VRIJE
UNIVERSITEIT
BRUSSEL

AKKA STREAMS

Assignment 1 - Software Architectures

Lennert Bontinck

Second session, 2020-2021

Student number: 568702

Computer Science: AI

Contents

General remarks

1.1 Notes on the assignment

None of the assignments for the Software Architectures course were submitted in the first examination period due to personal reasons. Because of this, all of the code written for this assignment is written specifically for the second examination period. One small *bug* in the code was resolved after email communication with the teaching assistants. This *bug* and its solution will be explained later. All other code was written by using the course material and the online Akka documentation[1].

1.2 Important files

All code written is available on the GitHub repository for this assignment (Bontinck, 2021). Rights to this private GitHub repository can be granted upon request. A copy of this GitHub repository is accompanied by this report. An overview of important files is given below:

- `README.md`
  - General information of the GitHub repository as well as the technical details about the used environment, running the project and validated output.
- `assignment.pdf`, `Lennert-Bontinck-SA1.pdf` and `report/`
  - The assignment PDF, this report and the source files of the report.
- `code/Lennert-Bontinck-SA1/`
  - The folder containing the code of the assignment solution.
  - Instructions on how to run the code are available in the `README.md` file.
  - `./src/main/resources/`: sub-folder containing the extracted input file and a shortened alternative. The results files are available under the sub-folder `./result/`.
  - `./src/main/scala/Lennert_Bontinck_SA1/`: sub-folder containing the main Scala files. Multiple files are made for separate objects and classes. All code is well documented and discussed later in this report.

[1] https://akka.io/docs/

Project defence

In what follows the implementation details of the assignment are discussed. Multiple figures were made to illustrate the working of different parts of the code. A legend showing the meaning of the parts of the figure is given in figure 2.5. For more technical details the reader is invited to read the comments in the code.

2.1 Main application structure

There are two main runnable applications: `Main` and `MainDisplay`. Both extend the App object and run a `RunnableGraph`. They differ in the way they handle output by having differing sinks. `Main` saves the output to text files in the output folder whilst `MainDisplay` prints the output to the terminal. The former is the assignment's requirement and the latter is an easier to debug alternative. A simplified representation of the Main application is given in figure 2.1. The different components will be explained in greater detail in what follows.

The most important remark to make for the main graph is the use of `alsoToMat` to reach two distinct sinks. Alternatively, a broadcast mechanism could have been used. These multiple sinks are required for the second term extension of the assignment, each saving a differing file. The discussed *bug* for which the teaching assistants help was received was simply the use of `to` instead of `toMat` for flowing to the sink, causing premature termination of the graph.

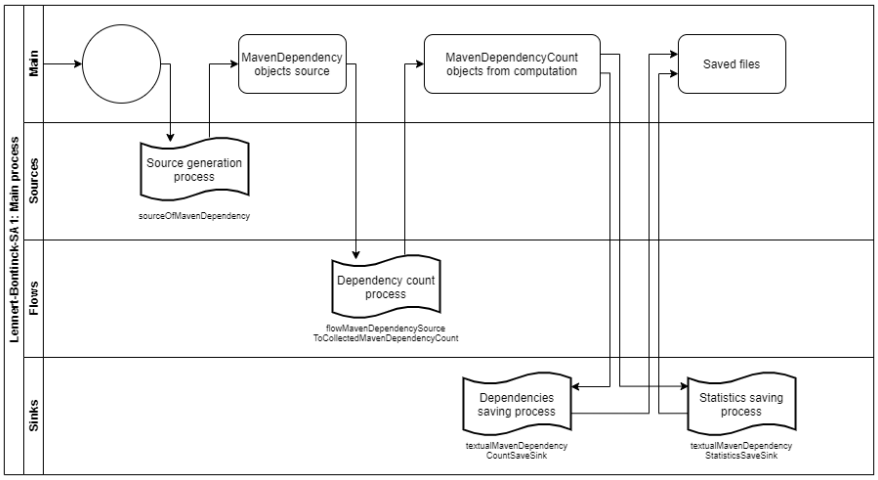

Figure 2.1: Simplified representation of the main application.

