School of Electrical and Information Engineering - Witwatersrand
ELEN 7046 Software Technologies and Techniques

Big Data Visualization using Commodity Hardware and Open Source Software - historic data sourcing

Matsobane Khwinana

779053

4 July 2016

Abstract:

Historic data sourcing component is one of the two sub-components that make up the Big Data source module of the Big Data Visualization using Commodity Hardware and Open Source Software project. This report showcases the design and development of the historic data sourcing module. The technologies used are discussed and the logical flow of the program is illustrated; the test driven development technique used during developed is explained with a snippet of test cases showcasing how the module was tested and the results are presented.

Table of Contents

Introduction

The historic data source component is the sub-component of the data sourcing module of the Big Data Visualization using Commodity Hardware and Open Source Software project undertaken by group 2 of the 2016 ELEN-7046 class at Wits University. As stated in the group project report [1]; the project provides a low cost solution to sourcing, processing and visualizing Big Data using non-commodity hardware and open source software. The data sourcing component was divided into two sub-components, namely streaming (for live data feed) and historic data extraction (using start and end dates, and counts). This report discusses the solution provided for the historic data sourcing sub-component or service to illustrate the contribution and made by the author on the project.

More background information on the problem at hand is provided. The primary use cases are listed in the Requirements section and the overview of the design is outlined; the technologies used in the project are briefly discussed; the Java EE reference architecture that is used in the project is brief explained; the testing method is also explained with some code snippets and the conclusion sums up the report; the references and appendices then follows.

Background

The most topical subjects during the inception of the project were the United States general elections and South African municipal elections. The project chose to visualize the sentiments on the social media regarding both elections focusing on the Hilary Clinton and Donald Trump for the US and the 3 leading political parties in SA, namely, ANC, DA and EFF.

Twitter was chosen as the data source for the group project, as it was felt this better portrayed the sentiment of the people, not just online and traditional media outlets. Also, the Twitter API proved easy to integrate with.

Twitter provides REST APIs for searching and retrieving data using various inputs such as the Twitter account name, a tweet (Twitter post) ID, hashtags and dates.

This component is tasked with the collecting the data from Twitter using the APIs provided focusing on the history data.

Requirements

Below is the list of primary use cases addressed by the historic data sourcing component.

Functional

- Extract tweets by hash tags using start and end date
- Extract and persist tweets by list of hashtags using random dates
- Extract and persist tweets by list of hashtags using start and end dates
- Distribute all persisted tweets to the data processing component FTP
- Distribute tweets for a given hashtag

Non-functional

- Must be scalable
- Must be portable
- Must be distributable
- Must support concurrent users

Approach

The Big Data collection component was divided into 2 sub-components: streaming and historic. The project followed a "divide and conquer" strategy to decompose the data collection problem into simpler sub-components. The divide and conquer strategy in software design, helps in avoiding a single point of failure, and it enables parallelism [2]. Dividing the data collection component into separately deployable smaller services provide few advantages including scaling, separation of concerns, issue isolation and ability to use different technologies and techniques.

Challenges

Two challenges were faced by the data sourcing components included the tweets (Twitter post or message) location data and the usage rates limits on the data provider APIs.

The location; as discussed in the project group report, most of the tweets did not have the location at which the tweets were made and the user profile's location was then assumed as the tweet location. However, the user profile location is recorded a free text which then allowed users to put anything they wished as their profile.

As a workaround the data sourcing module then used Google Maps Geocoding API to determine the geographic location coordinates. Because the Twitter users can put in whatever they like as location, some of the location coordinates could not be resolved.

Usage rate limits; both Twitter and Google Maps Geocoding API have usage rate limitations. Twitter API allows 100 tweets per request and only 450 tweets per 15 minute window [3], while Google Maps Geocoding API allows only 2500 requests per day; anything more that Twitter promises to block the requesting account whereas for Google you will be allowed more requests provided you pay [4].

