AKKA STREAMS

Amadou Sarjo Jallow
Vrije Universiteit Brussels
Software Architecture

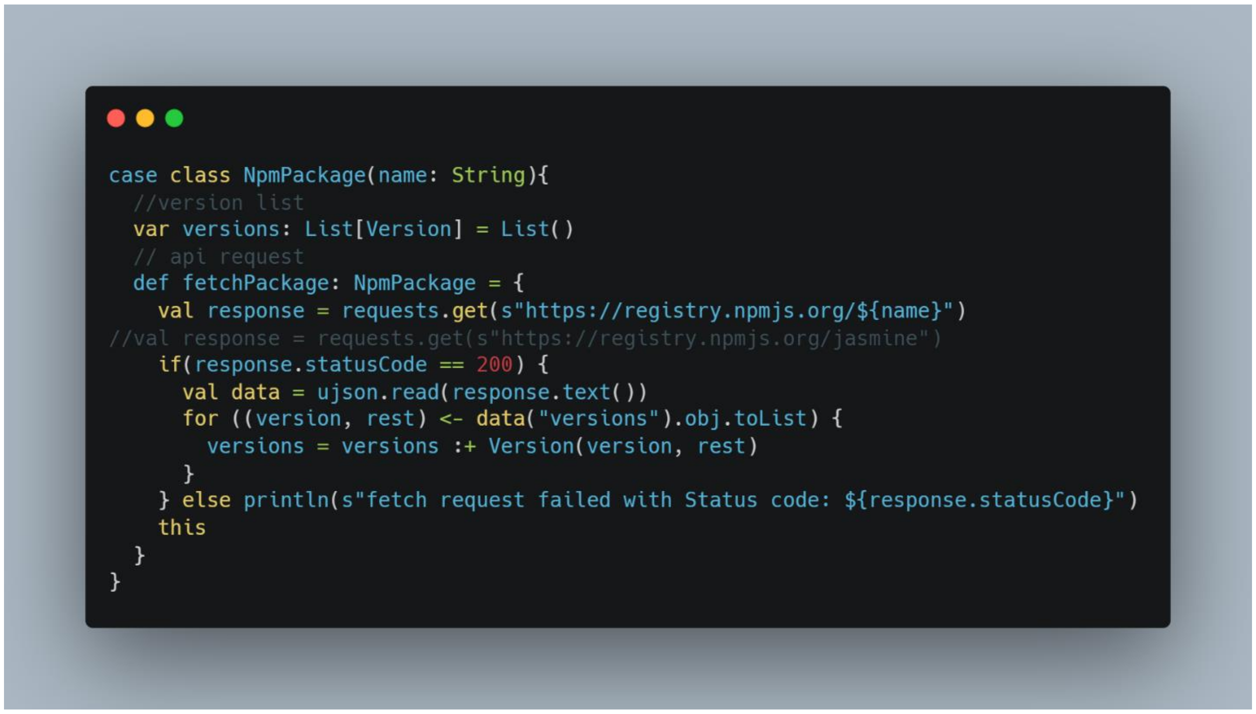
CONTENT

GENERAL FLOW

Running the application calls the NpmStreaming Object which is the main class. Inside this class, a stream flow starts which reads a text file and performs several transformations on the content of the file including instantiating the strings in the file to package objects which then are used to query the Npm package registry to retrieve information corresponding to those packages. Each package is mapped to versions which are then extracted. A series of transformations are then performed on the corresponding versions, and we end up outputting the dev and runtime dependency counts of each version and the list of keywords, if any , to the console.

