Lab 1 - Getting started with Java, AWS-SDK and S3

Jesus Omaña Iglesias & Pablo Castagnaro

Large Scale Distributed Systems - 2023/24 Edition

0.1 Changelog

2024/01/15

1 Objective of the lab

Create a small Java command called `TwitterFilter` that, given a collection of files containing tweets, copies all tweets in a given language into a new file, and uploads it to S3.

The application receives the following parameters:

- a 2 character string, which represents a language following the ISO 639-1 standard.
- the name of a local output file (where the output will be temporarily stored)
- the name for a bucket on S3 where to upload the resulting file
- a variable number of strings representing the paths of the files to be processed by the application (one or more files). Each file contains tweets in JSON format, one tweet per line

The command produces as output a local file containing all tweets in that language and then uploads it to Amazon S3 in a given *bucket* name with a given *key*.

Example: a call with the following parameters:

```
TwitterFilter es /tmp/output-es.txt test-bucket f1.json f2.json f3.json
```

will read 3 files (`f1.json f2.json f3.json`), and generate a new output file `/tmp/output-es.txt` containing all the tweets in the Spanish (`es`) language, then upload such file in an S3 bucket named `test-bucket`.

We divide the exercise in 4 parts.

1.1 Dependencies

To implement this lab, we will use two different external libraries. They should be added as dependencies to your Maven project.

- **GSON** Developed by Google, this library is one of the most used solutions for treating JSON content in Java. Latest version is `2.8.6`.
- **AWS-SDK for Amazon S3** Created by Amazon, facilitates the access to S3 from Java code. There are 2 versions of the SDK, use version `1.11.xxx`, possibly the latest version.

A seed project is available for you to bootstrap the development of this lab. This project implements the classic file structure for Maven projects and includes a main class file *TwitterFilter* that you don't need to modify. The code of the seed project is available in Aula Global in the file named *LSDS2024-Lab1-SeedProject.zip*

2 Parsing JSON (2.5 points)

The objective of this section is to parse tweets, and transform them from text lines representing a complex JSON format (with one JSON Object per line), to a simplified model, retaining only a few fields, named `SimplifiedTweet`.

We'll use Tweets from a dataset collected during Eurovision 2018. The dataset is available at the following *public* s3 path: `s3://lsds2022/twitter-eurovision-2018.tar.gz`. You should download (with the AWS CLI) and decompress this dataset onto your workstation.

2.1 Implement model class

The following snippet provides a partial implementation of this class, that you should extend by implementing, at least, the following components:

- the constructor
- the static parsing method that transforms a (JSON) string into an element of this class
- any other method that might be of use (think of a typical Java Object)

Partial model for a simplified tweet

```
public class SimplifiedTweet {

  private final long tweetId;      // the id of the tweet ('id')
  private final String text;          // the content of the tweet ('text')
  private final long userId;       // the user id ('user->id')
  private final String userName;   // the user name ('user'->'name')
  private final String language;      // the language of a tweet ('lang')
  private final long timestampMs;    // seconds from epoch ('timestamp_ms')

  public SimplifiedTweet(long tweetId, String text, long userId, String userName,
                         String language, long timestampMs) {
    // YOUR CODE GOES HERE!
  }

  /**
   * Returns a {@link SimplifiedTweet} from a JSON String.
   * If parsing fails, for any reason, return an {@link Optional#empty()}
   *
   * @param jsonStr
   * @return an {@link Optional} of a {@link SimplifiedTweet}
   */
  public static Optional<SimplifiedTweet> fromJson(String jsonStr) {
    // YOUR CODE GOES HERE!
  }

  @Override
  public String toString() {
       // Overriding how SimplifiedTweets are printed in console or the output file
       // The following line produces valid JSON as output
       return new Gson().toJson(this);
    }
}
```

A few remarks:

- Consider as valid tweets only those JSON objects that **contain all mandatory fields**, and discard everything else
- Feel free to implement any additional method you find convenient
- `toString()` is a method inherited from the class `Object`: all objects in Java inherit from class `Object`, thus all of them have access to the original implementation of such method on the `Object` class; the `@Override` annotation allow developers to change its parent behaviour.

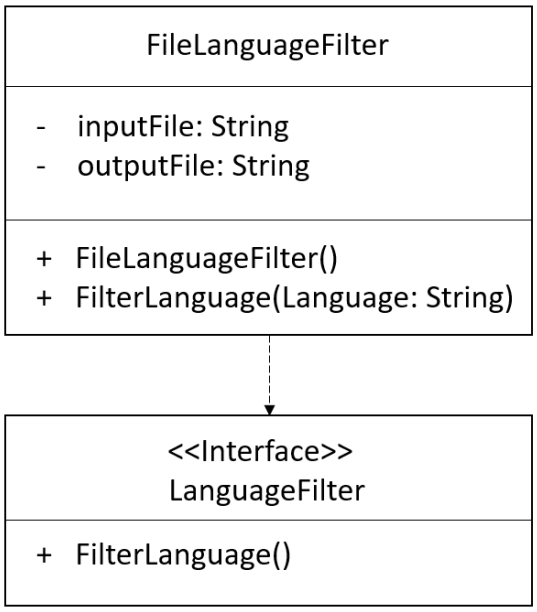

Figure 1: UML model of the S3Uploader class

3 Implement a Filter class (2.5 points)

This class will take a text file containing tweets and append to an output file only the tweets in a specified language.