As a workaround for the limitations a scheduler was incorporated into the design so the collection of data can be scheduled to work within the limitations. A caching solution was proposed as well, were previously determine locations can be cached to avoid requesting the coordinates of same locations repeatedly.

Design Overview

The component was required to extract and store data locally and later distribute it to the data processing component. The data extraction is done using Twitter's provided REST APIs. OAuth authentication is performed by the application into Twitter and once successful the application makes a subsequent call to Twitter to obtain the actual data and persist the data in the local data store.

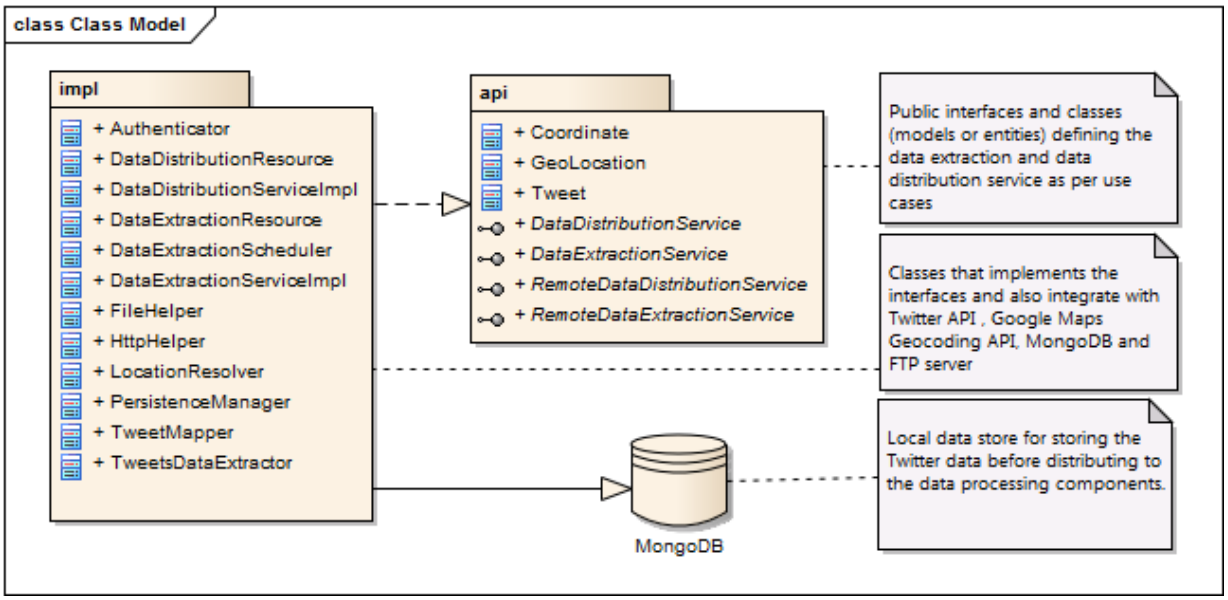
The class model diagram in the next page illustrates a high level view of the structure of the historic data collection component.

Due to the non-functional requirements stated, Java EE was the perfect candidate.

The API component includes the interfaces that describe the extraction and distribution use cases. The implementation module includes the actual implementation classes for the API; this module also contains the implementation of the REST endpoint. The project could have chosen to implement the endpoint component as a separate independently deployable module however due to the abstractions provided by the Java EE, there was no need separate the endpoint. Even though the endpoint is in the same component as the service implementations classes, they are not tightly coupled at all. The endpoint accesses the implementations via the public interfaces provided by the Java EE EJB (Enterprise

Java Beans) through dependency injection. However should the need to separate out the REST endpoint arises in the future this can be easily achieved.

The implementation module integrates with both Twitter and Google Maps Geocoding APIs. The design was not proposed upfront however it evolved and surfaced as each use case was implemented - one after the one.



