

Play2: A New Era of Web Application Development

Sadek Drobi • Zenexity

Today's Web has brought an end to the old programming model based entirely on a single-box architecture – a model that assumes no barriers of distribution and desperately seeks a unique, generalized representation of data. With the emergence of software as a service (SaaS) and Web services, data distribution is the norm, rather than the exception. Distributed data offers an opportunity to better reflect enterprise organization, which is commonly divided into services and business processes. It also fosters scalability through data replication and utilizes existing Web services to help us avoid reinventing the wheel. But with this change, new problems have arisen: stream management, scalability, and the management of various data formats.

Play is an open source framework for Web application development created in 2007 and opened to the community in 2009. It targets the Java Virtual Machine (JVM) and focuses on simplicity, developer productivity, and HTTP/Web friendliness. Play implements the classic *model-view-controller* (MVC) architecture with a route file that maps HTTP requests to controllers, which then take request information and produce an HTTP result representation, optionally using a view template. Play then serializes this result representation and returns it as a response to the client.

Here's an example of a route line in the routes file:

```
# Display a client.
GET    /greeting/:name
      controllers.Application.greeting
      (name: String)
```

The corresponding controller method is a function that takes a request and returns an HTTP result ("200 OK" in this case):

```
def greeting(name: String) = Action
  { request =>
    Ok("Hello "+ name)
  }
```

The name parameter passed to the `greeting` function comes from the target resource's URL, as specified in the routes file.

As the Play community grew, the evolution of the Web and associated technologies led the team behind Play to evolve the framework to address emerging Web programming challenges. Here, I examine the main features of Play2 and highlight how functional programming is the key to offering a programming model that addresses today's problems without introducing additional complexity.

Data Manipulation

Application and data distribution lead to heterogeneous data representations and formats. A fair amount of code in a given program is dedicated to producing and consuming these formats – that is, joining and transforming them into a model that's meaningful for a particular resulting functionality or service. A familiar example would be calling several Web services to retrieve data in popular XML and JSON formats, picking some data from each feed, and then combining these into a new JSON, XML, or even HTML document that represents the final result.

A standard approach in the old single-box architecture was to bend the datastore model to look exactly like the service model. This approach was motivated by the possibility of

completely mastering data and storing it all in one place. In the new Web application development model, you must have tools that allow for data manipulation and transformation.

Play2 natively supports the Scala programming language, which leverages a very rich collections API that makes data manipulation – such as grouping, transforming, and joining – much simpler by employing higher-order functions and immutable data structures. Play2 also extensively uses immutability and higher-order functions, making it both a natural fit and a coherent environment for Web-oriented architecture (WOA).

The following example uses the Scala collections API to perform some ordinary operations on a client list to obtain each client's monthly spending:

```
val monthlySpending =
  getOrdersForClient(id)
    .groupBy(order => order.
      month)
    .mapValues(orders =>
      orders.map(o =>
        o.amount).sum)
```

This code shows the use of higher-order functions for data manipulation, a simple, powerful practice that most mainstream languages ignore.

Play2 also chooses to deal directly with JSON, XML, and HTML formats instead of trying to abstract them using some runtime magic, which can often result in weaker and less useful custom structures. Indeed, these data formats have interesting recursive structures that can be handy when used in the right place. Figure 1 transforms the results of two Web calls that are in XML and JSON into HTML using truthful representations of these formats and the `map` higher-order function.

Signal Processing

The single-box architecture assumes that all components are always

```
val eventDescriptionXml = ...
val eventAttendeesJson = getAttendees(eventDescriptionXml \
  "id" text)
val event = new Event(
  id = eventDescriptionXml \ "id" text,
  name = eventDescriptionXml \ "name" text,
  desc = eventDescriptionXml \ "description" text,
  attendees = (eventAttendeesJson \ "list").map( a =>
    (a \ "email").as[String]
  )
)
(a)

@(event: Event)
<h1><a href="/event/@event.id">@event.name</a></h1>
Attendees:
<ul>
  @for( a <- event.attendees) {
    <li>@a</li>
  }
</ul>
(b)
```

Figure 1. XML and JSON formats manipulation. We can transform two Web calls in (a) XML and JSON into (b) an HTML template using Play2.

available, or, if they aren't, that we can do nothing about it except report an internal server error to the user and hope the component will be back shortly.

