UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Classe n. 35/S - Classe delle lauree specialistiche in ingegneria dell'informazione

MOOGLE: A WEB CRAWLER AND FULL-TEXT SEARCH APPLICATION FOR DISTRIBUTED PRIVATE DATA

Relatore:

Chiar.mo Prof. Stefano Paraboschi

Tesi di Laurea Specialistica

Paolo COFFETTI

Matricola n. 42519

ANNO ACCADEMICO 2013/2014

Abstract

You want to search for that sushi restaurant someone recommended you last month but you don't remember its name: you type "sushi restaurant" in your laptop and you get an old tweet from a friend talking about Tokyo Sushi. You also get a comment you wrote on Facebook, an SMS message you sent your brother and a bookmark in your browser, all about the same restaurant. And imagine that you can do this with your smart phone, your laptop, tablet or smart TV. Something so basic yet so far from the reality. This is ***Moogle*** *- My Own Google, the search engine for private data.*

This thesis describes the work I accomplished in order to design, develop and deploy a first solid version of Moogle following the iterative and incremental software development process outlined by Grady Booch.

Acknowledgments

First and foremost I'd like to thank my parents and sisters for their patience and the support they provided me during my studies.

I also want to express my gratitude to Professor Stefano Paraboschi for his precious guidance.

I wish to thank my girlfriend Boyang for her loving support and who, once again, had to deal with those long hours of my absence while working on this project.

A special acknowledgment goes to my dearest friends Marco Bontempi for his valuable and prompt technical advice and Simone Pilla for his daily backing.

Finally yet importantly, thanks to United Academics and especially to Louis Lapidaire for his frequent encouraging words and material support.

Table Of Contents

List Of Figures

CHAPTER 1

Introduction

A few years ago all our private data were stored in laptops: we used to manage emails with client programs like Mozilla Thunderbird and Microsoft Outlook, and to store our documents and photos in local folders.

But today much of our private data are stored in the shared web-based infrastructure known as "the cloud": we use web based email services like Google Gmail and Yahoo Mail; we store and share documents with services like Dropbox, Google Drive and Microsoft Skydrive; we share photos and interact with our friends in social networks like Facebook, Twitter and Instagram; we communicate using messaging applications like Google Talk, Facebook chat, Skype, SMS, Whatsapp; we have bookmarks in our laptops, in our office desktops and in our mobile phones.

Statistics prove the popularity of these services: 1.5 million emails are sent per second, 112 emails per day by average corporate user; Dropbox reached 100 millions users in Nov 2012; Facebook reaches 1 billion active users every month; in 2009 1 billion Facebook chat messages and 10 billions Whatsapp messages were sent every day.

Having our personal data in the cloud is handy but there is a drawback: the search is more complicated because the information is distributed over several services.

When a user wants to search for an address and he doesn't remember if he read it in a file in his laptop, in an email or in a Facebook post, he might waste much time in

searching through different platforms. What he wants instead is a single place to query in order to get results coming from all his private information distributed in the cloud. This is Moogle - My Own Google.

Moogle is a web application which provides users with the ability to search against all their distributed private data being them emails, documents, posts in social networks, SMS and chat messages in mobile phones or bookmarks.

Furthermore Moogle performs full-text search. This means that the actual content of documents is searched and not only those metadata that describe resources. Advanced text analysis is also performed while indexing data, with well-known techniques like stop-words removal, lexical analysis, stemming, synonyms injections, etc. Notice that these features are not always available in the cloud: Dropbox, Twitter and messaging applications in mobile phones f.i. just offer keyword-based search.

Moogle is the result of a project I have been working on in the last years. The project together with a sketch of its business and marketing plans have been pitched in some venture capitals meetings in Washington D.C. and Amsterdam and it raised some interest. This thesis describes the work I have done to build a first version of Moogle.

The project was named Moogle - My Own Google because it is a highly suggestive name. Yet it is just a working name used only in this initial phase.

1.1. Purpose

This thesis describes the work I accomplished in order to design, develop and deploy a first solid version of Moogle.

Due to limitations of resources not all features could be implemented, f.i. data from mobile phones are not fetched, but the entire set of the most relevant features were realized.

One of the most important aspect considered while designing the system was the ability to *evolve*: for this reason the software architecture uses extensively abstraction and inheritance such that f.i. new libraries could be easily developed to interact with services like LinkedIn or platforms like mobile phones.

As an academic project I paid particular attention to the software development process. During my studies I've always been interested in the work of Grady Booch, an American software engineer best known for his innovative work in software architecture and for developing the Unified Modeling Language. In the book *"Object-oriented analysis and design with applications"* (Addison-Wesley, 2007) he describes a software development process based on two perspectives: "the overall software development lifecycle (the *macro process*) and the analysis and design process (the *micro process*)". This is the discipline I followed during this work together with a broad usage of UML to document the main architectural decisions.

CHAPTER 2

Requirements

The system's requirements are presented by a summary of the vision document, key features, use case models and specifications.

2.1. Vision And Key Features

The vision for this project can be summarized easily.

Moogle will provide users with the capability to perform full-text search through their own private data hosted at service providers like Google Gmail, Google Drive, Facebook, Twitter and Dropbox.

This objective will be met only if the system developed is easy to use, intuitive, intelligent and the privacy of data is guaranteed.

The system will provide the following key features:

- Allows users to add and remove service providers;
- Fetches and indexes data from service providers on behalf of users and on a regular basis;
- Provides full-text search through the indexed data;

- Lets users navigate from search results directly to the original data hosted at service providers;
- Uses a web interface;
- Is implemented and deployed in a way that it can scale up easily.

2.2. Use Case Analysis

The practice of use case analysis was first formalized by Jacobson, I. in 1994. He defines a use case as *"a behaviourally related sequence of transactions performed by an actor in a dialogue with the system to provide some measurable value to the actor"* - The Object Advantage: Business Process Reengineering with Object Technology, Jacobson at al., 1994.

A use case is often a list of steps, typically defining interactions between an actor and a system, to achieve a goal.

"A use case is not a description of a single functional requirement but rather a description of something that provides significant value to the actor invoking it, in the form of scenarios." - Object Oriented Analysis and Design with Applications, Booch et al., Addison-Wesley, 2007.

The system contains the following **actors**:

- **User**: the main user of this system; he uses Moogle in order to search his own private data;
- **System Admin**: a role responsible for maintaining Moogle's application keys registered at providers.

The main **use cases** are as follows.

- *Manage Connected Providers*: add or remove providers from which Moogle will fetch and index data;
- *Search*: perform a search and view search results;
- *Manage Moogle's App Keys*: add or remove application keys for providers.

Use case specifications are discussed in the next sections.

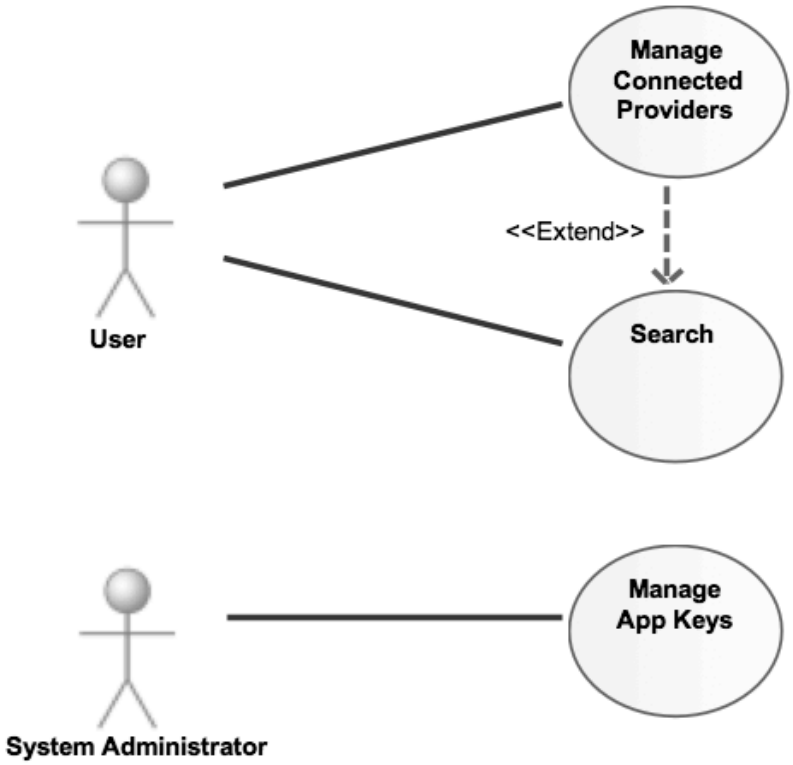

Figure 2-A Main Use Cases

2.2.1. Manage Connected Providers

Use case name: manage connected providers.

Actor: user.

Goal: the user wishes to add a new data provider such that Moogle can index his own data hosted at that provider. F.i. if the provider is Facebook, the user wants to add Facebook to his connected providers so that Moogle will fetch and index all posts in his Facebook account.

Preconditions: the user is authenticated.

Main flow:

1. The user selects a link to his dashboard.

2. Moogle looks up in the database what the available providers are and what have already been added by the user. Moogle shows the group of providers currently in use and the group of providers that can be added.
3. The user selects one of the providers that can be added.
4. Moogle redirects the user to the authorization page hosted at the provider's website. In that page the provider asks the user to authorize Moogle to access his own data.
5. The user authorizes Moogle to access his own data.
6. The user is automatically redirected back to his dashboard page in Moogle's website. Moogle displays a success message and adds the provider to the group of providers currently in use. This use case ends.

Alternate flow: all providers already added.
Condition triggering an alternate scenario: all available providers have already been added by the user.
Flow:

2. The group of providers that can be added is empty, because all available providers have already been added. This use case ends.

Alternate flow: remove provider.
Goal: the user wishes to remove a data provider such that Moogle quits indexing his own data hosted at that provider. Also, user wants the data that Moogle indexed from that provider (on the behalf of the user) to be removed from Moogle.
Condition triggering an alternate scenario: the user does not select one of the providers from the group of providers that can be added. Instead, he selects one from the group of providers currently in use.
Flow:

3. The user selects one of the providers that have already been added.

3.1. Moogle informs the user that the provider will be removed together with all the indexed data.
3.2. The user confirms.
3.3. Moogle disconnects the provider and resets the user's index for that provider.

3.4. Moogle displays a success message. This use case ends.

Alternate flow: authorization denied.
Condition triggering an alternate scenario: the user denies the authorization required by Moogle to access his data.
Flow:

5. The user chooses not to authorize Moogle to access his own data.

5.1. The user is automatically redirected back to his dashboard page in Moogle's website. Moogle displays a failure message. This use case ends.

2.2.2. Search

This use case by far is the most frequently invoked. As a result, it is critical to implement this use case effectively and to ensure that it meets all of the overall design goals, including the non-functional requirements.

Use case name: search.
Actor: user.
Goal: the user wishes to perform a search through his private data.
Preconditions: the user is authenticated.
Main flow:

1. The user selects a link to Moogle home page.
2. Moogle shows a page with a text input box, icons of all added providers and a "+" symbol if not all available providers have been added. All added providers are enabled by default. The user clicks on a provider in order to temporary disable the search on that provider.
3. The user types the search query in the text input box and clicks on the search button.
4. Moogle runs the query against all selected providers and displays a result page. Results are grouped by providers. Each result contains a snippet of the original

main text content and a link that leads to the original resource in the provider's website.

5. The user can navigate the resultset using navigation buttons. He can also go back to the step #1 by selecting a `back` link. When he is satisfied with the search results, this use case ends.

Extension point: add a provider.

Condition triggering an alternate scenario: the user clicks the "+" symbol in the search page.

Flow:

2. Moogle shows a page with a text input box, icons of all added providers and a "+" symbol if not all available providers have been added.

2.1. The user clicks on the "+" symbol. The scenario *Manage Connected Providers* is executed. The primary scenario is resumed from step #1.

2.2.3. Manage Moogle's App Keys

Use case name: manage Moogle's app key.

Actor: system admin.

Goal: the system admin wishes to change/add/remove Moogle's app keys. An app key is the credential got from a provider when registering Moogle as an application able to interact with the provider's API.

Preconditions: the system admin has registered Moogle as an app that can use a provider's API. The system admin is authenticated.

Main flow:

1. The system admin selects a link to Moogle's backend website.
2. Moogle checks whether the system admin has the right permissions to access the backend website. If so Moogle displays a link to the "Providers" page.
3. The system admin clicks the link to the "Providers" page. Moogle displays a list of all available providers and for each of them a link to the "Provider Details" page.

4. The system admin clicks the link to a "Provider Details" page. Moogle displays a form with some information about the provider and the related app keys.
5. The system admin edits/adds/removes keys and clicks the save button.
6. Moogle updates the keys and displays a success message. This use case ends.

2.3. Non-functional Requirements

The use cases establish the basic functional requirements for Moogle, that is, they tell what the system must do for its users. In addition, we have non-functional requirements and constraints that impact the requirements specified by our use cases, as listed here.

Non-functional requirements:

- The search functionality must be provided using a user friendly web interface, with minimalist design as used in many popular web search engines like Google and Bing;
- Privacy of data must be guaranteed;
- The search application must provide results within a few seconds (max 5 seconds);
- The system must have a designed-in capability to evolve and work with new service providers and new platforms like mobile phones.

CHAPTER 3

Software Development Process

"A successful software project is one in which the deliverables satisfy and possibly exceed the customer's expectations, the development occurred in a timely and economical fashion, and the result is resilient to change and adaptation. By this measure, we have observed several traits that are common to virtually all of the successful object-oriented systems we have Grady Booch encountered and noticeably absent from the ones that we count as failures: existence of a strong architectural vision; application of a well-managed iterative and incremental development lifecycle" - Object-Oriented Analysis and Design with Applications, Booch et al., Addison Wesley, 2007.

Designing a sound architecture and following an iterative and incremental software development cycle are the principles I followed while working on this project. In particular I found very helpful the methodology described by Booch at al in the book titled *"Object-Oriented Analysis and Design with Applications"* (Addison Wesley, 2007).

Iterative and incremental development is where the functionality of the system is developed in a successive series of releases (*iterative*) of increasing completeness (*incremental*). The selection of what functionality is developed in each iteration is driven by the mitigation of project risks, with the most critical risks being addressed first. The experience and results gained in one iteration are applied to the next one. With each iteration strategic and tactical decisions are gradually refined, ultimately

converging on a solution that meets the requirements and yet is simple, reliable, and adaptable. The iterative and incremental approach is at the heart of most modern software development methods, including agile methods like Extreme Programming and SCRUM.

Booch's approach to software development uses two perspectives: the **macro process** which is the overall software development lifecycle and the **micro process** which is the analysis and design step.

A key point in the definition of the macro and micro processes is the strong separation of concerns between them. While the choice of lifecycle style (waterfall, iterative, agile, plan-driven, ...) affects the macro process, the choice of analysis and design techniques (structured, object-oriented, ...) affects the micro process.

The macro process involves the following disciplines, executed in the following relative order:

- **Requirements**: establish agreement with the stakeholders on what the system should do and define the boundaries of the system;
- **Analysis and design**: transform the requirements into a design of the system; this includes evolving a robust architecture for the system and establishing the common mechanisms that must be used by disparate elements of the system;
- **Implementation**: implement and integrate the design, resulting in an executable system;
- **Test**: test the implementation to make sure that it fulfills the requirements;
- **Deployment**: ensure that the software product is available for its end users.

Plus there are three more disciplines executed throughout the entire lifecycle:

- **Project management**: manage the software development project: planning, staffing, monitoring and managing the risks;

- **Configuration and change management**: identify the configuration items, and control changes to those items;

- **Environment**: provide the software development environment to support the development team.

These disciplines are well-known in the classic software engineering world and most of the results they lead to are described in the chapter 4. The remainder of this chapter instead, describes what Booch calls milestones and phases for the macro process and the way I applied them.

3.1. Macro Process: Phases And Milestones

In an iterative and incremental macro process, the disciplines are repeated until a specific milestone is reached. A milestone defines objectives and ensures that the iterations make progress and converge on a solution, rather than iterating indefinitely.

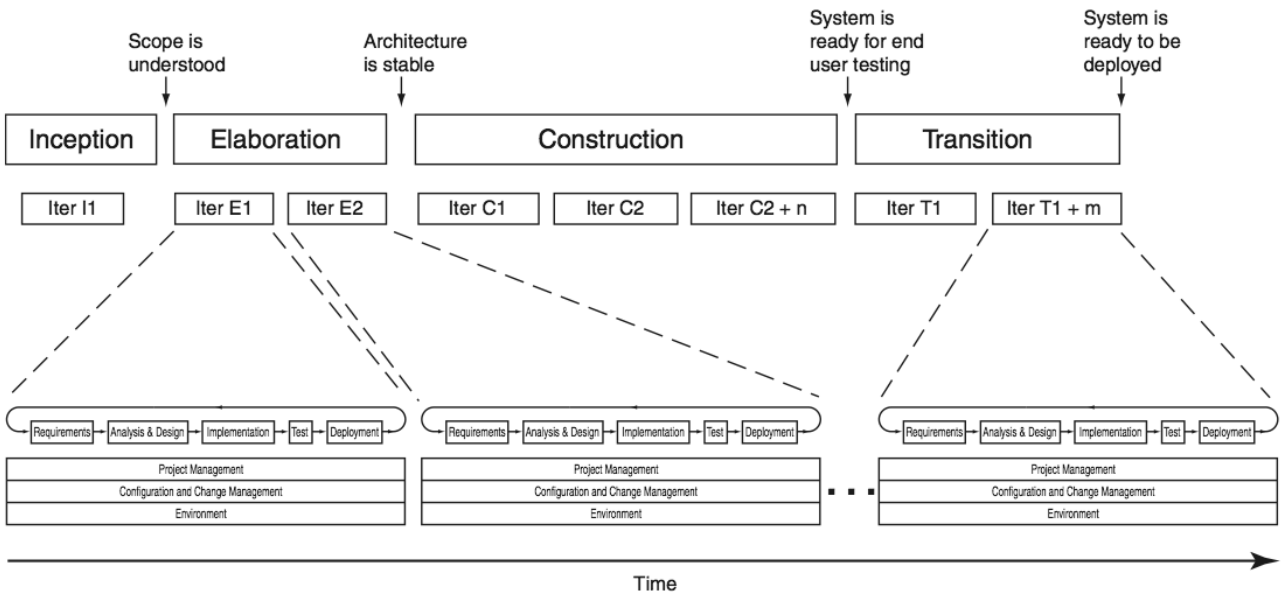

Figure 3-A Macro process, phases and milestones [source: Object-Oriented Analysis and Design with Applications, Booch et al., Addison Wesley, 2007]

The following sections describe each phase in detail.

1. Inception

The purpose is to ensure that the project is both valuable and feasible. During this phase the initial idea behind the entire project is not completely defined, but the main vision and its assumptions are validated. The main focus is on ensuring the technical feasibility of the project itself. The core requirements of the system are defined and the key risks are well understood.

The milestone is: "scope is understood".

Activities:

- I formalized my initial idea into a set of high-level requirements;
- I verified my assumptions, in particular:
    - the lack of full-text search on many providers;
    - the possibility to harvest private textual data from providers using their APIs.

2. Elaboration

The main focus is on the overall architecture that provides the foundation for all the iterations that follow. The intent is to identify architectural flaws early and to establish common policies that yield a simpler and more stable architecture.

The milestone is: "architecture is stable".

Activities:

- I sketched the overall architecture and refined it within a few iterations; this was the time when the two main components (Moogle Website and Magpie Web Crawler) were conceived;
- a couple of initial releases were created in order to partially satisfy the scenarios;
- many of the tools required to interact with provider's API and with subsystems like PostgreSQL, Redis and Solr were identified and experimented.

3. Construction

The construction phase is when the development moves from discovery into production, where production can be thought of as "a controlled methodological process of raising product quality to the point where the product can be shipped". The development is completed through a series of releases which are constantly instrumented, studied and evaluated as they incrementally grow end evolve to the actual production system.

The milestone is: "system is ready for end-user testing".

Activities:

- the development of the system was completed based on the baselined architecture produced during the previous phases;
- several releases were produced and tested in order to discover and fix defects.

4. Transition

During the Transition phase, the product is provided to the user community for evaluation and end-user testing. The feedback received is then used to improve the system. Configuration, installation, and usability issues are addressed during this phase.

The milestone is: "system is ready to be deployed".

Activities:

- I performed a end-user functional testing and fine-tuned the product;
- I focused mainly on the Magpie component and its web interface and on the text analysis performed by Solr.

CHAPTER 4

Architecture

Software architecture is the *"fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"* - IEEE Recommended Practice for Architectural Description of Software Intensive Systems (IEEE 1471).

The system is composed of two main logical processes: a batch task which periodically fetches and indexes data from providers and an interactive web application to provide users with full-text search. These two processes are implemented by **Magpie Web Crawler** and **Moogle Website**, respectively. These components are independent and completely de-coupled.

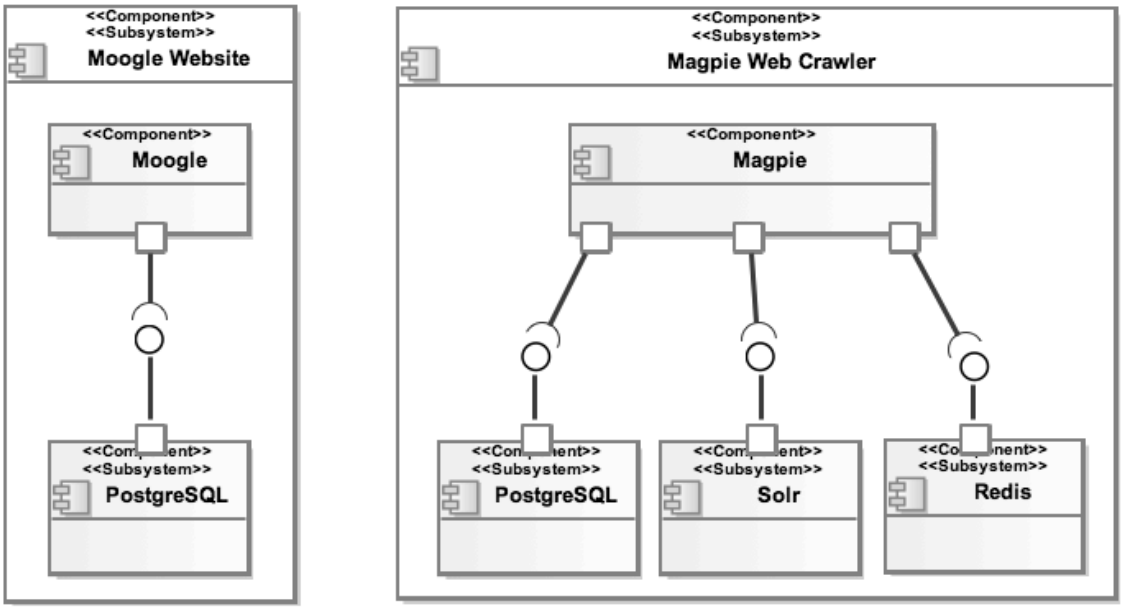

Figure 4-A Main components of the system

The next sections describe the internal structure of these components.

4.1. 4+1 Architectural View Model

The system architecture is described following the "4+1 architectural view model" by Philippe Kruchten, but with the few adaptations introduced by Booch et al. in the book *"Object-Oriented Analysis and Design with Applications"* - Addison Wesley, 2007.

"It describes the architecture of software-intensive systems, based on the use of multiple, concurrent views. The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are logical, development, process and physical view. In addition selected use cases or scenarios are utilized to illustrate the architecture serving as the 'plus one' view. Hence the model contains 4+1 views:
***Logical view**: the logical view is concerned with the functionality that the system provides to end-users[...].*
***Development view**: the development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view[...].*
***Process view**: the process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. [...].*
***Physical view**: the physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. This view is also known as the deployment view[...].*

***Scenarios**: the description of an architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as use case view."* - Wikipedia.

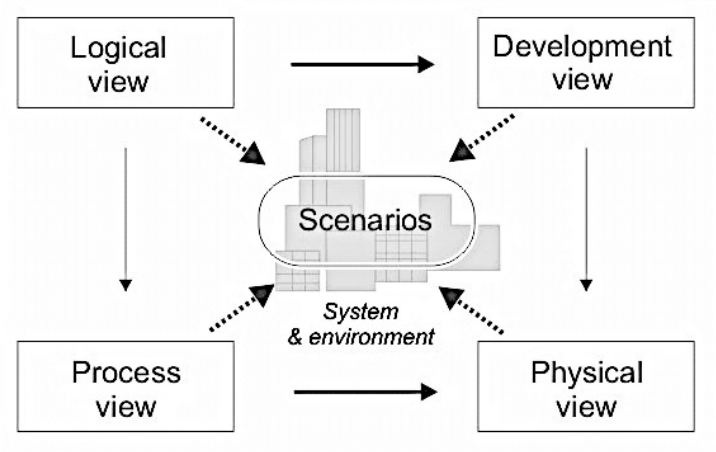

Figure 4-B 4+1 Architectural View Model [source: Wikipedia]

The following discussion is not meant to be a complete description of the system architecture, however all the most significant aspects are covered.

4.2. Magpie Web Crawler

Magpie Web Crawler is the largest and most interesting part of the entire system and it is responsible for crawling data providers and building a full-text index.

The architecture of Magpie is discussed in this chapter; as mentioned in the previous section, the *"4+1 architectural view model"* by Philippe Kruchten is used here, but with the few adaptations introduced by Booch et al. in the book "Object-Oriented Analysis and Design with Applications" - Addison Wesley, 2007.

The following discussion is not meant to be a complete description of Magpie's architecture, however all the most significant aspects are covered.

4.2.1. Logical View

Magpie is composed of three main components: crawler, downloader, indexer. The details of each component are described in the next sections.