Design rationale

API was separated from implementation to achieve loose coupling. The API module is made up of public interfaces and entity classes only, see the class diagram in Appendix A. The API is packaged as separate Java Archive (jar) file making the component independently shareable; meaning the interest users of the API need not have access to the implementation details of the service. This allows for easy swapping of implementation without affecting the API users. This made possible by using Java EE's Enterprise Java Beans (EJBs).

Technologies used

The data sourcing component uses Free and Open Source Software and tools for development and execution as it is in line with the rest of the group project. Besides the Java SE and Java EE, below are the tools and technologies that were used to realize this project.

Operating system (OS): For both development and execution Linux was the chosen free and open source. However, because of Java technology's write once and run

every mantra, the project could have still been developed in any other free OS. Ubuntu 15:10 LTS was used for development and execution.

Netbeans 8.1 IDE: Netbeans is a natural fit for development using the Java technology. It does not require lots of plugins as opposed to Eclipse for example. Through the experience acquired by the author by using both Eclipse and Netbeans in prior projects, Netbeans came out top compared to Eclipse, it is lightweight and has a better support for Apache Maven; Maven is not treated as a second class citizen in Netbeans, it is nicely integrated into the IDE.

Apache Maven: For the job at hand Maven proved to be a suitable tool for compiling source code and packaging of deployment artifacts. Its excellent dependency management facility came out top when compared to other source code compilation tools like Apache Ant for instance. Even though this project does not use lots of dependencies and it is easier to download and bundle are jar file to project in Maven than it is Ant. Another selling factor for Maven project is project templates called archetypes. Another tool that could be considered was Gradle; however Maven was good enough for job at hand.

Git and GitHub: For source code management, Git was the obvious choice because of its ease of use, local repository feature which helps a great deal offline development and its excellent workflow and branching functionality. Together with Git, GitHub was chosen for host the source to facilitate collaboration. Bitbucket is another option for hosting the source code, however it offers the similar functionality sets as GitHub therefore GitHub was sufficient to carry out the job.

MongoDB: The data format returned by Twitter is in JSON format and the project also chose JSON a format for data interchange. As a result a document-based NoSQL database was suitable for storing the data. There was no need for an SQL database and the nature of the data for the purpose this project did not require relational modelling. MongoDB was the suitable choice for this job.

The framework

Java EE is an industry standard for developing portable, robust, scalable and secure server-side Java applications [5]. Through the open source fashion, the community put together a specification that provides a set of standard APIs which must be implemented by Java EE frameworks also known as application servers. Examples of a free open source Java EE frameworks or application servers include JBoss Wildfly, Glassfish, Payara (enhanced Glassfish) and Apache TomEE among others.

The Java EE specification serves as a reference architecture which the frameworks are required to implement. On the surface the Java EE architecture can be seen as a 5-layered architecture with the layers being, the presentation layer, a services layer, a domain objects layer, the infrastructure or middleware layer, and the back-end layer. However, since Java EE 6, Web Archives (war) files traditionally used for developing web modules only, now can be used to package EJBs therefore blurring the lines in terms number of layers.

In addition to the five layers used by client applications, the Java EE frameworks generate some internal layers which host certain responsibilities. Among others in the enterprise services layer, the Java EE framework generate an EJBObject for every enterprise bean (the class that defines the business logic), which acts as an interception layer at which the enterprise services such as networking, transactions, concurrency, security and persistence are applied [5]. Enterprise Java Beans were used extensively in the data sourcing component to take advantage of this enterprise services.

Throughout its design, the Java EE implements most of Gang of Four patterns [3]. For example, Stateless EJB implements Facade, Singleton EJBs implements a Singleton, CDI events implements the Observer pattern and many more design patterns prevalent in the Java EE.

JBoss Wildfly 10 was chosen as the Java EE framework for this project; other free Java EE frameworks such as Glassfish, Payara or TomEE can be used as well; they all implement the same specification therefore they ought to produce the same results.

Class diagram

The class diagram in Appendix A illustrates the API module design. The table below describes the each of the classes shown on the class diagram.

