Software Architectures

Assignment 1: Akka Streams

Camilo Velazquez, Ahmed Zerouali
Emails: {cavelazq, azeroual}@vub.be
Office: {10F725}

Assignment

You will have to implement and report on your original solution to a problem using Akka Streams. You will find the input file needed for this assignment in *Assignments > Assignment 1.*

Deadline: January 9th 2022 by 23:59. The deadline is fixed and will not be extended for any reason.

Deliverables A report and source code. The report (in English) explains your solution to the problem and the main components used in your implementation. When possible, use simple diagrams to illustrate.

The report should be handed in as a single PDF file.
The file should follow the naming schema `FirstName-LastName-SA1.pdf`, for example: `Camilo-Velazquez-SA1.pdf`.

The source code is your implementation of the project zipped or exported from IntelliJ. Do not include files or folders resulting from the compilation process of your project, e.g., target/ or *.class. Submit the *report* and *source code* as a single zip file on the Software Architectures course page in Canvas, by clicking on *Assignments > Assignment 1.*

Plagiarism Note that copying – whether from previous years, from other students, or from the internet – will not be tolerated, and will be reported to the dean as plagiarism, who will decide appropriate disciplinary action (e.g., expulsion or exclusion from the examination session). If you use any other resources besides those provided in the lectures and in this document, remember to cite them in your report.

Grading Your solution will be graded and can become subject of an additional defense upon your or the assistants' request.

Problem Description

In this assignment you will implement a Pipe and Filter architecture for a system that processes a streamed list of packages from the NPM software registry[1], i.e., given a list of NPM packages, we want to know how many dependencies each package version has.

You are provided with a zipped template project (*template_project.zip*) with all dependencies needed for this project already included. You are also provided with the list of packages to process (*src/main/resources/packages.txt.gz*). Each of the lines in the packages' file contains the name of a package that should be streamed and analysed by your approach. You should create the necessary classes in order to map each row in the file to an object in Scala.

Figure 1 provides a graphical representation of the application that you need to implement. Once the rows have been converted to objects of a user-defined class, you should feed them to a Source. In a follow-up step, the Source streaming should be buffered with a maximum capacity of 10 packages. A backpressure strategy should be adopted in case that the mentioned limit is reached. Next, using the package name you should request the API endpoint of the NPM registry to get information about package versions and their dependencies (i.e., runtime dependencies are listed as *dependencies*, where development ones are listed as *devDependencies*). For example, to get information about the package *jasmine*, you can use `https://registry.npmjs.org/jasmine`.

To keep the API requests controlled and not stress the NPM registry, you should stream one package each 3 seconds, before querying its information from the API.

The response of the request will correspond to all information related to a package, e.g., name, description, versions, etc. From this response, you should be able to extract information related to the versions and their (runtime and development) dependencies.

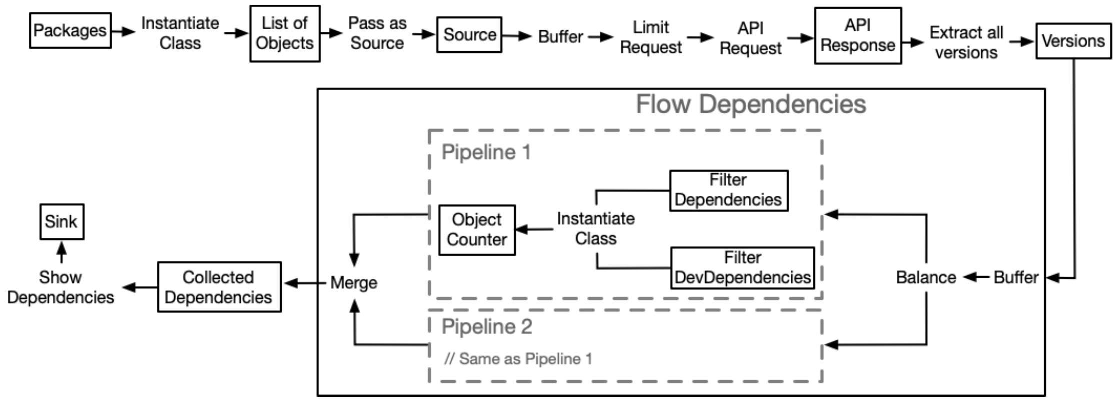


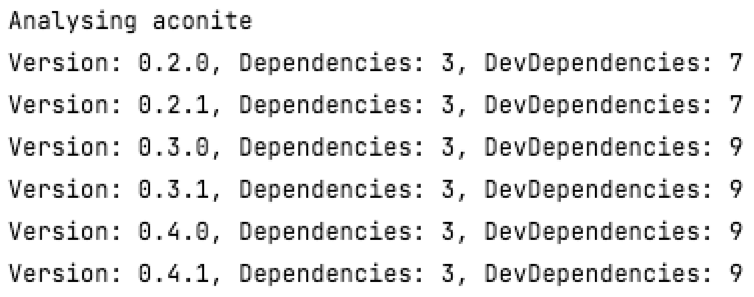
Figure 1: Application diagram.

A custom Flow shape (i.e., the Flow Dependencies box in Figure 1) should be created in order to process the number of dependencies in a parallel way. A buffer of 12 versions as maximum and following the backpressure strategy should control all versions of a

[1]https://www.npmjs.com/

package. The flow-shape should have a balanced approach to send groups of versions of a package to two pipelines. The pipelines should work in the same way by grouping the dependencies by type and counting them. You should create the necessary classes and objects to be able to store the number of dependencies per version. The two pipelines should return the identifier of a library version (e.g., 1.2.3) and the number of dependencies per type. Both pipelines will be merged in a single outlet as the output of the custom shape.

Finally, the collected dependencies per version should be displayed as standard output to the terminal. The screenshot in Figure 2 provides a real example of the expected output for the described process.



```
Analysing aconite
Version: 0.2.0, Dependencies: 3, DevDependencies: 7
Version: 0.2.1, Dependencies: 3, DevDependencies: 7
Version: 0.3.0, Dependencies: 3, DevDependencies: 9
Version: 0.3.1, Dependencies: 3, DevDependencies: 9
Version: 0.4.0, Dependencies: 3, DevDependencies: 9
Version: 0.4.1, Dependencies: 3, DevDependencies: 9
```

Figure 2: Screenshot of the expected output.

A line consists of the name of a package being analysed. The remaining lines contain the version identifier and the number of dependencies per type for the package.

The above requirements should be implemented using Akka Streams. You are also required to make use of the `GraphDSL`. **Motivate your design decisions** in the report as per the problem description and illustrate your approach using concepts from Akka Streams.

More information on Akka Streams and external libraries

- `https://doc.akka.io/docs/akka/2.5.32/stream/index.html`
- `https://doc.akka.io/docs/akka/2.5.32/stream/stream-graphs.html`
- `https://github.com/com-lihaoyi/requests-scala`
- `https://www.lihaoyi.com/post/uJsonfastflexibleandintuitiveJSONforScala.html`