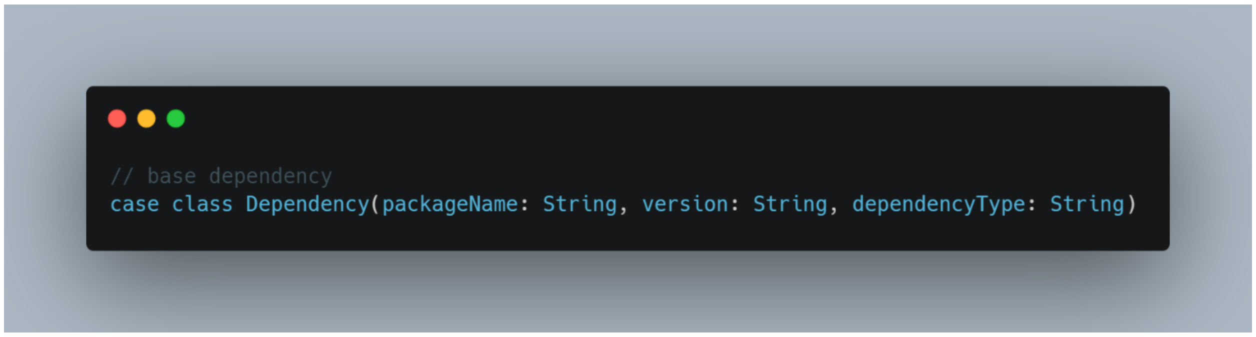
COMPONENTS

The NpmPackage class file contains the components used by the flows to transform incoming data and output the required results.



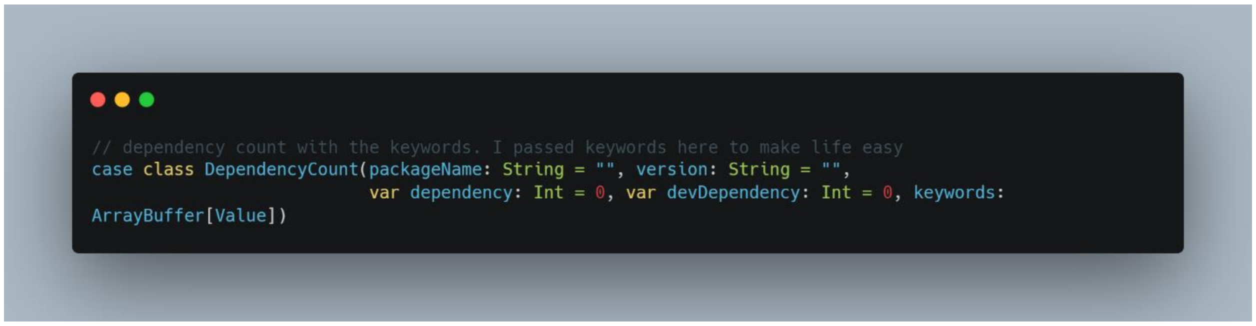
```scala
case class NpmPackage(name: String){
  //version list
  var versions: List[Version] = List()
  // api request
  def fetchPackage: NpmPackage = {
    val response = requests.get(s"https://registry.npmjs.org/${name}")
//val response = requests.get(s"https://registry.npmjs.org/jasmine")
    if(response.statusCode == 200) {
      val data = ujson.read(response.text())
      for ((version, rest) <- data("versions").obj.toList) {
        versions = versions :+ Version(version, rest)
      }
    } else println(s"fetch request failed with Status code: ${response.statusCode}")
    this
  }
}
```

An NpmPackage object is identified by its unique name and defines a fetchPackage method which maps each package to its versions when called. Each instance of the NpmPackage maps all its versions to the versions list variable.



