTF-8 representation of type `CharString` (an optimized version of a String proper for stream manipulation, and part of the play-extra-iteratees library)
- `Enumeratee.grouped(JsonIteratees.jsSimpleObject)` to have the stream consumed over and over again by the `JsonIteratees.jsSimpleObject` iteratee

The `jsSimpleObject` iteratee ignores whitespace and line breaks, which is convenient in this case because the tweets coming from Twitter are separated by a line break.

Set up a logging iteratee to print out the parsed JSON object stream, and connect it to the transformation pipeline you just set up using the `run` method of the enumerator.

This method tells the enumerator to start feeding data to the iteratee as soon as some is available.

Finally, by providing the `iteratee` reference to the `get()` method of the WS library, you effectively put the whole mechanism into motion.

If you run this example, you'll now get a stream of tweets printed out, ready to be manipulated for further use.

FASTER JSON PARSING Although the play-extra-iteratees library is very convenient, the JSON tooling it offers isn't optimized for speed; it serves as more of a showcase of what can be done with iteratees. If I wanted to build a pipeline for production use, or where performance matters a lot more than low memory consumption, I'd probably create my own enumeratee and make use of a fast JSON parsing library such as Jackson.

2.3 Streaming tweets to clients using a WebSocket

Now that we have streaming data being sent by Twitter, let's make it available to users of our web application using WebSockets. Figure 2.4 provides an overview of what we want to achieve.

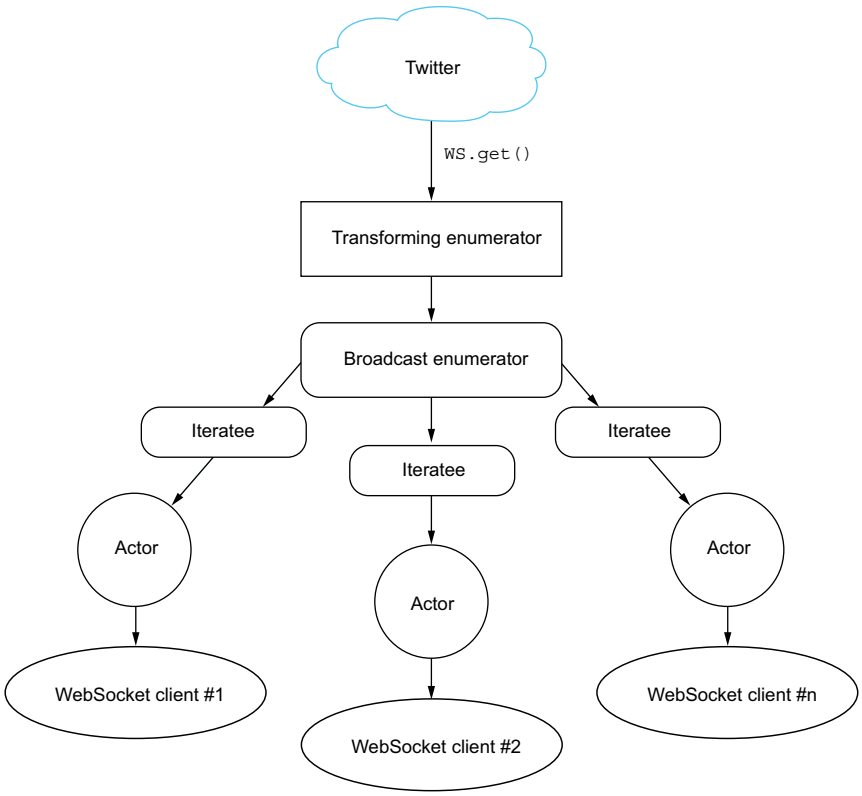


Figure 2.4 Reactive pipeline from Twitter to the client's browser

We want to connect once to Twitter and broadcast the stream we receive to the user's browser using the WebSocket protocol. We'll use an actor to establish the WebSocket connection for each client and connect it to the same broadcasted stream.

We'll proceed in two steps: first, we'll move the logic responsible for retrieving the stream from Twitter to an Akka actor, and then we'll set up a WebSocket connection that makes use of this actor.