Class	Description
DataExtractionService	A public interface defining the data extraction service as per use cases. The interface uses Tweet class to represent a Twitter post or message.
DataDistributionService	A public interface defining the data distribution service as per use cases. The interface uses Tweet class to represent a Twitter post or message.
Tweet	A class that represents a Twitter post or message. It encapsulates the GeoLocation.
GeoLocation	A class that represents geographical location of a Tweet.

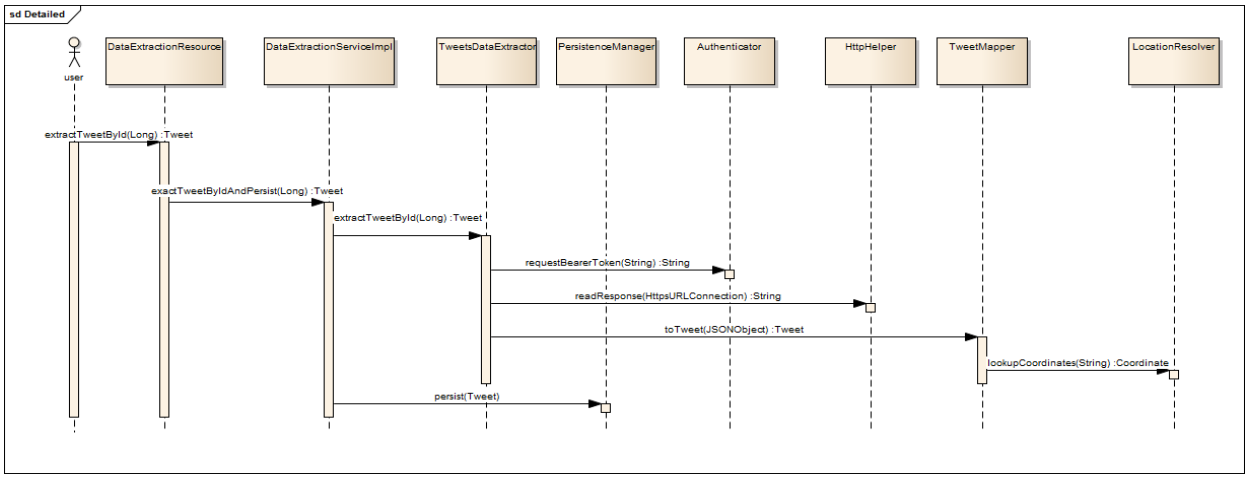
	It is made of an array of Coordinates; the type may have values such Polygon, Point, MultiPoint, etc.
Coordinate	A class that represents a geographical location coordinates, made of latitude and longitude.

The implementation class diagram in "Appendix B" shows the implementation module design; the table below describes the major classes in the diagram and omits other helper or utility classes because of the size of this report.

Class	Description
DataExtractionServiceImpl	An implementation class for the data extraction interface. The class delegates to the TweetsDataExtractor class for extraction of Twitter data and the PersistenceManager class for persisting the data in the persistent storage.
DataDistributionServiceImpl	An implementation class for the data distribution interface. The class delegates to PersistenceManager class for retrieving data from the storage and to FileHelper for distributing the data to an FTP location.
TweetsDataExtractor	A class that most of the work for extraction. However, the class delegates to other helper or utility classes for each of the steps of extraction. For example, authentication to Twitter API is handled by Authenticator, connecting to the Twitter API REST service is handled by the HttpHelper class, mapping the Twitter data to the Tweet entity is down by TweetMapper which also delegates to LocationResolver to help with Tweet location issue.
PersistenceManager	A class that handles the integration with the persistence storage, MongoDB. It is used for persisting, retrieving and deleting of data.
DataExtractionResource	A class that implements the REST endpoint for triggering the extraction of data.
DataDistributionResource	A class that implements the REST endpoint for triggering the distribution of data.
DataExtractionScheduler	A class that implements a scheduler. Another trigger for extraction. It is implemented using EJB timer service.