2.2 MD, MDC and MDS objects

There are a total of three case classes for this assignment. `MavenDependency` (`MD`) represents a singular Maven dependency as instantiated from the extracted file. It has a `library` (name), `dependency` and `dependency_type`. Thus multiple dependencies of an equal Maven library give rise to different `MD` objects. Its companion object provides an alternative constructor.

`MavenDependencyCount` (`MDC`) represents a Maven library and its dependencies. It has a `library` (name) and a dependency count per dependency type (`compile`, `provided`, `runtime` and `test`). Its companion object provides a function for combining two `MDC` objects. The `MavenDependencyStatistics` (`MDS`) object is used for the statistics part of the assignment. It has a `minimumDependencies` and a count of libraries eligible for the statistic per dependency type. Its companion object provides a function for adding a `MDC` object to the statistic object.

2.3 Source generation process

Figure 2.1 shows how `Main` makes use of a source generation process to provide a list of `MD` objects as source. This source is available as `sourceOfMavenDependency` under the `Sources` object. The process of generating this source is visualised in figure 2.2.

The process makes use of an initial `ByteString` source which is simply the extracted file. The raw `ByteString` source is then parsed as a CSV through `CsvParsing` whose records are then converted to a map of strings. This map can be used to instantiate `MD` objects resulting in a source of `MD` objects.

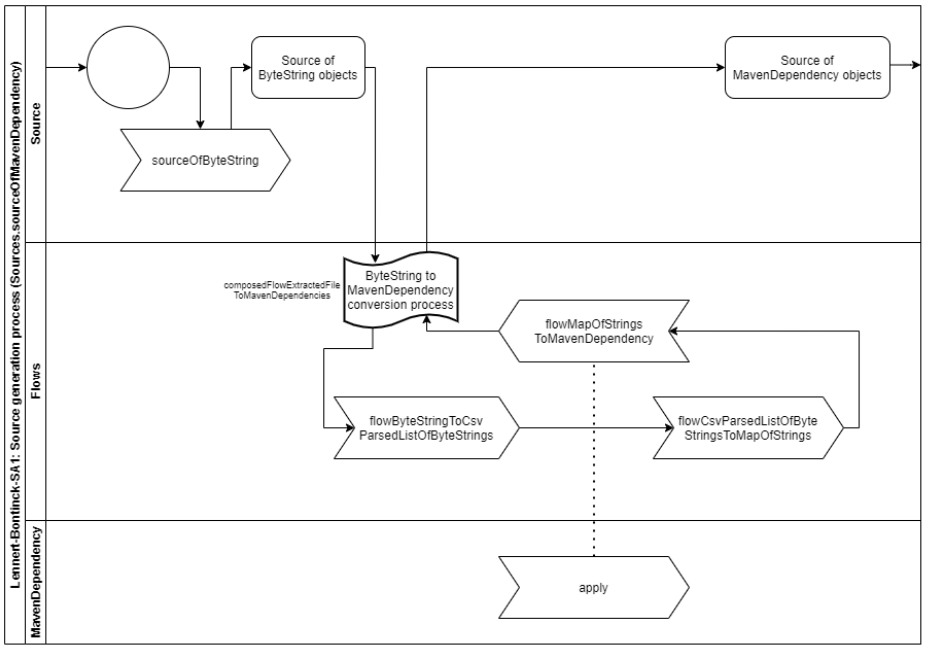

Figure 2.2: Simplified representation of the source generation process.

2.4 Source generation process

Figure 2.1 shows how `Main` makes use of the explained `MD` object source to collect `MDC` objects through a dependency count process. This dependency count process is available as `flowMavenDependencySourceToCollectedMavenDependencyCount` under the `Flows` object. This flow makes use of custom flow shapes and is visualised in figure 2.3.