2.3.1 Creating an actor

An actor is a lightweight object that's capable of sending and receiving messages. Each actor has a mailbox that keeps messages until they can be dealt with, in the order of reception. Actors can communicate with each other by sending messages. In most cases, messages are sent asynchronously, which means that an actor doesn't wait for a reply to its message, but will instead eventually receive a message with the answer to its question or request. This is all you need to know about actors for now—we'll talk about them more thoroughly in chapter 6.

To see an actor in action, start by creating a new file in the `actors` package, app/actors/TwitterStreamer.scala, with the following content.

Listing 2.8 Setting up a new actor

```
package actors

import akka.actor.{Actor, ActorRef, Props}
import play.api.Logger
import play.api.libs.json.Json

class TwitterStreamer(out: ActorRef) extends Actor {
  def receive = {
    case "subscribe" =>
      Logger.info("Received subscription from a client")
      out ! Json.obj("text" -> "Hello, world!")
  }
}

object TwitterStreamer {
  def props(out: ActorRef) = Props(new TwitterStreamer(out))
}
```

The receive method handles messages sent to this actor.

Handles the case of receiving a "subscribe" message

Sends out a simple Hello World message as a JSON object

Helper method that initializes a new Props object

You want to use your actor to represent a WebSocket connection with a client, managed by Play. You need to be able to receive messages, but also to send them, so you pass the `out` *actor reference* in the constructor of the actor. Play will take care of initializing the actor using the `akka.actor.Props` object, which you provide in the `props` method of the companion object `TwitterStreamer`. It will do so every time a new WebSocket connection is requested by a client.

An actor can send and receive messages of any kind using the `receive` method, which is a so-called *partial function* that uses Scala's pattern matching to figure out

which case statement will deal with the incoming message. In this example, you're only concerned with messages of type `String` that have the value `"subscribe"` (other messages will be ignored).

When you receive a subscription, you first log it on the console, and then (for the moment) send back the JSON object `{ "message": "Hello, world!" }`. The exclamation mark (`!`) is an alias for the `tell` method, which means that you "fire and forget" a message without waiting for a reply or a delivery confirmation.

> **SCALA TIP: PARTIAL FUNCTIONS** In Scala, a partial function `p(x)` is a function that's defined only for some values of `x`. An actor's `receive` method won't be able to handle every type of message, which is why this kind of function is a good fit for this method. Partial functions are often implemented using pattern matching with `case` statements, wherein the value is matched against several `case` definitions (like a `switch` expression in Java).

2.3.2 Setting up the WebSocket connection and interacting with it

To make use of your freshly baked actor, you need to create a WebSocket endpoint on the server side and a view on the client side that will initialize a WebSocket connection.

SERVER-SIDE ENDPOINT

We'll start by rewriting the `tweets` method of the `Application` controller (you may want to keep the existing method as a backup somewhere, because we'll reuse most of its parts later on). You'll notice that we're not creating a Play `Action` this time, because actions only deal with the HTTP protocol, and WebSockets are a different kind of protocol. Play makes initializing WebSockets really easy.

Listing 2.9 Setting up the WebSocket endpoint in app/controllers/Application.scala

```
// ...
import actors.TwitterStreamer

// ...

def tweets = WebSocket.acceptWithActor[String, JsValue] {
  request => out => TwitterStreamer.props(out)
}
```

That's it! You don't need to adjust the route in the routes file either, because you're essentially reusing the existing mapping to the /tweets route.

The `acceptWithActor[In, Out]` method lets you create a WebSocket endpoint using an actor. You specify the type of the input and output data (in this case, you want to send strings from the client and receive JSON objects) and provide the `Props` of the actor, given the `out` actor reference that you're using to communicate with the client.

SIGNATURE OF THE ACCEPTWITHACTOR METHOD The `acceptWithActor` method has a slightly uncommon signature of type `f: RequestHeader => ActorRef => Props`. This is a function that, given a `RequestHeader`, returns another function that, given an `ActorRef`, returns a `Props` object. This construct allows you to access the HTTP request header information for purposes such as performing security checks before establishing the WebSocket connection.