Sequence diagram

The sequence diagram below shows the execution flow of the data extraction service



When an extraction is triggered via the REST endpoint as illustrated in the sequence diagram. The endpoint class (DataExtractionResource) in invokes the DataExtractionServiceImpl to carry out the extraction execution. Even though the sequence diagram indicates that the REST endpoint depends on the implementation class, it is not the case. The extraction service interface is injected on the endpoint therefore there's no tight coupling between the endpoint and implementation. The implementation class invokes to the TweetsDataExtractor which invokes all its helpers to assist in connecting to Twitter API, authenticating and retrieving the data. Eventually the implementation class then invokes the persistence manager to persist the tweet to the database.

Testing

Test Driven Development (TDD) approach was followed during the development of the historic data source component. The technique was used alongside the Use Case Driven approach, were each use case is picked and every aspect of the use case is developed end-to-end, from interfacing with Twitter API to storing the data.

Using TDD a failing test would be writing first, and then code to make the test pass would be developed secondly and once the test passes, refactoring of the classes would follow and re-running the test.

The above was done iteratively, taking one use case at a time until all use cases were covered then the project was considered complete.

Below is a code snippet from one of the use cases:

Figure 1: Extract History Data test case snippet

```
@Test
public void testExtractHistoricEFFTweetsByHashtag(){
    //Given
    String hashtag = "EFFmanifesto";
    WebTarget target = this.client.target(extractHistoryTweetsByHashtagUrl);
    log.severe(target.getUri().toString());

    //When
    Response response = target.queryParam("hashtag",hashtag)
                              .queryParam(COUNT, 10)
                              .queryParam(SINCE, EFF_MANIFESTO_LAUNCH_DATE)
                              .queryParam(UNTIL, THIRDY_DAYS_AFTER_EFF_MANIFESTO_LAUNCH)
                              .request(MediaType.APPLICATION_JSON)
                              .accept(MediaType.APPLICATION_JSON)
                              .get();

    //Then
    assertThat(response.getStatus(), is(200));
    JsonArray tweets = response.readEntity(JsonArray.class);
    assertTrue(tweets.size()>0);
}
```

The test case was written for the use case "*Extract tweets by hash tags using start and end date*". It tests a REST endpoint that takes a hashtag, start date, end date and count parameters. After execution, the HTTP status from the REST service response should be 200, and the returned response payload should contain at least one tweet.

All test cases followed this pattern:

Given: the test input is specified

When: the code execution takes place

Then: the desired results expected are asserted

An extract of test result log can be found on Appendix C. Each test case the endpoint URL under test is logged and the name of the test case is logged as well. At the end a summary of results is logged, showing that all 7 test cases passed.

Recommendations

The API can be abstracted further; a class such as Tweet can be re-modelled to a more abstract definition to cater for other social media platforms. At the moment, it is too specific Twitter. Similarly the interfaces and can be made more abstract; the parameters such as “hashtag” is also specific to Twitter as well, perhaps “topic” may represent a better abstraction.

For the usage rates limitations, caching can be implemented to alleviate the problem by reducing number of requests to the Google API. With caching if a particular location has been looked up before and cached then subsequent requests for coordinates of the same location can be retrieved from the local cache instead calling Google API repeatedly.

When a location with the same exists in many places (for example, Doornfontein exists in Johannesburg and Bakenburg in Limpopo), the Google API returns all the places and the system picks the first location. A more intelligent algorithm maybe applied to better pick the correct location, perhaps the system can deduce some context from the tweet text and do better pick.

Conclusion

Java EE as the implementation platform for data sourcing proved to work well without issues. Test driven development used together with Use case Driven Development works seamlessly. Lots of learnings has been done throughout the project, including using the NoSQL database, interacting with public APIs and the constraints they provide and also dealing with un-structured data (missing locations, etc.) that prevails in Big Data. Free Open Source Software and tools proved to be reliable toolset for implementing the extraction of Big Data.

