Overview

Twitter's data is available through a RESTful API, which supports two methods of pagination:

- **Timeline pagination** is used with chronological data that may see frequent additions. Items in a Timeline are assigned an `"id"` that increases globally with every new post, and the timeline is served in reverse chronological order. Clients can traverse the list by passing the lowest `"id"` they have encountered as the `max_id` query parameter. The default `max_id`, -1, begins pagination from the start. Pagination concludes when the response is empty.
- **Cursor pagination** is used with data of an arbitrary order (such as by name or by sorted identifier). If further data are available, the response to the first request includes the `"next_cursor"` field that should be passed as the `cursor` query parameter to continue pagination through the data set. Pagination concludes when the `next_cursor` is not present in the response.

This exercise includes a replica of the Twitter API, `MockTwitterService`, which models the following endpoints:

- `/statuses/user_timeline`, which utilises timeline pagination and returns a sequence of tweets in reverse chronological order.
- `/lists/subscriptions`, which utilises cursor pagination and returns a sequence of subscriptions in arbitrary order.

The mock API may fail pseudo-randomly with the `ServiceUnavailable` response. Both endpoints support the optional `count` query parameter to modify the size of the page of data returned.

Objective

You are asked to implement the `TwitterRestClient` such that: (1) `getAllTweets()` retrieves *all* tweets of the `/statuses/user_timeline` endpoint; and (2) `getAllSubscriptions()` retrieves *all* subscriptions of the `lists/subscriptions` endpoint; and (3) the domain models declared in `models.scala` (`Tweet` and `Subscription`) follow the output produced by the `MockTwitterService`. You *must* implement against the `RestService` interface. Take care to gracefully retry if the service is unavailable; for development purposes, you may choose to set the argument `serviceUnavailableProbability = 0.0` when initialising the mock service.

Running:

```
$ sbt run
```

Testing:

```
$ sbt test
```

You may find it faster to enter the SBT REPL (`$ sbt`) first.

Resources

- Scala 2.11 Standard Library: https://www.scala-lang.org/files/archive/api/2.11.11
- Scalatest (specifically: matchers): http://www.scalatest.org/user_guide/using_matchers
- Play! Framework 2.4 Reference (specifically: the JSON packages) https://www.playframework.com/documentation/2.4.x/api/scala/index.html#play.api.libs.json.package
- Play! Framework 2.4 (specifically: Scala JSON guide) https://www.playframework.com/documentation/2.4.x/ScalaJson