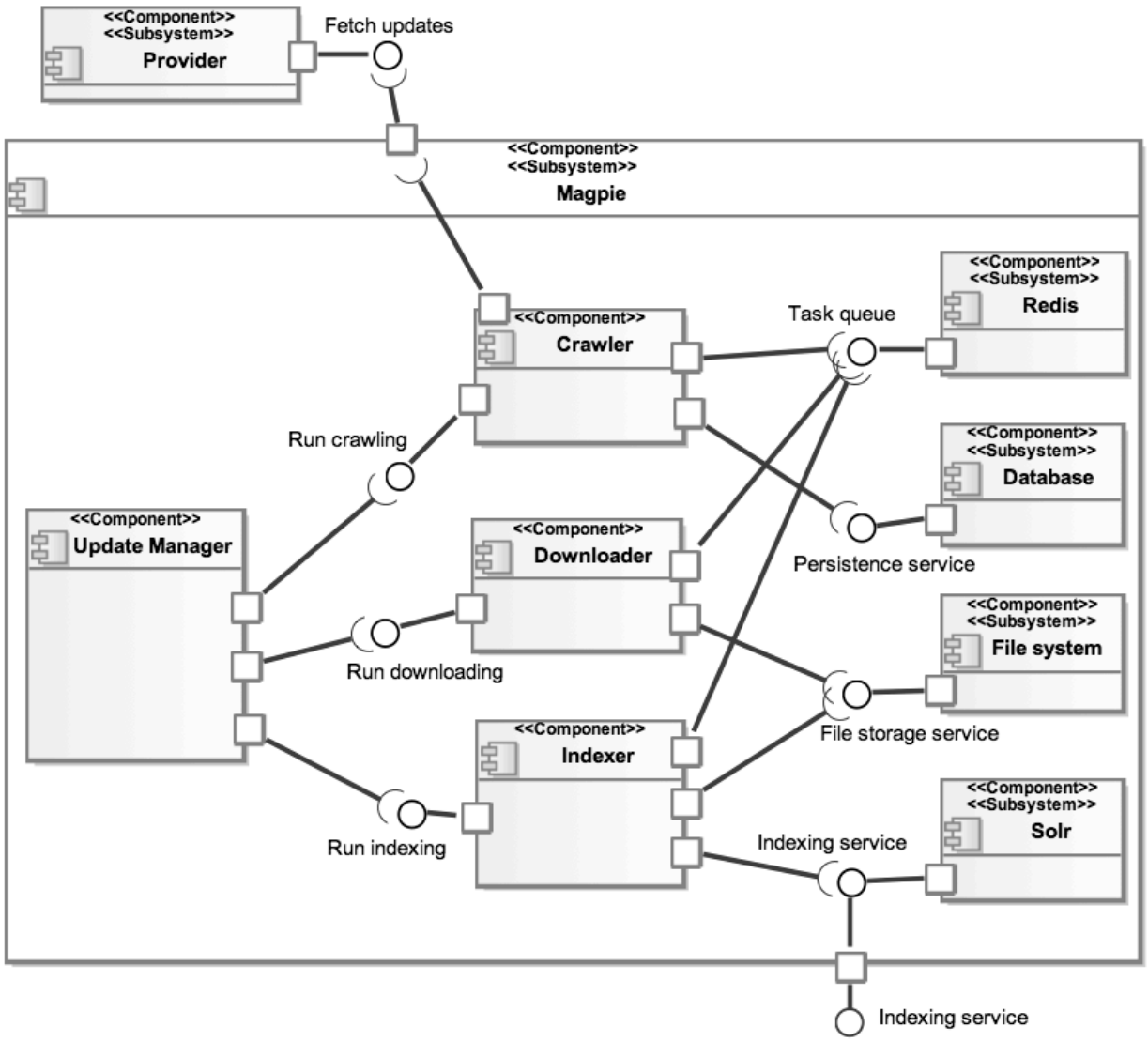

Figure 4-C Top-level component diagram for Magpie

Database Models

The main purpose of Magpie is to harvest textual private data from providers on behalf of users and create a full-text index. Thus a proper full-text engine (Apache Solr) is used to store private data, as discussed in the chapter 7. Full-Text Search Engine.

A classic relational database is used to store information about providers and OAuth access tokens. An object-relational mapping (ORM) is the interface between the crawler

and the actual DBMS engine. The chosen ORM is *SQLAlchemy* and it can work together with several DBMS engine like *PostgreSQL* and *MySQL*.

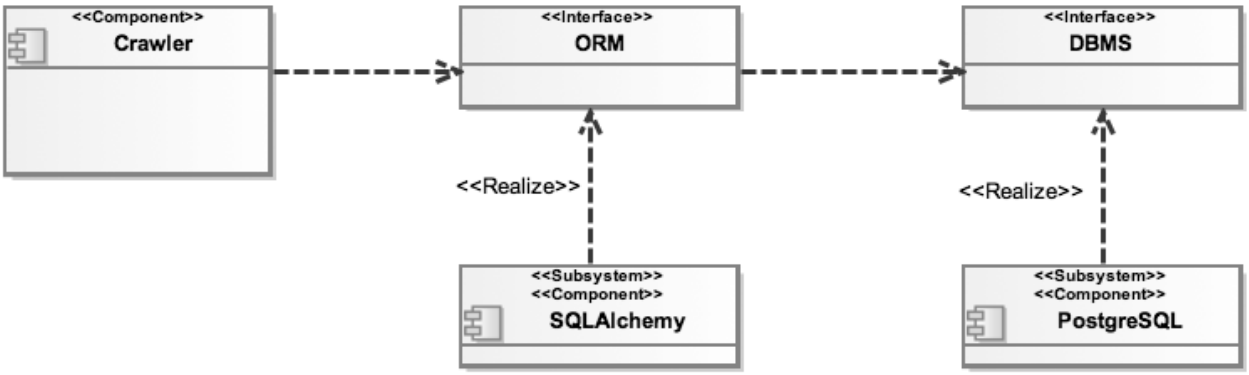

Figure 4-D SQLAlchemy as ORM interface to the database

The designed database models and their relationship are quite straightforward as shown in the next diagram.

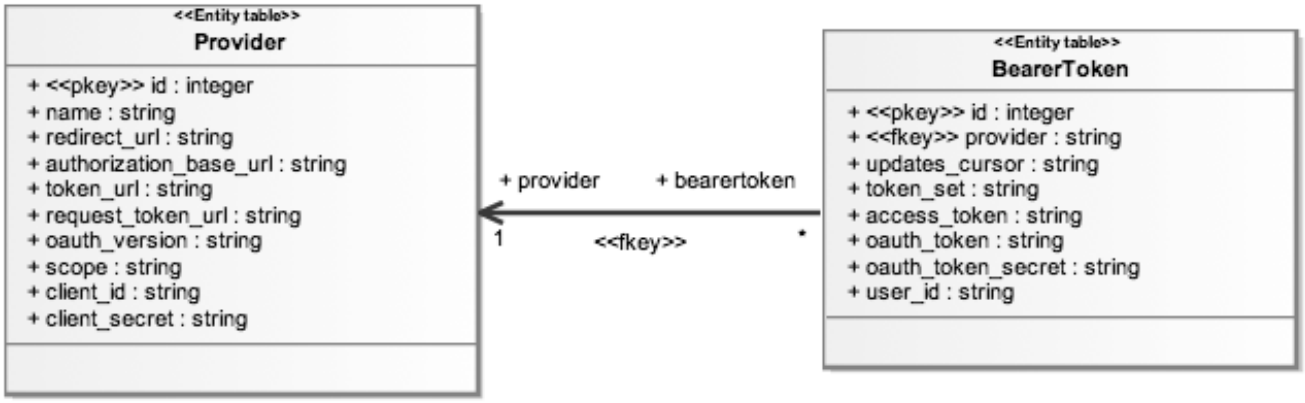

Figure 4-E ER class diagram

Two are the entities: Provider and BearerToken.

Provider

A provider is a web service which Magpie queries on behalf of users in order to collect private data. Magpie currently works with the following providers: Facebook, Twitter and Dropbox. While Google Gmail and Google Drive are queried directly by Moogle, as they offer their own integrated full-text search service.

A provider has an internal identifier, a verbose name and a collection of attributes (like URLs and application keys) required to perform queries on behalf of users.

BearerToken

A bearer token is a set of keys (usually an identifier and an access token) given by the provider at the end of the OAuth flow and meant to be used to authorize the access to some private data. So when a user connects a new provider and gives Moogle the authorization to access his own private data, Moogle receives a new bearer token from the provider and shares it with Magpie. A bearer token is all what Magpie needs in order to identify a user, this is the reason why there is no such a concept of user in Magpie. Also, dealing with bearer tokens rather than users, improves the overall privacy level. See chapter 5. Protocols And Data Entities for more information about bearer tokens.

Crawler

The crawler is the component in charge of querying providers and collect data. Many are the classes involved in this process as shown in the next diagram.

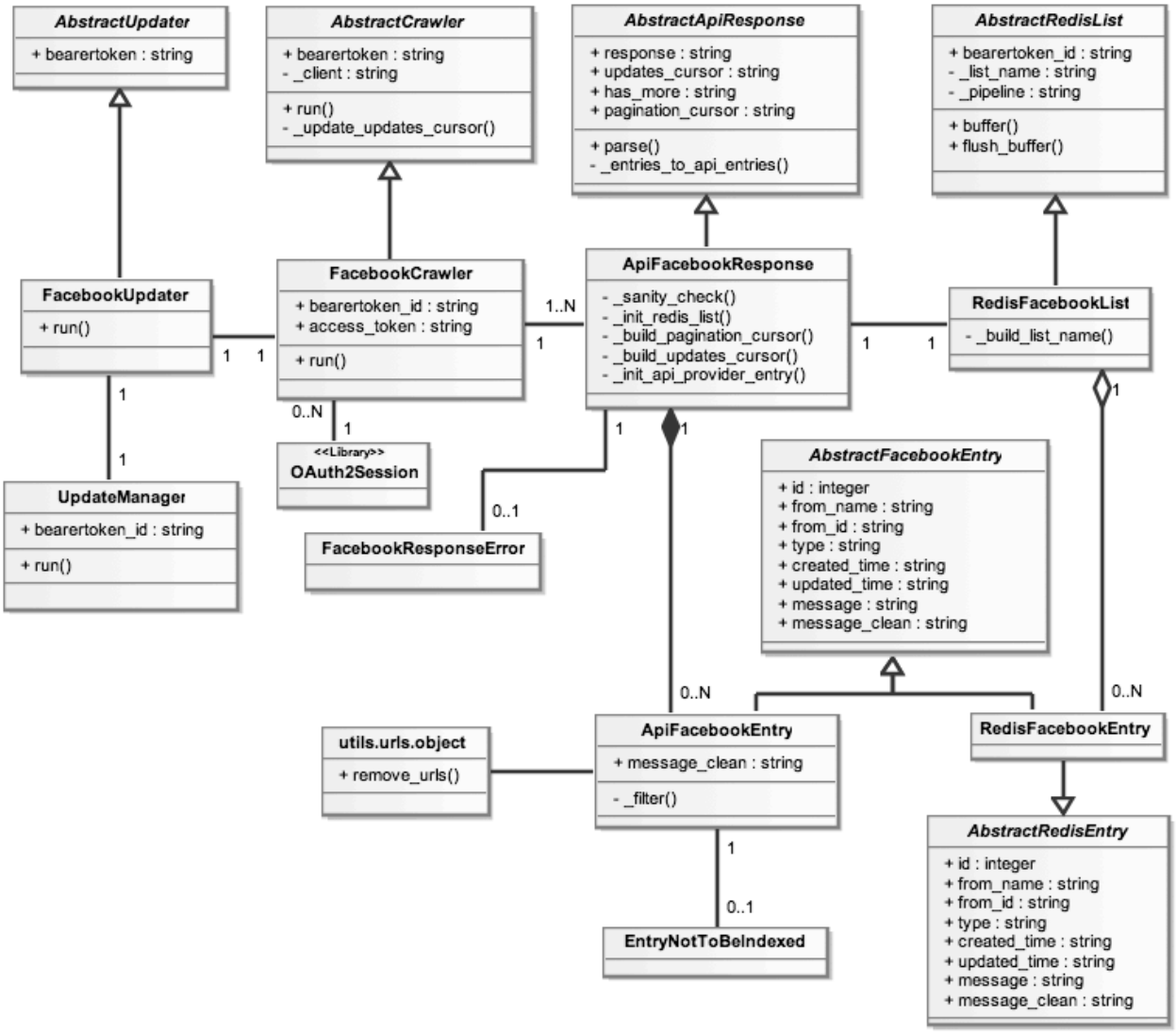

Figure 4-F Class diagram for the Facebook crawler component

FacebookUpdater

The update process is a complex operation involving many classes, so a *facade* pattern is the best solution in order to provide a simple interface. FacebookUpdater is the facade component which exposes a simple method *run()* and under the hood runs the FacebookCrawler first and the FacebookIndexer then.

FacebookCrawler

It is in charge of the communication with Facebook via Facebook API protocol (see chapter 5.2. Facebook API). It knows how to manage an OAuth session using a specific

library (requests-oauthlib) and a bearer token; it also initializes an ApiFacebookResponse instance with the response got from the provider.

ApiFacebookResponse

It is initialized with a raw response got from Facebook web service and knows how to parse it and to create an ApiFacebookEntry for each item included in the response. It also makes sure that the response is positive and syntactically correct by performing a sanity check and builds a pagination cursor used to navigate the resultset and an updates cursor to keep track of the most recent update fetched.

ApiFacebookEntry

A single Facebook post with all attributes identified in the chapter 5.2. Facebook API - Main Entities. It knows when the entry is not relevant and must be filtered out (when there is no textual content, f.i. the post is an image with no message or comments) and also how to create a new *message_clean* attribute by removing all URLs from the original message.

RedisFacebookList

It is an interface to Redis, used to queue tasks. It implements a FIFO queue whose items are Facebook posts which are pushed in the list during the crawling phase and extracted during the indexing phase. It knows how to create unique names for tasks (posts in this case) and task lists and it manages the communication with Redis using pipelines to optimize the performance. See chapter 6. Task Queue With Redis for more details about the Redis data structures in use (lists and hashes).

RedisFacebookEntry

A single task in the task queue. It is a simple data structure whose attributes are those identified in the chapter 5.2. Facebook API - Main Entities.

A crawler is designed to work only with a specific provider, so in Magpie there are three different crawler implementations: **FacebookCrawler**, **TwitterCrawler** and

DropboxCrawler. They have a similar behavior and expose the same interface, thus they have been designed as part of a hierarchical structure.

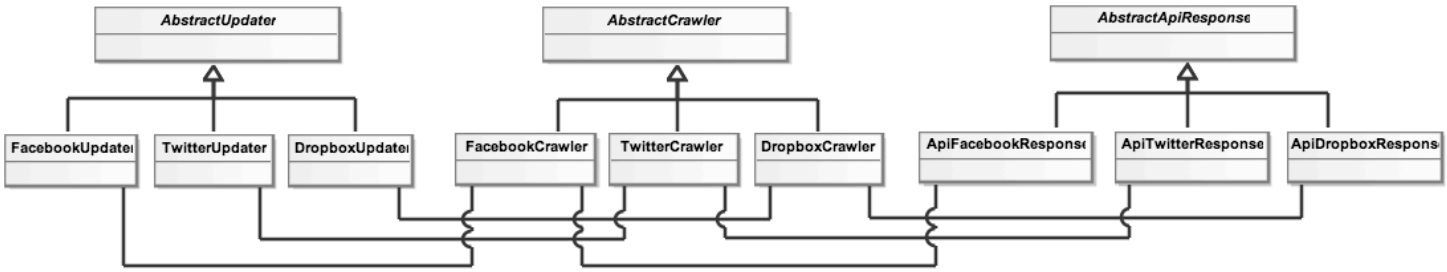

Figure 4-G Partial hierarchical structure of the different crawlers (AbstractRedisList, AbstractRedisEntry and their specialized classes have been removed)

Indexer

The indexer is the component in charge of indexing the data collected by the crawler. Many are the classes involved in this process as shown in the next diagram.

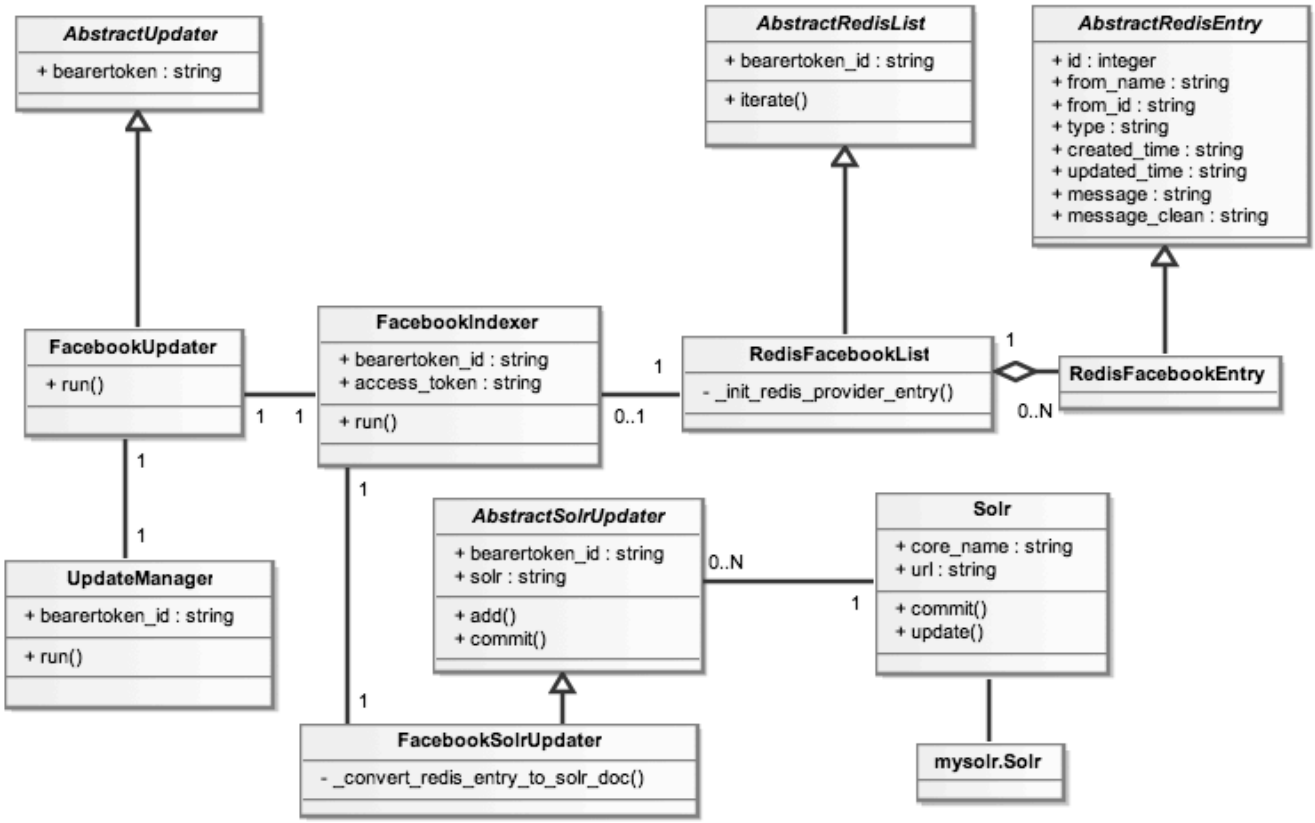

Figure 4-H Class diagram for the Facebook indexer component

FacebookUpdater

See previous Crawler section.

FacebookIndexer

Its responsibility is to extract tasks (Facebook posts) from a Redis list and send them to Solr in order to be text-analyzed and indexed.

RedisFacebookList

See previous Crawler section.

RedisFacebookEntry

See previous Crawler section.

FacebookSolrUpdater

It is an interface to Apache Solr, used to perform text-analysis and index Facebook posts. It uses the Solr class to manage the actual communication with the full-text search engine. It knows how to convert a Facebook post to a Solr document. See chapter 7. Full-Text Search Engine for more details about Solr, schema and documents format.

Solr

MySolr Python library is used to manage the actual operations with Apache Solr. This library lacks some important feature so the Solr class is a tiny wrapper around MySolr in a typical *decorator* pattern structure. It enriches some basic operations like *update* and *search* by performing a sanity check on the response, and adds new features like *add_file* (used by DropboxIndexer) and *delete_by_query*.

An indexer is designed to work only with a specific provider, so in Magpie there are three different indexer implementations: **FacebookIndexer**, **TwitterIndexer** and **DropboxIndexer**. They collaborate with three classes in charge of the communication with Solr. These classes, named **FacebookSolrUpdater**, **TwitterSolrUpdater** and

DropboxSolrUpdater, have a similar behavior and expose the same interface, thus they have been designed as part of a hierarchical structure.

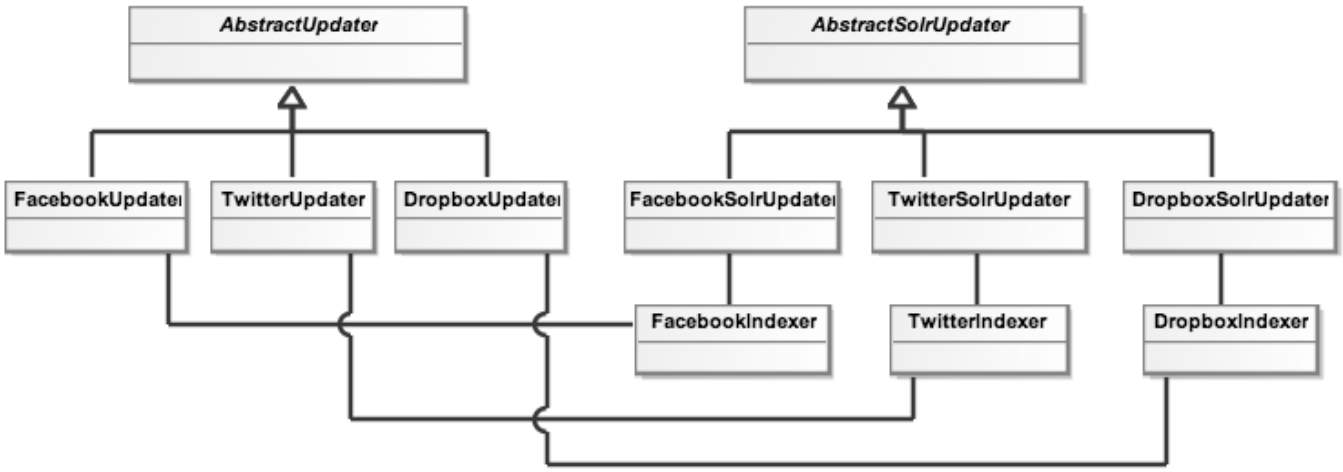

Figure 4-I Partial hierarchical structure of the different indexers (AbstractRedisList and its specialized classes have been removed from this diagram)

Downloader

Unlike the crawler and the indexer, the downloader is a component required only when dealing with data from Dropbox. Main entities in Dropbox are text files (see Appendix A. Entities) and the downloader is in charge of downloading these files and storing them to disk. Many are the classes involved in this process as shown in the next diagram.

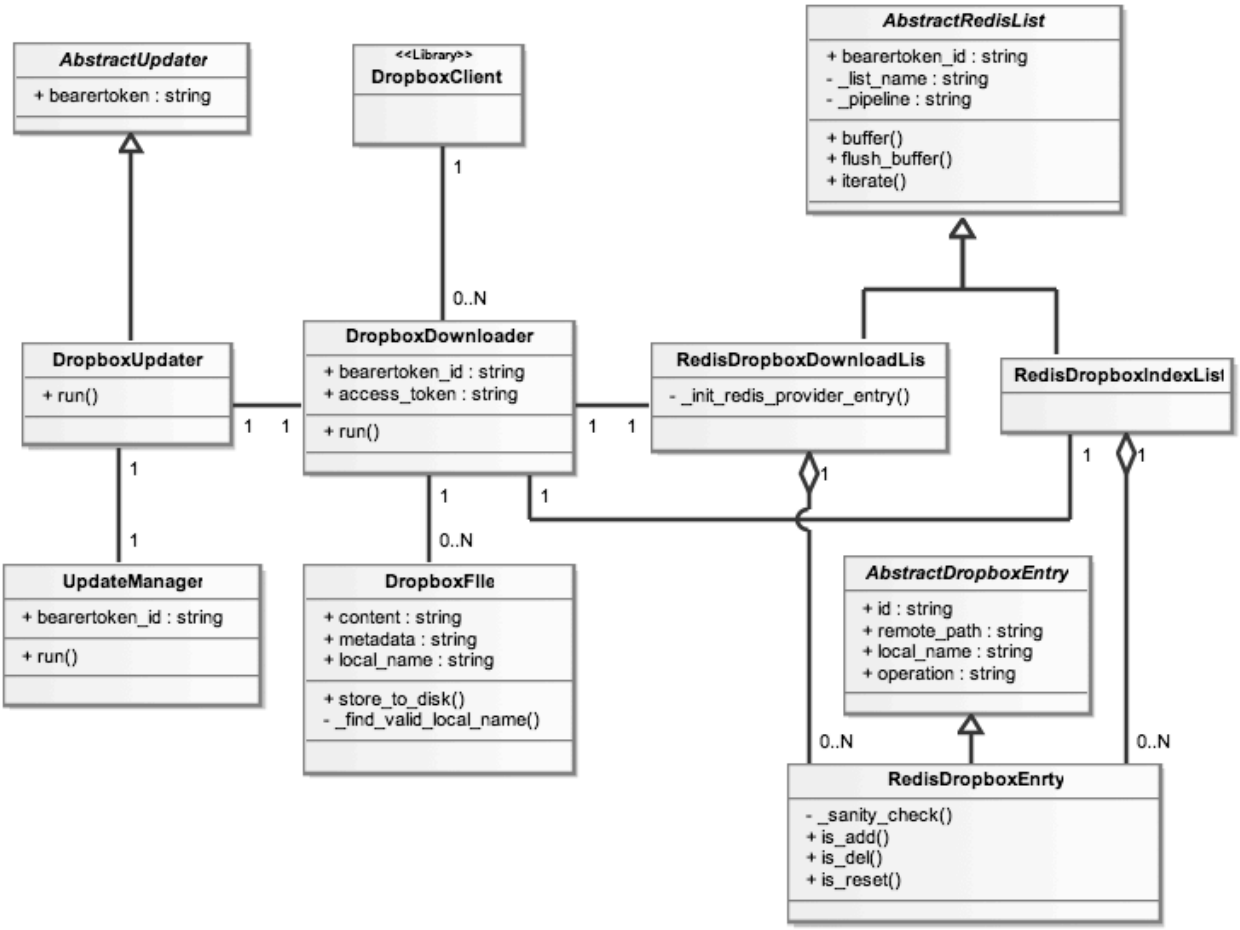

Figure 4-J Class diagram for the Dropbox downloader component

DropboxUpdater

See FacebookUpdater in the Crawler section.

DropboxDownloader

Its responsibility is to extract tasks (Dropbox files to be downloaded) from a RedisDropboxDownloadList, download them and push new tasks to RedisDropboxIndexList. It uses the official Python Dropbox library to manage the download operation.

DropboxFile

It models an actual text file in Dropbox; it is made of content and metadata, a set of attributes like size and path describing the file. It knows how to store a file to disk finding a valid unique name.

RedisDropboxDownloadList, RedisDropboxIndexList

They are interfaces to Redis, used to manage task queues. They implement FIFO queues whose items are Dropbox files to be downloaded first and indexed then. They know how to create unique names for tasks (files in this case) and task lists and they manage the communication with Redis using pipelines to optimize the performance. See chapter 6. Task Queue With Redis for more details about the Redis data structures in use (lists and hashes).

RedisDropboxEntry

See RedisFacebookEntry in the Crawler section.

4.2.2. Process View

There are mainly three categories of processes involved in Magpie: crawling, downloading and indexing. Each category includes three similar implementation of the same process, one for each provider. So f.i. there is a Facebook crawling process, a Twitter crawling process and a Dropbox crawling process. The only exception is the downloading process, which only exists for Dropbox.

This section describes the details of the most important tasks of such processes.

Crawling Process

Crawling is the process of running queries against providers and collect data.

The next figure shows the way FacebookCrawler is run in order to fetch the latest Facebook updates for a specific bearer token and push them in a task queue.