CLIENT-SIDE VIEW

We'll now create a client-side view that will establish the WebSocket connection using JavaScript. Instead of creating a new view template, we'll simply reuse the existing view template, app/views/index.scala.html, as follows.

Listing 2.10 Client-side connection to the WebSocket using JavaScript

```
@(message: String)(implicit request: RequestHeader)

@main(message) {
    <div id="tweets"></div>
    <script type="text/javascript">
        var url = "@routes.Application.tweets().webSocketURL()";
        var tweetSocket = new WebSocket(url);

        tweetSocket.onmessage = function (event) {
            console.log(event);
            var data = JSON.parse(event.data);
            var tweet = document.createElement("p");
            var text = document.createTextNode(data.text);
            tweet.appendChild(text);
            document.getElementById("tweets" ).appendChild(tweet);
        };

        tweetSocket.onopen = function() {
            tweetSocket.send("subscribe");
        };
    </script>
}
```

Initializes the WebSocket connection using a URL generated by Play

The container in which the tweets will be displayed

The handler called when a message is received

The handler called when the connection is opened

Sends a subscription request to the server

You start by opening a WebSocket connection to the `tweets` handler. The URL is obtained using Play's built-in reverse routing and resolves to ws://localhost:9000/tweets. Then you add two handlers: one for handling new messages that you receive, and one for handling the new WebSocket connection once a connection with the server is established.

USING URLS IN VIEWS It's also possible to make use of reverse routing natively in JavaScript. We'll look into that in chapter 10.

When a new connection is established, you immediately send a subscribe message using the `send` method, which is matched in the `receive` method of the `TwitterStreamer` on the server side.

Upon receiving a message on the client side, you append it to the page as a new paragraph tag. To do this, you need to parse the `event.data` field, as it's the string representation of the JSON object. You can then access the `text` field, in which the tweet's text is stored.

There's one change you need to make for your project to compile, which is to pass the `RequestHeader` to the view from the controller. In app/controllers/Application.scala, replace the `index` method with the following code.

Listing 2.11 Declaring the implicit `RequestHeader` to make it available in the view

```
def index = Action { implicit request =>
  Ok(views.html.index("Tweets"))
}
```

You need to take this step because in the index.scala.html view you've declared two parameter lists: a first one taking a message, and a second implicit one that expects a `RequestHeader`. In order for the `RequestHeader` to be available in the implicit scope, you need to prepend it with the `implicit` keyword.

Upon running this page, you should see "Hello, world!" displayed. If you look at the developer console of your browser, you should also see the details of the event that was received.

Scala tip: implicit parameters

Implicit parameters are a language feature of Scala that allows you to omit one or more arguments when calling a method. Implicit parameters are declared in the last parameter list of a function. For example, the index.scala.html template will be compiled to a Scala function that has a signature close to the following:

```
def indexTemplate(message: String)(implicit request: RequestHeader)
```

When the Scala compiler tries to compile this method, it will look for a value of the correct type in the implicit scope. This scope is defined by prepending the `implicit` keyword when declaring anonymous functions, as here with `Action`:

```
def index = Action { implicit request: RequestHeader =>
  // request is now available in the implicit scope
}
```

You don't need to explicitly declare the type of `request`; the Scala compiler is smart enough to do so on its own and to infer the type.

2.3.3 Sending tweets to the WebSocket

Play will create one new `TwitterStreamer` actor for each WebSocket connection, so it makes sense to only connect to Twitter once, and to broadcast our stream to all connections. To this end, we'll set up a special kind of broadcasting enumerator and provide a method to the actor to make use of this broadcast channel.

We first need an initialization mechanism to establish the connection to Twitter. To keep things simple, let's set up a new method in the companion object of the `TwitterStreamer` actor in app/actors/TwitterStreamer.scala.

