Streaming HTTP responses

Standard responses and Content-Length header

Since HTTP 1.1, to keep a single connection open to serve several HTTP requests and responses, the server must send the appropriate Content-Length HTTP header along with the response.

By default, you are not specifying a **Content-Length header** when you send back a simple result, such as:

```
def index = Action {
  Ok("Hello World")
}
```

Of course, because the content you are sending is well-known, Play is able to compute the content size for you and to generate the appropriate header.

Note: that for text-based content it is not as simple as it looks, since the **Content-Length** header must be computed according the character encoding used to translate characters to bytes.

Actually, we previously saw that the response body is specified using a **play.api.http.HttpEntity**:

```
def action = Action {
  Result(
    header = ResponseHeader(200, Map.empty),
    body = HttpEntity.Strict(ByteString("Hello world"), Some("text/plain"))
  )
}
```

This means that to compute the **Content-Length header** properly, Play must consume the whole content and load its content into memory.

Sending large amounts of data

If it's not a problem to load the whole content into memory, what about large data sets? Let's say we want to return a large file to the web client.

Let's first see how to create an Source[ByteString, _] for the file content:

```
val file = new java.io.File("/tmp/fileToServe.pdf")
val path: java.nio.file.Path = file.toPath
val source: Source[ByteString, _] = FileIO.fromPath(path)
```

The problem for large files is that we don't want to load them completely into memory. To avoid that, we just have to specify the **Content-Length header** ourselves. In the example below we are

creating a value **contentLength** for it and we are then passing it for the **HttpEntity.Streamed** function.

```
def streamedWithContentLength = Action {

  val file = new java.io.File("/tmp/fileToServe.pdf")
  val path: java.nio.file.Path = file.toPath
  val source: Source[ByteString, _] = FileIO.fromPath(path)

  val contentLength = Some(file.length())

  Result(
    header = ResponseHeader(200, Map.empty),
    body = HttpEntity.Streamed(source, contentLength,
Some("application/pdf"))
  )
}
```

By doing it this way, Play will consume the body in a lazy way, copying each chunk of data to the HTTP response as soon as it is available.

Serving files

Play provides an easy-to-use helper for a common task like serving a local file:

```
def file = Action {
  Ok.sendFile(new java.io.File("/tmp/fileToServe.pdf"))
}
```

This helper will also compute the **Content-Type header** from the file name, and add the **Content-Disposition header** to specify how the web browser should handle this response. The default is to show this file inline by adding the header **Content-Disposition: inline; filename=fileToServe.pdf** to the HTTP response.

You can also provide your own file name:

```
def fileWithName = Action {
  Ok.sendFile(
    content = new java.io.File("/tmp/fileToServe.pdf"),
    fileName = _ => "termsOfService.pdf"
  )
}
```

If you want to serve this file attachment:

```
def fileAttachment = Action {
  Ok.sendFile(
    content = new java.io.File("/tmp/fileToServe.pdf"),
    inline = false
  )
}
```

Now you don't have to specify a file name since the web browser will not try to download it, but will just display the file content in the web browser window. This is useful for content types supported natively by the web browser, such as text, HTML or images.

Chunked responses

For now, it works well with streaming file content since we are able to compute the content length before streaming it. But what about dynamically computed content, with no content size available?

For this kind of response we have to use Chunked transfer encoding.

Chunked transfer encoding is a data transfer mechanism in version 1.1 of the **Hypertext Transfer Protocol (HTTP)** in which a web server serves content in a series of chunks. It uses the **Transfer-Encoding HTTP** response header instead of the **Content-Length header**, which the protocol would otherwise require. Because the **Content-Length header** is not used, the server does not need to know the length of the content before it starts transmitting a response to the client (usually a web browser). Web servers can begin transmitting responses with dynamically-generated content before knowing the total size of that content.

The size of each chunk is sent right before the chunk itself, so that a client can tell when it has finished receiving data for that chunk. Data transfer is terminated by a final chunk of length zero.

The **advantage is that we can serve the data live**, meaning that we send chunks of data as soon as they are available. The drawback is that since the web browser doesn't know the content size, it is not able to display a proper download progress bar.

Let's say that we have a service somewhere that provides a dynamic **InputStream** computing some data. First we have to create a Source for this stream:

```
val data = getDataStream
val dataContent: Source[ByteString, _] =
StreamConverters.fromInputStream(() => data)
```

We can now stream these data using a **Ok.chunked**:

```
def chunked = Action {
  val data = getDataStream
  val dataContent: Source[ByteString, _] =
StreamConverters.fromInputStream(() => data)
  Ok.chunked(dataContent)
}
```

Of course, we can use any Source to specify the chunked data:

```
def chunkedFromSource = Action {
  val source = Source.apply(List("kiki", "foo", "bar"))
  Ok.chunked(source)
}
```

We can inspect the HTTP response sent by the server:

HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

4
kiki
3
foo
3
bar
0

Comet Sockets

Using chunked responses with Comet Sockets

A common use of **chunked responses** is to create a **Comet socket**.

A **Comet socket** is a chunked **text/html response** containing only **<script>** elements. For each chunk, we write a **<script>** tag containing **JavaScript** that is immediately executed by the web browser. This way we can send events live to the web browser from the server: for each message, wrap it into a **<script>** tag that calls a **JavaScript callback function**, and write it to the chunked response.

Because **Ok.chunked** leverages **Akka Streams** to take a **Flow[ByteString]**, we can send a **Flow** of elements and transform it so that each element is escaped and wrapped in the **Javascript** method. The **Comet helper** automates **Comet sockets**, pushing an initial blank buffer data for browser compatibility, and supporting both **String** and **JSON** messages.