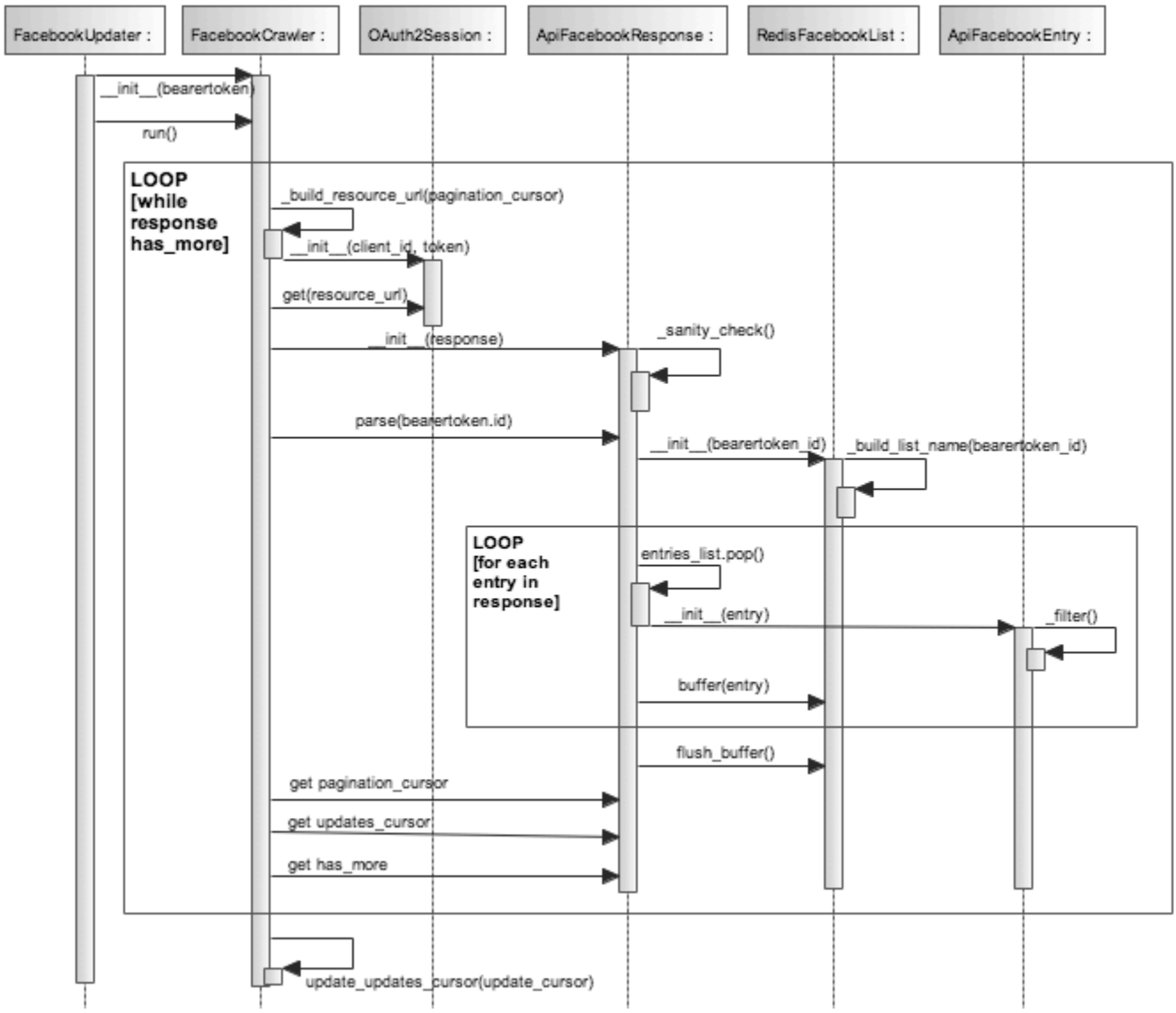

Figure 4-K Sequence diagram for the Facebook crawling process

FacebookCrawler opens an **OAuth2Session** to communicate with Facebook API and runs an update query. The response is used to initialize an **ApiFacebookResponse**, whose *parse* method takes care of scrolling all the entries (Facebook posts) included in the response, and to buffer them to the task queue in Redis. FacebookCrawler then reads the pagination cursor and the updates cursor from ApiFacebookResponse and in case it *has_more* results, a new update query is performed.

Indexing Process

Indexing is the process of sending textual data to a full-text search engine (Apache Solr) which performs text analysis and adds them to an inverted index.

The next figure shows the way FacebookIndexer is run in order to extract Facebook posts from a task queue for a specific bearer token and send them to Solr.

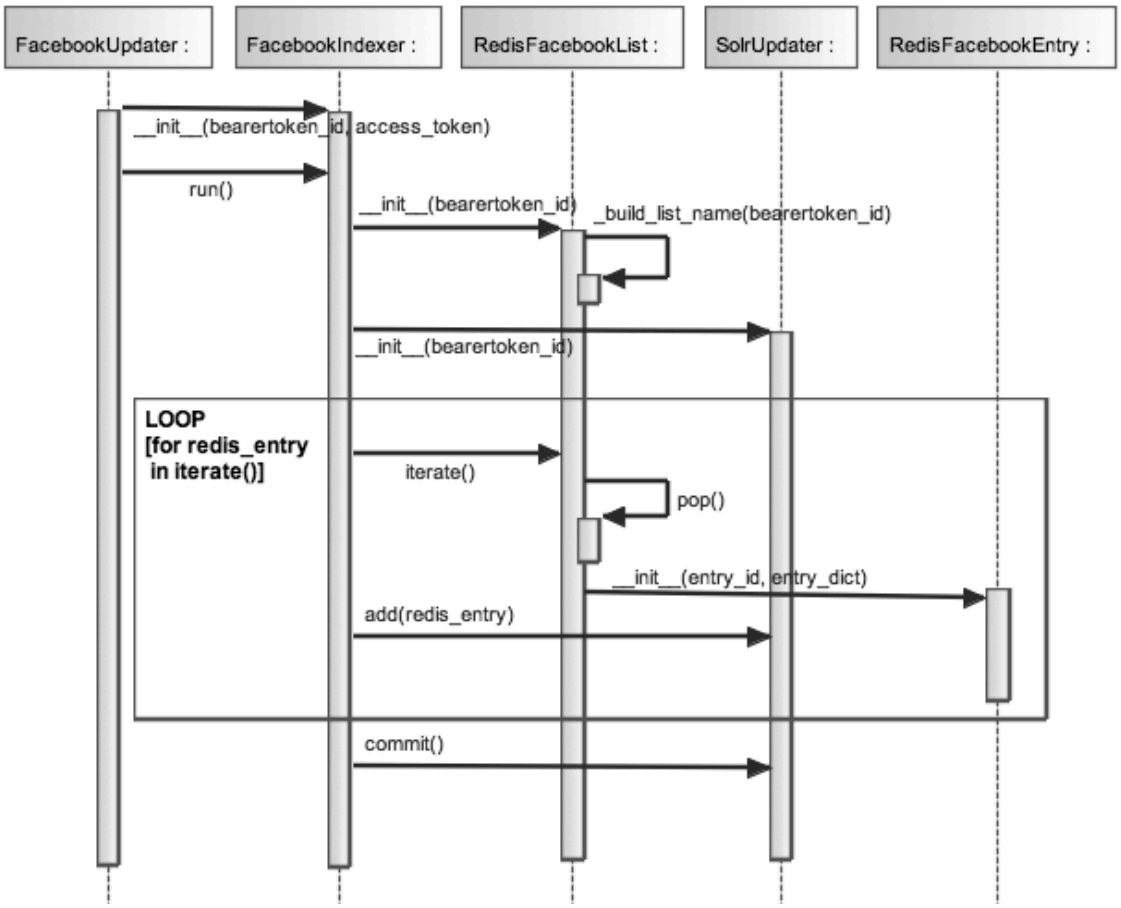

Figure 4-L Sequence diagram for the Facebook indexing process

FacebookIndexer establishes a connection with **RedisFacebookList**, which is a task queue manager for a specific bearer token. The *iterate* method of RedisFacebookList then performs a loop by popping elements from the queue and converting them to **RedisFacebookEntry**. FacebookIndexer sends them one-by-one to Solr and eventually, when the loop is over, invokes a hard-commit.

Downloading Process

Downloading is the process of downloading text files from Dropbox.

The next diagram shows the way DropboxDownloader is run in order to extract file entries from a task queue for a specific bearer token, to download the relative files from Dropbox and to add new tasks to the indexing queue.

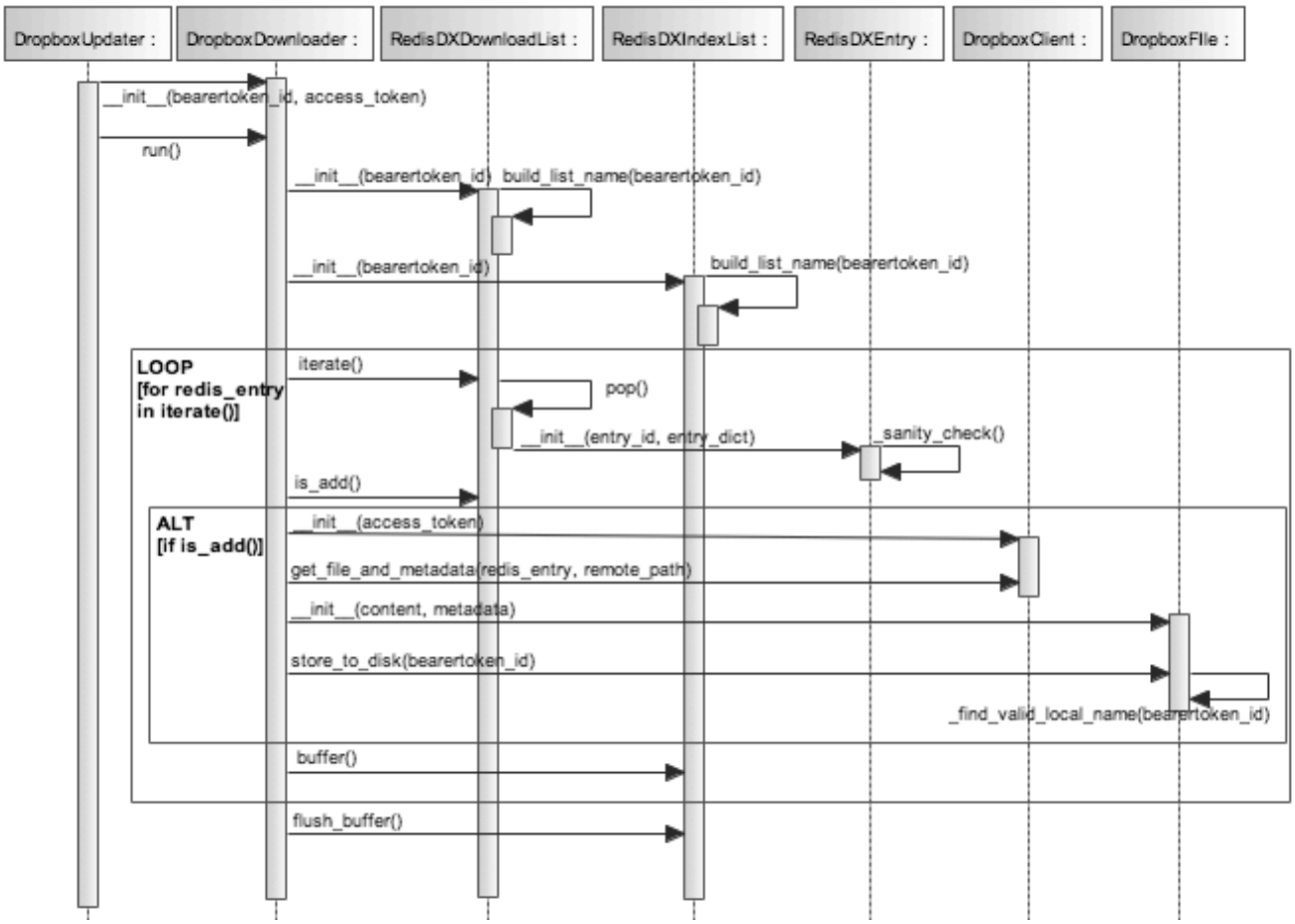

Figure 4-M Sequence diagram for the Dropbox downloading process

DropboxDownloader establishes a connection with **RedisDropboxDownloadList** and **RedisDropboxIndexList**. The *iterate* method of RedisDropboxDownloadList then performs a loop by popping elements one-by-one from the queue and converting them to **RedisDropboxEntry**. If this entry stands for a new file added to Dropbox, the actual file is downloaded using the DropboxClient library, converted to a DropboxFile

instance and stored to disk. Finally the entry is pushed to RedisDropboxIndexList, the task queue for the indexing process.

4.2.3. Development/Implementation View

"The development architecture focuses on the actual software module organization on the software development environment. The software is packaged in small chunks (program libraries, or subsystems) that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it." - The 4+1 View Model of Software Architecture, P. Kruchten, IEEE Software 12(6), 1995.

The next diagram shows an overview of the implementation model for Magpie and describes the system's tiers and packages. The diagram is a simplified version of the actual implementation and all abstract classes have been skipped for clarity's sake, however all the most significant aspects are covered.

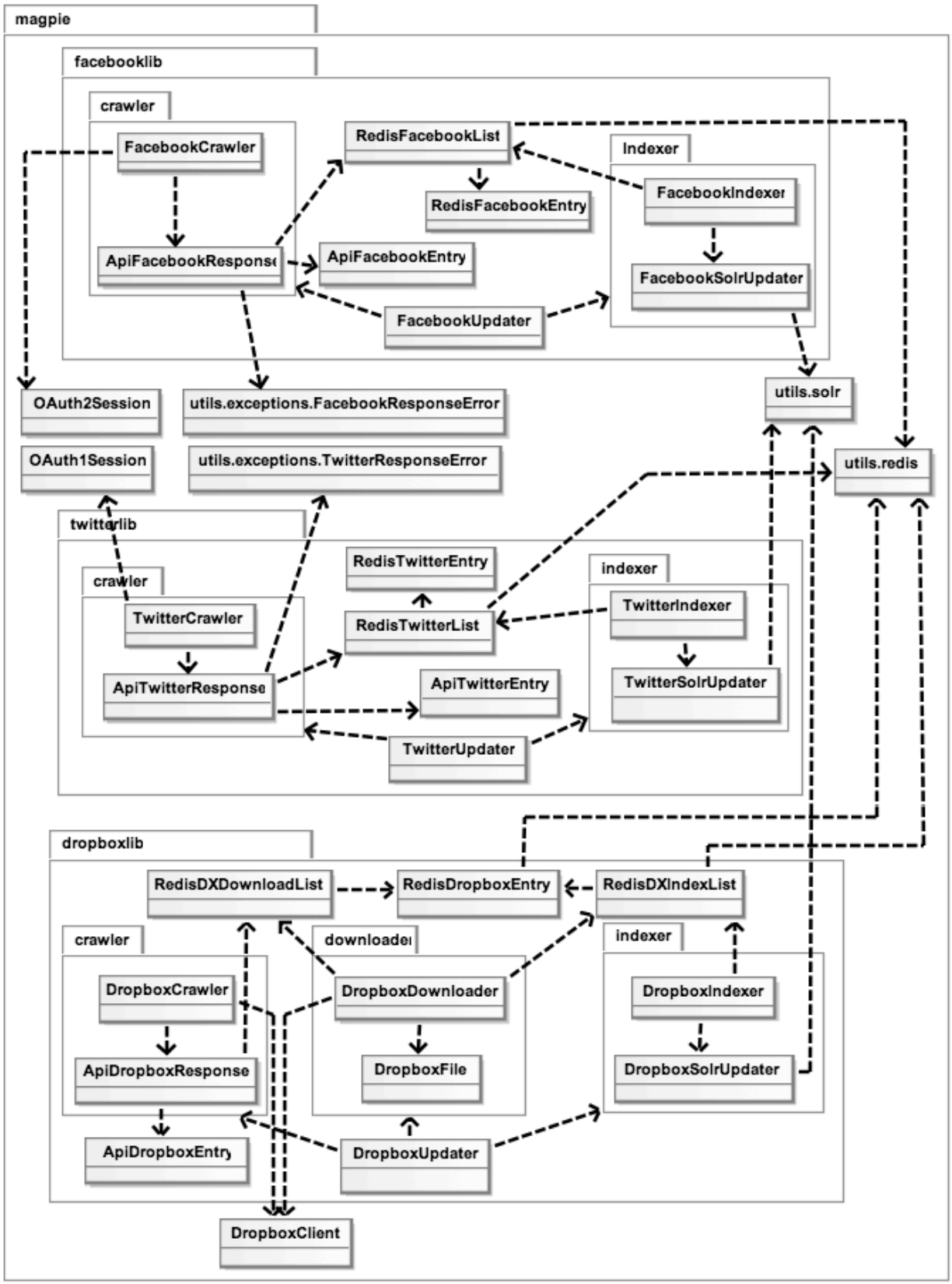

Figure 4-N Package diagram for Magpie

There are three main libraries: **facebooklib**, **twitterlib** and **dropboxlib**. Those libraries have a similar structure and they are composed of a **crawler** and an **indexer** package, plus a **downloader** package for Dropbox. Also some utilities packages to interact with Solr, Redis and the providers are included in the main Magpie package.

4.2.4. Physical/Deployment View

Magpie has four main nodes: the database, the message broker, the full-text search engine and the application itself. These nodes are initially deployed to a single machine, an **Amazon Elastic Compute Cloud** instance.

In the next figure, the main nodes and their connections are clearly identified.

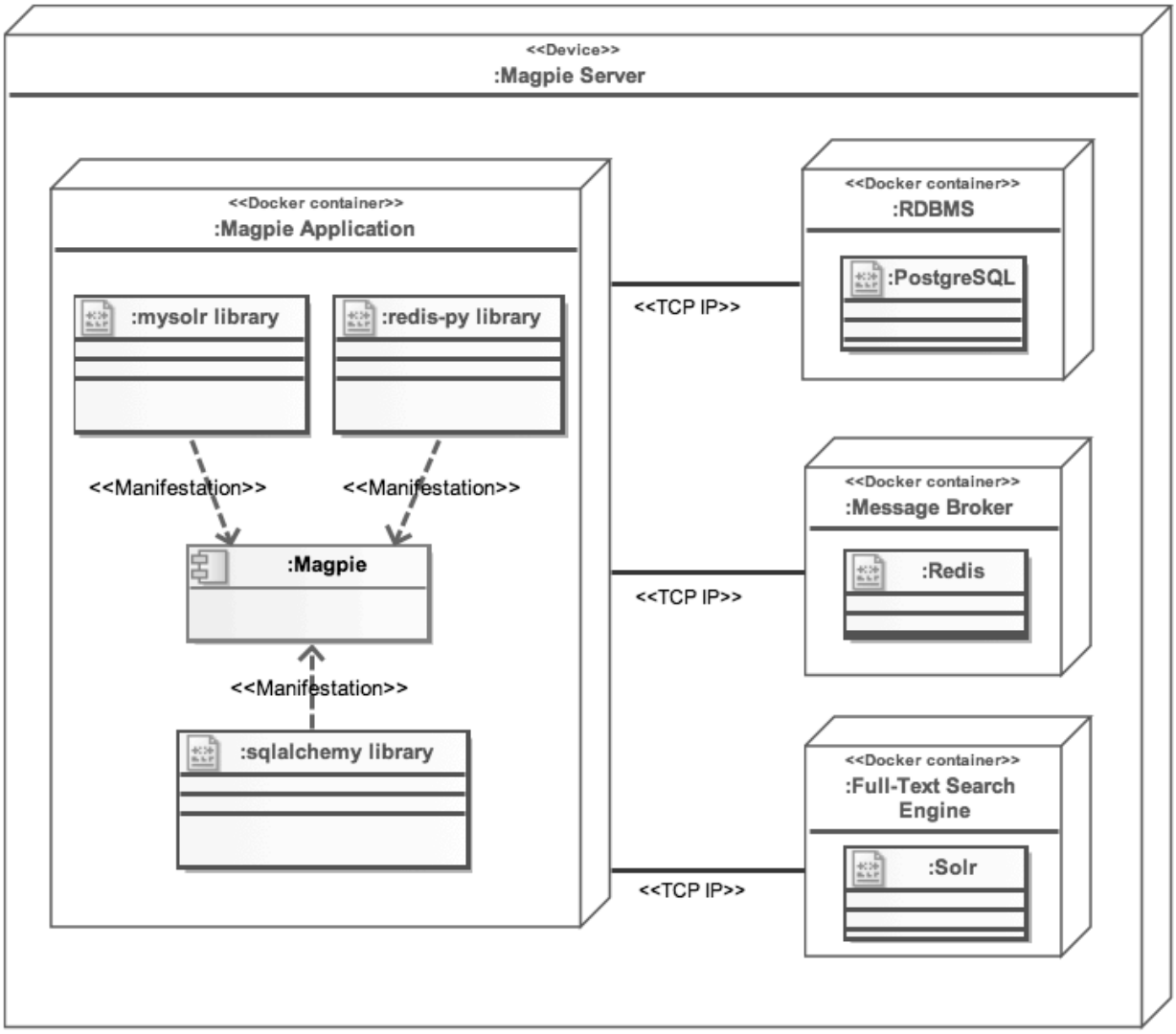

Figure 4-O Deployment diagram for Magpie

Each node is an independent execution environment implemented using a **Docker container**.

"Docker is an open-source project that automates the deployment of applications inside software containers. The Linux kernel provides cgroups for resource isolation (CPU, memory, block I/O, network, etc.) that do not require starting any virtual machines. The kernel also provides namespaces to completely isolate an application's view of the operating environment, including process trees, network, user ids and mounted file systems. LXC (LinuX Containers) combines cgroups and namespace support to provide

an isolated environment for applications; Docker is built on top of LXC, enabling image management and deployment services." - Wikipedia.

"Docker is a tool that can package an application and its dependencies in a virtual container that can run on any Linux server. This helps enable flexibility and portability on where the application can run, whether on premise, public cloud, private cloud, bare metal, etc." - Jay Lyman, senior analyst at 451 Research.

Thus Docker containers are completely independent self-contained environments which communicate with each other using regular TCP/IP connections. This setup would let us scale-up efficiently and easily: Docker containers can indeed be moved to a dedicated machine with no effort.

4.2.5. Use Case View

Use cases are extensively discussed in the chapter 2. Requirements.

4.3. Moogle Website

Moogle Website is built on top of *Django*, a free and open source web application framework, written in Python, which follows the model-view-controller architectural pattern. It is the main user interface of the entire Moogle project and it provides users with the following features:

- connect/disconnect providers (more technically: allow/disallow Moogle to access users' private data hosted at providers);
- perform full-text search;
- navigate the resultset and from there to the original entities hosted at providers (f.i. from tweets indexed by Moogle to the actual tweets hosted at Twitter).

As mentioned in the previous section, the *"4+1 architectural view model"* by Philippe Kruchten is used here, but with the few adaptations introduced by Booch et al. in the book "Object-Oriented Analysis and Design with Applications" - Addison Wesley, 2007.

The following discussion is not meant to be a complete description of Moogle Website's architecture, however all the most significant aspects are covered.

4.3.1. Logical View

Magpie is composed of three main components:

- *OAuthlib*: a library to manage the OAuth flow;
- *Snooper*: a set of classes to query providers (Google Gmail, Google Drive) and Magpie;
- *Solr*: *MySolr* Python library is used in order to manage the actual operations with Apache Solr. This library lacks some important feature so the Solr class is a tiny wrapper around MySolr in a typical *decorator* pattern structure. It enriches

some basic operations like *update* and *search* by performing a sanity check on the response.

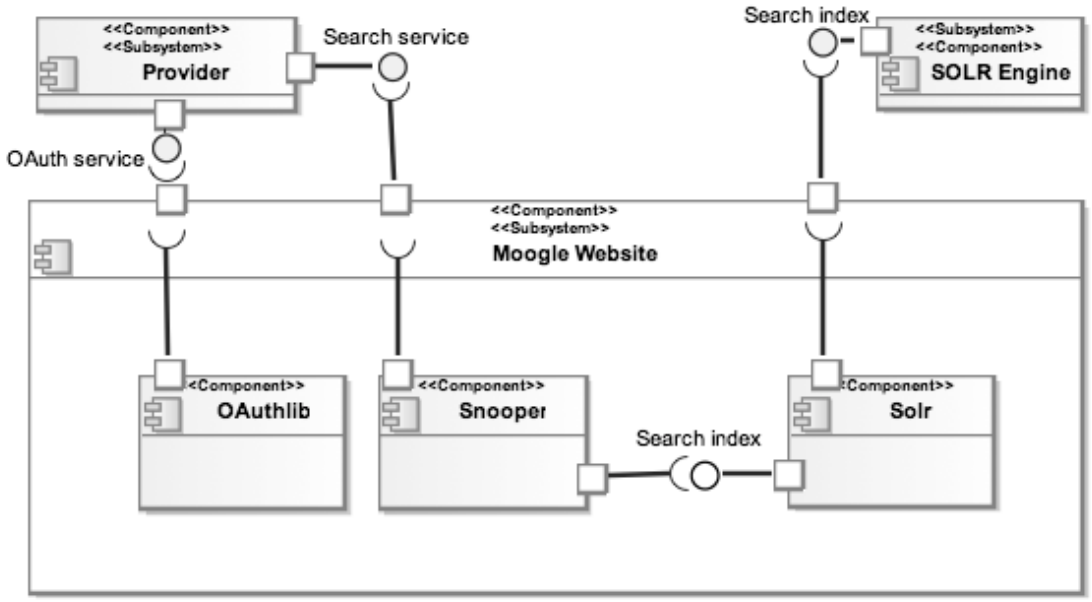

Figure 4-P Top-level component diagram for Moogle Website

Database Models

A classic relational database is used to store information about providers and OAuth access tokens. An object-relational mapping (ORM) is the interface between Django and the actual DBMS engine. The ORM is integrated in Django and it can work together with several DBMS engine like *PostgreSQL* and *MySQL*.

The designed database models and their relationship are quite straightforward as shown in the next diagram.

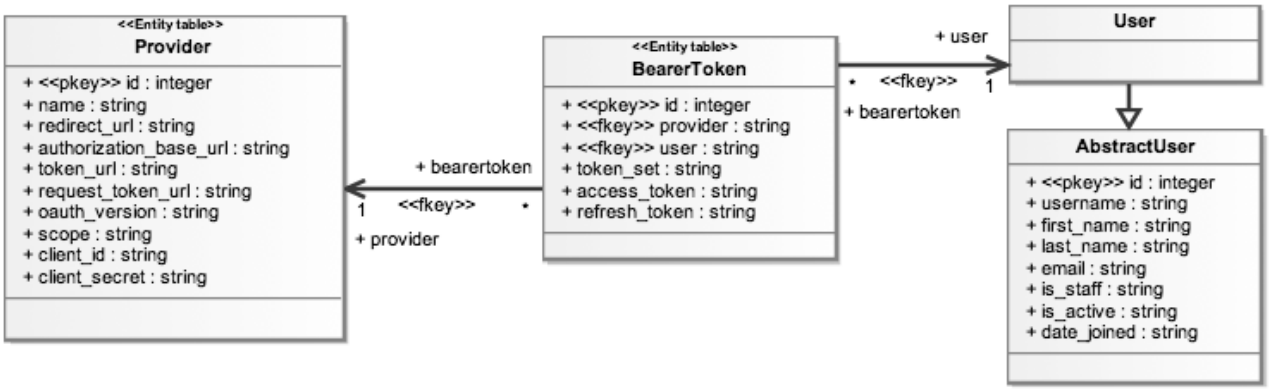

Figure 4-Q ER class diagram

4.3.2. Process View

The main process involved in Moogle Website is relative to the Search use case (see chapter 2.2.2. Search).

The next diagram shows the way the main `search` view first initializes and then sends a `search` message to FacebookSnooper which interacts with Solr for the actual search. The results are then rendered by Django in `facebook_result.html` page.

