TABLE OF CONTENTS

TEST & BUILD THE APPLICATION

Run Unit Tests

```
$ sbt clean test
```

Run Integration Tests

```
$ sbt clean it:test
```

Note:

Integration tests require connection to the live GitHub and Twitter APIs, thus you will need an internet connection. Also, before you run integration tests, you will need to edit `src/it/resources/application.conf` with your own authentication info. For CI we would create a GitHub and Twitter users specifically for this purpose.

To build a jar file with all dependencies:

```
$ sbt clean assembly
```

This application was developed using SBT 0.13 and Scala 2.11.

EXECUTE THE JAR IN THE COMMAND LINE

Note:

Before you can run the application, you need to provide GitHub's and Twitter's authentication info. I suggest copying ***src/main/resources/application.conf*** *(a template configuration file) and then editing it:*

```
$ cp src/main/resources/application.conf .
$ vi application.conf
```

You may then provide your own configuration to the app by passing the system property ***config.file*** *as follows:*

```
java -Dconfig.file=application.conf \
             -jar ./target/scala-2.11/grid-assembly-1.0.jar \
             -Dconfig.file=application.conf ...
```

Examples on how to call the commmand-line app follow in the next section.

To generate an authentication token for GitHub, follow these instructions. For Twitter, go the ***Application Management Page*** *in your Twitter account, then* ***Create New App*** *(if you don't already have one), then in your app navigate to* ***Application Settings > manage keys and access tokens****. Once you are there, you may need to go to* ***Token Actions*** *(at the bottom of the page) and click the button* ***Regenerate My Access Token and Token Secret*** *if you have just generated your app.*

Usage

```
java -jar <jar name> [options] keyword

keyword                  Search keyword to generate the output for.
-o, --output-file <file>
                         Name of the output file. If not provided, \
                         will print output to console.
--help                   prints this help text
```

Examples

```
# Generate grid to output file ‘output.json‘
java -Dconfig.file=application.conf \
     -jar ./target/scala-2.11/grid-assembly-1.0.jar reactive -o output.json

# Generate grid to output file ‘output.json‘ with log level set to ‘DEBUG‘
java -Dconfig.file=application.conf \
     -Dlog4j.logLevel=DEBUG \
     -jar ./target/scala-2.11/grid-assembly-1.0.jar reactive -o output.json
```

ON THE DESIGN CHOICES

Note:

I'm being very verbose here (I do try to be verbose as a rule of thumb, but not to this particular degree) since I'm trying to explain my methodology of work and justify my design decisions for the purpose of showcasing my skills. Therefore, I tried to write this as it was a technical article.

An important thing to know prior to design is the lingo of your business domain, so one can properly define and name required abstractions.

The application in this case builds a `Grid`, which is assembled from a collection of resources. In this particular case, the resources are data from the GitHub API (projects matching the input search keyword) and data from the Twitter API (tweets mentioning a project limited to a maximum number of 10). Each join of a given GitHub project with its correspondent tweets represents a `Cell` in the `Grid`.

Resources provide `Assets`, therefore data from GitHub for a particular project represents an `Asset` and data from Twitter for a particular tweet represents another `Asset`. Assets are collections of data provided by its respective resources:

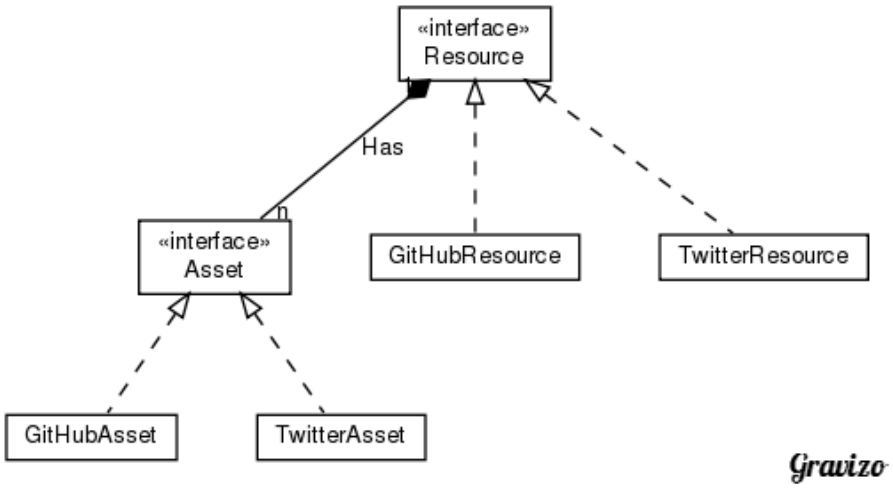

Figure 1: GitHub Asset and Resource Class Diagram

These resources are used at the end of the day to assemble each `Cell` on the `Grid`:

Where: *(11 x `Asset`) = (1 x `GitHubAsset` + 10 x `TwitterAsset`).*

I haven't defined an abstraction for `Cell`, which would be implemented by `GitHubTwitterCell`, as for the moment only a (GitHub x Twitter) `Cell` is

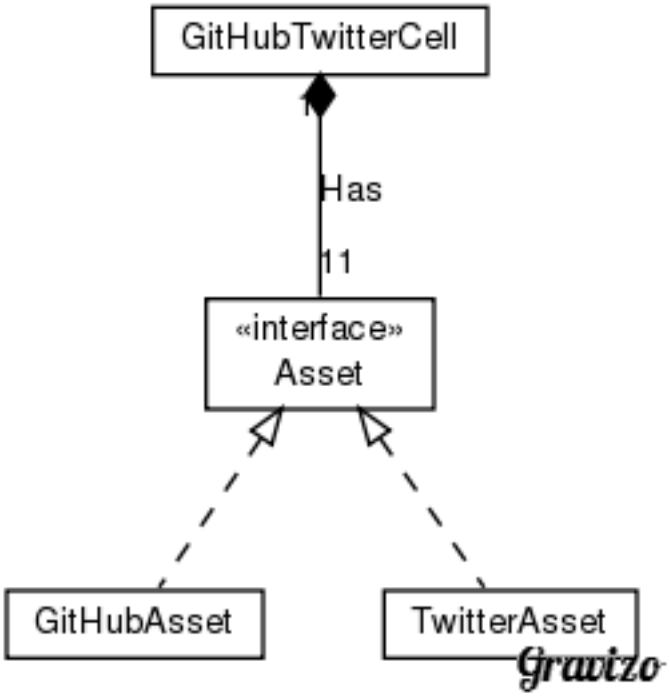

Figure 2: Asset and Resource Class Diagram

required. If we require multiple types of grids, an abstraction would come in handy, but given the requirements, I find this unlikely (but not impossible).

Now that we have defined abstractions for our data, an important thing to consider is how to decouple our implementation for the resources from the live GitHub and Twitter APIs. That's not only important for testing reasons (one would like to test ones code without having to rely on the live third-party REST APIs by using mock-ups, in our particular case scalamock) but also to decouple our business logic from specific third-party clients for the APIs (we might want to use different ones in case of production or performance issues or just because we have found a better client and we want to refactor for cleaner and simpler code).

I would rather use Scala (or Scala wrapper) libraries, but the ones available for Twitter at the moment seem like a work in progress, therefore I have opted for `twitter4j`. `buhtig`, a GitHub API client, is Scala and it seems in good shape. It uses `json4s` and it's very straightforward to use.

The APIs are decoupled from the resources through the interface `JsonApiClient`, which defines a method `search`, which gets a query string as an argument and returns a JSON object.

Note: *I could have used a Scala native type instead, but I think that a JSON return value (`JValue` from `json4s`) is a good abstraction for a REST API client and `json4s` is ubiquitous enough in the Scala community, thus using a Scala native type here (such as `String`) would be overkill.*

Finally, once we have all data abstractions and its specializations in place, we can define the final App:

`GitHubTwitterGridBuilder` is a Scala trait, which `App`, a Scala object and main command-line app, uses to generate the grid, a JSON output, from the input search keyword and resources.

`App` also uses another trait,`AppConf`, to read configuration properties from `application.conf`. For reading and parsing command-line arguments, `App` uses `scopt`.