Listing 2.12 Initializing the Twitter feed

```
object TwitterStreamer {
  def props(out: ActorRef) = Props(new TwitterStreamer(out))

  private var broadcastEnumerator: Option[Enumerator[JsObject]] = None

  def connect(): Unit = {
    credentials.map { case (consumerKey, requestToken) =>
      val (iteratee, enumerator) = Concurrent.joined[Array[Byte]]

      val jsonStream: Enumerator[JsObject] = enumerator &>
        Encoding.decode() &>
        Enumeratee.grouped(JsonIteratees.jsSimpleObject)

      val (be, _) = Concurrent.broadcast(jsonStream)
      broadcastEnumerator = Some(be)

      val url = "https://stream.twitter.com/1.1/statuses/filter.json"
      WS
        .url(url)
        .sign(OAuthCalculator(consumerKey, requestToken))
        .withQueryString("track" -> "reactive")
        .get { response =>
          Logger.info("Status: " + response.status)
          iteratee
        }.map { _ =>
          Logger.info("Twitter stream closed")
        }

    } getOrElse {
      Logger.error("Twitter credentials missing")
    }
  }
}
```

Initializes an empty variable to hold the broadcast enumerator

Sets up the stream transformation pipeline, taking data from the joined enumerator

Sets up a joined set of iteratee and enumerator

Initializes the broadcast enumerator using the transformed stream as a source

Consumes the stream from Twitter with the joined iteratee, which will pass it on to the joined enumerator

With the help of the broadcasting enumerator, the stream is now available to more than just one client.

A WORD ON THE CONNECT METHOD Instead of encapsulating the `connect()` method in the `TwitterStreamer` companion object, it would be better practice to establish the connection in a related actor. The methods exposed in the `TwitterStreamer` connection are publicly available, and misuse of them may seriously impact your ability to correctly display streams. To keep this example short, we'll use the companion object; we'll look at a better way of handling this case in chapter 6.

You can now create a `subscribe` method that lets your actors subscribe their WebSocket clients to the stream. Append it to the `TwitterStreamer` object as follows.

Listing 2.13 Subscribing actors to the Twitter feed

```
object TwitterStreamer {

  // ...

  def subscribe(out: ActorRef): Unit = {
    if (broadcastEnumerator.isEmpty) {
      connect()
    }
    val twitterClient = Iteratee.foreach[JsObject] { t => out ! t }
    broadcastEnumerator.foreach { enumerator =>
      enumerator run twitterClient
    }
}
```

In the `subscribe` method, you first check if you have an initialized `broadcastEnumerator` at your disposal, and if not, establish a connection. Then you create a `twitterClient` iteratee, which sends each JSON object to the browser using the actor reference.

Finally, you can make use of this method in your actor when a client subscribes.

Listing 2.14 `TwitterStreamer` actor subscribing to the Twitter stream

```
class TwitterStreamer(out: ActorRef) extends Actor {
  def receive = {
    case "subscribe" =>
      Logger.info("Received subscription from a client")
      TwitterStreamer.subscribe(out)
  }
}
```

When running the chain, you should now see tweets appearing on the screen, one after another. You can open multiple browsers or tabs to see more client connections being established.

This setup is very resource-friendly given that you only make use of asynchronous and lightweight components that don't block threads: when no data is sent from Twitter, you don't unnecessarily block threads waiting or polling. Instead, each time new data comes in, the parsing and subsequent communication with clients happen asynchronously.

PROPER DISCONNECTION HANDLING One thing we haven't done here is properly handle client disconnections. When you close the browser tab or otherwise disconnect the client, your `twitterClient` iteratee will continue trying to send new messages to the `out` actor reference, but Play will have

closed the WebSocket connection and stopped the actor, which means that messages will be sent to the void. You can observe this behavior by seeing Akka complain in the log about "dead letters" (actors sending messages to no-longer-existing endpoints). To properly handle this situation, you'd need to keep track of subscribers and check if each actor is still in the list of subscribers prior to sending each message. You can find an example of how this is done in the source code for this chapter, available on GitHub.

2.4 Making the application resilient and scaling out

We've built a pretty slick and resource-efficient application to stream tweets from our server to many clients. But to meet the failure-resilience criterion of a reactive web application, we need to do a bit more work: we need a good mechanism to detect and deal with failure, and we need to be able to scale out to respond to higher demand.