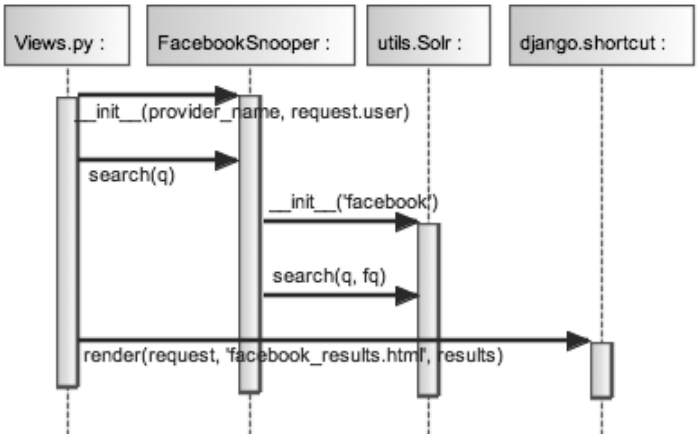

Figure 4-R Sequence diagram for the Facebook search process

4.3.3. Development/Implementation View

The next diagram shows an overview of the implementation model for Moogle website and describes the system's tiers and packages. The diagram is a simplified version of the actual implementation and some classes have been skipped for clarity's sake, however all the most significant aspects are covered.

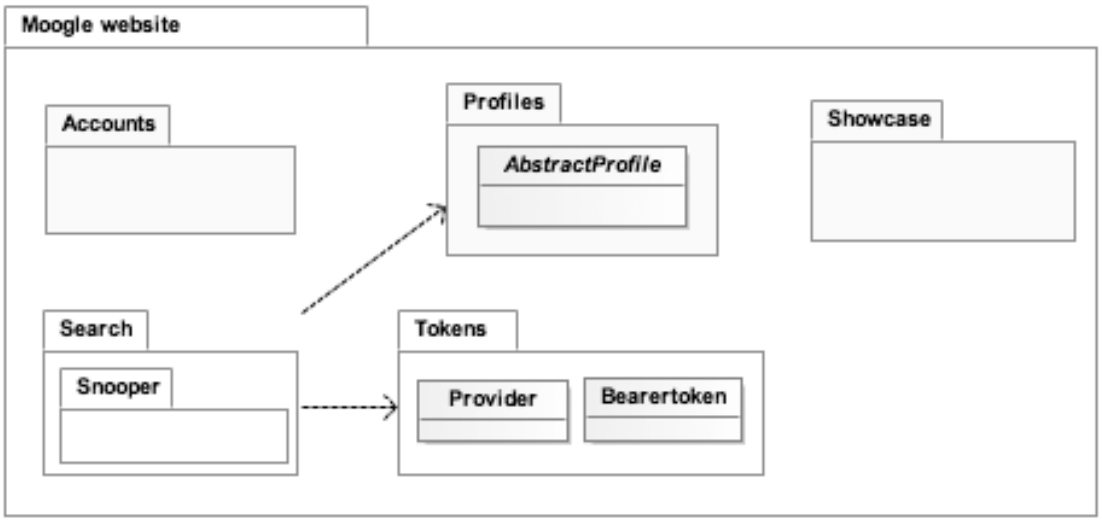

Figure 4-S Package diagram for Moogle Website

4.3.4. Physical/Deployment View

Moogle Website has two main nodes: the web server and the database. These nodes are initially deployed to a single machine, an **Amazon Elastic Compute Cloud** instance.

In the next diagram, the main nodes and their connections are clearly identified:

- *Django* is the Python framework which runs the codebase;
- *Gunicorn* is the Python Web Server Gateway Interface HTTP Server;
- *Nginx* is used as reverse proxy and web server to serve static files.

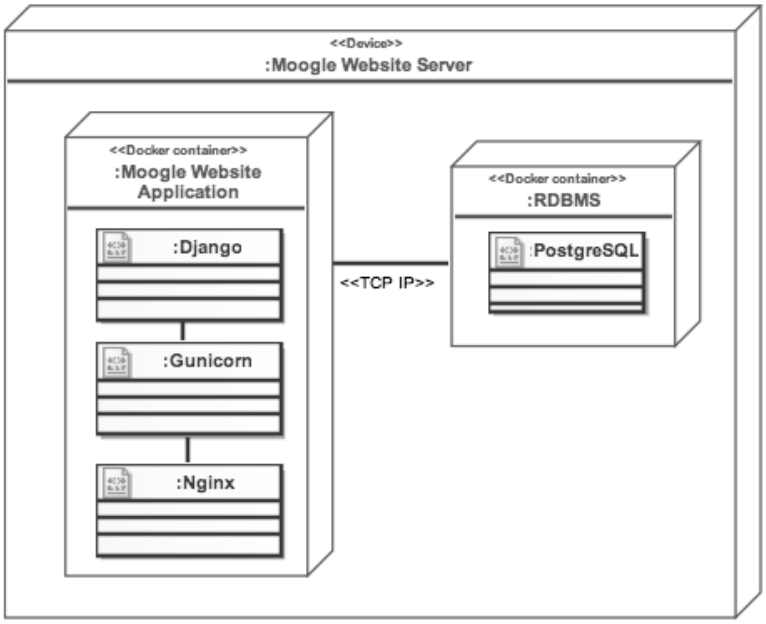

Figure 4-T Deployment diagram for Moogle Website

4.3.5. Use Case View

Use cases are extensively discussed in the chapter 2. Requirements.

CHAPTER 5

Protocols And Data Entities

Moogle connects to several providers in order to fetch data on behalf of users. Each provider supplies an Application Programming Interface (API) with methods designed to accomplish tasks available for that specific provider. Thus, even thought all APIs use the same technologies (like RESTful webservices), they all have different methods and mechanisms to manage things like updates cursors and pagination of results.

A common protocol instead is used to authorize applications to access users' private data: OAuth.

Data entities are the pieces of textual information that Moogle fetches and indexes. In some cases they have a straightforward structure, like tweets in Twitter, while in some other cases like Dropbox their structure is more complex.

This chapter digs into the detail of the protocols used by Moogle, identifies the data entities and lists all the possible ways to fetch them.

5.1. OAuth Protocol

"OAuth is an open standard for authorization. OAuth provides client applications a 'secure delegated access' to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner, or end-user. The client then uses the access token to access the protected resources hosted by the resource server." - Wikipedia.

OAuth is a popular protocol used to authorize applications to access users' private data. This section briefly introduces some of the key-elements of the protocol.

Bearer token: the authorization process ends with a secret key sent from the provider to the application. This key, named bearer token, grants the access to protected resources on behalf of a user.

Scope: a permission for a specific operation, like reading/writing tweets on Twitter.

5.2. Facebook API

Facebook API is named **Graph API**:
https://developers.facebook.com/docs/graph-api.

"The Graph API is the core of Facebook Platform, enabling developers to read from and write data into Facebook. The Graph API presents a simple, consistent view of the Facebook social graph, uniformly representing objects in the graph (e.g., people, photos, events, and pages) and the connections between them (e.g., friend relationships, shared content, and photo tags)." - Wikipedia.

Graph API must not be confused with OpenGraph, which is a protocol to describe the semantic content of web pages and to integrate them in the Social Graph:
http://en.wikipedia.org/wiki/Social_graph.

Graph API is a RESTful webservice which can be used directly or via a query language SQL-like named Facebook Query Language (FQL):
https://developers.facebook.com/docs/reference/fql/.
FQL has some limitations and will be dismissed soon, so I chose to use plain Graph API.

5.2.1. Authorization

Facebook uses OAuth2 to grant access to users' private data.

Bearer Token

Bearer tokens have the following form:
{

```
    "expires_in": "5183999",
    "expires_at": 1404039012.480244,
    "expires": "5183999",
    "token_type": "Bearer",
    "access_token":
"CAABkoVljmhJKghjgkKJG876KLJHjkgJHGkj876JGjhjhKEhQc6JWuOhhic6GylrDpjpZ
CZBEueb4kI4sT4PZA5lkA0yWPXeOr
qXnEjkghKJghKJghKJh78KJGHkGHkhgkjghKJghkjGHjkgh876trlkjnhjgxdfsDHJHKbv
fgda56rsdf55rtDEREWuiyfqiFwUhcZD"
}
```

Bearer Token Expiration

User Access Tokens last 60 days, the expiration date is written inside the token itself: `"expires_in": "5183152"`. The official doc states:
"User access tokens come in two forms: short-lived tokens and long-lived tokens. Short-lived tokens usually have a lifetime of about an hour or two, while long-lived tokens usually have a lifetime of about 60 days. You should not depend on these lifetimes remaining the same - the lifetime may change without warning or expire early."
https://developers.facebook.com/docs/facebook-login/access-tokens/#termtokens.

When a token is expired, the following HTTP response is given:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;
...
{
  "error": {
    "message": "Error validating access token: Session has expired on
Feb 6, 2014 4:00am. The current time is Feb 6, 2014 5:23am.",
    "type": "OAuthException",
    "code": 190,
    "error_subcode": 463
  }
}
```

5.2.2. Query For Basic Profile Information

Scope

No scope is required to query the basic profile information. But the scope `email` is required in order to get the email address.

Example

The query can be done via regular HTTP requests and the response is JSON-encoded data:

```
(GET) https://graph.facebook.com/v2.0/me
{
  "id": "3453454345",
  "name": "John Doe",
  "first_name": "John",
  "last_name": "Doe",
  "link": "https://www.facebook.com/johndoe",
  "gender": "male",
  "email": "johndoe@gmail.com",
  "timezone": 1,
  "locale": "en_GB",
  "verified": true,
  "updated_time": "2014-01-24T18:08:10+0000",
  "username": "johndoe",
}
```

Where `me` stands for the owner of the Bearer Token being used for the request.

5.2.3. Query For Private Textual Data

This chapter describes what the data exposed by Graph API are and identifies the set of data that Moogle handles in order to create a full-text search index.

Available Data

"Any status, link, photo or video posted on your wall by you or a friend of yours, with a status message." - Facebook docs. So basically any information posted on Facebook is available via Graph API.

Full-Text Search Availability

There is no full text search service available either via Facebook website or via Graph API. Moogle has to read pieces of information via Graph API and index them using a full-text search engine.

Main Entities

I. Status Update

A status update is recorded every time a user writes a new status message in his wall. So status updates do not include comments, links, and media (photos and videos). Technically `statuses` are an edge of `user` and a subset of `feeds`: https://developers.facebook.com/docs/graph-api/reference/v2.0/user/feed.

All status updates are available via Graph API at the url: `/v2.0/me/statuses`.
Where `me` stands for the owner of the Bearer Token being used for the request.

Status updates are limited in time: it is not possible to read status updates older than 2011, so it is actually impossible to get the entire history of a user. For this reason and because status updates do not include comments, links and media, I decided to index another entity: feed.

Scope

No scope is required for to query status updates.

Selected Fields

- `id`: the unique identifier of the status update.
- `message`: the textual content.
- `updated_time`: the creation timestamp.

Example

The query is done via regular HTTP requests and the response is JSON-encoded data.
See Appendix A. Entities – A.1. Facebook.

II. Feeds And Posts

Feeds are sets of items displayed on a wall. They can be *statuses*, *links*, *photos* or *videos* posted by the given user (on his own wall or someone else's wall) or posted on the given user's wall (by anyone). Technically a query to a feed returns `post` objects either of type `status` or `link` or `photo` or `video`:
https://developers.facebook.com/docs/graph-api/reference/v2.0/user/feed/
https://developers.facebook.com/docs/graph-api/reference/v2.0/post

All feeds are available via Graph API at the url: `/v2.0/me/feed`.
Where `me` stands for the owner of the Bearer Token being used for the request.

Unlike status updates, the entire feeds history is available via Graph API.

Scope

No scope is required for to query status updates.

Selected Fields

Feed queries return `post` objects of type `status` or `link` or `photo` or `video`: https://developers.facebook.com/docs/graph-api/reference/v2.0/post

The selected fields are:

- `id`: the unique identifier of the post.
- `message`: the textual content.
- `from`: the author.
- `type`: link or status or photo or video.
- `created_time`: the creation timestamp.
- `updated_time`: the last update timestamp.

Example

The query is done via regular HTTP requests and the response is JSON-encoded data. See Appendix A. Entities – A.1. Facebook.

Facebook Query Language (FQL)

Status updates and feeds can be queried using plain Graph API or via Facebook Query Language (FQL). FQL has major limitations, as discussed in the next two sections, thus I decided to query Facebook using plain Graph API.

I. Keyword Search Against Status

Example:

```
SELECT message FROM status WHERE uid = me() AND strpos(lower(message),
'foo') >=0
```

This is not a full-text search but just a keyword search. Plus FQL doesn't use pagination of results: this is a big limitation when querying the entire status updates history.

II. Keyword Search Against Stream

Example:

```
SELECT
action_links, actor_id, app_data, app_id, attachment, attribution,
call_to_action, claim_count, comment_info, created_time, description,
description_tags, expiration_timestamp, feed_targeting, filter_key,
impressions, is_exportable, is_hidden, is_popular, is_published,
like_info, message, message_tags, parent_post_id, permalink, place,
post_id, privacy, promotion_status, scheduled_publish_time,
share_count, share_info, source_id, subscribed, tagged_ids, target_id,
targeting, timeline_visibility, type, updated_time, via_id, viewer_id,
with_location, with_tags, xid, likes
FROM stream
WHERE source_id = me()
AND (
  strpos(lower(message), 'foo') >=0
  OR strpos(lower(attachment.name), 'foo') >=0
  OR strpos(lower(attachment.caption), 'foo') >=0
  OR strpos(lower(attachment.href), 'foo') >=0
  OR strpos(lower(attachment.description), 'foo') >=0
)
```

Again this is not a full-text search but just a keyword search with no pagination of results.

Pagination Of Results

Graph API results are paginated using a `previous` and a `next` link to navigate amongst results:
https://developers.facebook.com/docs/graph-api/using-graph-api/#paging

Example:

```
"paging": {
```

```
  "previous":
"https://graph.facebook.com/1522956429/statuses?limit=25&since=1390148
940&__paging_token=10203300267177108",
  "next":
"https://graph.facebook.com/1522956429/statuses?limit=25&until=1343575
899&__paging_token=4415609238035"
}
```

Sub-elements like comments and likes can be paginated using cursors to be used in the next query. Example:

```
"paging": {
  "cursors": {
    "after": "MTAyMDE5MTk5NDQ3MDY1MTE=",
    "before": "MTAxNTIzODU3MTAzODE1NjU="
  }
}
```

Real-Time Updates

Facebook has recently introduced Real Time Updates: webhooks which send updates directly to all applications subscribed to changes in certain pieces of data. This makes applications more efficient, as they know exactly when a change has happened, and don't need to rely on continuous or even periodic poll of Graph API.
https://developers.facebook.com/docs/graph-api/real-time-updates/
http://en.wikipedia.org/wiki/Webhook

This feature has not been implemented yet in Moogle, but it is listed as a future improvement (see chapter 8. Conclusions and Future Developments).

5.3. Twitter API

Twitter API is named **REST API** and it is implemented as a RESTful webservice: https://dev.twitter.com/docs/api/1.1.

"... we provide an API method for just about every feature you can see on our website. Programmers use the Twitter API to make applications, websites, widgets, and other projects that interact with Twitter. Programs talk to the Twitter API over HTTP, the same protocol that your browser uses to visit and interact with web pages." - Twitter.

REST API must not be confused with Streaming API which is discussed in the next Streaming Updates section.

5.3.1. Authorization

Twitter uses OAuth1.0a to grant access to users' private data.

Bearer Token

Bearer tokens have the following form:

```
{
    "oauth_token": "123456789-
hgTfRD567nLwtciUDFjWhGffDR678jgf5RFjkGf567",
    "screen_name": "johndoe",
    "oauth_token_secret":
"hGtFd6i8TR7kiGjlmcM55q1nvjhGfdSt678jHREdfTgH76",
    "user_id": "123456789"
}
```

Bearer Token Expiration

"The access tokens yielded to you through the OAuth process do not have an explicit expiration datetime attached to them. They will be valid for use when making requests on behalf of your users for as long as the user has granted your application access. A user can revoke access to your application at any time." - Twitter.

When a token has been revoked, the following HTTP response is given:

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json;
...
{
  "errors": [
    {"message":"Invalid or expired token","code":89}
  ]
}
```

5.3.2. Query For Basic Profile Information

Scope

There is no such a concept of **scope** in OAuth 1.0a. But detailed settings can be defined in the application management page, see Appendix B. Providers Details. In particular the option `Application Type Access` can have values:

- Read only
- Read and Write
- Read, Write and Access direct messages

Moogle requires a `Read only` access.

Example

The query can be done via regular HTTP requests and the response is JSON-encoded data:

```
(GET) https://api.twitter.com/1.1/account/verify_credentials.json
{
  "id": 135348443,
  "id_str": "135348443",
  "name": "John Doe",
  "screen_name": "johndoe",
  "location": "Milky Way",
  "description": "This is my twitter",
  "url": "http://t.co/l2h46P9ias",
  "entities":  {
    "url":  {
      "urls":  [
         {
          "url": "http://t.co/l2h46P9ias",
          "expanded_url": "http://johndoe.io",
          "display_url": "johndoe.io",
          "indices":  [
            0,
            22
          ]
        }
      ]
    },
    "description":  {
      "urls":  []
    }
  },
  "protected": false,
  "followers_count": 15,
  "friends_count": 31,
  "listed_count": 0,
  "created_at": "Wed Apr 21 02:05:39 +0000 2010",
  "favourites_count": 0,
  "utc_offset": 3600,
  "time_zone": "Ljubljana",
  "geo_enabled": false,
  "verified": false,
  "statuses_count": 13,
```

```
  "lang": "en",
  "status":  {
    "created_at": "Sun Jan 19 17:39:13 +0000 2014",
    "id": 424959240944377860,
    "id_str": "424959240944377856",
    "text": "I love Pizza!",
    "source": "web",
    "truncated": false,
    "in_reply_to_status_id": null,
    "in_reply_to_status_id_str": null,
    "in_reply_to_user_id": null,
    "in_reply_to_user_id_str": null,
    "in_reply_to_screen_name": null,
    "geo": null,
    "coordinates": null,
    "place": null,
    "contributors": null,
    "retweet_count": 0,
    "favorite_count": 0,
    "entities":  {
      "hashtags":  [],
      "symbols":  [],
      "urls":  [],
      "user_mentions":  []
    },
    "favorited": false,
    "retweeted": false,
    "lang": "sl"
  },
  "contributors_enabled": false,
  "is_translator": false,
  "profile_background_color": "C0DEED",
  "profile_background_image_url":
"http://abs.twimg.com/images/themes/theme1/bg.png",
  "profile_background_image_url_https":
"https://abs.twimg.com/images/themes/theme1/bg.png",
  "profile_background_tile": false,
  "profile_image_url":
"http://abs.twimg.com/sticky/default_profile_images/default_profile_5_
normal.png",
  "profile_image_url_https":
"https://abs.twimg.com/sticky/default_profile_images/default_profile_5
_normal.png",
  "profile_link_color": "0084B4",
```

```
  "profile_sidebar_border_color": "C0DEED",
  "profile_sidebar_fill_color": "DDEEF6",
  "profile_text_color": "333333",
  "profile_use_background_image": true,
  "default_profile": true,
  "default_profile_image": true,
  "following": false,
  "follow_request_sent": false,
  "notifications": false
}
```

5.3.3. Query For Private Textual Data

This chapter describes what the data exposed by REST API are and identifies the set of data that Moogle handles in order to create a full-text search index.

Available Data

Any tweet (regular tweet or re-tweet) and any private message is available via REST API. If a tweet includes a media (a photo or a video), the media is hosted at Twitter and its URL included in the tweet.

Full-Text Search Availability

There is no full text search service available either via Twitter website or via REST API. Moogle has to read pieces of information via REST API and index them using a full-text search engine.

Main Entities

I. Tweet

"Tweets are the basic atomic building block of all things Twitter. Users tweet Tweets, also known more generically as "status updates." Tweets can be embedded, replied to, favorited, unfavorited and deleted." - Twitter. 140 characters long, they can include URL.

II. Scope

See previous Scope section.

III. Selected Fields

- `id_str`: the unique identifier of the tweet;
- `user`:
    - `id_str`: the unique identifier of the author;
- `text`: the textual content;
- `lang`: language used in the `text`. All available languages: https://api.twitter.com/1.1/help/languages.json to find all languages;
- `in_reply_to_screen_name`: the author of the original tweet, for retweets;
- `retweeted`: true for retweets;
- `created_at`: the creation timestamp;
- `entities`:
    - `urls`:
        - `expanded_url`: full version of any URL included in `text`;
    - `user_mentions":
        - `screen_name`: any user mention in `text` in the form @username;
    - `hashtags`:
        - `text`: any hash tag in `text` in the form #hashtag;
    - `symbols`:

- text`: any symbol in `text` in the form $SYMBOL.

See also:
https://dev.twitter.com/docs/platform-objects/tweets

API Endpoints and Calls

I. Search/tweets

Returns a collection of relevant Tweets matching a specified query:
https://dev.twitter.com/docs/api/1.1/get/search/tweets.

This endpoint provides a limited view of all tweets: *"Please note that Twitter's search service and, by extension, the Search API is not meant to be an exhaustive source of Tweets. Not all Tweets will be indexed or made available via the search interface."* - Twitter. It seems that only the most popular tweets are available: because of this limitation, I decided not to use this endpoint.

Base URL

https://api.twitter.com/1.1/search/tweets.json

Example

See Appendix A. Entities – Twitter.

II. Statuses/user_timeline

"Returns a collection of the most recent Tweets posted by the user indicated by the screen_name or user_id parameters... This method can only return up to 3,200 of a user's most recent Tweets... Requests per rate limit window: 180/user, 300/app" - Twitter: https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline.

Also for this endpoint tweets are limited, but the given threshold (3200) is big so that all tweets could be actually fetched for most of the accounts. This is the endpoint I chose to query with Moogle in order to collect and index tweets.

Base URL

https://api.twitter.com/1.1/statuses/user_timeline.json

Example

See Appendix A. Entities – Twitter.

III. Streaming User

Streams messages for a single user. This endpoint has always been unavailable in the last year (2014), plus a streaming connection is not suitable for Moogle as discussed in the next Streaming Updates section:
https://dev.twitter.com/docs/api/1.1/get/user

Base URL

https://userstream.twitter.com/1.1/user.json

IV. Streaming Statuses/filter

Meant to track public tweets that match one or more predicate. Because of this and because it is a streaming connection (see next Streaming Updates section), this endpoint is not suitable for Moogle:
https://dev.twitter.com/docs/api/1.1/post/statuses/filter.

Base URL

https://stream.twitter.com/1.1/statuses/filter.json

Pagination Of Results

"The Twitter API has several methods, such as GET statuses/user_timeline, GET statuses/home_timeline and GET search/tweets, which return a timeline of Tweet data.

Such timelines can grow very large, so there are limits to how much of a timeline a client application may fetch in a single request. Applications must therefore iterate through timeline results in order to build a more complete list." - Twitter.

Unlike Facebook, Twitter leaves the management of the pagination of results to applications. This is done using the `max_id` and the `since_id` parameters: https://dev.twitter.com/docs/working-with-timelines

Details are discussed in the chapter 4. Architecture – Magpie.

Streaming Updates

"The set of streaming APIs offered by Twitter give developers low latency access to Twitter's global stream of Tweet data. A proper implementation of a streaming client will be pushed messages indicating Tweets and other events have occurred, without any of the overhead associated with polling a REST endpoint." - Twitter.

A streamline connection is a long-lasting connection where data flow from the server to the client application in a slow and persistent stream. This is a different logic from the traditional polling and also from the webhooks-based real-time updates implemented by Facebook. *"Connecting to the streaming API requires keeping a persistent HTTP connection open. In many cases this involves thinking about your application differently than if you were interacting with the REST API"* - Twitter. It is an attempt to get rid of the traditional polling logic which generates traffic overload, but the nature of this strategy, unlike webhooks, does not match Moogle requirements. Moogle is a multi-user application and managing a long-lasting connection per user is not efficient.

5.4. Dropbox API

Dropbox API is named **Core API**:
https://www.dropbox.com/developers/core/docs

"The Core API provides a flexible way to read and write to Dropbox. It includes support for advanced functionality like search, revisions, and restoring files. While Drop-ins and the Sync API are simpler to integrate and use, the Core API can be a better fit for server-based apps." - Dropbox.

Core API must not be confused with Sync API, which is meant exclusively for mobile applications. Core API is a RESTful webservice to read and write files on Dropbox on the behalf of a user and get updates when a file has changed.

5.4.1. Authorization

Dropbox uses OAuth2 to grant access to users' private data.

Bearer Token

Bearer tokens have the following form:

```
{
    "access_token": "hg67FdRf67JAAAXXXAAABTfWu-CjhYgj-
jHy654EfGhPpolVPQUFiXNnfdy7hThg",
    "uid": "207239643",
    "token_type": "bearer"
}
```

Bearer Token Expiration

Bearer tokens do not expire but they can be revoked by the user or by Dropbox administrators:
https://www.dropbox.com/developers/core/bestpractices.

"In the case where a token is no longer authorized, the REST API will return an HTTP Error 401 Unauthorized response. Re-authenticating is typically all that is necessary to regain access." - Dropbox.

5.4.2. Query For Basic Profile Information

Scope

There is no such a concept of **scope** in Core API, even though it uses OAuth 2.0. But detailed settings can be defined in the application management page, see Appendix B. Providers Details. In particular two options:

- Permission type: all file types or specific file type.
- Allowed file types: text files, documents, images, videos, audio files, ebooks.

Moogle requires access only to text files (TXT, Markdown, Code files, HTML, etc.) and documents (Word, Excel, PowerPoint, PDF, CSV, etc.).

Example

The query can be done via regular HTTP requests and the response is JSON-encoded data:

```
(GET) https://api.dropbox.com/1/account/info
{
    "referral_link":
"https://www.dropbox.com/referrals/r1a2n3d4m5s6t7",
```

```
    "display_name": "John Doe",
    "uid": 12345678,
    "country": "US",
    "quota_info": {
        "shared": 253738410565,
        "quota": 107374182400000,
        "normal": 680031877871
    },
    "email": "johndoe@gmail.com",
    "allowed_file_types": [
        "",
        ".aeh",
        ".applescript",
        ".as",
        ".as3",
        ...
    ]
}
```

5.4.3. Query For Private Textual Data

This chapter describes what the data exposed by Core API are and identifies the set of data that Moogle handles in order to create a full-text search index.

Available Data

Any file, of the chosen file types, stored in Dropbox is available via Core API.

Full Text Search Availability

There is no full text search service available either via Dropbox website or via Core API. The only search functionality implemented is a file-name search. Moogle has to download text files via Core API and index them using a full-text search engine.

Main Entities

I. Text File And Document Plus Metadata

Main entities are regular text files like plain text, PDF, Microsoft Word, Markdown, etc. Each file is described by a set of metadata.

Scope

See previous Scope section.

Metadata

Information about a file or a folder.

Example:


```
{
    "size": "225.4KB",
    "rev": "35e97029684fe",
    "thumb_exists": false,
    "bytes": 230783,
    "modified": "Tue, 19 Jul 2011 21:55:38 +0000",
    "client_mtime": "Mon, 18 Jul 2011 18:04:35 +0000",
    "path": "/Getting_Started.pdf",
    "is_dir": false,
    "icon": "page_white_acrobat",
    "root": "dropbox",
    "mime_type": "application/pdf",
    "revision": 220823
}
```


Selected Metadata

Apart from the actual content of a text file, the selected metadata are:

- `bytes`: size of the file.
- `modified`: a last edit timestamp.
- `path`: full path of the file relative to the Dropbox root folder.