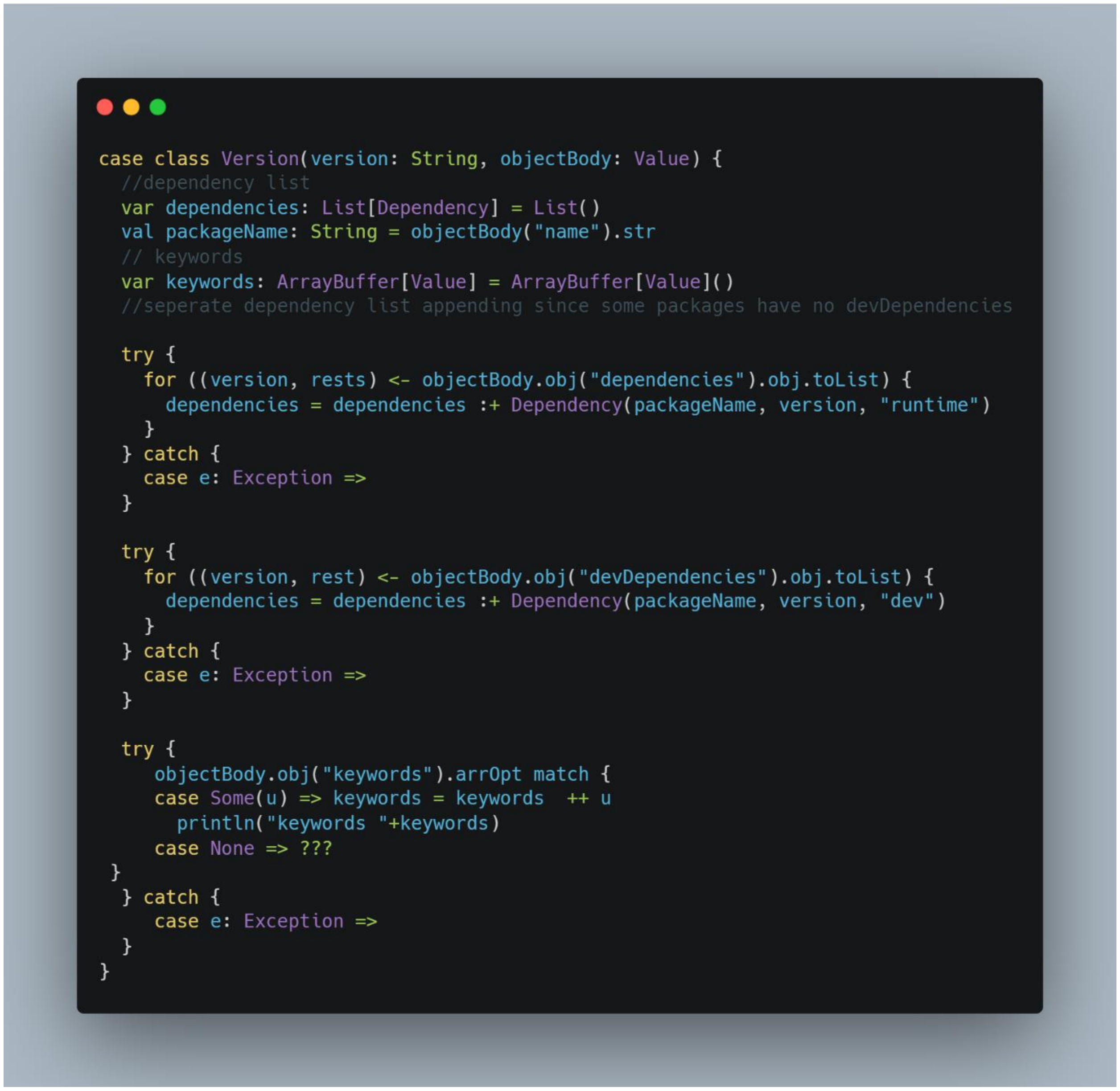
```scala
// base dependency
case class Dependency(packageName: String, version: String, dependencyType: String)
```

A Dependency object contains the package name, its version and the dependencyType which can be either “dev” or “runtime”. Each version of a package instantiates its dependencies based on this definition.



```scala
// dependency count with the keywords. I passed keywords here to make life easy
case class DependencyCount(packageName: String = "", version: String = "",
                           var dependency: Int = 0, var devDependency: Int = 0, keywords:
ArrayBuffer[Value])
```

A DependencyCount object contains for each version of a package, the “dev” and “runtime” dependency counts of the version, the name of the package and list of keywords, if any. I decided to pass the keywords here to make life easier.