That completes the design. Here's the class diagram for the app:

To finalize, I also would like to be emphatic about the fact that I follow clean code principles. Sometimes I got commentary about "not having enough comments in my code". I try to make my code expressive:

> "Every time you write a comment, you should grimace and feel the failure of your ability of expression." - Robert C. Martin

If you don't understand what one of my classes does by its name (the same goes for methods and variables) I accept the criticism and will try to make it more expressive, but I will stick to my guns and use better naming to better

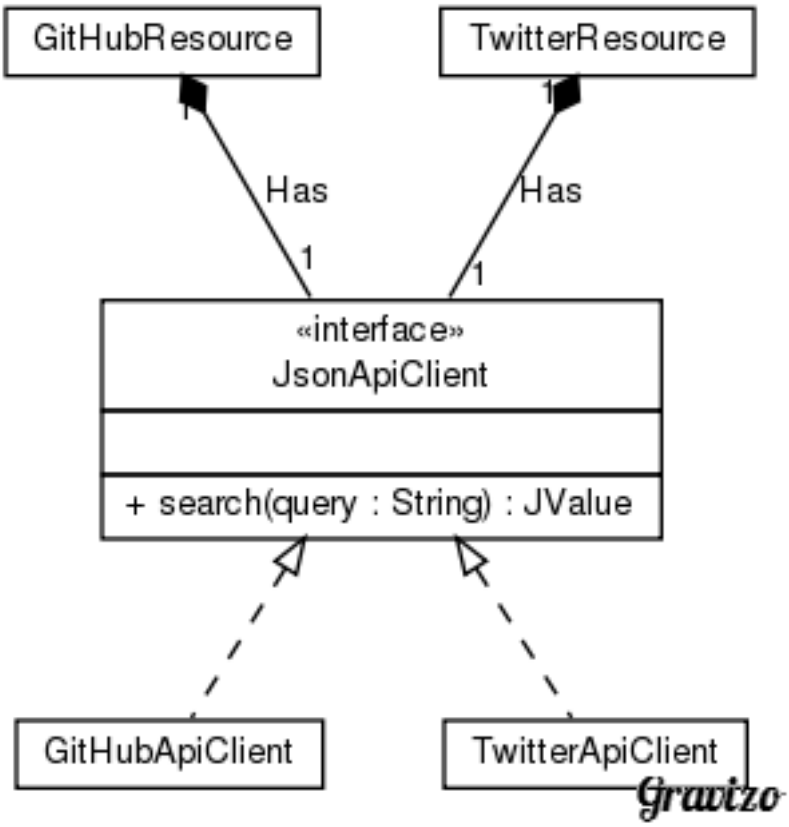

Figure 3: Asset and Resource Class Diagram

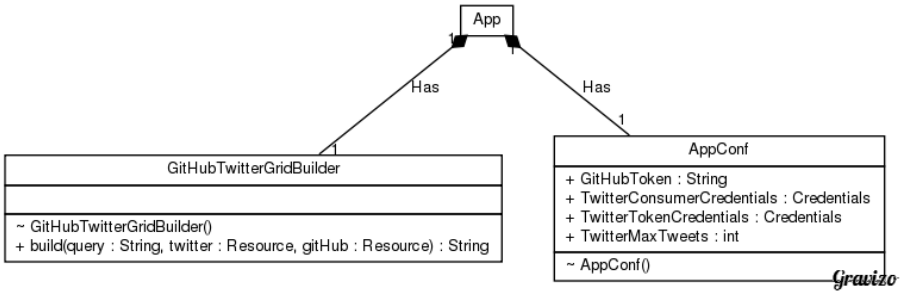

Figure 4: App Class Diagram

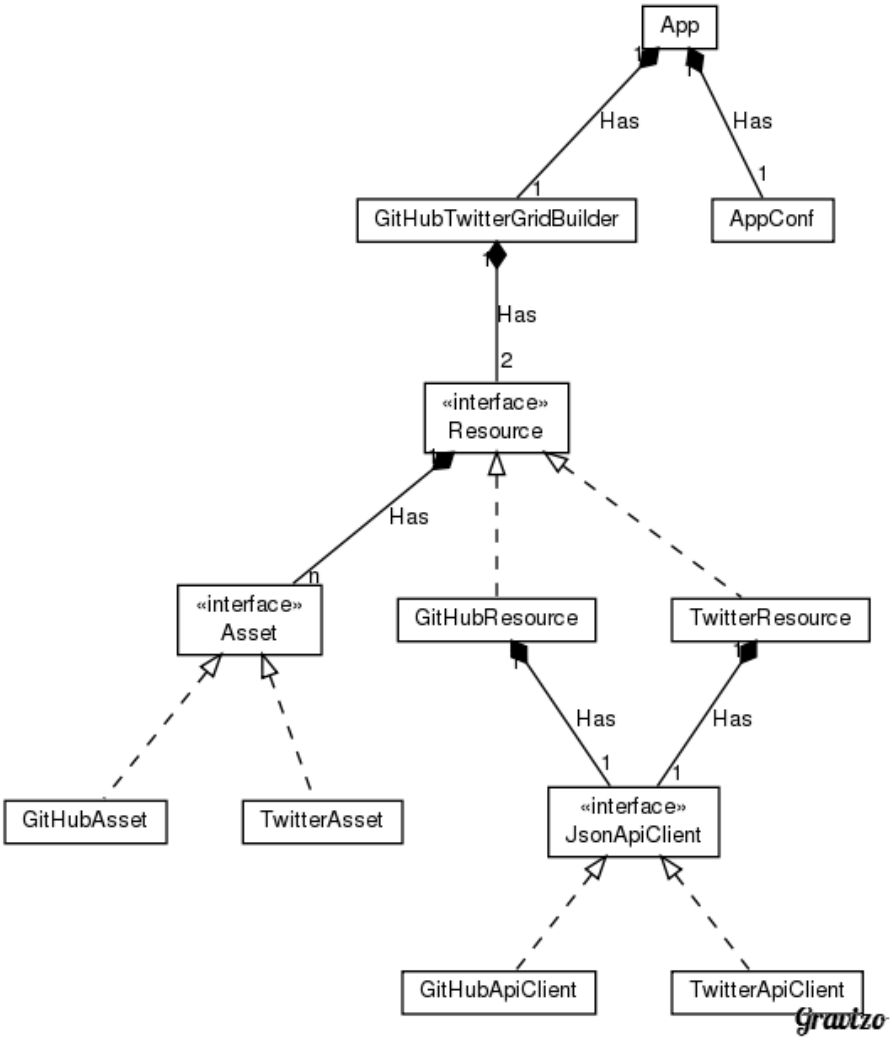

Figure 5: App Class Diagram

communicate the code's intent. I avoid cluttering my code with comments, unless they are meant for ScalaDoc for a public API.

It's also worth to mention that I always run IntelliJ's `Analyze > Inspect Code` before I commit any code.

TEST COVERAGE REPORTS

A SBT plugin, `sbt-coverage`, has been used to generate coverage reports. At the moment `sbt-scoverage` does not generate a unified report with both unit and integration tests, so they need to be generated separately:

For unit tests:

```
$ sbt clean coverage test
$ sbt coverageReport
```

For integration tests:

```
$ sbt clean coverage it:test
$ sbt coverageReport
```

After generating each report, you will find them in `target/scala-2.11/scoverage-report/index.html`.