API Endpoints And Calls

I. Delta

"A way of letting you keep up with changes to files and folders in a user's Dropbox. You can periodically call /delta to get a list of delta entries, which are instructions on how to update your local state to match the server's state." - Dropbox.

Delta is the endpoint for updates. When a user edits a file a new entry relative to the edited file is added to the delta list. A delta entry is a list of couples where the first item is the full path of the file and the second item is the metadata dictionary:
https://www.dropbox.com/developers/core/docs#delta

See an example in Appendix A. Entities.

II. Get File

Downloads a file along with its metadata:
https://www.dropbox.com/developers/core/docs#files-GET

Metadata are stored in the `x-dropbox-metadata` HTTP header in JSON format, while the file content is in the actual HTTP response content.

Pagination Of Results

The pagination of results for /delta queries is accomplished using two parameters: a `cursor` and an `has_more` flag. When the `has_more` flag is set to true, it means that more results are available and they can be fetched appending the given cursor to the next calls.

Real-Time Updates

Dropbox has recently introduced Webhooks: they are a way for web applications to get real-time notifications when users' files change in Dropbox. This makes applications more efficient, as they know exactly when a change has happened, and don't need to rely on continuous or even periodic poll of Core API:
https://www.dropbox.com/developers/webhooks/tutorial

"With webhooks, your web app can be notified when your users' files change, rather than requiring your app to frequently poll Dropbox for changes. Every time a user's files change, the webhook URI you specify will be sent an HTTP request telling it which users have changes." - Dropbox.

This feature has not been implemented yet in Moogle, yet it is listed as a future improvement.

5.5. Google API

Google provides a vast set of APIs to interact with its services:
https://developers.google.com/apis-explorer/#p/

"Google APIs (or Google AJAX APIs) is a set of [...] APIs developed by Google that allows interaction with Google Services and integration of rich, multimedia, search or feed-based Internet content into web applications." - Wikipedia.

Moogle works with two Google products: Gmail and Drive. The involved APIs are named: **Gmail API** and **Drive API**.

5.5.1. Authorization

Google uses OAuth2 to grant access to users' private data for most of its products. Gmail API uses IMAP authentication based on bearer tokens generated during the OAuth step: details are discussed in the next chapter.

Bearer Token

Bearer tokens have the following form:

```
{
    "token_type": "Bearer",
    "refresh_token": "1/HygFdre49IhawQ-hgrdg76GHhs9VNmPwLA4PF089ijh",
    "access_token": "ya32.KIU-XX-jUhgFre45TgfcGh-
jkUHgt56FdretNplKo9f5ea7anrG8HTR43DFvhh",
    "id_token": "kjkj897fdggh...kjHgrWsd45Dfg",
    "expires_at": 1400622901.529355,
    "expires_in": 3600
}
```

Bearer Token Expiration

Bearer tokens last for 3600 seconds (1 hour) and they include a refresh token. The refresh token must be used in order to get a new valid bearer token: https://developers.google.com/accounts/docs/OAuth2WebServer#refresh

5.5.2. Query For Basic Profile Information

Scope

Two scopes are required in order to get the user's name and email address:

```
"https://www.googleapis.com/auth/userinfo.email",
"https://www.googleapis.com/auth/userinfo.profile",
```

The email address is used by Gmail API.

Example

The query can be done via regular HTTP requests and the response is JSON-encoded data:

```
(GET) https://www.googleapis.com/userinfo/v2/me
{
  "id": "12345678901234567890",
  "email": "johndoe@gmail.com",
  "verified_email": true,
  "name": "John Doe",
  "given_name": "John",
  "family_name": "Doe",
  "link": "https://profiles.google.com/12345678901234567890",
  "gender": "male",
  "locale": "en"
}
```

Where `me` stands for the owner of the Bearer Token being used for the request.

5.6. Google Drive API

Google Drive API is a RESTful webservice to read, write and search the content of files on Drive on the behalf of a user and get updates when a file has changed: https://developers.google.com/drive/web/about-sdk

5.6.1. Query For Private Textual Data

This chapter describes what the data exposed by Drive API are and identifies the endpoints and API methods that Moogle deals with in order to provide a full-text search feature.

Available Data

Any file stored in Drive is available via Drive API.

Full Text Search Availability

"Drive automatically indexes the content of most common file types (e.g. .html, .xml, .txt ...) for search as soon as they are uploaded. Also, Drive uses OCR to find text in images or PDF files and Google Goggles to examine and index identifiable objects, people and places. For other resources such as drawings or unique file types, apps can provide their own indexable text when they save files to Drive." - Google.

Google provides a great full text search service which can be used via Drive website or via Drive API. Thus in this case, Moogle does not need to download textual files in order to index them, but it behaves like a proxy to smooth the access to the original Drive search.

Main Entities

I. Text File And Document Plus Metadata

Main entities are regular text files like plain text, PDF, Microsoft Word, Markdown, etc. Each file is described by a set of metadata.

Scope

In order to use the search endpoint a read-only scope is required:

```
"https://www.googleapis.com/auth/drive.readonly",
```

Metadata

Information about a file or a folder. They are the actual entities included in a response to a search query. See example in Appendix A. Entities – Dropbox.

Selected Metadata

The selected metadata includes:

- `title`: the title of the document;
- `originalFilename`: the file name;
- `fileExtension`: e.g. "pdf";
- `mimeType`: e.g. " application/pdf ";
- `kind`: file or folder;
- `fileSize`: the file size in bytes;
- `modifiedDate`: the last edit timestamp;
- `createdDate`: the creation timestamp;
- `ownerNames`: a list of names;
- `owners`: a list of dictionaries where each element is a owner with his display name and picture;
- `thumbnailLink`: link to a thumbnail for the document;
- `webContentLink`: link to download the document;
- `alternateLink`: link to open the document in Drive website;
- `iconLink`: link to a icon for the document;

- `exportLinks`: link to download the document converted in pdf, rtf, html, Word, Open Office or plain text format;
- `embedLink`: link to preview the document.

API Endpoints And Calls

I. Files.list

Files.list is the endpoint for search. The search can be performed against metadata, like title and owner, or against the actual textual content of documents. The latter is a proper full-text search and is accomplished using the `fullText` field, like:

```
(GET)
https://www.googleapis.com/drive/v2/files?q=fullText+contains+%27foo%27
```

See a full example in Appendix A. Entities.

Pagination Of Results

The pagination of results for files.list queries is accomplished using the `nextLink` parameter which is returned when the response contains more results. It is an URL which must be used for the next call.

5.7. Google Gmail API

"The Gmail IMAP and SMTP servers have been extended to support authorization via the industry-standard OAuth 2.0 protocol... IMAP and SMTP use the standard Simple Authentication and Security Layer (SASL), via the native IMAP AUTHENTICATE and SMTP AUTH commands, to authenticate users. The SASL XOAUTH2 mechanism enables clients to provide OAuth 2.0 credentials for authentication. The SASL XOAUTH2 protocol documentation describes the SASL XOAUTH2 mechanism in great detail, and libraries and samples which have implemented the protocol are available." - Google.

Google extended the IMAP protocol adding two notable features:

- authorization via OAuth protocol.
- full-text search.

The result is not perfect, especially the design is sometimes over complicated yet it is still a powerful tool:
https://developers.google.com/gmail/xoauth2_protocol

In order enable an application to access a Gmail account using this API, a user must enable the IMAP access. This could be accomplished in Gmail website following: Settings > Forwarding and POP/IMAP.

5.7.1. Query For Private Textual Data

This chapter describes what the data exposed by Gmail API are and identifies the endpoints and API methods that Moogle deals with in order to provide a full-text search feature.

Available Data

Any email stored in Gmail is available via Gmail API.

Full Text Search Availability

Google extended the IMAP protocol adding a search command `X-GM-RAW` to perform full-text search:
https://developers.google.com/gmail/imap_extensions#extension_of_the_search_command_x-gm-raw

"To provide access to the full Gmail search syntax, Gmail provides the X-GM-RAW search attribute. Arguments passed along with the X-GM-RAW attribute when executing the SEARCH or UID SEARCH commands will be interpreted in the same manner as in the Gmail web interface." - Google.

The new protocol is not well documented yet, so an experimental approach is required in order to get good results. Using this command Moogle can act like a proxy to smooth the access to the original full text-search without having to download and index emails.

Main Entities

I. Email

Main entities are regular emails as in the traditional IMAP protocol.

Scope

There is no such a concept of scope, but as per the traditional IMAP protocol during the folder selection step it is possible to specify a read-only access.

Extended IMAP Protocol

The extended IMAP protocol is composed of four main steps.

I. Authentication

The authentication step follows the traditional IMAP protocol, but the bearer token is used instead of a password.

II. Folder Selection

Selecting the right folder is important because it defines the scope of the search. To perform an extensive search the folder with all mail must be selected. This folder has different names for different languages: `"[Gmail]/All Mail"` for English, `"[Gmail]/Tutti i messaggi"` for Italian. In order to identify this folder, Moogle runs a `LIST` command first and parses the response.

III. Search

Full-text search is performed using the X-GM-RAW parameter, as already explained in the previous Full-Text Search Availability section.

The basic syntax is as follows:

```
a005 SEARCH X-GM-RAW "has:attachment in:unread"
And the response is a list of email identifiers which can be later
used with a `FETCH` command.
* SEARCH 123 12344 5992
a005 OK SEARCH (Success)
```

IV. Fetch

The FETCH command is used to download emails from the IMAP server. Moogle does not need the entire content of an email message, but only a few pieces of information. The performance of the FETCH operation can be then improved using field filters.

Some filtering options:

- `(RFC822)`: fetch the entire message;
- `(UID BODY[TEXT])`: the UID and the text (UID is the unique message identifier, unique within the entire mailbox);
- `((UID X-GM-MSGID X-GM-THRID)`: the UID and the identifiers which can be used to build a direct link to the message in Gmail website;
- `(BODY[HEADER])`: only the headers (from, to, subject, date, ...);
- `(BODY[HEADER.FIELDS (SUBJECT FROM TO DATE)])`: only the given headers: subject, from date, to date;
- `(BODY[TEXT]<0.100>)`: only the first 100 bytes of the content;
- `(BODY.PEEK[1]<0.100>)`: only the first 100 bytes of the first content part (the content is divided in parts like plain, html, attachment, ...);
- `(BODY.PEEK[1.MIME])`: only the mime of the first part.

The filter used by Moogle is:

```
(UID X-GM-MSGID X-GM-THRID BODY[HEADER.FIELDS (SUBJECT FROM TO DATE)]
BODY.PEEK[1.MIME] BODY.PEEK[1]<0.100>)
```

This fetches the UID together with the identifiers required to build a direct link to the message in Gmail website. Plus the subject, from and to dates, the mime and the first 100 bytes of the content.

See a full example in Appendix A. Entities.

CHAPTER 6

Task Queue With Redis

6.1. Redis Overview

Redis is a high-performance, key-value data store. More specifically, it is a type of database that is commonly referred to as NoSQL or non-relational. In Redis, there are no tables, and there is no database defined or enforced way of relating data together. Data are stored in memory using only five different types of basic structures: this is what makes Redis very efficient in any I/O operation.

Redis also supports the writing of its data to disk automatically. Two different forms of persistence are available: the first method is a point-in-time dump either when certain conditions are met (f.i. a number of writes in a given period) or when one of the two dump-to-disk commands is called. The other method uses an append-only file that writes every command that alters data in Redis to disk as it happens.

Given these features, Redis could be used as general-purpose memory caching system, similar to Memcached. A typical application for this kind of software is to speed up dynamic database-driven websites.

Redis is different compared to other key-value stores for two main reasons:

- *"Redis is a different evolution path in the key-value DBs where values can contain more complex data types, with atomic operations defined on those data*

types. Redis data types are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers." - Redis documentation.

- *"Redis is an in-memory but persistent on disk database, so it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than memory. Another advantage of in memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk, so Redis can do a lot, with little internal complexity. At the same time the two on-disk storage formats (RDB and AOF) don't need to be suitable for random access, so they are compact and always generated in an append-only fashion (Even the AOF log rotation is an append-only operation, since the new version is generated from the copy of data in memory)."* - Redis documentation.

Generally speaking, many software developers choose to store data in Redis only when the performance or functionality of Redis is necessary, using other relational or non-relational data storage for data where slower performance is acceptable, or where data is too large to fit in memory economically.

"One common use of databases is to store long-term reporting data as aggregates over fixed time ranges. To collect these aggregates, it's not uncommon to insert rows into a reporting table and then later to scan over those rows to collect the aggregates, which then update existing rows in an aggregation table. Rows are inserted because, for most databases, inserting rows is a very fast operation (inserts write to the end of an on-disk file, not unlike Redis's append-only log). But updating an existing row in a table is fairly slow (it can cause a random read and may cause a random write).
By using Redis instead of a relational or other primarily on-disk database, you can avoid writing unnecessary temporary data, avoid needing to scan over and delete this temporary data, and ultimately improve performance. These are both simple examples, but they demonstrate how your choice of tool can greatly affect the way you solve your problems." - *Redis in Action* by J.L. Carlson, Manning, 2013.

Another common use of Redis is to manage task (message) queue: f.i. when handling requests from web clients, some operations like media processing might take a long time and we don't want clients to wait too long. We can defer those operations by putting information about our task to be performed inside a queue, which we process later. This method of deferring work to some task processor is called a task queue. Currently there are many different pieces of software designed specifically for task queues: IBM WebSphere MQ, Oracle Advanced Queuing, Java Message Service, ActiveMQ, RabbitMQ, Gearman, Amazon SQS, and others. But Redis as a general-purpose data store can be used for basic task queues.

6.2. Data Structures

Redis allows us to store keys that map to any one of four different data structure types: STRING, LIST, SET and HASH. Their implementation and semantics are similar to those same structures built in a variety of programming languages.

6.2.1. String

Strings are sequences of chars, similar to strings that other languages or other key-value stores have. If a string can be parsed as a number, operations like INCR and DECR are available.

Some Basic Commands

- `GET`: fetches the data stored at the given key.
- `SET`: sets the value stored at the given key.
- `DEL`: deletes the value stored at the given key.
- `APPEND`: appends a value to a key.
- `INCR`: increment the integer value of a key by one.
- `DECR`: decrements the integer value of a key by one.

Usage example:

```
redis> set name John
OK
redis> get name
"John"
redis> append name Doe
(integer) 7
redis> get name
"JohnDoe"
redis> del name
(integer) 1
redis> set counter 1
```

```
OK
redis> incr counter
(integer) 2
```

6.2.2. List

Lists in Redis are ordered sequences of strings implemented as linked-list structures. They allow items to be pushed or pulled from the front or back and fetched from a given position. They are the structure required to manage queues.

Some Basic Commands:

- `RPUSH`: pushes the value onto the right end of the list.
- `LPUSH`: pushes the value onto the left end of the list.
- `LRANGE`: fetches a range of values from the list.
- `LINDEX`: fetches an item at a given position in the list.
- `LPOP`: pops the value from the left end of the list and returns it.
- `RPOP`: pops the value from the right end of the list and returns it.

Usage example:

```
redis> rpush numbers two three four
(integer) 3
redis> lpush numbers one
(integer) 4
redis> lrange numbers 0 -1
1) "one"
2) "two"
3) "three"
4) "four"
redis> lindex numbers 1
"two"
redis> rpop numbers
"four"
redis> lpop numbers
"one"
redis> lrange numbers 0 -1
1) "two"
```

```
2) "three"
```

6.2.3. Set

Sets are unordered sequences of unique strings.

Some Basic Commands:

- `SADD`: adds the item to the set.
- `SMEMBERS`: returns the entire set of items.
- `SISMEMBER`: checks if an item is in the set.
- `SREM`: removes the item from the set, if it exists.
- `SUNION`: adds multiple sets.
- `SINTER`: intersects multiple sets.
- `SDIFF`: subtracts multiple sets.

Usage example:

```
redis> sadd numbers1 one two three
(integer) 3
redis> sadd numbers2 three four five
(integer) 3
redis> sinter numbers1 numbers2
1) "three"
```

Note: there is also a sorted version of Set named Zset.

6.2.4. Hash

Hashes store a mapping of keys to values. They are similar to Python dictionaries.

Some Basic Commands:

- `HSET`: stores the value at the key in the hash

- `HGET`: fetches the value at the given hash key
- `HGETALL`: fetches the entire hash
- `HDEL`: removes a key from the hash, if it exists

Usage example:

```
redis> hset person1 name John
(integer) 1
redis> hset person1 lastname Doe
(integer) 1
redis> hgetall person1
1) "name"
2) "John"
3) "lastname"
4) "Doe"
```

6.3. Redis For Task Queues

Moogle uses Redis to manage task queues. During an update session, Moogle queries providers and fetches data like Facebook status updates, Twitter tweets, etc. Data are paginated and the navigation to the next pages is managed using cursors. In most cases cursors are valid only for a short time, so the next queries must be performed rapidly: Moogle must then store the data in a temporary storage with high write performance and process them in a second stage. Redis is the right tool for this task

Furthermore this design solution decouples the activities of querying data from providers and indexing them. Decoupled systems are usually more flexible and scale better.

A traditional RDBMS system could have been used for this purpose, but Redis was chosen for three main reasons:

- no need of long term persistence: data are temporary and they are thrown away after being processed;
- data have a simple structure and they are not linked together;
- high write performance are required.

Also a truly message-oriented middleware like Celery + RabbitMQ could have been used, but Redis was chosen for two main reasons:

- tasks are only of one kind, actually they are not even real tasks but pure data (textual content to be indexed);
- no need of a complex logic to manage tasks: tasks are just pushed and pulled in traditional FIFO queues.

6.4. Data Model in Redis

Tasks stored in Redis are the main data entities described in chapter 5. Protocols And Data Entities. Those data are pushed to Redis by the crawlers, which fetch them from the providers, and they are extracted by the indexers, which send them to a full-text search engine.

A FIFO strategy for tasks management can be easily implemented in Redis using lists. But the main data entities to be stored have multiple fields (in Python code they are dictionaries), thus plain lists are not enough: they are implemented with two data structures as follows.

- list: unique identifiers of data entities are stored in a list and they are managed with a FIFO strategy;
- hash: hashes are used to store all fields of data entities; they are named after their unique identifier.

During an update session a new task list is created with the following features:

- the name of the list includes the provider name and the bearer token identifier (which stands for the user whose data we are about to index);
- the items in the list are data entities identifiers, f.i. a twitter id.

A hash for each item in the list is also created, with the following features:

- the hash name contains the bearer token identifier and the data entity identifier;
- the content of the hash is a collection of couples where the first element is the field name (f.i. `created_at`) and the second is the value (f.i. `Wed May 07 2014 14:09`).

More details about this data structure are listed in the next sections.

6.4.1. Facebook Status Update

List Format

- name: `facebook:token:<BID>`
- content: `[<PID>, <PID>, <PID>, ...]`

Where `<BID>` is the bearer token identifier owned by the user whose data we are about to index; and `<PID>` is a Facebook post identifier.

Example

```
redis> lrange facebook:token:49 0 -1
...
1118) "18807449704_10152498964779705"
1119) "18807449704_10152499457484705"
1120) "18807449704_630987226992994"
1121) "18807449704_10152498920824705"
```

Hash Format

- name: `facebook:token:<BID>:<PID>`
- content:

```
{ "type": "status",  # Or "link" or "photo" or "video".
  "message": "text",  # Text, sometimes with links.
  "message_clean": "text",  # Pure text with no links.
  "from_id": "123456780",
  "from_name": "John Doe",
  "created_time": "2014-05-23T02:41:35+0000",  # Date ISO 8601
format.
  "updated_time": "2014-05-23T02:41:35+0000"  # Date ISO 8601
format.
}
```

The meaning of `<BID>` and `<PID>` is explained in the previous section.

Example:

```
redis> hgetall facebook:token:49:18807449704_630987226992994
```

```
 1) "created_time"
 2) "2014-05-23T02:41:35+0000"
 3) "updated_time"
 4) "2014-05-23T02:41:35+0000"
 5) "from_name"
 6) "Sunrise Sunset"
 7) "message_clean"
 8) "Last day to be a backer/make a pledge on KICKSTARTER (ends Fri
1:30pm EDT) NYC musician MARK BIRNBAUM.\nAn adventure captured by
C\xc3\xa9dric B. von Sydow."
 9) "type"
10) "video"
11) "from_id"
12) "646739705417746"
13) "message"
14) "Last day to be a backer/make a pledge on KICKSTARTER (ends Fri
1:30pm EDT) NYC musician MARK BIRNBAUM.\nAn adventure captured by
C\xc3\xa9dric B. von Sydow.\n\n
https://www.kickstarter.com/projects/von-sydow/melody-memories"
```

6.4.2. Twitter Tweets

List Format

- name: `twitter:token:<BID>`
- content: `[<TID>, <TID>, <TID>, ...]`

Where `<BID>` is the bearer token identifier owned by the user whose data we are about to index; and `<TID>` is a Twitter tweet identifier.

Example

```
redis> lrange twitter:token:48 0 -1
1) "480500908783325184"
2) "480485829027303424"
3) "480470850676981761"
...
```

Hash Format

- name: `twitter:token:<BID>:<TID>`
- content:

```
{ "text": "my tweet",  # Text, sometimes with links.
  "text_clean": "my tweet",  # Pure text with no links.
  "lang": "en",  # Identified language.
  "retweeted": "False",  # True if this is a retweet.
  "created_at": "Sun Jun 22 00:01:58 +0000 2014",  # A date.
}
```

The meaning of `<BID>` and `<TID>` is explained in the previous section.

Example:

```
redis> hgetall twitter:token:48:480500908783325184
 1) "retweeted"
 2) "False"
 3) "text_clean"
 4) "1-tap root for Android, better sleep, and customized desktops:
here were our best downloads this week. "
 5) "lang"
 6) "en"
 7) "created_at"
 8) "Sun Jun 22 00:01:58 +0000 2014"
 9) "text"
10) "1-tap root for Android, better sleep, and customized desktops:
here were our best downloads this week. http://t.co/HQGVM022y6"
```

6.4.3. Dropbox Files

The data processing for Dropbox requires one more step compared to Facebook and Twitter: after the query, data entities (text files) must be downloaded before being indexed. The download process is managed using its specific task queue.

Lists Format

I. Download List

- name: `dropbox:dw:token:<BID>`
- content: `[<HT>, <HT>, <HT>, ...]`

Where `<BID>` is the bearer token identifier owned by the user whose data we are about to index, and `<HT>` is the MD5 hash of the remote path used as Dropbox file identifier.

Example

```
redis> lrange dropbox:dw:token:55 0 -1
1) "584575760c24a8d293e95f719e6a9b4e"
2) "4f5d1a3d7fd6180bb6c6b96c2b02a7f5"
3) "6af9e30311f33895a47e1f6325858a5d"
...
```

II. Index List

- name: `dropbox:ix:token:<BID>`
- content: `[<HT>, <HT>, <HT>, ...]`

The meaning of `<BID>` and `<HT>` is explained in the previous section.

Example

```
redis> lrange dropbox:ix:token:55 0 -1
1) "584575760c24a8d293e95f719e6a9b4e"
2) "4f5d1a3d7fd6180bb6c6b96c2b02a7f5"
3) "6af9e30311f33895a47e1f6325858a5d"
...
```

Hashes Format

I. Download List

- name: `dropbox:dw:token:55:<BID>:<HT>`
- content:

```
{ "operation": "+",  # If the file was added/edited, "-" if the
file was deleted.
  "local_name": "",  # Not used in the download list.
  "remote_path": "/documents/thesis.pdf"  # Full file path on
Dropbox.
}
```

The meaning of `<BID>` and `<HT>` is explained in the previous section.

Example:

```
redis> hgetall dropbox:dw:token:55:584575760c24a8d293e95f719e6a9b4e
1) "operation"
2) "+"
3) "local_name"
4) ""
5) "remote_path"
6) "/documents/thesis.pdf"
```

II. Index List

- name: `dropbox:ix:token:55:<BID>:<HT>`
- content:

```
{ "operation": "+",  # If the file was added/edited, "-" if the
file was deleted.
  "local_name": thesis.pdf",  # The file name used for the local
copy of the file which has just been downloaded.
  "remote_path": "/documents/thesis.pdf"  # Full file path on
Dropbox.
  }
```

The meaning of `<BID>` and `<HT>` is explained in the previous section.

Example:

```
redis> hgetall dropbox:ix:token:55:584575760c24a8d293e95f719e6a9b4e
1) "local_name"
2) "thesis.pdf"
3) "operation"
4) "+"
5) "remote_path"
6) "/documents/thesis.pdf"
```

CHAPTER 7

Full-Text Search Engine

"In text retrieval, full-text search refers to techniques for searching a single computer-stored document or a collection in a full text database. Full-text search is distinguished from searches based on metadata or on parts of the original texts represented in databases (such as titles, abstracts, selected sections, or bibliographical references). [...] In a full-text search, a search engine examines all of the words in every stored document as it tries to match search criteria (text specified by a user)." - Wikipedia.

Full-text search is one of the main techniques behind all web search engines and one of the keys to their success. **Apache Solr** is a popular open-source enterprise search engine that is optimized to search large volumes of text-centric data and return results sorted by relevance. The next sections describe some of the main features of Solr.

7.1. Apache Solr: Key Concepts

Full-text search engines like Solr are designed to solve a specific class of problem quite well: problems requiring the ability to search across large amounts of unstructured text and pull back the most relevant results.

They are optimized to handle data exhibiting four main characteristics:

- *Text-centric*: unstructured text data based on natural language; such search engines are specifically designed to extract the implicit structure of text into their index to improve searching;
- *Read-dominant*: data are intended to be accessed efficiently, as opposed to updated frequently by meaning that documents are read far more often than they're created or updated;
- *Document-oriented*: document is a self-contained collection of fields, in which each field only holds data and doesn't contain nested fields; so a document has a flat structure and doesn't depend on other documents;
- *Flexible schema*: documents in such search index don't need to have a uniform structure, they can have different fields;

Furthermore Solr is a NoSQL technology.
"One of the main challenges facing software architects is handling the massive volume of data consumed and produced by a huge, global user base. In addition, users expect online applications to always be available and responsive. To address the scalability and availability needs of modern web applications, we've seen a growing interest in specialized, nonrelational data storage and processing technologies, collectively known as NoSQL (Not only SQL)." - Solr In Action by T.Grainger and T.Potter, Manning, 2014.

But full-text search engines like Solr aren't general-purpose data-storage and processing solutions; they are intended instead to power keyword search, ranked retrieval, and information discovery. So Solr does complement relational and NoSQL databases rather than replace them.

7.1.1. Common Use Case

Keyword search is the most typical way users begin working with search engines. It would be rare for a user to want to fill out a complex search form initially. With keyword search users want to type in a few simple keywords and get back great results.

Keyword search sounds like a simple task of matching query terms to documents, but there are several issues that must be addressed to provide a great user experience: relevant results must be returned quickly, within a few seconds; spelling correction and autosuggestions are needed in order to improve the user input (users expect search engines to "do what I mean, not what I say"); synonyms of query terms must be recognized; documents containing linguistic variations of query terms must be matched; phrase handling is needed; stopwords like "a", "an" and "of" must be handled properly.

