Architecture and Design of Data Lake core [1]

Master I Internship

Le Nhu Chu Hiep

July 2, 2022

[1] ICTLab - Dr.TRAN Giang Son

Contents

Acknowledgement

First and for all, I want to thanks Dr. TRAN Giang Son for allowing to join on his Data Lake project a teammember and supporting me a lots during this project development. I also want to thanks Mrs. Hoang Thi Van Anh to always help us on solving the master problem during this covid time so that I and my friend could fully focus on our internship project. Finally, I would like to spend my thanks to all ICTLab memeber for their kindness, enthusiasm on their academy jobs even in this most difficult time, I learned a lot experience from that.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Information, Communication and Technology Laboratory (ICTLab) is a research laboratory joined between USTH and its Vietnamese/France partners. ICTLab research objective focus on building and applying computer models for the assessment and management of complex environmental systems as the way to inform and support human decision. Collecting, storing and sharing data is a common phrase in the most of ICTLab research project and also cause the cost of time in preparing and sharing process especially when the data is pretty large in scale. Moreover, large data in different project and between different researcher normally duplicates multiple times because of difficult in managing same resource in normal file system which lead to waste of cluster storage space. Therefore, a demand on having a proper storage management tool to store, control and share data in ICTLab in an effective way is raised up. [7]

Data Lake system known as a repository centralization system for storing and managing data in raw format. First times discussed by James Dixon - the chief of technology officer at Pentaho, Data Lake was introduced as an alternative solution for limitation of traditional data warehouse especially with information siloing issues. Unlike traditional system, Data Lake allow to store and analyze a large number of data not only structured data but also semi and unstructured data in centralization model. Data Lake is supported by many big guys in cloud service domain (like Amazone, Google or Cloudera) and used by many company to manage their own data including company user sensitive information. Behind the scene of idea, common core of Data Lake is a centralized repository applying storage distributed technology with a well design metadata management system that allows to store, control and audit data in scale. By having the power in idea but simple in basic design, Data Lake is great suitable hint for solving ICTLab problem in managing data. [5]

1.2 Internship Objective

Although building Data Lake in general is not complicated, the question on an design satisfying scalable property as well as preventing Data Lake to become an Data Swarm, an useless data repository because of lacking searching method for target user, is not easy. A

micro-service style is suitable solution for these problem which support system on scale both horizontal and vertical direction. By designing small and independent service working together, the system could run on one or multiple cluster sharing same network and add as much as possible number of power node to execute and expand system. Each service itself includes multiple clone function nodes for distributing processing throughput and preventing system down with single node failure. It is also good point when new feature could be developed and attached to system pretty simple in this style.

Leading by Dr. TRAN Giang Son, the developer team, working on Data Lake project in ICTLab, aim on a prove of concept design and implementation of a basic Data Lake oriented micro-service. The core services which store and manage data in lake are combination of many ideas and complicate system interaction and will not be discussed in this internship. The main internship objective focus on **ingestion service** - a small service with role of gateway for putting data into system. Despite of small role, this service itself is still a bit big and ambiguous, hence, the real and explicit objective of internship is a crawler system, a kind of ingestion service, that fetch and put data into lake storage.

More specific, the crawler system will be designed to work with API (application programming interface) instead of site HTML because of simple usage and population of API in current data sources.

1.3 Thesis Organization

After this Introduction chapter, next chapters will discussed more about crawler system and its circumstance information.

- chapter 2: Background, more information on term and technology used in system
- chapter 3: Methodology, design and method of system
- chapter 4: Libraries, list of important libraries supporting system
- chapter 5: Conclusion, summary result and future work

Chapter 2

Background

This chapter provide useful information on the main concern of this internship: Data Lake Ingestion, API and Crawler. Chapter is expected to provide a clear picture on each topic including issues that developer will face when building system.

2.1 Data Lake Ingestion

For a data relating system, ingestion is an initial and important phase to put data into store zone. The correction and useful of further system process depending on the correctness and meaningful of data in ingestion phase. The **right** data ingested strongly relate to type and target of system consuming it. For instance, a database management system will require input data is table like coming with metadata about source format and destination detail (database, table, etc.) or file system allow put data as a block of file with minimum filename requested.

Data Lake, specially the basic one mentioned in this reports, work close to a file centralization system with some extra data processing on top and the data ingestion expected is the raw octet-stream plus optional metadata that could be provided as an JSON string. Since the key idea of this system about managing and processing a big data and highly depending on metadata to drive any upper process than storing, the most important data part ironically is metadata despite of the fact this is only optional. If the metadata is not enriched enough from beginning, the cost of exploiting time during processing plus potential of missing track on data is dramatically high and would be more worsted in scale system. Understanding the important of metadata with data life cycle in Data Lake, ingestion service performing ingestion phase need to have proper design with high priority factor concentrate on the way metadata provided and possibly, simple and flexible are necessary non-functional properties in here.

There are many type of ingestion service depending on source and attribute of data. The non-stop tracking sensor as security camera or temperature measurement provide a real time stream with less to miss metadata relating to stream content, on the other hand, traditional enterprise database will provide a huge batch of data with carefully enriched metadata and sometime could be considered to processed data but require an active method to connect and capture. Fortunately, many companies containing second type resource do not force

direct connection but provide an simple access interface commonly through HTTP/HTTPS (such as API most time or HTML website) for reaching their own data. The crawler system, main objective in report, is designed to capture the resource through this type of interface.