This assumption doesn't hold in modern Web development. Indeed, several distributed services are involved in most Web services; some are essential to the final Web service result, others are optional, and some are available from multiple servers (via replication). Continuing to send an internal server error to the Web service client when any of these services is absent or in error has become unreasonable. Instead, we must be more explicit in dealing with signals returned from different services. Take, for example, an HTTP service call; it could return a "200 OK" status, but it could also return a "404 Not Found" (signaling that the requested resource doesn't exist), a "400 Bad Request," or any other useful HTTP status that could help our Web service decide how to deal with the situation. Depending on the circumstances, it could retry, deduce

the related error and return it to the user, or even ignore the error.

Mainstream languages use runtime exceptions to deal with situations that might not be handled at the call site but rather in a different functional layer. As the name suggests, using this approach should be exceptional; thus it isn't optimized for ubiquitous use.

To address this issue more adequately, Play2 doesn't try to hide or abstract such signals, but rather deals with them using all the facilities functional programming provides. Among others, the `Either` and `Option` types are very convenient for dealing with multiple signals at a time, given functions such as `map`, `flatMap`, `filter`, and so on.

`Option[A]` represents a value that can either be present or not. Its two subclasses, `Some[A]` and `None` do just this. On the other hand, `Either[A,B]` represents a value that can be either A or B. It also has subclasses, `Left[A]` and `Right[B]`. Both types implement `map` and `flatMap` methods, which let us focus on a

```
val response: Either[Response,Response] =
  WS.url("http://someservice.com/post/123/comments").focusOnOk

val responseOrUndesired: Either[Result,Response] = response.
  left.map {
    case Status(4,0,4) => NotFound
    case Status(4,0,3) => NotAuthorized
    case _ => InternalServerError
  }

val comments: Either[Result,List[Comment]] =
  responseOrUndesired.right.map(r => r.json.as[List[Comment]])

// in the controller

comment.fold(identity, cs => Ok(html.showComments(cs)))
```

Figure 2. Dealing with multiple Web service signals. This example shows how to manage focus on a particular response case while maintaining the other cases using the *Either* type in Play2.

potentially contained interesting value in a context without dropping the other possibility. In the *Either* type, we choose our focus (*Left* or *Right*) and pass a function to then transform it (see Figure 2).

Reactive Composition

When your application is based on distributed services, and answering the client's requests involves calling and composing the results of several such services, then you can't ignore the latency involved and couple your resource consumption to the performance of servers you depend on.

Let's imagine our service's purpose is to retrieve the user's profile. This will involve calling the service that can provide this basic information, another service that provides the user's last orders, and a third service that returns last visited items. This means that for one Web request-response, the application sends three requests, with possible interdependence among them.

The classic model is to give each incoming request an execution thread that it will keep until the server sends the response back to the user. However, when calling services on a different server, our machine won't

be doing any computation, but will rather just be waiting for responses while holding scarce resources such as threads and their dedicated memory. This approach is, of course, a blocking I/O model. If the user's request takes one second to complete (waiting on the three other requests to complete), we'll be blocking one thread and its associated memory unnecessarily for that one second. Consequently, our server will have a hard time scaling up to more than 100 or so users per second, while the CPU is idle most of the time. Modern operating systems, however, offer primitives for doing nonblocking I/O using a notification mechanism that lets the caller know that some data is available for consumption.

The Play2 architecture is based entirely on a reactive model that uses *Promises* to reflect these nonblocking opportunities. A *Promise* is a type that represents a value that will eventually be present. It lets you use the *map*, *flatMap*, and other functions to focus on transforming the value by passing transformation functions, taking care of synchronization, combining different callbacks, and handling other plumbing code. Not only can you

send your requests asynchronously, but you can also collect notifications and decide to react on them. *Promises* provide a way to describe the service result composition in a declarative way while having control over error cases. Note that the API used for composing *Promises* is similar to the one for composing collections and *Option* and *Either* types.

Play2 lets you return a *Promise* of an HTTP response instead of an actual response, and it will write the actual response to the user's socket once it's available, without blocking (see Figure 3). Alternatively, we can use the "for" expression syntax, which the compiler will translate to *map* and *flatMap* calls (see Figure 4).

Play2 provides the opportunity to return a *Promise* of result representing the response that will be eventually sent to the user. This *Promise* could be a result of composing multiple *Promises* of different non-blocking/reactive calls.

Reactive Streams

Reactive programming with *Promises* fits best when the request and the response are each entirely contained in a single chunk. However, Web programming presents many situations in which fitting the entire request or response in one chunk isn't reasonable space-wise (with regard to either memory or disk). This is especially the case for big file transfers and progressive stream computing. We must also consider the increasing interest in the latest HTTP specifications that enable message streaming such as *WebSocket* (bidirectional messaging), *Server-Sent Events* (<http://dev.w3.org/html5/eventsources/>), and the classic HTTP 1.1 chunked transfer mode.