Another important factor is ranked retrieval which is *"a way to return top documents for a query. In an SQL query to a relational database, a row either matches a query or it doesn't, and results are sorted based on one or more of the columns. A search engine returns documents sorted in descending order by a score that indicates the strength of the match of the document to the query. How the strength of the match is calculated depends on a number of factors, but in general a higher score means the document is more relevant to the query."* - Solr In Action by T.Grainger and T.Potter, Manning, 2014.

7.1.2. Document

Solr is basically a document storage and retrieval engine. Every piece of data submitted to Solr for processing is a document. A document could be a newspaper article, a resume or social profile, or, in an extreme case, an entire book.

Each document contains one or more fields, each of which is modeled as a particular field type: string, tokenized text, Boolean, date and time, latitude and longitude, etc. Each field is defined in Solr schema as a particular field type, which allows Solr to know how to handle the content as it's received.

Although Solr has a flexible schema for each document, it's not completely schema-less. All field types must be defined, and all field names should be specified in Solr

schema.xml. This does not mean that every document must contain every field, only that all possible fields must be mappable to a particular field type should they appear in a document and need to be processed.

7.1.3. Inverted Index

Solr is built on *Apache Lucene*, a popular, Java-based, open source, information retrieval library for building and managing an inverted index: a specialized data structure for matching query terms to text-based documents.

"An inverted index (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents." - Wikipedia.
While a traditional database representation of multiple documents would contain a document's identifier mapped to one or more content fields containing all of the terms in that document, an inverted index inverts this model and maps each term in the corpus to all of the documents in which it appears.

The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database.

7.1.4. Precision And Recall

Precision and recall are the two key measures to evaluate the performance of a search engine.

The precision is the measure of the accuracy of a search result set. More technically it is a number between 0.0 and 1.0, defined as:

```
# correct matches / # total results returned
```

If all of the documents that were supposed to be returned are returned, and no more, the Precision of the query would be 1.0. If, however, all documents in the data store are returned, then the precision would be much lower. Precision is a measure of how good each of the results of a query is, but it pays no attention to how thorough it is.

Because precision only considers the overall accuracy of the results that come back and not the comprehensiveness of the result set, we need to counterbalance the precision measurement with one that takes thoroughness into account: recall.

Recall is a measure of how thorough the search results are. More technically, recall is defined as:

```
# correct matches / (# correct matches + # missed matches)
```

Precision is high when the results returned are correct while recall is high when the correct results are present. Though there is clearly tension between the two, precision and recall are not mutually exclusive and maximizing for both measures is the ultimate goal of search engines.

7.1.5. Fields, Schema.xml

Each document in a Solr index is made up of fields, and each field has a specific type that determines how it is stored, searched, and analyzed. When we consider a Twitter tweet as a document, some of its fields are: author, date, text, retweet, language. Those attributes are great fields candidates because they contain information that a typical user could use to build a query.

Identifying the right fields in documents is a key point when designing search applications. All fields in a document could be added to the index, but this leads to performance issues when dealing with large scale systems to support millions of documents and high query volumes.

One of the fields in a document must be used to uniquely identify each document in the index. In a tweet f.i. this can be the unique identifier assigned by Twitter itself. This special field is referred to as *unique key*.

Each field can be defined as *indexed* and/or *stored*. Indexed fields are those which users could use to develop a meaningful query. For instance, in a search application for books, users might want to search by title and author and hardly by editor. But still, it might be useful to display the editor's name in the search results. Such fields that aren't useful from a search perspective but are still useful for displaying search results are called stored fields. Of course, a field may be indexed and stored, such as the author and date in a search application for tweets.

In Solr a file named *schema.xml* is the place where fields and field types are defined. Solr uses the field definitions from this file to figure out what kind of text analysis needs to be performed for each fields in the documents in order to add their content as terms to the inverted search index.

Each field in schema.xml must define a type attribute that identifies the *field type* to be used for the text analysis. Also, as already previously discussed, there are no nested fields in Solr due to its flat document structure. This means that there is no relational structure which allows documents to be joined together.

Copy fields are a special structure which allows a field to be populated from one or more other fields. They are used in particular for two kind of activities: populate a single catch-all field with the contents of multiple fields; apply different text analysis to the same field content to create a new searchable field.

7.1.6. Field Types

Solr provides a number of built-in field types for structured data, such as numbers, dates, and geospatial coordinates. These fields are usually referred to as *structured nontext fields* as no text analysis is required before storing them into the inverted index.

The following diagram shows a hierarchy of the most popular field types in Solr.

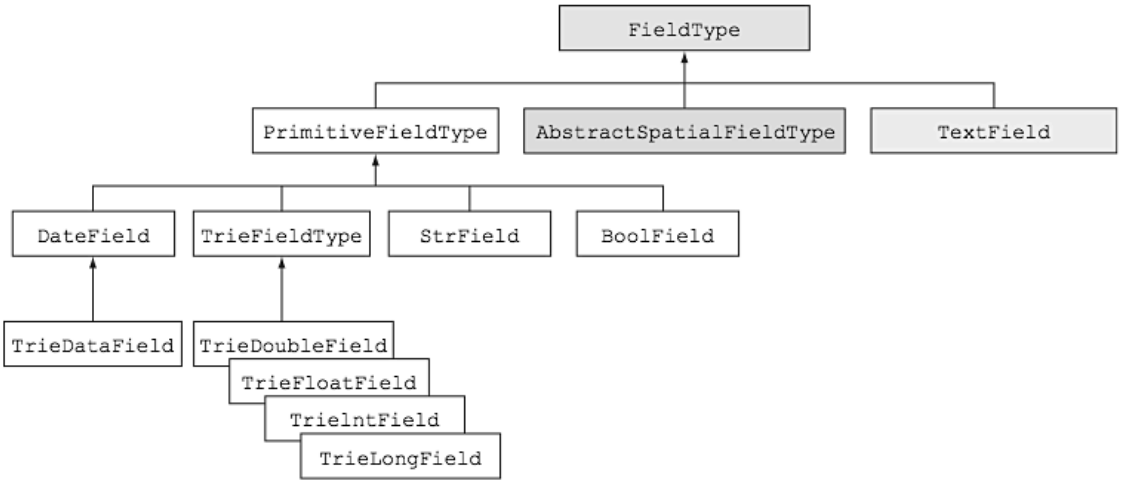

Figure 7-A Class diagram of the most commonly used field types

The string field type *StrField* is a special case: it is meant to store textual data, but which does not require text analysis. For example, the *lang* field of a tweet contains a standard ISO-639-1 language code used to identify the used language, such as `en`. Because the language code is already standardized, Solr does not need to make any change to it during indexing and query processing.

When none of the predefined field types meet the requirements, then it is possible to define new custom field type to perform a tailored text analysis.

7.1.7. Indexing Process

At a high level, the Solr indexing process is composed of three key tasks:

- Conversion of documents from their native format into a format supported by Solr (XML or JSON usually);
- Dispatch documents to Solr using one of several interfaces, typically HTTP POST;
- Text analysis and commit to the inverted index.

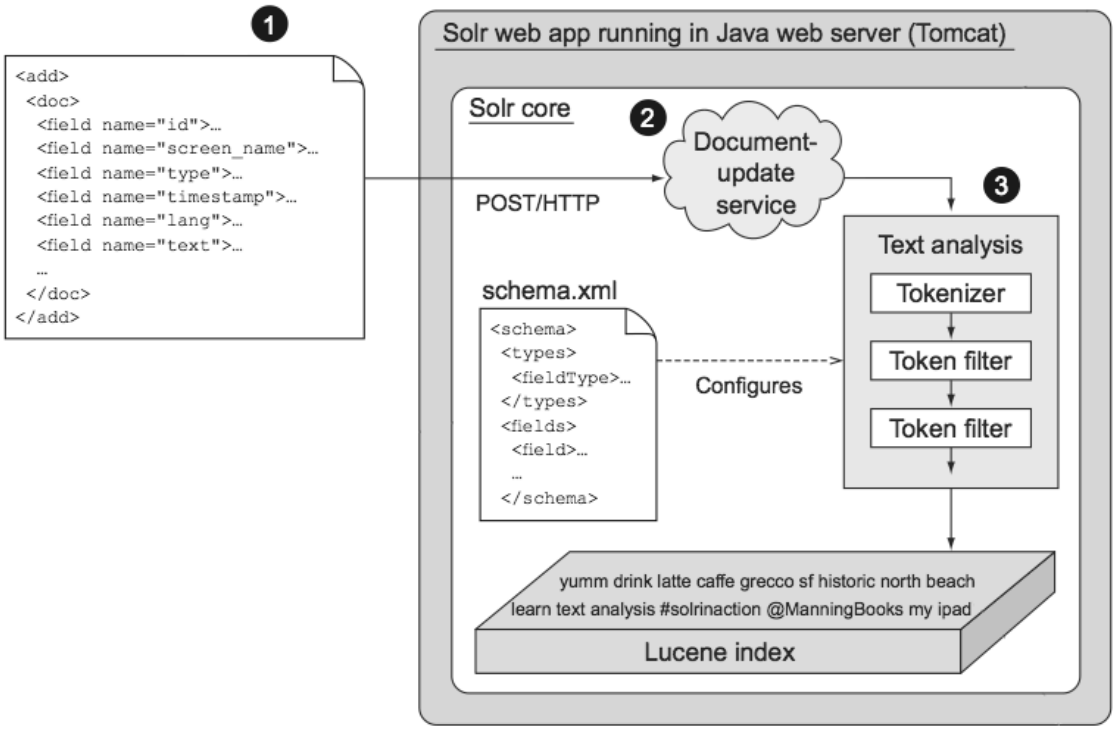

Figure 7-B High-level overview of three main steps in indexing documents

These steps are discussed in the next sections.

7.1.8. Dispatch documents

Solr supports all the most popular documents formats like XML, JSON, CSV, PDF, HTML, Microsoft Word, Open Office, etc. Documents are usually first converted to one of those formats and then sent to the document update service over HTTP POST requests.

But Solr offers also a number of powerful utilities for adding documents from other systems, most notably *Data Import Handler* and *ExtractingRequestHandler*. The Data Import Handler (DIH) is an extension that pulls data into Solr directly from external sources like websites or relational databases.

ExtractingRequestHandler, commonly called *Solr Cell*, instead allows binary files like PDF, MS Office, and OpenOffice documents to be indexed. Behind the scenes, Solr Cell uses the Apache Tika project to detect the type of document and to parse it in order to extract text and metadata.

7.1.9. Commit

When a document is added to Solr it is not visible until it's committed to the index. There are two types of commits: *hard commit* and *soft commit*.

During a hard commit Solr flushes all uncommitted documents to disk and refreshes an internal component called a searcher so that the newly committed documents can be searched and they are safely persisted to durable storage. However a hard commit can be an expensive operation that can impact query performance.

A soft commit is a mechanism to make documents searchable in near real-time by skipping the costly aspects of hard commits, such as flushing to durable storage. It is a

less expensive operation that can be executed quite often, however a hard commit is still needed at some point to ensure that documents are eventually flushed to disk.

7.1.10. Text Analysis

"Text analysis removes the linguistic variations between terms in the index and terms provided by users when searching, so that a user's query for 'buying a new house' can match a document titled 'purchasing a new home'." - Solr In Action by T.Grainger and T.Potter, Manning, 2014.

Text analysis is what allows users to find information they seek using natural language without having to think about all the possible forms of their search terms.

Solr has an extensive framework for doing basic text analysis tasks, such as removing very common words (*stop words*) and doing more complex analysis tasks.

Each field type defined in the file *schema.xml* defines the format and how fields are analyzed for indexing and queries. If all the fields in a documents collection contained structured data like language codes and timestamps, then there is no need of text analysis at all and so probably Solr is not the right tools to use. Relational databases instead are very efficient at indexing and searching such structured data; but dealing with unstructured text is what Solr is made for.

Text analysis is performed by three chained components as follows.

Analyzer

A field type must define at least one analyzer. But in practice, it's common to define two separate analyzer elements: one for indexing and another for analyzing the text entered by users when searching. This is because often additional analysis is required for

processing queries beyond what's necessary for indexing a document. For example, adding synonyms is typically done during query text analysis only to avoid inflating the size of the index and to make it easier to manage synonyms.

An analyzer is made of a single tokenizer and zero or more token filters.

Tokenizer

Each analyzer breaks the text analysis process into two phases: tokenization and token filtering.

In the tokenization phase, text is split into a stream of tokens using some form of parsing. The most basic tokenizer is a *WhitespaceTokenizer* that splits text on whitespace only. More common is *StandardTokenizer*, which performs intelligent parsing to split terms on whitespace and punctuation and correctly handles URLs, email addresses, and acronyms.

Token Filter

A token filter performs one of three actions on a token:

- Transformation: changes the token to a different form such as lowercasing all letters or stemming;
- Token injection: adds a token to the stream, as with the synonym filter;
- Token removal: removes tokens, as with the stopword filter.

Filters can be chained together to apply a series of transformations on each single token.

7.2. Schema Design

The schema file is the foundation of a great search application with Solr. The next sections illustrate the schema design for Moogle.

We have initially assumed English as the main language of documents and thus the designed text analysis is based on this assumption. As future developments we will consider adding other languages.

7.2.1. Documents Identification

Determining what a document should represent in the Solr index drives the entire schema design process. This task could be sometimes challenging, but in this case it is straightforward: the documents are the main entities identified for each provider (see Appendix A. Entities). That means tweets for Twitter, posts for Facebook and text files for Dropbox.

Documents are first converted to JSON and then sent to Solr using HTTP POST requests. Their structure is quite simple, examples are provided in the next sections.

Note that bearer token ids are used to uniquely identify the owner of a document. This is to guarantee data privacy.

Note also that Twitter and Facebook documents include a "*text_clean*" and "*message_clean*" field: this field is a copy of the main textual field with URLs removed. More details are discussed in the chapter 4. Architecture – Magpie.

Twitter Document Example

```
[
  {
    "id": "12345678901",
    "bearertoken_id": "98765",
    "lang": "en",
    "created_at": "2014-02-23T22:37:57Z",
    "retweeted": false,
    "text_original": "Check out my new snowboard at
http://t.co/bqQXwQycIU",
    "text_clean": "Check out my new snowboard"
  }
]
```

Facebook Document Example

```
[
  {
    "id": "93878937897",
    "bearertoken_id": "135348442",
    "from_id": "456567657",
    "from_name": "John Doe",
    "type": "status",
    "created_time": "2014-02-23T22:37:57Z",
    "updated_time": "2014-02-23T22:37:57Z",
    "message_original": "Check out my new snowboard at
http://www.mywebsite.com/snowboard",
    "message_clean": "Check out my new snowboard at"
  }
]
```

Dropbox Document Example

A Solr document for Dropbox is made of a text file (PDF, Microsoft Word, Open Office, ...), from which Apache Tika automatically extracts some information, plus a set of metadata.

```
[
  {
```

```
    "id": "93878937897",
    "bearertoken_id": "135348442",
    "remote_path": "/myfolder/Myfile.txt",
    "modified_at": "2014-02-23T22:37:57Z",
    "mime_type": "text/plain; charset=ISO-8859-1",
    "bytes": 562
  }
]
```

7.2.2. Multi-Provider, Multi-User

The search application we are designing must index data coming from 3 different data providers. Documents coming from different providers contain different fields and each of them requires a specific text analysis. Plus the search run by a user must be limited to the set of documents owned by the user itself. So we have to design a multi-provider and multi-user search application.

Solr Cores

Solr supports running multiple cores in a single engine; each core has a separate index and configuration. A popular use case of Solr's multicore support is data partitioning, such as having one core for recent documents and another core for older documents, known as chronological sharding.

The multicore feature is the best solution for the multi-provider requirement: using a core for each provider lets us write tailored schemas and isolate indexes.

Filter Queries

Queries are performed in Solr using two special parameters, among the other: *q* and *fq*. The *fq* parameter stands for *filter query*, and the *q* parameter stands for *query*. Both *fq* and *q* serves to limit query results to a set of matching documents, but the *q* parameter

also supplies the relevancy algorithm with a list of terms to be used for relevancy scoring.

"Separating filter queries from the main query serves two purposes. First, because filter queries are often reusable between searches (they typically do not contain arbitrary keywords), it's possible to cache their results in the filter cache [...]. Second, because the relevancy scoring operation must perform calculations on every term in the query (q) for every matching document, by separating some parts of your query into a filter query (fq), those parts in the fq parameter will not result in unnecessary additional relevancy calculations being performed." - Solr In Action by T. Grainger and T. Potter, Manning, 2014.

Thus filter queries are the best solution for the multi-user requirement: each query must specify a bearer token as filter query parameter. This limits the query to the set of data owned by that bearer token without affecting the scoring.

7.2.3. Field Types

Most of the fields in documents could be handled by built-in field type. But a new custom field type is necessary in order to perform a tailored text analysis on the main textual content of documents.

Documents from different providers require a similar text analysis; the next sections describe the general strategies applied to solve common issues while the full *schema.xml* files are listed in the Appendix C. Solr Schema.

One main custom field type is defined in the *schema.xml* file of each core. This field type is named *"tweet"* in Twitter's core, *"post"* in Facebook's core, *"dropbox_file"* in Dropbox's core and it defines two analyzers: a index analyzer and a query analyzer.

Index Analyzer

The index analyzer is meant to process data by performing a proper text analysis and splitting data into tokens to be added to the inverted index.

It is composed of the following elements:

- Pattern replace char filter: to collapse repeated letters;
- Whitespace tokenizer: to split text on white spaces;
- Word delimiter filter: to split works like "i-pad" in single tokens;
- Stop filter: to remove stop words;
- Lower case filter: to remove the case sensitivity;
- ASCII folding filter: to change diacritical marks to ASCII equivalent, like "caffè" to "caffe";
- KStem filter: to perform stemming.

The details of each element are discussed in the next sections.

Query Analyzer

The query analyzer is meant to process the input query before actually running it against the index.

It is composed of the following elements:

- Pattern replace char filter;
- Whitespace tokenizer;
- Word delimiter filter;
- Lower case filter;
- ASCII folding filter;
- Stop filter;
- Synonym filter: to inject synonyms into the query;
- KStem filter.

The details of each element are discussed in the next sections.

Pattern Replace Char Filter

A character filter is a preprocessor on an incoming stream of characters before they are passed on to the tokenizer for parsing. The pattern replace char filter is used to filter characters using regular expressions.

We want to get rid of terms with repeated letters, so popular in tweets and posts, like "goooo". In order to do so we use the regular expression `([a-zA-Z])\1+` to identify and group sequences of repeated letters (`\1` matches a repeat of the first group) and `$1$1` as replacement.

The full syntax is:

```
<charFilter class="solr.PatternReplaceCharFilterFactory" pattern="([a-zA-Z])\1+" replacement="$1$1"/>
```

Whitespace Tokenizer

Whitespace tokenizer is the most basic tokenizer that splits text on whitespace only.

Standard tokenizer instead splits text on whitespace and punctuation, preserves internet domain names and split hyphenated terms like "i-pad". Standard tokenizer is not the right choice for our purpose because it splits also on `#` and `@`, symbols used in mentions and hashtags (in Twitter and Facebook), so we must use whitespace tokenizer.

The full syntax is:

```
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
```

Word Delimiter Filter

Word delimiter filter is a great complement to the basic whitespace tokenizer. It splits a token into subwords using various parsing rules applied f.i. to words with hyphenated terms or mixed-case letters.

By default this filter also strips off the `#`, `@` and `$` characters from hashtags, mentions and symbols (which are element of tweets and posts in Facebook and Twitter). But we can customize this behavior providing a simple *"types"* mapping file with a list of character to be treated as word delimiters.

In our case the file is named `wdfftypes.txt` and the content is the following:

```
\# => ALPHA
@ => ALPHA
$ => ALPHA
```

The syntax of the word delimiter filter, together with some comments, is:

```
<!-- Split tokens into subwords based on parsing rules:
      (1 means enabled, 0 means disabled)
      * generateWordParts (1): "i-pad" to "i pad".
      * splitOnCaseChange (0): "SolrInAction" to "Solr In Action".
      * splitOnNumerics (0): "r2d2" to "r 2 d 2".
      * stemEnglishPossessive (1): "SF's" to "SF".
      * preserveOriginal (0): include the original token.
      * catenateWords (1): "i-pad" to "ipad".
      * generateNumberParts (1): "02-820" to "02 820".
      * catenateNumbers (0): "02-820" to "02820".
      * catenateAll (0): concatenate all number and word parts.
      * types: a file where we list some symbols (like #, @, $) which
must not be considered as word delimiters.
-->
<filter class="solr.WordDelimiterFilterFactory"
        generateWordParts="1"
        splitOnCaseChange="0"
        splitOnNumerics="0"
        stemEnglishPossessive="1"
        preserveOriginal="0"
        catenateWords="1"
        generateNumberParts="1"
        catenateNumbers="0"
        catenateAll="0"
        types="wdfftypes.txt"/>
```

Stop Filter

The stop filter removes stop words from the token stream during analysis. Stop words are common words like "and", "a", "or" that add little value to helping users find relevant documents. Removing stop words during indexing helps reduce the size of the index and can improve search performance as it reduces the number of documents Solr has to process and the number of terms that are scored in the relevancy calculation for queries that contain stop words.

In general, stop word removal is language-specific and Solr provides custom stop word lists for many languages.

The full syntax is:

```
<filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt"/>
```

Lower Case Filter

The lower case filter changes the letters in all tokens to lower case. This filter, applied to both the index analyzer and the query analyzer, makes it possible to match the query term "Moogle" with the indexed term "moogle" f.i..

The full syntax is:

```
<filter class="solr.LowerCaseFilterFactory"/>
```

ASCII Folding Filter

Diacritical marks like in the word "caffé" can easily break the matching among query terms and indexed terms in the same way as letter case does. For this reason Solr provides the ASCII folding filter to transform characters into their ASCII equivalents, if available. In this case the word "caffé" would be transformed to "caffe".

The full syntax is:

```
<filter class="solr.ASCIIFoldingFilterFactory"/>
```

KStem Filter

Stemming transforms words into a base form using language-specific rules. Solr provides a number of stemming filters based on different algorithms. The KStem filter is based on the Krovetz stemmer which is considered a "light" stemmer, as it makes use of inflectional linguistic morphology. The popular Porter stemmer instead is more aggressive.

The full syntax is:

```
<filter class="solr.KStemFilterFactory"/>
```

Synonym Filter

Synonym filter injects synonyms for important terms into the token stream. A common strategy is to inject synonyms only during query-time analysis in order to reduce the size of the index and to make it easier to maintain changes to the synonym list.

The full syntax is:

```
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
```

7.2.4. Fields

The fields defined for the different cores are a direct mapping of the documents identified in the previous Documents Identification section. Each field belongs to a field type which is either a built-in type or a custom type discussed in the previous sections.

The next sections list all the relevant fields used in the different cores, while the entire *schema.xml* files are in the Appendix C. Solr Schema.

Twitter Fields

```
<fields>
  ...
  <field name="id" type="string" indexed="true" stored="true"
required="true" multiValued="false"/>
  <field name="bearertoken_id" type="string" indexed="true"
stored="false"/>
  <field name="lang" type="string" indexed="true" stored="true"/>
  <field name="created_at" type="tdate" indexed="true" stored="true"/>
  <field name="retweeted" type="boolean" indexed="false"
stored="true"/>
  <field name="text_original" type="string" indexed="false"
stored="true"/>
  <field name="text_clean" type="tweet" indexed="true"
stored="false"/>
  <!-- Catch all field -->
  <field name="text" type="tweet" indexed="true" stored="false"
multiValued="true"/>
</fields>
```

Facebook Fields

```
<fields>
  ...
  <field name="id" type="string" indexed="true" stored="true"
required="true" multiValued="false"/>
  <field name="bearertoken_id" type="string" indexed="true"
stored="false"/>
  <field name="from_id" type="string" indexed="true" stored="true"/>
  <field name="from_name" type="string" indexed="true" stored="true"/>
  <field name="type" type="string" indexed="true" stored="true"/>
  <field name="created_time" type="tdate" indexed="true"
stored="true"/>
  <field name="updated_time" type="tdate" indexed="true"
stored="true"/>
```

```
  <field name="message_original" type="string" indexed="false"
stored="true"/>
  <field name="message_clean" type="post" indexed="true"
stored="false"/>
  <!-- Catch all field -->
  <field name="text" type="post" indexed="true" stored="false"
multiValued="true"/>
</fields>
```

Dropbox Fields

```
<fields>
  ...
  <field name="id" type="string" indexed="true" stored="true"
required="true" multiValued="false"/>
  <field name="bearertoken_id" type="string" indexed="true"
stored="false"/>
  <field name="remote_path" type="string" indexed="true"
stored="true"/>
  <field name="modified_at" type="tdate" indexed="true"
stored="true"/>
  <field name="mime_type" type="string" indexed="true" stored="true"/>
  <field name="bytes" type="long" indexed="true" stored="true"/>
  ...
</fields>
```

CHAPTER 8

Conclusion And Future Developments

As expected in any software engineering effort, the objectives of this work are manifested in the set of requirements identified through use-case analysis, reported in Chapter 2. By focusing on requirements, the system architecture was initially outlined and two main components identified: Magpie Web Crawler and Moogle Website. Following an iterative and incremental development cycle these components were built and all the requirements met.

The most challenging objective was the non-functional requirement that states that the system must have a designed-in capability to evolve and work with new service providers and new platforms like mobile phones. This requires the architecture to be designed using strong abstraction and inheritance among classes. Data providers use similar technologies like RESTful webservices yet the details of each API are very different f.i. in the way pagination of results and cursors updates are managed. Thus a higher level of abstract Python classes was built to define the general structure of crawlers and a lower level of specific classes was implemented for each provider.

Many innovative and interesting technologies were used. Amongst them:

- *Django Web Framework*: a free and open source web application framework, written in Python, which follows the model–view–controller architectural pattern;

- *Redis*: an open-source, networked, in-memory, key-value data store with optional durability;
- *Apache Solr*: an open source enterprise search platform from the Apache Lucene project with advanced features like full-text search, hit highlighting, faceted search, dynamic clustering, database integration, and rich document (e.g., Word, PDF) handling;
- *OAuth protocol*: an open standard protocol for authorization used by many service providers;
- *Docker*: an open-source project that automates the deployment of applications inside software containers.

As clearly stated in the introduction, this work is only a first solid version of Moogle. The project will soon be extended to include new features, in particular:

- bookmarks indexing: when a bookmarks is added to the browser, the web page is fetched and indexed by Moogle (part of this work has already be done in a project I have contributed to: https://github.com/bookieio/Bookie);
- new platforms, new providers: indexing data in mobile phones (SMS, chat messages), connecting different email services like Yahoo Mail, Outlook.com and other services like LinkedIn, GitHub;
- language identification and language-specific text analysis (now only English language is assumed to be used in every document);
- a feasibility study focused on porting the backend of Moogle to consumer electronics devices like routers, smart external hard drives and smart televisions: this would solve any concern regarding the privacy of indexed data.

APPENDIX A

Entities

Entities are the main textual pieces of information fetched from providers. They belong to users and Moogle, when explicitly authorized, can read, download and index them.

This chapter lists the main entities for all providers supported in Moogle.