First, the incoming stream of `MD` objects is grouped on their library name to create a substream for each library. `flowMavenDependencyToMavenDependencyCountParallel` is a custom flow shape responsible for making a balanced approach of counting the dependencies by adding a `Balance[MavenDependency]` having two ports. These two ports then perform the same custom flow shape `flowMavenDependencyToMavenDependencyCount` to perform the actual counting and are merged on the end. All of this is included in `flowMavenDependencyToMavenDependencyCountParallel`.

`flowMavenDependencyToMavenDependencyCount` is shown in detail. The counting happens by broadcasting the incoming `MD` objects to four separate flows. These flows generate `MDC` objects having either a count of one or zero for the field they are counting. All of these `MDC` objects are then merged to a singular `MDC` object by using `toSingleMavenDependencyCount`, which uses `flowMultipleMavenDependencyCountsToSingle` which in it turns uses `mergeMavenDependencyCount` from the `MDC` companion object.

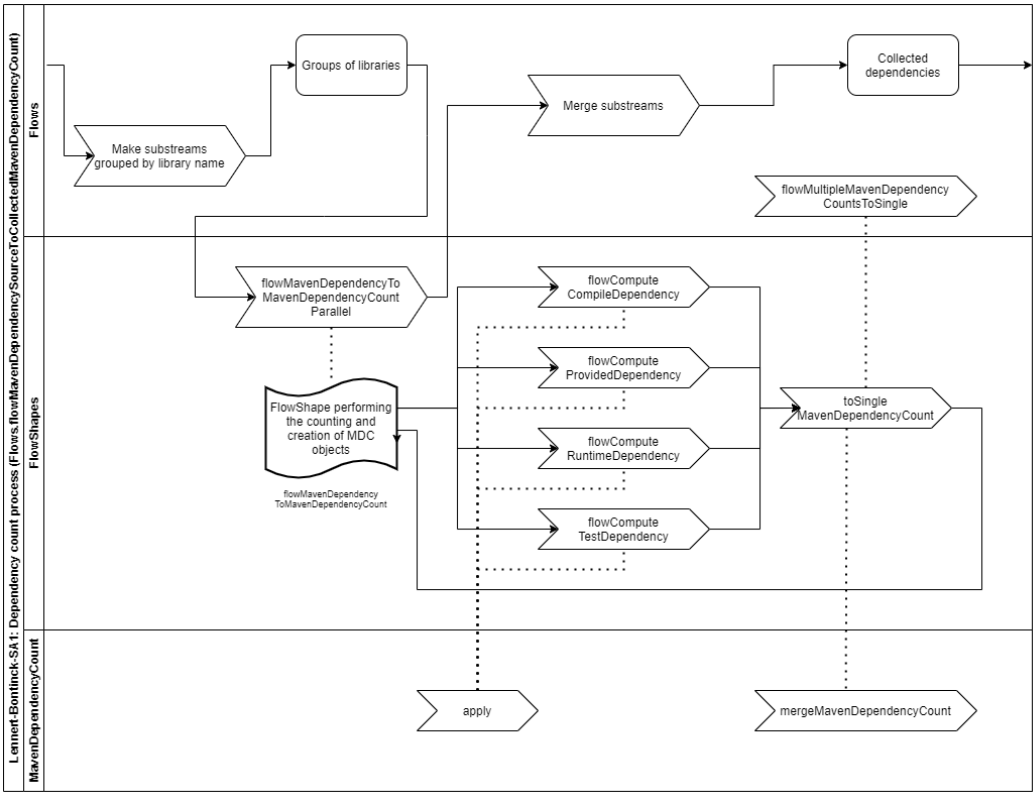

Figure 2.3: Simplified representation of the counting process.

2.5 Saving process

Figure 2.1 shows how `Main` makes use of the explained `MDC` objects to save dependency count and statistical information to a text file. This saving process is available as the `textualMavenDependencyCountSaveSink` and `textualMavenDependencyStatisticsSaveSink` sink under the `Sinks` object. Figure 2.3 visualises the saving process.