Thus, we need a model that lets us create, consume, and manipulate different data streams reactively but easily. Play2 implements a variation of the *Iteratee* I/O model explained next that natively integrates *Promises*.

Iteratee was initially introduced in the Haskell programming language community as a model for doing composable I/O.

Many materials, including Play2 documentation, provide a detailed insight into the Iteratee programming model. I focus here on showing tasks for which you can use Iteratees and progressive stream programming within Play2.

Stated simply, an `Iteratee` describes how to progressively consume data chunks and compute a result from them. An `Enumerator` is a data source that takes an `Iteratee` and eventually returns an `Iteratee`, possibly in a new state. Clearly, the source controls the execution, and the `Iteratee` will react only to the passed data chunks. On each chunk, an `Iteratee` can return one of three states: "Done" with a computed result, "Cont" with a callback accepting more input, or "Error" with a message and the chunk that caused the error. This provides a powerful primitive for implementing a full API that deals with data streams reactively, including creating, transforming, filtering, combining, and consuming them. To illustrate, let's look at a few scenarios for handling data streams.

Progressive Stream Processing with File Uploads

In an HTML form, a user chooses a 10-Mbyte file and hits the upload button. The user's browser then sends the file as chunks to the Web server. Most Web servers offer no choice but to put the whole file into memory or to save it on disk. In many cases, the content goes directly to a dedicated service, such as Amazon S3 storage, to avoid the complexity of managing and storing files. This simply means that temporarily storing the file on the server is completely useless. Actually, with simple math, we can determine that 100 such users would consume almost 1 Gbyte of memory

```
def userInfo(...):Promise[JsValue] = {
  val profilePromise = WS.url(...).get()
  val attachedEventsPromise = WS.url(...).get()
  val topArticlesPromise = WS.url(...).get()

  profilePromise.flatMap { profile =>
    attachedEventsPromise.flatMap { events =>
      topArticlesPromise.map{ articles =>
        Json.obj(
          "profile" -> profile,
          "events" -> events,
          "articles" -> articles )
      }}}
}
```

Figure 3. Composition of reactive IO calls. Using `promises` in Play2 allows you to compose reactively nonblocking IO calls with `map` and `flatMap`.

```
for {
  profile <- profilePromise
  events <- attachedEventsPromise
  articles <- topArticlesPromise
} yield Json.obj(
  "profile" -> profile,
  "events" -> events,
  "articles" -> articles )

// in the controller

def showInfo(...) = Action { rq =>
  Async {
    actorInfo(...).map(info => Ok(info))
  }
}
```

Figure 4. Returning a Promise that will eventually yield an HTTP response to the user.

or disk space for their files. A better scenario would be to directly forward the chunks into the service without storing anything. This is what Play2 offers through the `Iteratee` I/O in its action API:

```
val bodySize:BodyParser[Int] =
  BodyParser { rh =>
    Iteratee.fold[Array[Byte]]
      (0){ (s,bytes) =>
        s + bytes.length
      }
  }
```

```
def countBodySize =
  Action(bodySize) { rq =>
    Ok("request's body size is "+
      rq.body)
  }
```

A more sophisticated signature of an action in Play2 is the one that takes a `BodyParse`: an `Iteratee` that parses the request body and returns a body or some HTTP result. The HTTP result allows the body parser to return directly to the client (in case the body isn't appropriate, for instance).

One question that comes to mind while looking at this example is, what happens if the user uploads the file faster than the server uploading to the dedicated service? One thing we must avoid at any price is buffering the chunks in the server, thus filling memory. Play2, being completely reactive, will slow down the upload speed depending on the body parser's speed:

```
val slowBodySize:BodyParser[Int] =
  BodyParser { rh =>
    Iteratee.fold1[Array[Byte]]
      (0){ (s,bytes) =>
        Promise.timeout(s + bytes.
          length, 100)
      }
  }
```

Each time this body parser receives a chunk of bytes, it's taking, reactively, 100 milliseconds to increase the computed size. Play2 will adapt the upload rate accordingly.

Server-Sent Events

In modern Web programming, open uni- or bidirectional sockets for messages and notifications between the server and the client are gaining momentum, be it WebSocket, HTTP Server-Sent Events, or Comet (long polling) through HTTP 1.1 chunked transfer mode. This enables interactivity between the client and server and has applications in business and gaming.