A.1. Facebook

A.1.1. Status Update

```
(GET) https://graph.facebook.com/v2.0/18807449704/statuses
{
  "data": [
    {
      "id": "10152486784504705",
      "from": {
        "category": "Media/news/publishing",
        "name": "Mashable",
        "id": "18807449704"
      },
```

```
      "message": "Have a question about net neutrality or yesterday's
FCC hearing? Mashable reporter Jason Abbruzzese will be answering your
questions live for the next 30 minutes.",
      "updated_time": "2014-05-16T19:25:59+0000",
      "likes": {
        "data": [
          {
            "id": "716076371785370",
            "name": "Jordan XX XXX"
          },
          {
            "id": "68100688",
            "name": "Seif XXX"
          }
        ],
        "paging": {
          "cursors": {
            "after": "MTAyMDA5NzY3Njc4NTYyNTY=",
            "before": "NzE2MDc2MzcxNzg1Mzcw"
          },
          "next":
"https://graph.facebook.com/v2.0/10152486784504705/likes?limit=25&afte
r=MTAyMDA5NzY3Njc4NTYyNTY="
        }
      },
      "comments": {
        "data": [
          {
            "id": "10152486784504705_10152486837759705",
            "from": {
              "category": "Media/news/publishing",
              "name": "Mashable",
              "id": "18807449704"
            },
            "message": "That's all the time we have for today. Thanks
to everyone for your questions and to Jason for providing the answers!
You can read more of our net neutrality coverage here:
http://mashable.com/category/net-neutrality/",
            "can_remove": false,
            "created_time": "2014-05-16T19:58:17+0000",
            "like_count": 1,
            "user_likes": false
          },
          {
```

```
            "id": "10152486784504705_10152486820669705",
            "from": {
              "id": "10203736588760626",
              "name": "Melissa XXXX"
            },
            "message": "Sadly even signing the petition or letting
whomever know we disagree with this feels like a losing battle.",
            "can_remove": false,
            "created_time": "2014-05-16T19:48:57+0000",
            "like_count": 3,
            "user_likes": false
          }
        ],
        "paging": {
          "cursors": {
            "after": "NA==",
            "before": "Mjk="
          },
          "next":
"https://graph.facebook.com/v2.0/10152486784504705/comments?limit=25&a
fter=NA=="
        }
      }
    },
    {
      "id": "10152412203769705",
      "from": {
        "category": "Media/news/publishing",
        "name": "Mashable",
        "id": "18807449704"
      },
      "message": "Have questions about Heartbleed, the major Internet
security threat? Ask Lance Ulanoff for the next 30 minutes.",
      "updated_time": "2014-04-09T17:57:14+0000",
      "comments": {
        "data": [
          {
            "id": "10152412203769705_30454061",
            "from": {
              "id": "10101530038108253",
              "name": "Jack XXX"
            },
            "message": "Where did it get it's name, Heartbleed?",
            "can_remove": false,
```

```
            "created_time": "2014-04-09T18:29:24+0000",
            "like_count": 0,
            "user_likes": false
          }
        ],
        "paging": {
          "cursors": {
            "after": "NTU=",
            "before": "Nzk="
          },
          "next":
"https://graph.facebook.com/v2.0/10152412203769705/comments?limit=25&a
fter=NTU="
        }
      }
    },
    {
      "id": "10152261447699705",
      "from": {
        "category": "Media/news/publishing",
        "name": "Mashable",
        "id": "18807449704"
      },
      "message": "The complete guide to LinkedIn etiquette:
http://on.mash.to/1dmsFSO.",
      "updated_time": "2014-01-20T11:00:00+0000",
      "likes": {
        "data": [
          {
            "id": "1434815483453160",
            "name": "Rasel ShXXXXXakib"
          },
          {
            "id": "241566822716068",
            "name": "Niloy KhXXan"
          }
        ],
        "paging": {
          "cursors": {
            "after": "MTAxNTI1MTA4MDA1MTY1OTE=",
            "before": "MTQzNDgxNTQ4MzQ1MzE2MA=="
          },
```

```
          "next":
"https://graph.facebook.com/v2.0/10152261447699705/likes?limit=25&afte
r=MTAxNTI1MTA4MDA1MTY1OTE="
        }
      },
      "comments": {
        "data": [
          {
            "id": "10152261447699705_29521239",
            "from": {
              "id": "10202841309728293",
              "name": "Ryan XXXX"
            },
            "message": "Step 1: Avoid LinkedIn",
            "can_remove": false,
            "created_time": "2014-01-20T11:05:12+0000",
            "like_count": 13,
            "user_likes": false
          },
          {
            "id": "10152261447699705_29521326",
            "from": {
              "id": "739056782812316",
              "name": "Harun XXXX"
            },
            "message": "@Mashable, as much as I love LinkedIn, there's
a feature that I'm startled about. The ability to send a message to
several people at the same time (up to 50). Can you kindly tell me if
it's still available?? I need to know if I should just use Outlook
with the CSV-acquired contacts. ",
            "can_remove": false,
            "created_time": "2014-01-20T11:19:35+0000",
            "like_count": 0,
            "user_likes": false
          }
        ],
        "paging": {
          "cursors": {
            "after": "MQ==",
            "before": "MTg="
          }
        }
      }
    }
```

```
  ],
  "paging": {
    "previous":
"https://graph.facebook.com/v2.0/18807449704/statuses?limit=25&since=1
400268359&__paging_token=enc_AewNx0PEgi8yJS9DqSm-rdRd4BScbRcN1-dc-
7xyaJeN-IzvLejHyR7l06k5JgECOXwCUAqC0yKBrKNht-bDlCLF",
    "next":
"https://graph.facebook.com/v2.0/18807449704/statuses?limit=25&until=1
388487600&__paging_token=enc_AexOBBjbPWPrS70ZVABmfq77dNj2sCD9Qso09i9eL
ll5zPGz6htmqQbzRXd5q7eehnJpCd4r32m49cWVDSfo593F"
  }
}
```

A.1.2. Feed

```
(GET) https://graph.facebook.com/v2.0/18807449704/feed
{
  "data": [
    {
      "id": "18807449704_10152556618999705",
      "from": {
        "category": "Media/news/publishing",
        "name": "Mashable",
        "id": "18807449704"
      },
      "to": {
        "data": [
          {
            "category": "Product/service",
            "name": "Facebook",
            "id": "20531316728"
          }
        ]
      },
      "message": "Your Facebook friends have a bigger impact on you
than you think.",
      "message_tags": {
        "5": [
          {
            "id": "20531316728",
            "name": "Facebook",
```

```
            "type": "page",
            "offset": 5,
            "length": 8
          }
        ]
      },
      "picture": "https://fbexternal-
a.akamaihd.net/safe_image.php?d=AQDzTPRpg-
KiQgZp&w=154&h=154&url=http%3A%2F%2Frack.3.mshcdn.com%2Fmedia%2FZgkyMD
E0LzA2LzE4LzQ5L0ZhY2Vib29rMTEwLmIwMWFlLmpwZwpwCXRodW1iCTEyMDB4NjI3Iwpl
CWpwZw%2F981a535a%2F28b%2FFacebook-110.jpg",
      "link": "http://mashable.com/2014/06/18/facebook-feelings-
study/?utm_cid=mash-com-fb-main-link",
      "name": "TMI: Your Mood on Facebook Can Affect Your Friends",
      "caption": "mashable.com",
      "description": "A team of social scientists led by Cornell
University found that your emotions on Facebook can positively or
negatively affect your friends.",
      "icon": "https://fbstatic-
a.akamaihd.net/rsrc.php/v2/y5/r/sXJx2UP7quc.png",
      "actions": [
        {
          "name": "Comment",
          "link":
"https://www.facebook.com/18807449704/posts/10152556618999705"
        },
        {
          "name": "Like",
          "link":
"https://www.facebook.com/18807449704/posts/10152556618999705"
        }
      ],
      "privacy": {
        "value": ""
      },
      "type": "link",
      "status_type": "shared_story",
      "created_time": "2014-06-18T20:37:41+0000",
      "updated_time": "2014-06-18T20:37:41+0000",
      "shares": {
        "count": 28
      },
      "likes": {
        "data": [
```

```
          {
            "id": "859855834040282",
            "name": "Virginia XXXXXXX"
          },
          {
            "id": "262080623977802",
            "name": "Nelu XXXXXX"
          }
        ],
        "paging": {
          "cursors": {
            "after": "MjYyMDgwNjIzOTc3ODAy",
            "before": "ODU5ODU1ODM0MDQwMjgy"
          },
          "next":
"https://graph.facebook.com/v2.0/18807449704_10152556618999705/likes?l
imit=25&after=MjYyMDgwNjIzOTc3ODAy"
        }
      },
      "comments": {
        "data": [
          {
            "id": "10152556618999705_10152557354339705",
            "from": {
              "id": "10152517934707148",
              "name": "Tiina XXXXXXX"
            },
            "message": "no kidding o_O",
            "can_remove": false,
            "created_time": "2014-06-18T20:45:43+0000",
            "like_count": 0,
            "user_likes": false
          }
        ],
        "paging": {
          "cursors": {
            "after": "Mw==",
            "before": "Mw=="
          }
        }
      }
    },
    {
      "id": "18807449704_688216377899054",
```

```
      "from": {
        "category": "Media/news/publishing",
        "category_list": [
          {
            "id": "108366235907857",
            "name": "Newspaper"
          },
          {
            "id": "220974118003804",
            "name": "Military Base"
          },
          {
            "id": "177246315652372",
            "name": "Entertainer"
          }
        ],
        "name": "Salute to Heroes Newspaper",
        "id": "613622568691769"
      },
      "to": {
        "data": [
          {
            "category": "Media/news/publishing",
            "name": "Mashable",
            "id": "18807449704"
          }
        ]
      },
      "message": "http://youtu.be/H3BS0Tu8JJ4",
      "picture": "https://fbexternal-
a.akamaihd.net/safe_image.php?d=AQCy3K-
NO0KYfRwn&w=130&h=130&url=http%3A%2F%2Fi1.ytimg.com%2Fvi%2FH3BS0Tu8JJ4
%2Fmaxresdefault.jpg",
      "link": "http://youtu.be/H3BS0Tu8JJ4",
      "source":
"http://www.youtube.com/v/H3BS0Tu8JJ4?autohide=1&version=3&autoplay=1"
,
      "name": "Salute To Heroes Newpaper Presents: Sound Off",
      "description": "Salute To Heroes Newpaper Presents: Sound Off
The US Military is the BEST military in the whole world. Salute to
Heroes Newspaper, is a military newspaper se...",
      "icon": "https://fbstatic-
a.akamaihd.net/rsrc.php/v2/yj/r/v2OnaTyTQZE.gif",
      "actions": [
```

```
        {
          "name": "Comment",
          "link":
"https://www.facebook.com/18807449704/posts/688216377899054"
        },
        {
          "name": "Like",
          "link":
"https://www.facebook.com/18807449704/posts/688216377899054"
        }
      ],
      "privacy": {
        "value": ""
      },
      "type": "video",
      "created_time": "2014-06-18T18:40:31+0000",
      "updated_time": "2014-06-18T18:40:31+0000"
    },
    {
      "id": "18807449704_10152555529349705",
      "from": {
        "category": "Musician/band",
        "name": "Cantora Elisangela",
        "id": "247752215379658"
      },
      "to": {
        "data": [
          {
            "category": "Media/news/publishing",
            "name": "Mashable",
            "id": "18807449704"
          }
        ]
      },
      "message": "!!!!!!!",
      "actions": [
        {
          "name": "Comment",
          "link":
"https://www.facebook.com/18807449704/posts/10152555529349705"
        },
        {
          "name": "Like",
```

```
          "link":
"https://www.facebook.com/18807449704/posts/10152555529349705"
        }
      ],
      "privacy": {
        "value": ""
      },
      "type": "status",
      "created_time": "2014-06-17T23:03:46+0000",
      "updated_time": "2014-06-17T23:03:46+0000"
    },
    {
      "id": "18807449704_10152546315584705",
      "from": {
        "category": "University",
        "category_list": [
          {
            "id": "108051929285833",
            "name": "College & University"
          },
          {
            "id": "145118935550090",
            "name": "Medical & Health"
          }
        ],
        "name": "The University of Utah",
        "id": "7576735635"
      },
      "to": {
        "data": [
          {
            "category": "Internet/software",
            "category_list": [
              {
                "id": "177721448951559",
                "name": "Workplace & Office"
              }
            ],
            "name": "Glassdoor",
            "id": "17433690754"
          },
          {
            "category": "Media/news/publishing",
            "name": "Mashable",
```

```
            "id": "18807449704"
          }
        ]
      },
      "message": "Salt Lake City was named one of the top 10 cities
with the most satisfied employees by Glassdoor and Mashable. One of
the workplaces responsible for all this happiness that was cited in
the report: The U. Learn more at http://bit.ly/1qMmj8m",
      "message_tags": {
        "87": [
          {
            "id": "17433690754",
            "name": "Glassdoor",
            "type": "page",
            "offset": 87,
            "length": 9
          }
        ],
        "101": [
          {
            "id": "18807449704",
            "name": "Mashable",
            "type": "page",
            "offset": 101,
            "length": 8
          }
        ]
      },
      "picture": "https://fbcdn-sphotos-g-a.akamaihd.net/hphotos-ak-
xfp1/t1.0-
9/q72/s480x480/10346526_10152114366460636_7113548677516644262_n.jpg",
      "link":
"https://www.facebook.com/universityofutah/photos/a.10151577423370636.
1073741837.7576735635/10152114366460636/?type=1",
      "name": "Mobile Uploads",
      "icon": "https://fbstatic-
a.akamaihd.net/rsrc.php/v2/yz/r/StEh3RhPvjk.gif",
      "actions": [
        {
          "name": "Comment",
          "link":
"https://www.facebook.com/18807449704/posts/10152546315584705"
        }
      ],
```

```
      "privacy": {
        "value": ""
      },
      "type": "photo",
      "object_id": "10152114366460636",
      "created_time": "2014-06-14T17:24:04+0000",
      "updated_time": "2014-06-14T17:24:04+0000",
      "likes": {
        "data": [
          {
            "id": "262302543950467",
            "name": "Ashlin XXXXXXXXX"
          },
          {
            "id": "165148910166467",
            "name": "University of Utah Counseling Center"
          }
        ],
        "paging": {
          "cursors": {
            "after": "MTAyMDQyNjYwNjM3NTk3MzE=",
            "before": "MjYyMzAyNTQzOTUwNDY3"
          },
          "next":
"https://graph.facebook.com/v2.0/18807449704_10152546315584705/likes?l
imit=25&after=MTAyMDQyNjYwNjM3NTk3MzE="
        }
      },
      "comments": {
        "data": [
          {
            "id": "10152114366460636_10152116898470636",
            "from": {
              "id": "277677425744985",
              "name": "Vania xx XXXXXXXX"
            },
            "message": "Wonderful",
            "can_remove": false,
            "created_time": "2014-06-15T19:50:58+0000",
            "like_count": 0,
            "user_likes": false
          },
          {
            "id": "10152114366460636_10152114415580636",
```

```
            "from": {
              "id": "473090146158809",
              "name": "Mary XXXXXX"
            },
            "message": "I work here and I love it :)",
            "can_remove": false,
            "created_time": "2014-06-14T17:42:49+0000",
            "like_count": 0,
            "user_likes": false
          }
        ],
        "paging": {
          "cursors": {
            "after": "MQ==",
            "before": "MTI="
          }
        }
      }
    }
  ],
  "paging": {
    "previous":
"https://graph.facebook.com/v2.0/18807449704/feed?limit=25&since=14031
24013",
    "next":
"https://graph.facebook.com/v2.0/18807449704/feed?limit=25&until=14030
85599"
  }
}
```

A.2. Twitter

A.2.1. Tweet Via Search/Tweets.json

```
(GET)
https://api.twitter.com/1.1/search/tweets.json?q=%23freebandnames&sinc
e_id=24012619984051000&max_id=250126199840518145&result_type=mixed&cou
nt=4

{
  "statuses": [
    {
      "coordinates": null,
      "favorited": false,
      "truncated": false,
      "created_at": "Mon Sep 24 03:35:21 +0000 2012",
      "id_str": "250075927172759552",
      "entities": {
        "urls": [

        ],
        "hashtags": [
          {
            "text": "freebandnames",
            "indices": [
              20,
              34
            ]
          }
        ],
        "user_mentions": [

        ]
      },
      "in_reply_to_user_id_str": null,
      "contributors": null,
      "text": "Aggressive Ponytail #freebandnames",
      "metadata": {
        "iso_language_code": "en",
```

```
        "result_type": "recent"
      },
      "retweet_count": 0,
      "in_reply_to_status_id_str": null,
      "id": 250075927172759552,
      "geo": null,
      "retweeted": false,
      "in_reply_to_user_id": null,
      "place": null,
      "user": {
        "profile_sidebar_fill_color": "DDEEF6",
        "profile_sidebar_border_color": "C0DEED",
        "profile_background_tile": false,
        "name": "Sean Cummings",
        "profile_image_url":
"http://a0.twimg.com/profile_images/2359746665/1v6zfgqo8g0d3mk7ii5s_no
rmal.jpeg",
        "created_at": "Mon Apr 26 06:01:55 +0000 2010",
        "location": "LA, CA",
        "follow_request_sent": null,
        "profile_link_color": "0084B4",
        "is_translator": false,
        "id_str": "137238150",
        "entities": {
          "url": {
            "urls": [
              {
                "expanded_url": null,
                "url": "",
                "indices": [
                  0,
                  0
                ]
              }
            ]
          },
          "description": {
            "urls": [

            ]
          }
        },
        "default_profile": true,
        "contributors_enabled": false,
```

```
        "favourites_count": 0,
        "url": null,
        "profile_image_url_https":
"https://si0.twimg.com/profile_images/2359746665/1v6zfgqo8g0d3mk7ii5s_
normal.jpeg",
        "utc_offset": -28800,
        "id": 137238150,
        "profile_use_background_image": true,
        "listed_count": 2,
        "profile_text_color": "333333",
        "lang": "en",
        "followers_count": 70,
        "protected": false,
        "notifications": null,
        "profile_background_image_url_https":
"https://si0.twimg.com/images/themes/theme1/bg.png",
        "profile_background_color": "C0DEED",
        "verified": false,
        "geo_enabled": true,
        "time_zone": "Pacific Time (US & Canada)",
        "description": "Born 330 Live 310",
        "default_profile_image": false,
        "profile_background_image_url":
"http://a0.twimg.com/images/themes/theme1/bg.png",
        "statuses_count": 579,
        "friends_count": 110,
        "following": null,
        "show_all_inline_media": false,
        "screen_name": "sean_cummings"
      },
      "in_reply_to_screen_name": null,
      "source": "<a
href=\"http://itunes.apple.com/us/app/twitter/id409789998?mt=12\"
rel=\"nofollow\">Twitter for Mac</a>",
      "in_reply_to_status_id": null
    }
  ],
  "search_metadata": {
    "max_id": 250126199840518145,
    "since_id": 24012619984051000,
    "refresh_url":
"?since_id=250126199840518145&q=%23freebandnames&result_type=mixed&inc
lude_entities=1",
```

```
    "next_results":
"?max_id=249279667666817023&q=%23freebandnames&count=4&include_entitie
s=1&result_type=mixed",
    "count": 4,
    "completed_in": 0.035,
    "since_id_str": "24012619984051000",
    "query": "%23freebandnames",
    "max_id_str": "250126199840518145"
  }
}
```

A.2.2. Tweet Via Statuses/User_timeline.json

```
(GET)
https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=tw
itterapi&count=2
[
  {
    "coordinates": null,
    "favorited": false,
    "truncated": false,
    "created_at": "Wed Aug 29 17:12:58 +0000 2012",
    "id_str": "240859602684612608",
    "entities": {
      "urls": [
        {
          "expanded_url": "https://dev.twitter.com/blog/twitter-
certified-products",
          "url": "https://t.co/MjJ8xAnT",
          "indices": [
            52,
            73
          ],
          "display_url": "dev.twitter.com/blog/twitter-c\u2026"
        }
      ],
      "hashtags": [

      ],
      "user_mentions": [
```

```
      ]
    },
    "in_reply_to_user_id_str": null,
    "contributors": null,
    "text": "Introducing the Twitter Certified Products Program:
https://t.co/MjJ8xAnT",
    "retweet_count": 121,
    "in_reply_to_status_id_str": null,
    "id": 240859602684612608,
    "geo": null,
    "retweeted": false,
    "possibly_sensitive": false,
    "in_reply_to_user_id": null,
    "place": null,
    "user": {
      "profile_sidebar_fill_color": "DDEEF6",
      "profile_sidebar_border_color": "C0DEED",
      "profile_background_tile": false,
      "name": "Twitter API",
      "profile_image_url":
"http://a0.twimg.com/profile_images/2284174872/7df3h38zabcvjylnyfe3_no
rmal.png",
      "created_at": "Wed May 23 06:01:13 +0000 2007",
      "location": "San Francisco, CA",
      "follow_request_sent": false,
      "profile_link_color": "0084B4",
      "is_translator": false,
      "id_str": "6253282",
      "entities": {
        "url": {
          "urls": [
            {
              "expanded_url": null,
              "url": "http://dev.twitter.com",
              "indices": [
                0,
                22
              ]
            }
          ]
        },
        "description": {
          "urls": [
```

```
          ]
        }
      },
      "default_profile": true,
      "contributors_enabled": true,
      "favourites_count": 24,
      "url": "http://dev.twitter.com",
      "profile_image_url_https":
"https://si0.twimg.com/profile_images/2284174872/7df3h38zabcvjylnyfe3_
normal.png",
      "utc_offset": -28800,
      "id": 6253282,
      "profile_use_background_image": true,
      "listed_count": 10775,
      "profile_text_color": "333333",
      "lang": "en",
      "followers_count": 1212864,
      "protected": false,
      "notifications": null,
      "profile_background_image_url_https":
"https://si0.twimg.com/images/themes/theme1/bg.png",
      "profile_background_color": "C0DEED",
      "verified": true,
      "geo_enabled": true,
      "time_zone": "Pacific Time (US & Canada)",
      "description": "The Real Twitter API. I tweet about API changes,
service issues and happily answer questions about Twitter and our API.
Don't get an answer? It's on my website.",
      "default_profile_image": false,
      "profile_background_image_url":
"http://a0.twimg.com/images/themes/theme1/bg.png",
      "statuses_count": 3333,
      "friends_count": 31,
      "following": null,
      "show_all_inline_media": false,
      "screen_name": "twitterapi"
    },
    "in_reply_to_screen_name": null,
    "source": "<a href=\"http://sites.google.com/site/yorufukurou/\"
rel=\"nofollow\">YoruFukurou</a>",
    "in_reply_to_status_id": null
  }
]
```

A.3. Dropbox

A.3.1. Delta Entry

```
(POST) https://api.dropbox.com/1/delta
PARAMETERS: cursor=GHjghjGJH...KJh78GgjhKJHjkh
{
  "cursor": "AAGPN8T1bTQZTYXiLitF....l-mQIlqYrmGEq1IhFQg",
  "reset": true,
  "has_more": false,
  "entries": [
    [
      "/temp/moogletest",
      {
        "bytes": 0,
        "thumb_exists": false,
        "path": "/temp/moogletest",
        "is_dir": true,
        "revision": 241716,
        "root": "dropbox",
        "rev": "3b0340265d3a0",
        "icon": "folder",
        "size": "0 bytes",
        "modified": "Mon, 27 Jan 2014 20:17:24 +0000"
      }
    ],
    [
      "/temp/moogletest/io sono leggy",
      {
        "bytes": 0,
        "thumb_exists": false,
        "path": "/temp/moogletest/Io Sono Leggy",
        "is_dir": true,
        "revision": 355504,
        "root": "dropbox",
        "rev": "56cb00265d3a0",
        "icon": "folder",
        "size": "0 bytes",
        "modified": "Mon, 12 May 2014 21:47:19 +0000"
```

```
        }
      ]
    ]
  }
```

A.4. Google Drive

A.4.1. File.list

```
(GET)
https://www.googleapis.com/drive/v2/files?q=fullText+contains+%27foo%2
7
{
  "nextPageToken":
"EAIalgELEgBSjQEKgwEKXPjzo2D_____wxOWxc_nieAA_wH__kNvc21vLnVzZXIoMDAwM
DAwNTA0ZTIzYWUxZlUpLmRpcl9lbnRyeSgzNDQ5MDgzNDQ4NjNcLjdcLndhejduaFlBMGR
nKQABEGQhJFON5eTddy05AAAAAJ9cDABIOFAAWgsJ_dHv6RsExFcQASAB8ISVfAEMQAAiC
wnMg5aEcgAAACAG",
  "nextLink":
"https://www.googleapis.com/drive/v2/files?pageToken=EAIalgELEgBSjQEKg
wEKXPjzo2D_____wxOWxc_nieAA_wH__kNvc21vLnVzZXIoMDAwMDAwNTA0ZTIzYWUxZlU
pLmRpcl9lbnRyeSgzNDQ5MDgzNDQ4NjNcLjdcLndhejduaFlBMGRnKQABEGQhJFON5eTdd
y05AAAAAJ9cDABIOFAAWgsJ_dHv6RsExFcQASAB8ISVfAEMQAAiCwnMg5aEcgAAACAG&q=
fullText+contains+'foo'",
  "kind": "drive#fileList",
  "etag":
"\"AkM7BvofPa_Jxo7Kxgh76A7i7OE/DY1Mdvx4cwWjCB30hR1SvQR9_Iw\"",
  "selfLink":
"https://www.googleapis.com/drive/v2/files?q=fullText+contains+'foo'"
  "items": [
    {
      "mimeType": "application/pdf",
      "version": "6568876",
      "appDataContents": false,
      "thumbnailLink":
"https://lh4.googleusercontent.com/dKCkfil_dAYZbYyoIbNmSMj7Kd3uDbSijmd
WD-5zNKnsfkzm4hEUAoRyAGyvSXDKX1W07Q=s220",
      "labels": {
        "restricted": false,
        "starred": false,
        "viewed": true,
        "hidden": false,
        "trashed": false
      },
```

```
      "etag": "\"AkM7BvofPa_Jxo7Kxgh76A7i7OE/MTM3ODQ1MTg1NDc5MA\"",
      "lastModifyingUserName": "Nat XXXXX",
      "writersCanShare": true,
      "owners": [
        {
          "picture": {
            "url": "https://lh5.googleusercontent.com/-
fwqgMTR0A4I/AAAAAAAAAAI/AAAAAAAAAHI/PCcGdBXC7-I/s64/photo.jpg"
          },
          "kind": "drive#user",
          "isAuthenticatedUser": false,
          "displayName": "Nat XXXX",
          "permissionId": "09121083344335304860"
        }
      ],
      "id": "0B9MYC3YXZ2GiSm5u677gJHGj",
      "lastModifyingUser": {
        "picture": {
          "url": "https://lh5.googleusercontent.com/-
fwqgMTR0A4I/AAAAAAAAAAI/AAAAAAAAAHI/PCcGdBXC7-I/s64/photo.jpg"
        },
        "kind": "drive#user",
        "isAuthenticatedUser": false,
        "displayName": "Nat XXXX",
        "permissionId": "09121083344335304860"
      },
      "title": "The Tragedy of Othello - William Shakespeare.pdf",
      "ownerNames": [
        "Nat XXXX"
      ],
      "lastViewedByMeDate": "2014-01-22T19:45:04.362Z",
      "parents": [
        {
          "isRoot": false,
          "kind": "drive#parentReference",
          "id": "0B9MYC3YXZ2GiNHRDOHJOYUpYMEk",
          "selfLink":
"https://www.googleapis.com/drive/v2/files/0B9MYC3YXZ2GiSm5uQlZBSmhQd1
U/parents/0B9MYC3YXZ2GiNHRDOHJOY",
          "parentLink":
"https://www.googleapis.com/drive/v2/files/0B9MYC3YXZ2GiNHRDOHJO"
        }
      ],
      "shared": true,
```

```
      "originalFilename": "The Tragedy of Othello - William
Shakespeare.pdf",
      "webContentLink":
"https://docs.google.com/uc?id=0B9MYC3YXZ2GiSm5uQlZBSmhQd1U&export=dow
nload",
      "editable": false,
      "markedViewedByMeDate": "2014-01-21T18:32:22.870Z",
      "quotaBytesUsed": "339226",
      "modifiedDate": "2013-09-06T07:17:34.790Z",
      "createdDate": "2013-09-06T07:17:34.790Z",
      "md5Checksum": "4c6eac5bace91daf6fd0a1d0c8ca1f44",
      "iconLink":
"https://ssl.gstatic.com/docs/doclist/images/icon_10_pdf_list.png",
      "kind": "drive#file",
      "alternateLink":
"https://docs.google.com/file/d/0B9MYC3YXZ2GiSm5uQlZBSmhQd1U/edit?usp=
drivesdk",
      "copyable": true,
      "downloadUrl": "https://doc-00-0c-
docs.googleusercontent.com/docs/securesc/537ui5akpqoqmq0pgr1e7hhfjoscf
96k/bqo841r6emsjnc2q6ipu0h8bt3g4l52d/1403287200000/09121083344335304860/0243250639944594507 9/0B9MYC3YXZ2GiSm5uQlZBSmhQd1U?h=16653014193614665626&e=download&gd=true",
      "userPermission": {
        "kind": "drive#permission",
        "etag":
"\"AkM7BvofPa_Jxo7Kxgh76A7i7OE/9mvWHaophF87yLFvL63tCoXTcEA\"",
        "role": "reader",
        "type": "user",
        "id": "me",
        "selfLink":
"https://www.googleapis.com/drive/v2/files/0B9MYC3YXZ2GiSm5uQlZBSmhQd1
U/permissions/me"
      },
      "fileExtension": "pdf",
      "headRevisionId":
"0B9MYC3YXZ2GiQjdGSS9iRUVQWHIxR3JWcG1UN3lpcStsRzFNPQ",
      "selfLink":
"https://www.googleapis.com/drive/v2/files/0B9MYC3YXZ2GiSm5uQlZBSmhQd1
U",
      "fileSize": "339226"
    }
  ]
}
```

A.5. Gmail

A.5.1. Fetched Email

```
[
    (b' 24294 (X-GM-THRID 1447735411271508216 X-GM-MSGID
1447735411271508216 UID 110780
      BODY[HEADER.FIELDS (SUBJECT FROM TO DATE)] {172}',
     b'Date: Sat, 22 Dec 2007 06:28:42 -0800\r\nFrom: "Gmail Team"
       <mail-noreply@google.com>\r\nTo: "John Doe"
<johndoe@gmail.com>\r\nSubject:
       =?ISO-8859-1?Q?Gmail_is_different._This_is_all_you_have_to_\r\n
       =?ISO-8859-1?Q?know_before_you_start_using_it.?=\r\n\r\n'),

    (b' BODY[1]<0> {100}',
     b'To spice up your inbox with colors and themes, check out the
Themes tab under
       Settings'),

    (b' BODY[1.MIME] {122}',
     b'Content-Type: text/plain; charset=ISO-8859-1\r\nContent-
Transfer-Encoding:
       quoted-printable\r\nContent-Disposition: inline\r\n\r\n'),

     b')'
]
```

APPENDIX B

Providers Details

Providers are organizations that provide web services like social networks, mailboxes, file storage and synchronization services. Users utilize those services to store and share entities which are pieces of information like Twitter tweets, Facebook posts or emails.