Saving consists of generating the right `ByteString` stream which can then be saved by `FileIO.toPath`. For the first sink, this just consists of converting each input `MDC` object to a textual representation in the specified form. For the second sink, the input `MDC` objects need conversion to `MDS` objects first before the specified textual representation can be made. In order to do this, the function `mergeMavenDependencyCount` from the companion object of the `MDS` class can be used in a folding manner. This is done in `flowMavenDependencyCountToMavenDependencyStatistics` where `minimumDependencies` can also be specified.

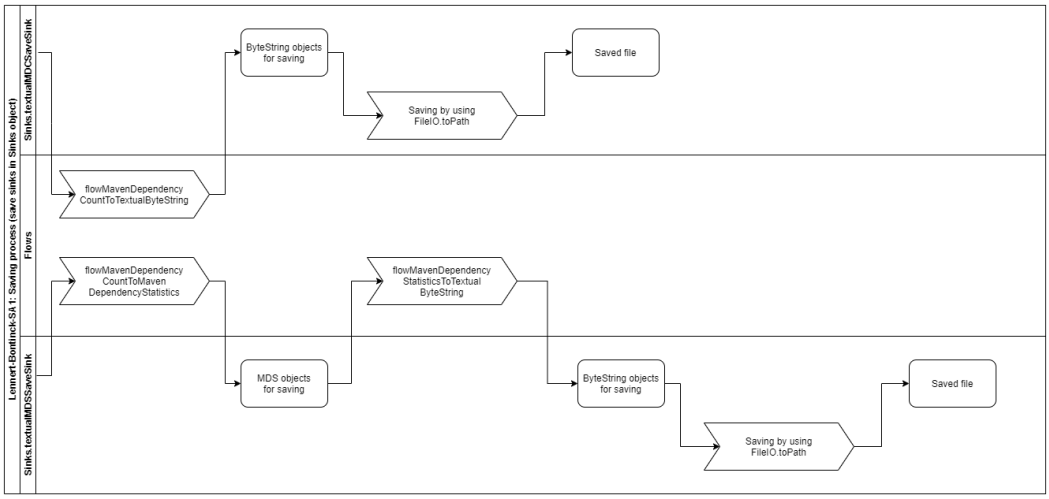

Figure 2.4: Simplified representation of the saving process.

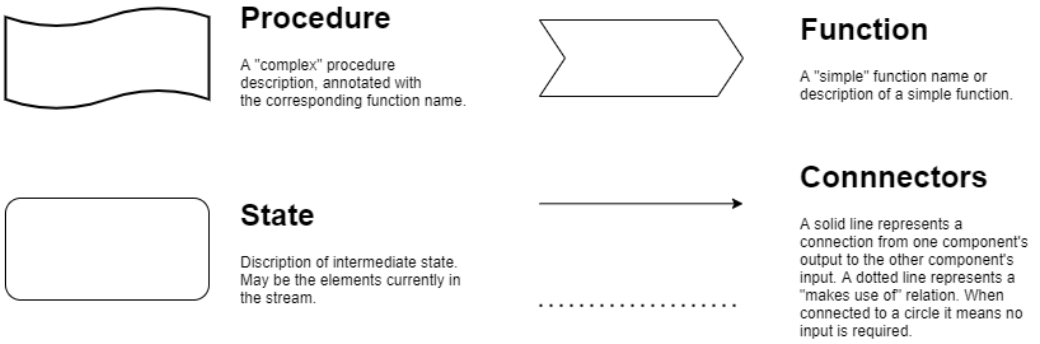

Figure 2.5: Legend for the figures.

References

Bontinck, L. (2021). *Assignment 1 of software architecture course* [GitHub commit: TODO]. Retrieved July 20, 2021, from https://github.com/pikawika/VUB-SA-assignment-1

De Smet, R. (2020). *Vub latex huisstijl* [GitHub commit: d91f55799abd390a7dac92492f894b9b5fea2f47]. Retrieved November 2, 2020, from https://gitlab.com/rubdos/texlive-vub