Using Comet with String Flow

In order to push the String messages through flow, follow the example:

```
def cometString = Action {
  implicit val m = materializer
  def stringSource: Source[String, _] = Source(List("kiki", "foo", "bar"))
  Ok.chunked(stringSource via Comet.string("parent.cometMessage")).as(Con-
tentTypes.HTML)
}
```

Although in order for this to work you will need the following imports and to have injected the materializer:

```
import akka.stream.Materializer
import akka.stream.scaladsl.Source
import play.api.http.ContentTypes
import play.api.libs.Comet
import play.api.mvc._

class Application @Inject()(materializer: Materializer) extends Controller
```

Using Comet with JSON Flow

```
def cometJson = Action {
  implicit val m = materializer
  def jsonSource: Source[JsValue, _] = Source(List(JsString("jsonString")))
  Ok.chunked(jsonSource via
Comet.json("parent.cometMessage")).as(ContentTypes.HTML)
}
```

Using Comet with iframe

The HTML file should have a helper, an example would look like:

```
<script type="text/javascript">
        var cometMessage = function(event) {
          console.log('Received event: ' + event)
        }
</script>

<iframe src="/comet"></iframe>
```

An example Comet helper could be found on the project:

https://github.com/playframework/play-scala-streaming-example

The life cycle of an application

Applications life cycle has two states: **stopped** and **running**. These are the times when application state changes. At different times we may need to perform some operation right or after a state change is about to occur.

Play uses Netty server which has three states:

- **StaticApplication** – which is used for the production mode, where the code changes do not affect the running application
- **ReloadableApplication** – used for development mode where the continues compilation is enabled and developers are able to see the impact of their changes after their changes are saved and the application is running
- **TestApplication** – is used for testing as the name implies where a fake application is started when the tests are getting run

Imagine three scenarios:

- Prior to starting the application, we need to ensure that the /opt/dev/ appName directory exists and is accessible by the application. A method in our application called ResourceHandler.initialize does this task.
- Create the required schema on startup using the DBHandler.createSchema method. This method does not drop the schema if it already exists. This ensures that the application's data is not lost on restarting the application and the schema is generated only when the application is first started.
- Create e-mail application logs when the application is stopped using the Mailer.sendLogs method. This method sends the application logs as an attachment in an e-mail to the emailId set in a configuration file as adminEmail. This is used to track the cause for the application's shutdown.

Play has the functionality for us to hook into the application life cycle and complete such tasks. This can be achieved through the use of **GlobalSettings** trait which has methods for us to achieve it, the methods can be overridden by the Global object if required.

To cater the three scenarios previously mention what we need to do is to create a Global object.

```
object Global extends GlobalSettings {
  override def beforeStart(app: Application): Unit = {
    ResourceHandler.initialize
  }
  override def onStart(app: Application):Unit={
    DBHandler.createSchema
  }
  override def onStop(app: Application): Unit = {
    Mailer.sendLogs
  }
}
```

Dependency Injection

Dependency injection is a design pattern that helps you to separate your component behaviour from dependency resolution. DI is all about decoupling client and service code, where the client might be another service. Services expose information what dependencies they need and instead of creating dependent service implementations inside the service itself, references to dependent services are “injected”. By doing this the code is easier to understand, test and reuse.

Play has the support for both **compile time** and **runtime** dependency injection.

Runtime dependency injection is when dependency graph is created and validated at runtime. In case a dependency is not found for a component, an error will only be shown when you run your application.

DI achieves several goals:

1. Allows you to bind different implementations for the same component. Which is useful for testing, where you’re able to instantiate components using mock dependencies or by injecting alternate implementations
2. Allows you to avoid global static state, while they allow you to achieve the first goal they require careful set up. Plays now deprecated static APIs require a running application which makes testing less flexible. Also having more than one instance available at a time makes it possible to run tests in parallel.

Declaring runtime DI dependencies

If you have a controller which requires another component as dependency, then you can use the **@Inject** annotation. It can be used on fields or on constructors, although the recommended practice is to use it on constructor.

```
import javax.inject._
import play.api.libs.ws._

class MyComponent @Inject() (ws: WSClient) {
  // ...
}
```

Note: the annotation has to come after the class name but before the constructor parameters, and it must have parentheses.

Component lifecycle

Di system manages the lifecycle of injected components, creating them as needed and injecting them into other components. It works like this:

- Every time a component is needed, a new instance of it is created. If it's used multiple times, multiple instances are created. In case you want a single instance of a component you need to mark it as a singleton.
- Instances are created lazily when they are needed. If a component is never used by another component, then it won't be created at all.
- Instances are not automatically cleaned up, beyond normal garbage collection. Components will be garbage collected when they're no longer referenced, but the framework won't do anything special to shut down the component, like calling a close method. However, Play provides a special type of component, called the **ApplicationLifecycle** which lets you register components to shut down when the application stops.

Singletons

If you had a component that holds some state like cache or a connection to some resource, creating a component like this is expensive. Then you would want to have only one instance of that component and to achieve it you would need to use the **@Singleton** annotation.

```
import javax.inject._

@Singleton
class CurrentSharePrice {
  private var price = 0

  def set(p: Int) = price = p
  def get = price
}
```

Stopping/cleaning up

In case some components like thread pools need to be cleaned up when play shuts down, play has previously mentioned **ApplicationLifecycle** component that can be used to create hooks that will serve for that purpose.

```
import scala.concurrent.Future
import javax.inject._
import play.api.inject.ApplicationLifecycle

@Singleton
class MessageQueueConnection @Inject() (lifecycle: ApplicationLifecycle) {
  val connection = connectToMessageQueue()
  lifecycle.addStopHook { () =>
    Future.successful(connection.stop())
  }

  //...
}
```

Note: It's important that the components you have the hooks on are singletons, otherwise it can become a memory leak since a new stop hook will be created each time that component was created.