References

[1] 2016 ELEN 7046 Group 2, "Big Data Visualization using Commodity Hardware and Open Source Software," Wits University, Johannesburg, 2016.

[2] H. Makabee, "Divide-and-Conquer: Coping with Complexity," Effective Software Design, 2011. [Online]. Available: https://effectivesoftwaredesign.com/2011/06/06/divide-and-conquer-coping-with-complexity/. [Accessed 28 June 2016].

[3] Twitter, "Rate Limits: Chart," Twitter, 2016. [Online]. Available: https://dev.twitter.com/rest/public/rate-limits. [Accessed 30 June 2016].

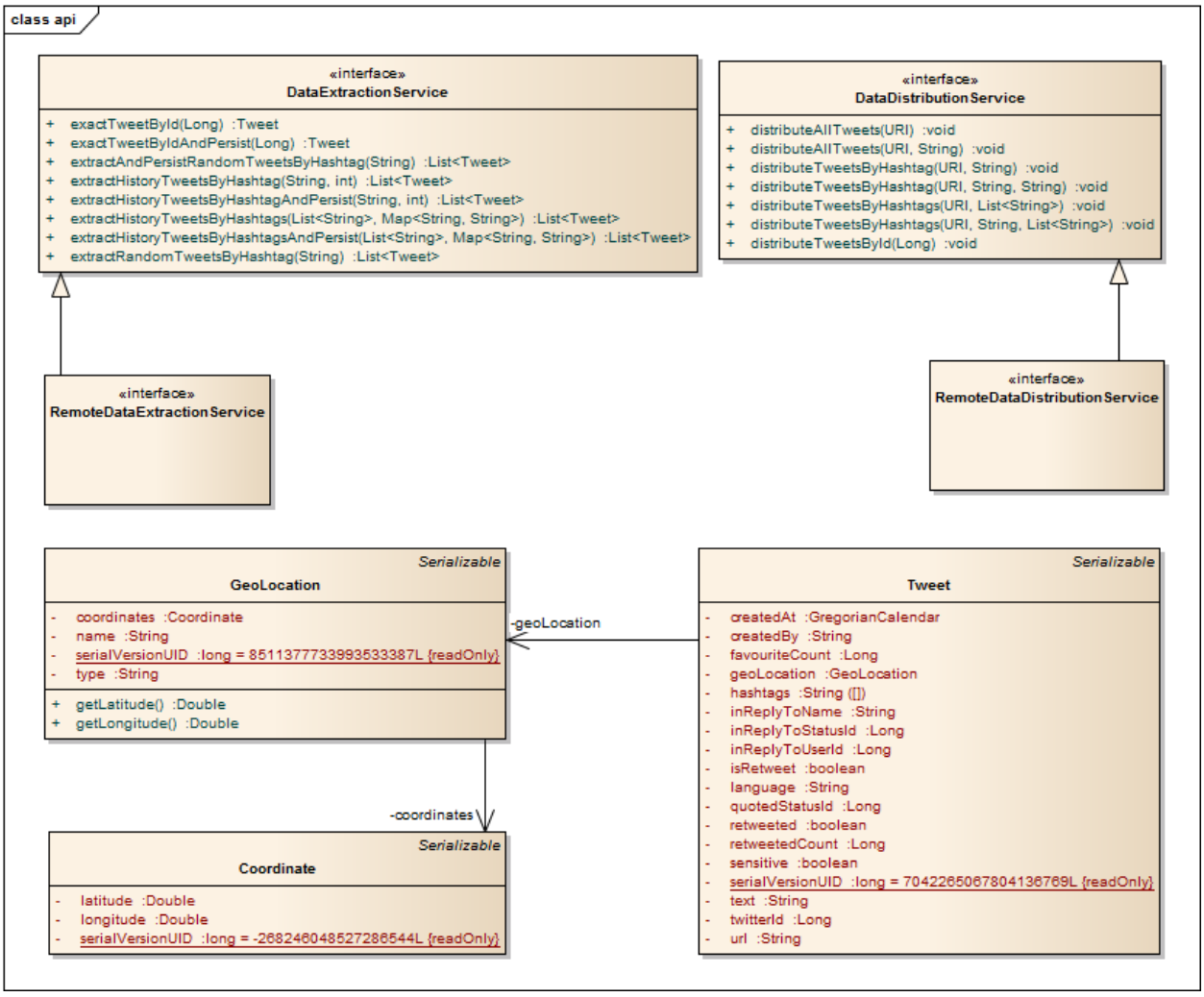
[4] Google, "Google Maps Geocoding API Usage Limits," Google Inc, 2016. [Online]. Available: https://developers.google.com/maps/documentation/geocoding/usage-limits. [Accessed 29 June 2016].