2.4.1 Making the client resilient

To be completely resilient, our application would need to be able to deal with a multitude of failure scenarios, ranging from Twitter becoming unavailable to our server crashing. We'll look into a first level of failure handling on the client side here, in order to alleviate the pain inflicted on our users if the stream of tweets were to be interrupted. We'll cover the topic of responsive clients in depth in chapter 8.

If the connection with the server is lost, we should alert the user and attempt to reconnect. This can be achieved by rewriting the `<script>` section of our index.scala.html view, as follows.

Listing 2.15 Resilient version of the JavaScript

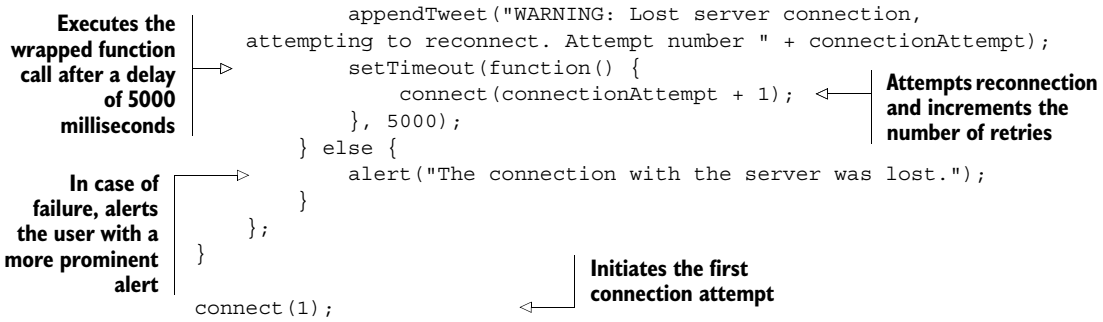
```
function appendTweet(text) {
    var tweet = document.createElement("p");
    var message = document.createTextNode(text);
    tweet.appendChild(message);
    document.getElementById("tweets").appendChild(tweet);
}

function connect(attempt) {
    var connectionAttempt = attempt;
    var url = "@routes.Application.tweets().webSocketURL()";
    var tweetSocket = new WebSocket(url);
    tweetSocket.onmessage = function (event) {
        console.log(event);
        var data = JSON.parse(event.data);
        appendTweet(data.text);
    };
    tweetSocket.onopen = function() {
        connectionAttempt = 1;
        tweetSocket.send("subscribe");
    };
    tweetSocket.onclose = function() {
        if (connectionAttempt <= 3) {
```

Encapsulates the WebSocket connection logic in a reusable function

Attempts up to three connection retries

The onclose handler, called when the WebSocket connection is closed




```
            appendTweet("WARNING: Lost server connection,
    attempting to reconnect. Attempt number " + connectionAttempt);
            setTimeout(function() {
                connect(connectionAttempt + 1);
            }, 5000);
        } else {
            alert("The connection with the server was lost.");
        }
    };
}

connect(1);
```

To avoid repeating the same code twice, you start by moving the logic for displaying a new message into the `appendTweet` method and the logic for establishing a new WebSocket connection into the `connect` method. The latter now takes as its argument the connection attempt count, so you know when to give up trying and can then inform the user about the progress.

The `onclose` handler of the WebSocket API is invoked whenever the connection with the server is lost (or can't be established). This is where you plug in your failure-handling mechanism: when the connection is lost, you inform the user in an unobtrusive manner (by appending a warning message to the existing tweet stream) and then attempt to reconnect after a waiting period of five seconds. If you haven't succeeded after three reconnection attempts, you alert the user in a more direct fashion (in this example, by using a native browser alert). If you succeed at reconnecting, you reset the connection attempt count to 1.

FURTHER COPING MECHANISMS It's not uncommon for a web application to lose connection with the server. One popular mechanism implemented in many clients, such as Gmail, is to wait for increasing amounts of time between two reconnection attempts (first a few seconds, then a minute, and so on), while still informing the user and also giving them a means to reestablish the connection manually by clicking a link or button. This disconnection scenario is quite frequent with mobile devices and laptops, so it's good for an application to have an automated reconnection mechanism in place to optimize the user experience.