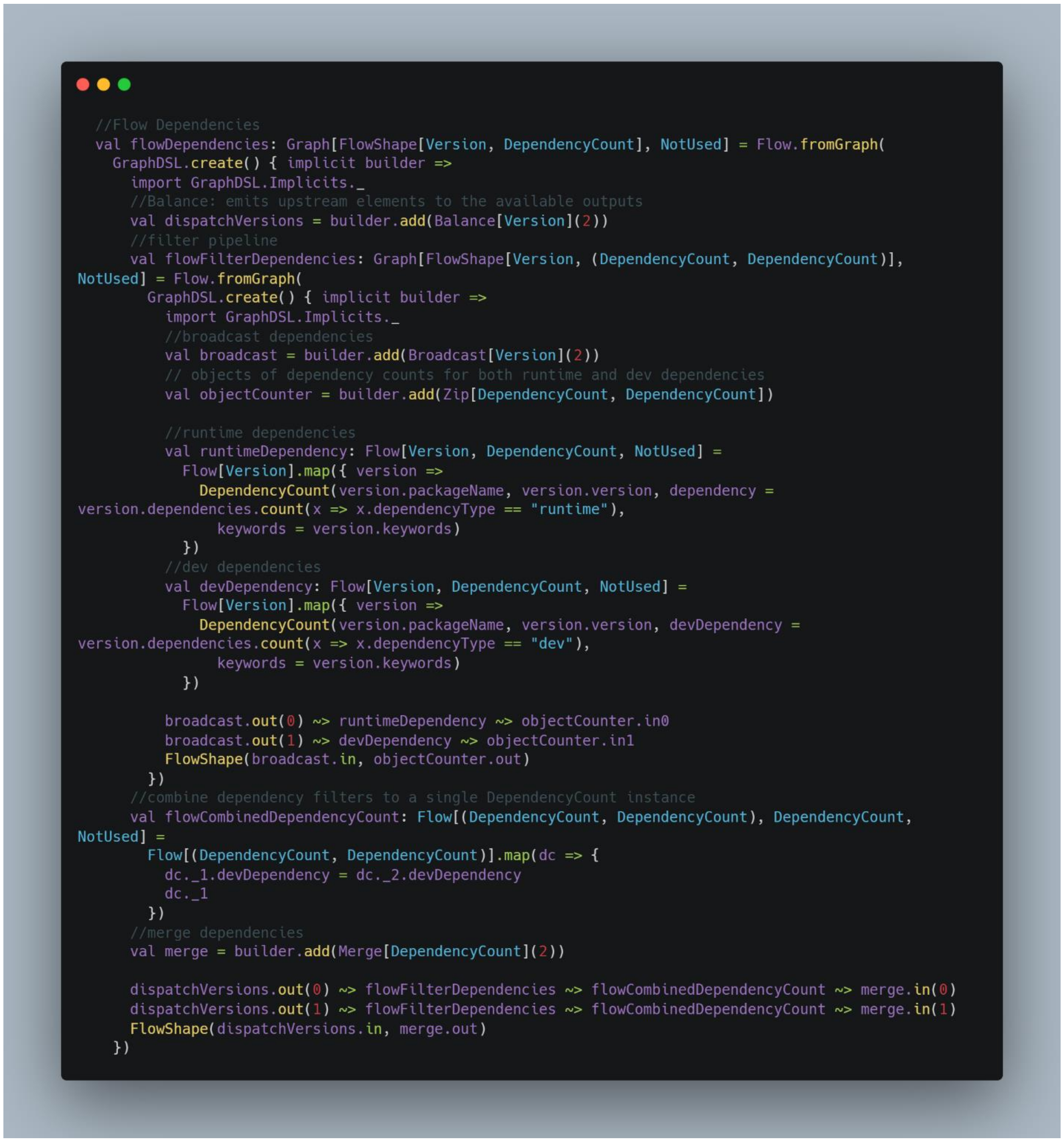
```scala
case class Version(version: String, objectBody: Value) {
  //dependency list
  var dependencies: List[Dependency] = List()
  val packageName: String = objectBody("name").str
  // keywords
  var keywords: ArrayBuffer[Value] = ArrayBuffer[Value]()
  //seperate dependency list appending since some packages have no devDependencies

  try {
    for ((version, rests) <- objectBody.obj("dependencies").obj.toList) {
      dependencies = dependencies :+ Dependency(packageName, version, "runtime")
    }
  } catch {
    case e: Exception =>
  }

  try {
    for ((version, rest) <- objectBody.obj("devDependencies").obj.toList) {
      dependencies = dependencies :+ Dependency(packageName, version, "dev")
    }
  } catch {
    case e: Exception =>
  }

  try {
     objectBody.obj("keywords").arrOpt match {
     case Some(u) => keywords = keywords  ++ u
       println("keywords "+keywords)
     case None => ???
 }
  } catch {
     case e: Exception =>
  }
}
```