[5] Oracle, "Java Platform, Enterprise Edition: The Java EE Tutorial," Oracle, 2014. [Online]. Available: https://docs.oracle.com/javaee/7/tutorial/overview002.htm. [Accessed 1 July 2016].

Appendices

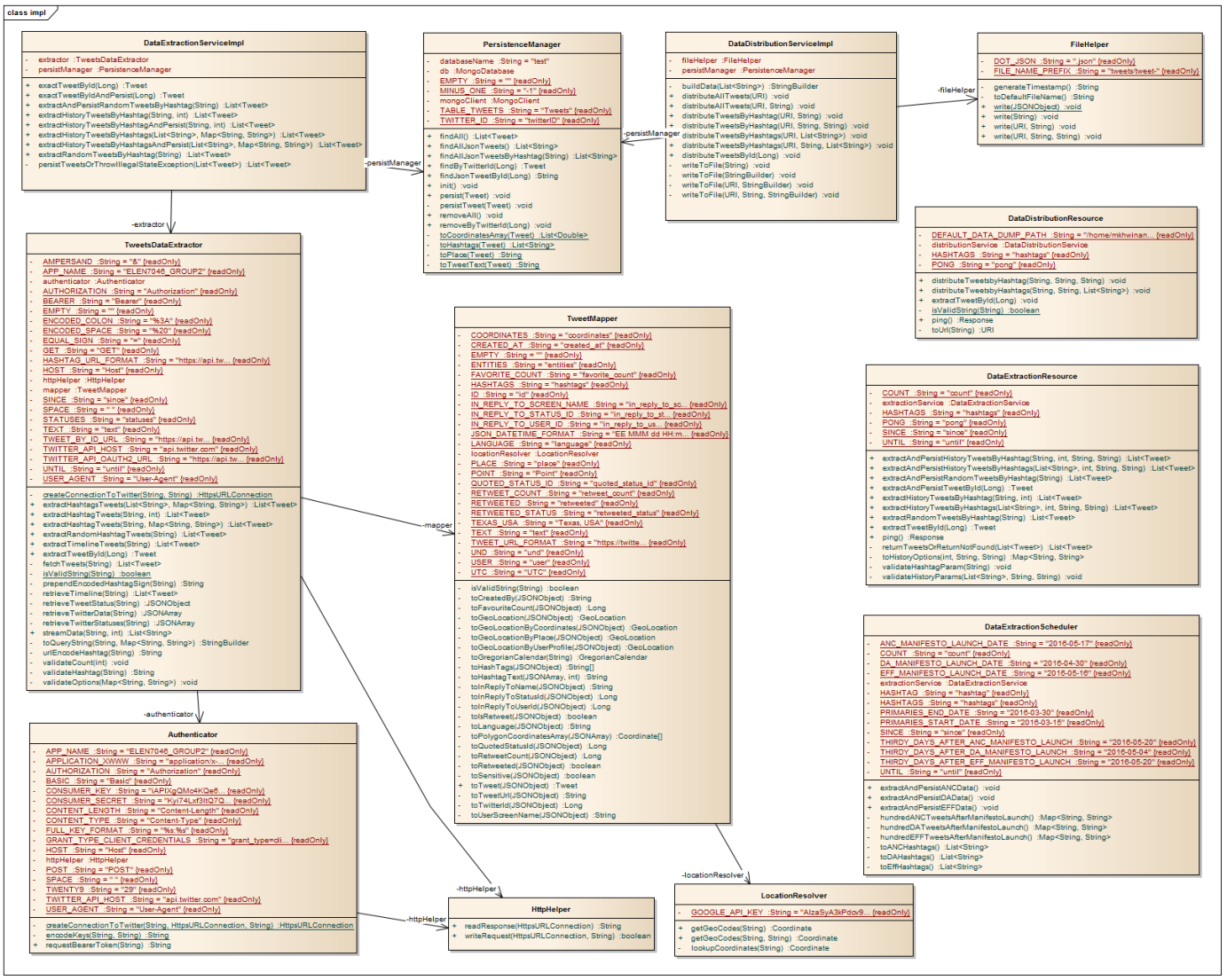
Appendix A

Figure 2: API class diagram



Appendix B

Figure 3: Implementation class diagram



Appendix C

Figure 4: Test results

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest
Jul 02, 2016 6:56:57 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest testPing
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-data
Jul 02, 2016 6:56:57 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest
testExtractHistoricTrumpTweetsDuringPrimaries
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-data/extractAndPersistHistoryByHashtags
Jul 02, 2016 6:57:01 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest testExtractTweetById
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-data/byId/729674502339055617
Jul 02, 2016 6:57:03 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest testExtractTweetById
INFO: {"twitterId":729674502339055617,"text":"RT @DLin71: Transcript of fascinating Trump interview on “Meet the Press”
https://t.co/IdaSdR4Exx","createdBy":"Neverevernever,ever!","createdAt":1462802932000,"favouriteCount":0,"hashtags":nul
l,"inReplyToName":"",
"inReplyToStatusId":-1,"inReplyToUserId":-1,"quotedStatusId":-
1,"isRetweet":true,"retweeted":false,"retweetedCount":916,"language":"",
"sensitive":false,"url":"https://twitter.com/leilanitexas/status/729674502339055617","geoLocation":{"coordinates":{"lat
itude":31.9685988,"longitude":-99.9018131},"type":"Point","name":" Mugwump, TX"}}