This chapter lists the main online resources for all providers supported in Moogle.

B.1. Facebook

Application management

Facebook applications can be created and managed at:
https://developers.facebook.com/apps

Graph API

Documentation:

https://developers.facebook.com/docs/graph-api/reference
https://developers.facebook.com/docs/graph-api/using-graph-api/

Facebook Query Language (FQL) specific documentation:
https://developers.facebook.com/docs/technical-guides/fql/

Facebook provides a web tool to explore Graph API:
https://developers.facebook.com/tools/explorer

Introspection query, which returns all fields and edges of an object:
https://developers.facebook.com/docs/graph-api/using-graph-api/v2.0#introspection
Example: `/v2.0./me?metadata=1`

Python libraries

Facebook does not provide official libraries to interact with its API. But there are a bunch of third party libraries:
https://developers.facebook.com/docs/other-sdks
In particular, for Python:
https://github.com/pythonforfacebook/facebook-sdk
https://facebook-sdk.readthedocs.org/en/latest/

Note: as this library is not officially maintained by Facebook and it is not compatible with Python 3 at the time of writing, I decided not to use it.

B.2. Twitter

Application management

Twitter applications can be created and managed at:
https://dev.twitter.com/apps

REST API

Documentation:
https://dev.twitter.com/docs/api/1.1

Twitter provides a web tool to explore Graph API:
https://dev.twitter.com/console

Meaning of each field in a tweet:
https://dev.twitter.com/docs/platform-objects/tweets

Python libraries

Twitter does not provide official libraries to interact with its API. But there are a bunch of third party libraries:
https://dev.twitter.com/docs/twitter-libraries
In particular, for Python:
https://github.com/sixohsix/twitter

Note: as none of these libraries are officially maintained by Twitter, I decided not to use them.

B.3. Dropbox

Application management

Dropbox applications can be created and managed at:
https://www.dropbox.com/developers/apps

Core API

Documentation:
https://www.dropbox.com/developers/core/docs

Official Python library

Documentation:
https://www.dropbox.com/developers/core/docs/python

B.4. Google

Application management

Google applications can be created and managed at:
https://cloud.google.com/console/project

API

APIs for all services:
https://developers.google.com/apis-explorer/#p/

Google Drive API:
https://developers.google.com/drive/v2/reference/
Google Drive search endpoint:
https://developers.google.com/drive/v2/reference/files/list

Google Gmail API:
https://developers.google.com/gmail/oauth_overview
https://developers.google.com/gmail/xoauth2_protocol

Google provides a web tool to explore its API:
https://developers.google.com/oauthplayground/

Official Python library

Google has a big set of official libraries:
https://developers.google.com/discovery/libraries
In particular, for Python:
https://developers.google.com/api-client-library/python/

Note: as the official Python library does not support Gmail API, I decided not to use it.

APPENDIX C

Solr Schema

Excerpts of the schema.xml files for the 3 Solr cores.

C.1. Twitter

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="twitter" version="1.5">
  <fields>
    <field name="_version_" type="long" indexed="true" stored="true"/>
    <field name="_root_" type="string" indexed="true" stored="false"/>
    <field name="id" type="string" indexed="true" stored="true"
required="true" multiValued="false"/>
    <field name="bearertoken_id" type="string" indexed="true"
stored="false"/>
    <field name="lang" type="string" indexed="true" stored="true"/>
    <field name="created_at" type="tdate" indexed="true"
stored="true"/>
    <field name="retweeted" type="boolean" indexed="false"
stored="true"/>
    <field name="text_original" type="string" indexed="false"
stored="true"/>
    <field name="text_clean" type="tweet" indexed="true"
stored="false"/>
    <!-- <field name="urls" type="string" indexed="true"
stored="false" multiValued="true"/> -->
    <!-- Catch all field -->
```

```
    <field name="text" type="tweet" indexed="true" stored="false"
multiValued="true"/>
  </fields>
  <uniqueKey>id</uniqueKey>
  <copyField source="text_clean" dest="text"/>
  <!-- <copyField source="urls" dest="text"/> NO!!! We don't want text
analysis on urls -->
  <types>
    ...
    <!-- positionIncrementGap: prevent phrase queries from matching
the end of one value and the beginning
         of the next value in multivalued fields -->
    <fieldType name="tweet" class="solr.TextField"
positionIncrementGap="100">
      <analyzer type="index">
        <!-- Collapse repeated letters to a max of 2, e.g. "yummmm" to
"yumm" -->
        <charFilter class="solr.PatternReplaceCharFilterFactory"
                    pattern="([a-zA-Z])\1+"
                    replacement="$1$1"/>
        <!-- Split only on whitespaces, so it preserves hashtags,
mentions, hypheneted terms -->
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- Split tokens into subwords based on parsing rules:
              (1 means enabled, 0 means disabled)
              * generateWordParts (1): "i-pad" to "i pad"
              * splitOnCaseChange (0): "SolrInAction" to "Solr In
Action"
              * splitOnNumerics (0): "r2d2" to "r 2 d 2"
              * stemEnglishPossessive (1): "SF's" to "SF"
              * preserveOriginal (0): include the original token
              * catenateWords (1): "i-pad" to "ipad"
              * generateNumberParts (1): "02-820" to "02 820"
              * catenateNumbers (0): "02-820" to "02820"
              * catenateAll (0): concatenate all number and word parts
              * types: a file where we list some symbols (like #, @,
$) which must not be considered as word delimiters.
        -->
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1"
                splitOnCaseChange="0"
                splitOnNumerics="0"
                stemEnglishPossessive="1"
                preserveOriginal="0"
```

```
                catenateWords="1"
                generateNumberParts="1"
                catenateNumbers="0"
                catenateAll="0"
                types="wdfftypes.txt"/>
        <!-- Remove English stopwords -->
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"/>
        <!-- Change case to lower case -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- Change diacritical marks to ASCII equivalent, e.g.
"caffè" to "caffe" -->
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <!-- Stemming for English -->
        <filter class="solr.KStemFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <!-- Collapse repeated letters to a max of 2, e.g. "yummmm" to
"yumm" -->
        <charFilter class="solr.PatternReplaceCharFilterFactory"
                    pattern="([a-zA-Z])\1+"
                    replacement="$1$1"/>
        <!-- Split only on whitespaces, so it preserves hashtags,
mentions, hypheneted terms -->
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- See comment on index analyzer -->
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1"
                splitOnCaseChange="0"
                splitOnNumerics="0"
                stemEnglishPossessive="1"
                preserveOriginal="0"
                catenateWords="1"
                generateNumberParts="1"
                catenateNumbers="0"
                catenateAll="0"
                types="wdfftypes.txt"/>
        <!-- Change case to lower case -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- Change diacritical marks to ASCII equivalent, e.g.
"caffè" to "caffe" -->
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <!-- Remove English stopwords -->
```

```
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"/>
        <!-- Inject synonyms -->
        <filter class="solr.SynonymFilterFactory"
                synonyms="synonyms.txt"
                ignoreCase="true"
                expand="true"/>
        <!-- Stemming for English -->
        <filter class="solr.KStemFilterFactory"/>
      </analyzer>
    </fieldType>
  </types>
</schema>
```

C.2. Facebook

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="facebook" version="1.5">
  <fields>
    <field name="_version_" type="long" indexed="true" stored="true"/>
    <field name="_root_" type="string" indexed="true" stored="false"/>
    <field name="id" type="string" indexed="true" stored="true"
required="true"
           multiValued="false"/>
    <field name="bearertoken_id" type="string" indexed="true"
stored="false"/>
    <field name="from_id" type="string" indexed="true" stored="true"/>
    <field name="from_name" type="string" indexed="true"
stored="true"/>
    <field name="type" type="string" indexed="true" stored="true"/>
    <field name="created_time" type="tdate" indexed="true"
stored="true"/>
    <field name="updated_time" type="tdate" indexed="true"
stored="true"/>
    <field name="message_original" type="string" indexed="false"
stored="true"/>
    <field name="message_clean" type="post" indexed="true"
stored="false"/>
    <!-- <field name="urls" type="string" indexed="true"
stored="false" multiValued="true"/> -->
    <!-- Catch all field -->
    <field name="text" type="post" indexed="true" stored="false"
multiValued="true"/>
  </fields>
  <uniqueKey>id</uniqueKey>
  <copyField source="message_clean" dest="text"/>
  <!-- <copyField source="urls" dest="text"/> NO!!! We don't want text
analysis on urls -->
  <types>
    ...
    <!-- positionIncrementGap: prevent phrase queries from matching
the end of one value and the beginning
         of the next value in multivalued fields -->
    <fieldType name="post" class="solr.TextField"
positionIncrementGap="100">
      <analyzer type="index">
```

```
        <!-- Collapse repeated letters to a max of 2, e.g. "yummmm" to
"yumm" -->
        <charFilter class="solr.PatternReplaceCharFilterFactory"
                    pattern="([a-zA-Z])\1+"
                    replacement="$1$1"/>
        <!-- Split only on whitespaces, so it preserves hashtags -->
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- Split tokens into subwords based on parsing rules:
              (1 means enabled, 0 means disabled)
              * generateWordParts (1): "i-pad" to "i pad"
              * splitOnCaseChange (0): "SolrInAction" to "Solr In
Action"
              * splitOnNumerics (0): "r2d2" to "r 2 d 2"
              * stemEnglishPossessive (1): "SF's" to "SF"
              * preserveOriginal (0): include the original token
              * catenateWords (1): "i-pad" to "ipad"
              * generateNumberParts (1): "02-820" to "02 820"
              * catenateNumbers (0): "02-820" to "02820"
              * catenateAll (0): concatenate all number and word parts
              * types: a file where we list some symbols (like #, @,
$) which must not be considered as word delimiters.
        -->
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1"
                splitOnCaseChange="0"
                splitOnNumerics="0"
                stemEnglishPossessive="1"
                preserveOriginal="0"
                catenateWords="1"
                generateNumberParts="1"
                catenateNumbers="0"
                catenateAll="0"
                types="wdfftypes.txt"/>
        <!-- Remove English stopwords -->
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"/>
        <!-- Change case to lower case -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- Change diacritical marks to ASCII equivalent, e.g.
"caffè" to "caffe" -->
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <!-- Stemming for English -->
        <filter class="solr.KStemFilterFactory"/>
```

```
      </analyzer>
      <analyzer type="query">
        <!-- Collapse repeated letters to a max of 2, e.g. "yummmm" to
"yumm" -->
        <charFilter class="solr.PatternReplaceCharFilterFactory"
                    pattern="([a-zA-Z])\1+"
                    replacement="$1$1"/>
        <!-- Split only on whitespaces, so it preserves hashtags -->
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- See comment on index analyzer -->
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1"
                splitOnCaseChange="0"
                splitOnNumerics="0"
                stemEnglishPossessive="1"
                preserveOriginal="0"
                catenateWords="1"
                generateNumberParts="1"
                catenateNumbers="0"
                catenateAll="0"
                types="wdfftypes.txt"/>
        <!-- Change case to lower case -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- Change diacritical marks to ASCII equivalent, e.g.
"caffè" to "caffe" -->
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <!-- Remove English stopwords -->
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"/>
        <!-- Inject synonyms -->
        <filter class="solr.SynonymFilterFactory"
                synonyms="synonyms.txt"
                ignoreCase="true"
                expand="true"/>
        <!-- Stemming for English -->
        <filter class="solr.KStemFilterFactory"/>
      </analyzer>
    </fieldType>
  </types>
</schema>
```

C.3. Dropbox

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="dropbox" version="1.5">
  <fields>
    <field name="_version_" type="long" indexed="true" stored="true"/>
    <field name="_root_" type="string" indexed="true" stored="false"/>
    <!-- Dropbox specific fields. -->
    <field name="id" type="string" indexed="true" stored="true"
required="true"
           multiValued="false"/>
    <field name="bearertoken_id" type="string" indexed="true"
stored="false"/>
    <field name="remote_path" type="string" indexed="true"
stored="true"/>
    <field name="modified_at" type="tdate" indexed="true"
stored="true"/>
    <field name="mime_type" type="string" indexed="true"
stored="true"/>
    <field name="bytes" type="long" indexed="true" stored="true"/>
    <!-- Case insensitive version of remote_path and mime_type.
         Keep in mind that "Dropbox treats file names in a case-
insensitive but
         case-preserving way. To facilitate this, the path strings
above are lower-cased
         versions of the actual path. The metadata dicts have the
original, case-preserved
         path."

https://www.dropbox.com/developers/core/docs/python#DropboxClient (see
delta() method)
    -->
    <field name="remote_path_ci" type="string_case_insensitive"
indexed="true" stored="false"/>
    <!-- Common metadata fields, named specifically to match up with
      SolrCell metadata when parsing rich documents such as Word, PDF.
      Some fields are multiValued only because Tika currently may
return
      multiple values for them. Some metadata is parsed from the
documents,
      but there are some which come from the client context:
        "content_type": From the HTTP headers of incoming stream
```

```
        "resourcename": From SolrCell request param resource.name
    -->
    <field name="title" type="dropbox_file" indexed="true"
stored="true" multiValued="true"/>
    <field name="subject" type="dropbox_file" indexed="true"
stored="true"/>
    <field name="description" type="dropbox_file" indexed="true"
stored="true"/>
    <field name="comments" type="dropbox_file" indexed="true"
stored="true"/>
    <field name="author" type="text_general" indexed="true"
stored="true"/>
    <field name="keywords" type="dropbox_file" indexed="true"
stored="true"/>
    <field name="category" type="dropbox_file" indexed="true"
stored="true"/>
    <field name="resourcename" type="dropbox_file" indexed="true"
stored="true"/>
    <field name="url" type="text_general" indexed="true"
stored="true"/>
    <field name="content_type" type="string" indexed="true"
stored="true" multiValued="true"/>
    <field name="last_modified" type="date" indexed="true"
stored="true"/>
    <field name="links" type="string" indexed="true" stored="true"
multiValued="true"/>
    <!-- Main body of document extracted by SolrCell.
         NOTE: This field is not indexed by default, since it is also
copied to "text"
         using copyField below. This is to save space. Use this field
for returning and
         highlighting document content. Use the "text" field to search
the content. -->
    <field name="content" type="dropbox_file" indexed="false"
stored="true"
           multiValued="true"/>
    <!-- Catchall field, containing all other searchable text fields
(implemented
         via copyField further on in this schema  -->
    <field name="text" type="dropbox_file" indexed="true"
stored="false" multiValued="true"/>
  </fields>
  <uniqueKey>id</uniqueKey>
  <!-- Dropbox specific fields. -->
```

```
  <copyField source="remote_path" dest="text"/>
  <copyField source="remote_path" dest="remote_path_ci"/>
  <!-- Text fields from SolrCell to search by default in our catch-all
field -->
  <copyField source="title" dest="text"/>
  <copyField source="author" dest="text"/>
  <copyField source="description" dest="text"/>
  <copyField source="keywords" dest="text"/>
  <copyField source="content" dest="text"/>
  <copyField source="content_type" dest="text"/>
  <copyField source="resourcename" dest="text"/>
  <copyField source="url" dest="text"/>
  <!-- Create a string version of author for faceting -->
  <copyField source="author" dest="author_s"/>
  <types>
    ...
    <!-- Dropbox specific type -->
    <!-- positionIncrementGap: prevent phrase queries from matching
the end of one value and the beginning
         of the next value in multivalued fields -->
    <fieldType name="dropbox_file" class="solr.TextField"
positionIncrementGap="100"
             autoGeneratePhraseQueries="true">
      <analyzer type="index">
        <!-- Collapse repeated letters to a max of 2, e.g. "yummmm" to
"yumm" -->
        <charFilter class="solr.PatternReplaceCharFilterFactory"
                    pattern="([a-zA-Z])\1+"
                    replacement="$1$1"/>
        <!-- Split only on whitespaces, so it preserves hashtags -->
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- Split tokens into subwords based on parsing rules:
              (1 means enabled, 0 means disabled)
              * generateWordParts (1): "i-pad" to "i pad"
              * splitOnCaseChange (0): "SolrInAction" to "Solr In
Action"
              * splitOnNumerics (0): "r2d2" to "r 2 d 2"
              * stemEnglishPossessive (1): "SF's" to "SF"
              * preserveOriginal (0): include the original token
              * catenateWords (1): "i-pad" to "ipad"
              * generateNumberParts (1): "02-820" to "02 820"
              * catenateNumbers (0): "02-820" to "02820"
              * catenateAll (0): concatenate all number and word parts
```

```
             * types: a file where we list some symbols (like #, @,
$) which must not be considered as word delimiters.
        -->
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1"
                splitOnCaseChange="1"
                splitOnNumerics="0"
                stemEnglishPossessive="1"
                preserveOriginal="0"
                catenateWords="1"
                generateNumberParts="1"
                catenateNumbers="1"
                catenateAll="0"/>
        <!-- Remove English stopwords -->
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"/>
        <!-- Change case to lower case -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- Change diacritical marks to ASCII equivalent, e.g.
"caffè" to "caffe" -->
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <!-- Stemming for English -->
        <filter class="solr.KStemFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <!-- Collapse repeated letters to a max of 2, e.g. "yummmm" to
"yumm" -->
        <charFilter class="solr.PatternReplaceCharFilterFactory"
                    pattern="([a-zA-Z])\1+"
                    replacement="$1$1"/>
        <!-- Split only on whitespaces, so it preserves hashtags -->
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- See comment on index analyzer -->
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1"
                splitOnCaseChange="0"
                splitOnNumerics="0"
                stemEnglishPossessive="1"
                preserveOriginal="0"
                catenateWords="1"
                generateNumberParts="1"
                catenateNumbers="0"
                catenateAll="0"/>
```

```
        <!-- Change case to lower case -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- Change diacritical marks to ASCII equivalent, e.g.
"caffè" to "caffe" -->
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <!-- Remove English stopwords -->
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"/>
        <!-- Inject synonyms -->
        <filter class="solr.SynonymFilterFactory"
                synonyms="synonyms.txt"
                ignoreCase="true"
                expand="true"/>
        <!-- Stemming for English -->
        <filter class="solr.KStemFilterFactory"/>
      </analyzer>
    </fieldType>
    <!-- Dropbox specific type. -->
    <!-- String filed case insensitive.
         Keep in mind that "Dropbox treats file names in a case-
insensitive but
         case-preserving way. To facilitate this, the path strings
above are lower-cased
         versions of the actual path. The metadata dicts have the
original, case-preserved
         path."

https://www.dropbox.com/developers/core/docs/python#DropboxClient (see
delta() method)
    -->
    <fieldType name="string_case_insensitive" class="solr.TextField"
positionIncrementGap="100">
      <analyzer type="index">
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldType>
  </types>
</schema>
```

Bibliography

ALCHIN, M. 2013. Pro Django. Apress, New York.

AMBLER, S. 2002. Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. John Wiley & Sons, Inc., New York.

ATZENI, P. CERI, S. PARABOSCHI, S., TORLONE, R. 2014. Basi di dati - Modelli e linguaggi di interrogazione, 4a edizione. McGraw-Hill, Milan.

BECK, K. 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, Mass.

BECK, K., AND ANDRES, C. 2004. Extreme Programming Explained: Embrace Change, 2nd ed. Addison-Wesley, Reading, Mass.

BECK, K., ET AL. 2001. Manifesto for agile software development. http://agilemanifesto.org (July 2014).

BISHT, S. 2013. Robot Framework Test Automation. Packt Publishing, Birmingham, UK.

BOOCH, G. 1991. Object-Oriented Design with Applications. Benjamin/Cummings, Redwood City, CA.

BOOCH, G. 1994. Object Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City, CA.

BOOCH, G., MARTIN, R. C., AND NEWKIRK, J. 1998. Object Oriented Analysis and Design with Applications, 2nd ed. Addison Wesley, Reading, Mass.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. Unified Modeling Language - User's Guide. Addison-Wesley, Reading, Mass.

BROWN, W. J., MALVEAU, R. C., MCCORMICK, H., AND MOWBRAY, T. 1998. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York, NY.

CARLSON, J. 2013. Redis in Action. Manning Publications, NY.

COCKBURN, A. 2001. Agile Software Development: Software through People. Addison-Wesley, Reading, Mass.

COPELAND, R. 2008. Essential SQLAlchemy. O'Reilly Media, Sebastopol, CA.

FOWLER, M. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, Mass.

FOWLER, M. 2004. Model Driven Architecture. Published on the Web at: http://martinfowler.com/bliki/ModelDrivenArchitecture.html (July 2014).

FOWLER, M., PARSONS, R. 2008. The ThoughtWorks Anthology: Essays on Software Technology and Innovation (Pragmatic Programmers). Pragmatic Bookshelf, USA.

FREDERICK P. BROOKS, JR. 1995. The Mythical Man-Month: Essays on Software Engineering Anniversary Edition. Addison Wesley, Reading, Mass.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading, Mass.

GRAINGER, T., POTTER, T. 2014. Solr in Action. Manning Publications, NY.

GREENFELD, D., ROY, A. 2014. Two Scoops of Django: Best Practices For Django 1.6. Two Scoops Press, USA.

INGERSOLL, G.S., MORTON, T.S., FARRIS, A.L. 2013. Taming Text - How To Find, Organize, And Manipulate It. Manning Publications, NY.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, G. 1999. Unified Software Development Process. Addison-Wesley, Reading, Mass.

KROLL, P., AND KRUCHTEN, P. 2003. The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process. Addison-Wesley, Reading, Mass.

KRUCHTEN, P. 2003. Rational Unified Process: An Introduction, 3rd ed. Addison-Wesley, Reading, Mass.

KRUCHTEN, P. 1995. The "4+1" View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50. https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf (July 2014).

KUC, R. 2013. Apache Solr 4 Cookbook. Packt Publishing, Birmingham, UK.

LUTZ, M. 2013. Learning Python, 5th Edition. O'Reilly Media, Sebastopol, CA.

LUTZ, M. 2011. Programming Python, 4th Edition. O'Reilly Media, Sebastopol, CA.

LUTZ, M. 2014. Python Pocket Reference, 5th Edition. O'Reilly Media, Sebastopol, CA.

MARTIN, C.R., MICAH, M. 2006. Agile Principles, Patterns, and Practices in C#. Prentice Hall, USA.

MCCANDLESS, M., HATCHER, E., GOSPODNETIC, O. 2010. Manning Publications, NY.

PERCIVAL, H.J.W. 2014. Test-Driven Development with Python. O'Reilly Media, Sebastopol, CA.

PHILLIPS, D. 2010. Python 3 Object Oriented Programming. Packt Publishing, Birmingham, UK.

PILGRIM, M. 2009. Dive Into Python 3. Apress, USA. http://www.diveintopython3.net (July 2014).

PRESSMAN, R. S. 2004. Software Engineering: A Practitioner's Approach, 6th ed. McGraw-Hill, New York, NY.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ.

RUSSELL, M. 2014. Mining the Social Web. 2014. O'Reilly Media, Sebastopol, CA.

SOMMERVILLE, I. 2004. Software Engineering, 7th ed. Addison-Wesley, Reading, Mass.

SHEIKO, D. 2013. Instant Testing with QUnit. Packt Publishing, Birmingham, UK.

TURNQUIST, G. L. 2011. Python Testing Cookbook. Packt Publishing, Birmingham, UK.

UNMESH, G. 2013. Selenium Testing Tools Cookbook. Packt Publishing, Birmingham, UK.

Webber, J., Parastatidis, S., Robinson, I. 2010. REST in Practice. O'Reilly Media, Sebastopol, CA.

WELLS, D. 2003. Extreme programming: A gentle introduction. Published on the Web at: http://www.extremeprogramming.org (July 2014).

YOUNKER, J. 2008. Foundations of Agile Python Development. Apress, USA.