Each Version instance defines variables, dependencies and keywords which maps all its dependencies and keywords, if any, based on their definitions. So basically, dependencies and keywords are defined each time a version is instantiated.



```
  //Flow Dependencies
  val flowDependencies: Graph[FlowShape[Version, DependencyCount], NotUsed] = Flow.fromGraph(
    GraphDSL.create() { implicit builder =>
      import GraphDSL.Implicits._
      //Balance: emits upstream elements to the available outputs
      val dispatchVersions = builder.add(Balance[Version](2))
      //filter pipeline
      val flowFilterDependencies: Graph[FlowShape[Version, (DependencyCount, DependencyCount)],
NotUsed] = Flow.fromGraph(
        GraphDSL.create() { implicit builder =>
          import GraphDSL.Implicits._
          //broadcast dependencies
          val broadcast = builder.add(Broadcast[Version](2))
          // objects of dependency counts for both runtime and dev dependencies
          val objectCounter = builder.add(Zip[DependencyCount, DependencyCount])

          //runtime dependencies
          val runtimeDependency: Flow[Version, DependencyCount, NotUsed] =
            Flow[Version].map({ version =>
              DependencyCount(version.packageName, version.version, dependency =
version.dependencies.count(x => x.dependencyType == "runtime"),
                keywords = version.keywords)
            })
          //dev dependencies
          val devDependency: Flow[Version, DependencyCount, NotUsed] =
            Flow[Version].map({ version =>
              DependencyCount(version.packageName, version.version, devDependency =
version.dependencies.count(x => x.dependencyType == "dev"),
                keywords = version.keywords)
            })

          broadcast.out(0) ~> runtimeDependency ~> objectCounter.in0
          broadcast.out(1) ~> devDependency ~> objectCounter.in1
          FlowShape(broadcast.in, objectCounter.out)
        })
      //combine dependency filters to a single DependencyCount instance
      val flowCombinedDependencyCount: Flow[(DependencyCount, DependencyCount), DependencyCount,
NotUsed] =
        Flow[(DependencyCount, DependencyCount)].map(dc => {
          dc._1.devDependency = dc._2.devDependency
          dc._1
        })
      //merge dependencies
      val merge = builder.add(Merge[DependencyCount](2))

      dispatchVersions.out(0) ~> flowFilterDependencies ~> flowCombinedDependencyCount ~> merge.in(0)
      dispatchVersions.out(1) ~> flowFilterDependencies ~> flowCombinedDependencyCount ~> merge.in(1)
      FlowShape(dispatchVersions.in, merge.out)
    })
```

The flowDependencies flow takes in a Version object and returns its DependencyCount which contains the “dev” and “runtime” dependencies, and the list of keywords associated with that version.

The flowFilterDependencies is the actual flow which returns a tuple containing the “dev” and “runtime” dependency counts of a version. This tuple is later combined into one tuple and passed downstream.

Finally, the sink prints the result to the console.