Jul 02, 2016 6:57:03 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest
testExtractAndStoreHistoricTrumpTweetsDuringPrimaries
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-data/historyByHashtags
Jul 02, 2016 6:57:07 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest
testExtractHistoricEFFTweetsByHashtag
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-data/historyByHashtag
Jul 02, 2016 6:57:10 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest testExtractAndStoreTweetById
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-data/byIdAndPersist/729674502339055617
Jul 02, 2016 6:57:12 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest testExtractAndStoreTweetById
INFO: {"twitterId":729674502339055617,"text":"RT @DLin71: Transcript of fascinating Trump interview on “Meet the Press”
https://t.co/IdaSdR4Exx","createdBy":"Neverevernever,ever!","createdAt":1462802932000,"favouriteCount":0,"hashtags":nul
l,"inReplyToName":"",
"inReplyToStatusId":-1,"inReplyToUserId":-1,"quotedStatusId":-
1,"isRetweet":true,"retweeted":false,"retweetedCount":916,"language":"",
"sensitive":false,"url":"https://twitter.com/leilanitexas/status/729674502339055617","geoLocation":{"coordinates":{"lat
itude":31.9685988,"longitude":-99.9018131},"type":"Point","name":" Mugwump, TX"}}
Jul 02, 2016 6:57:12 PM za.ac.wits.cpd.service.twitconpro.test.DataExtractionServiceTest
testExtractAndPersistHistoricEFFTweetsByHashtag
INFO: http://localhost:8080/historic-data-extraction-service/rest/extract-
data/extractAndPersistHistoryByHashtag?hashtag=EFFmanifesto&count=10&since=2016-04-30&until=2016-05-30
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 19.267 sec
```