SERVER-SIDE FAILURE HANDLING So far we've only handled failures on the client side; we haven't looked into mechanisms to deal with failure handling on the server side. This is not, unfortunately, because there are no failures on the server side, but rather because this topic is too big to cover in this chapter's example application. Don't worry, though. We'll revisit this aspect of the application in detail in chapters 5 and 6.

2.4.2 Scaling out

We now have a pretty slick and resource-efficient application that can stream tweets to many clients. But what if we were to build a fairly popular application, and we wanted

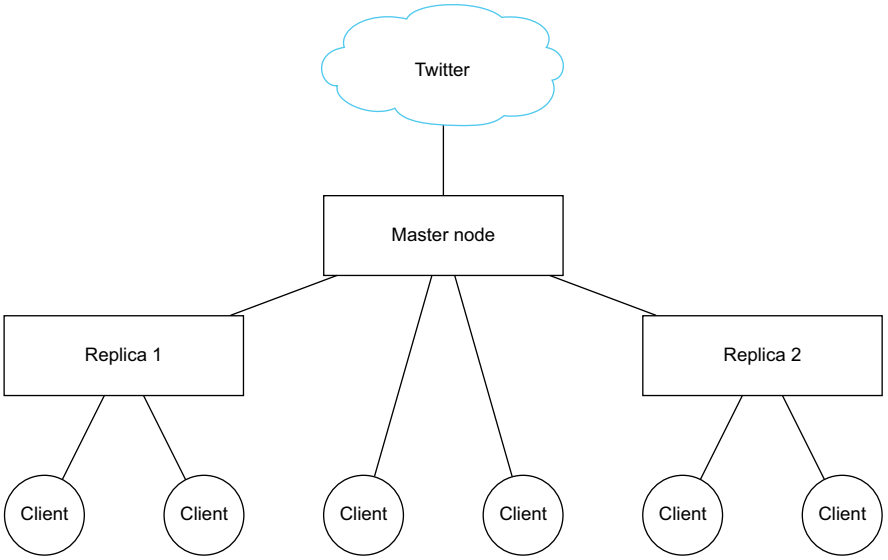


Figure 2.5 Scaling out using replica nodes

to handle more connections than a single node could manage? One mechanism we'll consider is *replica nodes* that could replicate our initial connection, as shown in figure 2.5.

Let's say we wanted to reuse the same connection to Twitter (because Twitter doesn't let us reuse the same credentials many times, and we don't want to create a new user and get new API credentials for each node). We already have a mechanism in place that lets clients view the stream using WebSockets, and we also have a mechanism to broadcast an incoming Twitter stream to WebSocket clients. The only thing we need in order to have working replica nodes that connect to a master node is a means to configure them and get them to connect to our master node instead of Twitter.

To achieve this, we'll set up a new subscription mechanism that allows other nodes to consume data from the initial stream (the one coming from Twitter). We'll set up a new controller action to stream out the content and make the necessary modifications to run the application in *replica* mode.

First, you need to set up a means for the `controller` method to subscribe to the stream.

Listing 2.16 Subscribing other nodes to the broadcast Twitter feed

```
def subscribeNode: Enumerator[JsObject] = {
  if (broadcastEnumerator.isEmpty) {
    connect()
  }
  broadcastEnumerator.getOrElse {
    Enumerator.empty[JsObject]
  }
}
```

This method, like the existing `subscribe` method, first makes sure that the connection to Twitter is initialized, and then simply returns the broadcasting enumeratee. You can now use the enumeratee in a controller method in your `Application` controller.

Listing 2.17 Streaming the replicated Twitter feed in the controller

```
class Application extends Controller {

  // ...

  def replicateFeed = Action { implicit request =>
    Ok.feed(TwitterStreamer.subscribeNode)
  }
}
```

The `feed` method simply feeds the stream provided by the enumerator as an HTTP request.