To use these streams effectively, however, we need a high-level API for managing different aspects of the streams, such as dispatching to different users or roles, handling security, filtering, and broadcasting. Play2 implements a set of methods on top of Iteratees (acting as reactive stream consumers), Enumerators (which act as the source), and Enumeratees (which act as adapters) and uses the Iteratee I/O model to implement these sockets and protocols.

To create a Stream, we can use one of the available APIs for creating an Enumerator. We use a callback

function each time the consumer is ready for more input:

```
val aStreamOfDates =
  Enumerator.fromCallback
    { () =>
      Promise.timeout(Some(new
        Date), 100 milliseconds)
    }
```

Another way to create a stream is to imperatively push chunks into it:

```
val channel = Enumerator.
  pushee[String] { onStart =
    pushee =>
      pushee.push("Hello")
      pushee.push("World")
  }
```

Enumeratees, on the other hand, are instrumental for manipulating streams. Play2 includes a set of APIs for creating Enumeratees that vary from simple transformation to sophisticated progressive search or chunks grouping:

```
// create an Enumeratee using
  the map method on Enumeratee
val toInt:
  Enumeratee[String,Int] =
  Enumeratee.map[String]{ s =>
    s.toInt }
```

This Enumeratee will transform each chunk from string to Int. Play2 accepts an Enumerator for pushing into an out socket:

```
def dateStream = {
  val stream = aStreamOfDates &>
    Enumeratee.map( d =>
      d.toString ) ><>
    EventSource[String]()

  Ok.feed(stream).
    withHeaders(CONTENT_TYPE
      ->"text/event-stream")
}
```

This code illustrates publishing a stream of date values. EventSource

is an Enumeratee that will wrap each data chunk into the necessary metadata for the Server-Sent Event protocol.

Enumerators à la Carte

One way to model applications publishing different streams is to implement a set of primitive streams and then combine them differently in different contexts or even dynamically on demand. Let's imagine we're publishing monitoring data about a system:

```
object AvailableStreams {

  val cpu: Enumerator[JsValue] =
    ( /* code here */ )
  val memory:
    Enumerator[JsValue] =
    ( /* code here */ )
  val threads:
    Enumerator[JsValue] =
    ( /* code here */ )
  val heap: Enumerator[JsValue] =
    ( /* code here */ )

}
```

We can now combine these for a certain screen/user using the interleaved operator '>-':

```
val physicalMachine =
  AvailableStreams.cpu >-
  AvailableStreams.memory
val jvm = AvailableStreams.
  threads >- AvailableStreams.
  heap
```

We can even have a system of user preferences for widgets:

```
def usersWidgets
  Composition(prefs:
  Preferences) = {
  // do the composition
  dynamically
}
```

Interleaving Enumerators is just one of many different ways to

combine these reactive streams of data. These operations are instrumental to handling easily concurrent reactive streams and are only possible due to the powerful Iteratee model.

Extending Streams from Other Streams

Another way to model streams is to start with a general stream of different events and filter or adapt it depending on the context. Let's imagine a stream of a system of operations; the event type can be one of two:

```
trait Event
case class Operation(amount:
  Int) extends Event
case class
  TechnicalLog(message:
  String) extends Event
```

Given a stream of events

```
val systemStream:
  Enumerator[Event] =
  ( /* code here */ ),
```

we can make a stream for public users by filtering out technical logs:

```
val publicStream = systemStream
  &> Enumeratee.collect { case
  e:Operation(_) => e }
```


However, we can also let the user choose his or her level of interest as a range, and produce an appropriate stream:

```
def forRange(min: Int,
  max: Int) = publicStream &>
  Enumeratee.filter( op =>
  op.amount > min && op.amount
  < max)
```

In addition to collect and filter Enumeratees, Play2 implements other operations such as take, takeWhile, drop, dropWhile, grouped, search (progressive search), scanLeft, and zip – and, of course, all these operate on a fully reactive stream. The Iteratee model lets users reason about reactive concurrent streams as if they were Lists, handling all the synchronization and notifications propagation.

This overview offers a few insights into some Play2 features and illustrates its strategy for putting the Web's power in developers' hands, rather than abstracting and concealing its existence. To gain a broader perspective about the Play2 Web framework, you can download it at www.playframework.org – where you'll also find documentation – and experiment with samples. You can also get help on Play's mailing list at <http://groups.google.com/group/play-framework>. □

Sadek Drobi is CTO of Zenexity and a software engineer specializing in the design and implementation of enterprise applications with a particular focus on bridging the gap between the problem domain and the solution domain. As a core Play developer, he works on the design and implementation of the Play framework. You can follow him on Twitter at [@sadache](https://twitter.com/sadache) or via his blog at <http://sadache.tumblr.com>.

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.