Write a class named `FileLanguageFilter` that implements the following interface:

```
public interface LanguageFilter {

  /**
   * Process
   * @param language
   * @return
   */
  void filterLanguage(String language) throws Exception;
}
```

The implementation should take into account the following:

- implement the given interface in a class named FileLanguageFilter.
- the input to this class should be a single text file
- the unit of reading/writing is a line (including the newline)

- each line should be parsed into an `Optional<SimplifiedTweet>`
- there will be always a single output file for the whole application
- in case of any failure, it's ok to propagate the exception
- check the Java Review slides and in particular the `BufferedReader` and `BufferWriter` to recall how to read/write files.

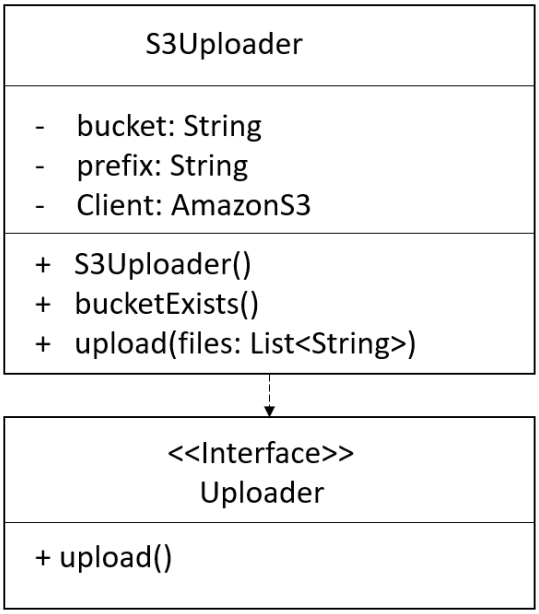

Figure 2: UML model of the FileLanguageFilter class

4 Implement an uploader (2.5 points)

Write a class named `S3Uploader` that takes a file as input and uploads it to an S3 bucket under a given key.

Your class should implement the following interface:

```
public interface Uploader {

    /**
     * Uploads a file to the target specified through its implementation
     * @param file the file to upload
     */
    upload(List<String> files);

}
```

The constructor of your `S3Uploader` class should have the following parameters:

- implement the given interface in a Java Class named `S3Uploader`.
- `BucketName`: name of the bucket where the file will be uploaded.
- `Prefix`: this parameter sets the string to be concatenated before the file name when creating the key (file path) inside of the bucket. Example: if the destination is `s3://sample.bucket/some/prefix/file1`, then the bucket would be `sample.bucket` and the prefix would be `some/prefix`

– `Credentials profile name`: by default, the AWS console shows a profile named `default` that most of you have copied in your `.aws/credentials` file. This name can be changed to upf in the credentials file or, if you prefer, you can also change the name in the main method so the default profile is accessed. Info about named profiles can be found here.

Some suggestions:

– You'll need to instantiate a `AmazonS3` instance with the appropriate credentials. The credentials should use the ones you have configured in the AWS section. Retrieving credentials should use the appropriate `CredentialsProvider`. Check here: (see "default credential profiles")

– Does the bucket exist? What happens if it doesn't?

About the AWS SDK dependency

– In this section, you need the AWS-SDK for Java: `https://aws.amazon.com/sdk-for-java/`

– You **don't need the full SDK** (`aws-java-sdk`), you can just use the S3 dependency (`aws-java-sdk-s3`)

5 Benchmarking (2.5 point)

Using the JAR you have created, the given twitter data, and your S3 acquired knowledge, each student in the same group should:

1. create a destination bucket with the following pattern: `lsds2024.lab1.output.<USER-ID>`, where `<USER-ID>` should be changed for your user identifier within the UPF (the number starting with U and followed by digits, like `Uxxxxxx`)
2. run the program with the following language parameters and report the obtained results (number of resulting tweets and computation time for each of them)
   – Tweets in Spanish ('es') (output to `lsds2024.lab1.output.<USER-ID>/es`)
   – Tweets in English ('en') (output to `lsds2024.lab1.output.<USER-ID>/en`)
   – Tweets in Catalan ('ca') (output to `lsds2024.lab1.output.<USER-ID>/ca`)
3. Provide also a brief description of your runtime environment. Be sure to report the time unit that you are using.
4. Did you encounter any issue when performing the calculation?

Please, include a section *Benchmark* in the README.md within your project where you describe such results for your implementation.

6 Additional points (Optional)

6.1 Unit tests for the SimplifiedTweet class (1 point)

To make sure that your implementation is correct, it's very convenient in these cases to write unit tests for it. To help with the task, you can use a real tweet from the collection as a "resource" for your test, and verify that the expected fields are correctly read and set in the Java model.

Try to write, for instance, these tests:

- a test that parses a real tweet (for instance, the first line of one of the files from the twitter collection)
- a test that parses invalid JSON
- a test that parses valid JSON where one of the fields is missing

If you do the above extensions, please add a section named *Extensions* to the README in your repo detailing your choices

7 Final remarks. Important!

- Your code must be submitted through Aula Global (one submission per group)
- Your code must compile. Non-compiling code won't be evaluated!
- Your submission must include a README file, where you answer all the questions in the statement
- Your submitted code must be **clean**, that is: it doesn't include unnecessary files, output files, compiled classes, but only your source code. Make sure to run a `mvn clean` before uploading code to Aula Global and exclude unnecessary files (for instance, IDE specific files or directories)
- Your code should be able to create a jar containing all your non-test classes (`mvn package` should work)
- The `TwitterFilter` class should be runnable, that is: a command like the following should work:

```
java -cp jarfile edu.upf.TwitterFilter arg1 arg2 ...  argN
```

8 Deadlines

The deadline is Feb, 15th at 23:59:59. Submission after the deadline **won't be accepted**, sorry!