Note:: *If you generate both reports in succession, the last one will override the results from the previous one.*

Test Coverage Analysis

Since reports for unit and integration tests are not merged together in a single report, we need to analyse them separately.

- `GitHubAsset`, `TwitterAsset` and `AppConf` only store data (Scala equivalent of POJOs), thus unit tests are not necessary.
- `GitHubResource`, `TwitterResource`, `GitHubTwitterGridBuilder` and `GitHubApiClient.SearchException` have 100% line coverage in the unit test reports.
- `GitHubApiClient`, `TwitterApiClient` and `JsonApiClient.SearchException` have 100 % line coverage in the integration tests.
- `App` is the command-line app, so the proper way to test it is through a system test.

CODE QUALITY ANALYSIS

I always run IntelliJ's `Analyze > Inspect Code` before I commit any code. I'm also using scalastyle-sbt-plugin. This project includes a `scalastyle` config:

`./scalastyle-config.xml`

`SonarQube` is recommended, but I don't have a running server with Scala extension at the moment (only Java, `//TODO`).

HOW TO VIEW THIS DOCUMENT

This document is better viewed using IntelliJs `GFM Plugin. In case it isn`t available, there is a PDF version of this document in the same directory.

For my own reference, to convert markdown to PDF use pandoc with the following command:

```
pandoc README.md -f markdown -t latex -s -o README.pdf
```

USEFUL RESOURCES

- GitHub REST API Repository Search
- Twitter REST API Search