You now need to provide a new route for this action in conf/routes:

```
GET     /replicatedFeed          controllers.Application.replicateFeed
```

If you now visit http://localhost:9000/replicatedFeed, you'll see the stream of JSON documents displayed with continuous additions to the page.

You now have almost everything in place to set up a replica node. The last thing you need to do is connect to the master node instead of the original Twitter API. You can do this very easily by replacing the URL used in a replica node with the master node's URL. In a production setup, you'd use the application configuration for this. To keep things simple for this example, we'll use a JVM property that can easily be passed along. Add the following logic in the `connect()` method of the `TwitterStreamer` companion object, replacing the existing URL declaration:

```
val maybeMasterNodeUrl = Option(System.getProperty("masterNodeUrl"))
val url = maybeMasterNodeUrl.getOrElse {
  "https://stream.twitter.com/1.1/statuses/filter.json"
}
```

Now, start a new terminal window and start another Activator console (don't close the existing running application):

```
activator -DmasterNodeUrl=http://localhost:9000/replicatedFeed
```

Then run the application on another port:

```
[twitter-stream] $ run 9001
```

Upon visiting http://localhost:9001, you'll see the stream from the other node. You can start more of those nodes on different ports to check if the replication works as expected. Given how the setup works, you can also chain more replicating nodes by passing the URL of a replicating node as `masterNodeUrl` to another node.

FAILURE HANDLING IN A REPLICATED SETUP Although scaling out makes your application capable of handling a higher demand in terms of connections, it also makes failure handling quite a bit more complicated. Given the limitation of only one node being able to connect to Twitter, you're in a situation where there is a single point of failure—if this node were to go down, you'd be in trouble. In a real system, you'd seek to avoid having a single point of failure, and instead have a number of master nodes. You'd also need to devise a mechanism to cope with the loss of a master server.

2.5 Summary

In this chapter, we built a reactive web application using Play and Akka. We used a few key techniques for reactive applications:

- Using asynchronous actions for handling incoming HTTP requests
- Streaming and transforming tweets asynchronously using iteratees, enumeratees, and enumerators
- Establishing WebSocket connections using an Akka actor and connecting it to the stream
- Dealing with failure on the client side
- Scaling out using a simple replication model

Throughout the remainder of the book, we'll explore these topics in more depth. In the next chapter, we'll visit one building block of reactive web applications by looking into functional programming concepts.

WEB DEVELOPMENT/SCALA

Reactive Web Applications

Manuel Bernhardt

Reactive applications build on top of components that communicate asynchronously as they react to user and system events. As a result, they become scalable, responsive, and fault-tolerant. Java and Scala developers can use the Play Framework and the Akka concurrency toolkit to easily implement reactive applications without building everything from scratch.

Reactive Web Applications teaches web developers how to benefit from the reactive application architecture and presents hands-on examples using Play, Akka, Scala, and Reactive Streams. This book starts by laying out the fundamentals required for writing functional and asynchronous applications and quickly introduces Play as a framework to handle the plumbing of your application. The book alternates between chapters that introduce reactive ideas (asynchronous programming with futures and actors, managing distributed state with CQRS) and practical examples that show you how to build these ideas into your applications.

What's Inside

- Reactive application architecture
- Basics of Play and Akka
- Examples in Scala
- Functional and asynchronous programming

For readers comfortable programming with a higher-level language such as Java or C#, and who can read Scala code. No experience with Play or Akka needed.

Manuel Bernhardt is a passionate engineer, author, and speaker. As a consultant, he guides companies through the technological and organizational transformation to distributed computing.

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

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]

"You'll come away with a solid understanding of how reactive web applications are architected, developed, tested, and deployed."

—From the Foreword by James Roper, lead developer of the Play Framework

"Good theory and good practice, with powerful examples."

—Steve Chaloner
Objectify

"How to be reactive in your application development… Eye-opening."

—David Torrubia Íñigo
Fon Wireless, Ltd

"A complete and exhaustive source of best practices for large-scale, real-world reactive platforms."

—Antonio Magnaghi, PhD
OpenMail

ISBN 13: 978-1-63343-009-9
ISBN 10: 1-63343-009-X
54499
9 781633 430099