2.2 API

Application programming interface (API) is actually an interface type to connect between pieces to pieces of program or offer service for another program. Unlike user interface which allows user to interact with software, API aims to a rich information with less representation style communication for increasing effective in data transmission on the connection. This type of interface, normally, is difficult for human end user but is preferred by developer to design multiple software incorporation.

In history, the API is older than the term itself since the first usage started from 1940s and 50s with the name "library catalog" used by Maurice Wilkes and David Wheeler, API have huge developing steps from official record with term in 1960s and 70s as well as applied in database design in same years then became web interface in 1990s during internet expansion time and evolved diverse style in 2000s which still be used until now (REST API is a famous style born in this time). [2]

Remote Procedure Call (RPC) is a type of API transfer binary data with data schema predefined between source and destination program. RPC is an effective type for the communication requesting fast, low latency and minimum in throughput. However, constraint of schema leads to miss flexibility, difficult on modifying/updating and hence is not suitable for web system which is normally dynamic, flexible and changes frequently. Another kind of API uses plain text to transfer data following convention format. This type API is more preferred in web system since it is flexible and not break down system even if the server update data schema as long as used piece does not change although the transfer cost is higher than RPC. Additional information, old format used in API transfer by plain text (normally through http/https protocol) was XML but dominated now is JSON and new format is actually less wasted space than old in current internet situation.

After the explosion of internet, website is most common way to share resource between different places and HTML website was the convenient interface for doing it. But nowadays, there are more than one way to access internet, not only website with HTML representation but also any application type with its own design representation, for some case, end user even requires data only. Therefore, API specially REST API become most concern and development target for enterprise and organization who want to share their resource including scientific data comparing HTML tags data. For this reason, this reports is concentrated on crawling API than traditional HTML. Moreover, although having its own difficult, API crawling is less interfered by noise and side effect information comparing to HTML especially caused by dynamic website.

2.3 Crawler System

A Crawler commonly relating to Web Crawler is the internet accessing program which systematically browses World Wild Web for exploiting and indexing web content. Web crawler, historically, was created to collect and index internet content serving for search engines. Sometimes, web crawler is used for another purpose than indexing content like copying content of competitor or collecting data for integrating onto another bigger system. The idea behind the crawler is straightforward. In that time, internet is bunch of HTML documents including hyper link connecting each other, then the simple way to exploit internet is acting as human user, visiting each website, getting document then retrieving hyper link from that and repeating visit action with new discovered links. Then, crawler system programmatically designed to emulate that process in larger scale and that is the reason it also called spider bot. [9]

In this report, the term "crawler process", defined in more general meaning, is the process that try to access the resource of any system for collecting data automatically. In this way, not only Internet, the action of accessing an enterprise database for exploiting database tables and using it to collect all table content is also considered as a crawler activity and same for case file share system or cloud base resource.

Following above definition, the crawler system, at least, need to satisfy two conditions. Firstly, it depends on a well interface for accessing target resource system and understanding response returned by this interface. Secondly, system need to have a data-driven mechanism to exploit and collect data from resource which requires a good understanding on target system architecture and data structure. Noted that besides of required condition, a good crawler system is also requested to adapt some another attributes like about speed, space consume or infinity loop detection. However, those attributes should be discussed in other more advance report, this report try to concentrate on basic properties only.

Chapter 3

Methodology

Previous chapter is a general picture about factors impacting on expected crawler system, this chapter will propose an architecture design of system as well as the solutions for satisfying those factors.

3.1 System Architecture

Although designing system is not an easy job requiring some real life experiences, there are several fixed steps in any system designing process that allow this work to become more easier. Firstly, the expected functional requirement must be clear, it should be, in general, presented as a use-case diagram showing system function and their relationship to outside factor (like user or third system). Non-functional system is another concern, but because the lack of capability and experience on crawling domain, this internship avoids to concentrate on it. After defining well use-case diagram and analyzing time, the abstraction components which represent for real system functional object will be sketched out to illustrate whole system elements. Those components keep different roles and work together to form up all requirement functions in use-case diagram. Those working combination and its relationship will be performed as the sequence diagram and provide to reader a general view of mechanism after system box. This section, by following above 3 steps, try to draw out a overview of the crawler architecture in most understandable way.

3.1.1 Use-case Diagram

Depending on internship objective, the main functional requirement is crawling process. But for supporting this process, the crawler need to have initial information to run crawler. Therefore, there are 2 use-cases for system:

- **interact user data:** this function allows user to interact, create and update info for crawling process. Moreover, since there will be more than one user in system, user information is also concerned.
- **run crawling process:** this function runs crawl process with proper parameter to fetch and/or collect data.

Also note that there will be at lease three actors joining in system activity:

- **User:** the main person uses system.
- **Data Resource:** this is unofficially actor present the place system will access to collect data.
- **Data Lake Storage:** the lake storage where system will send collected data into for storing. It is considered as an actor because Data Lake applying micro-service model, thus Crawler System and Lake Storage are two different programs.

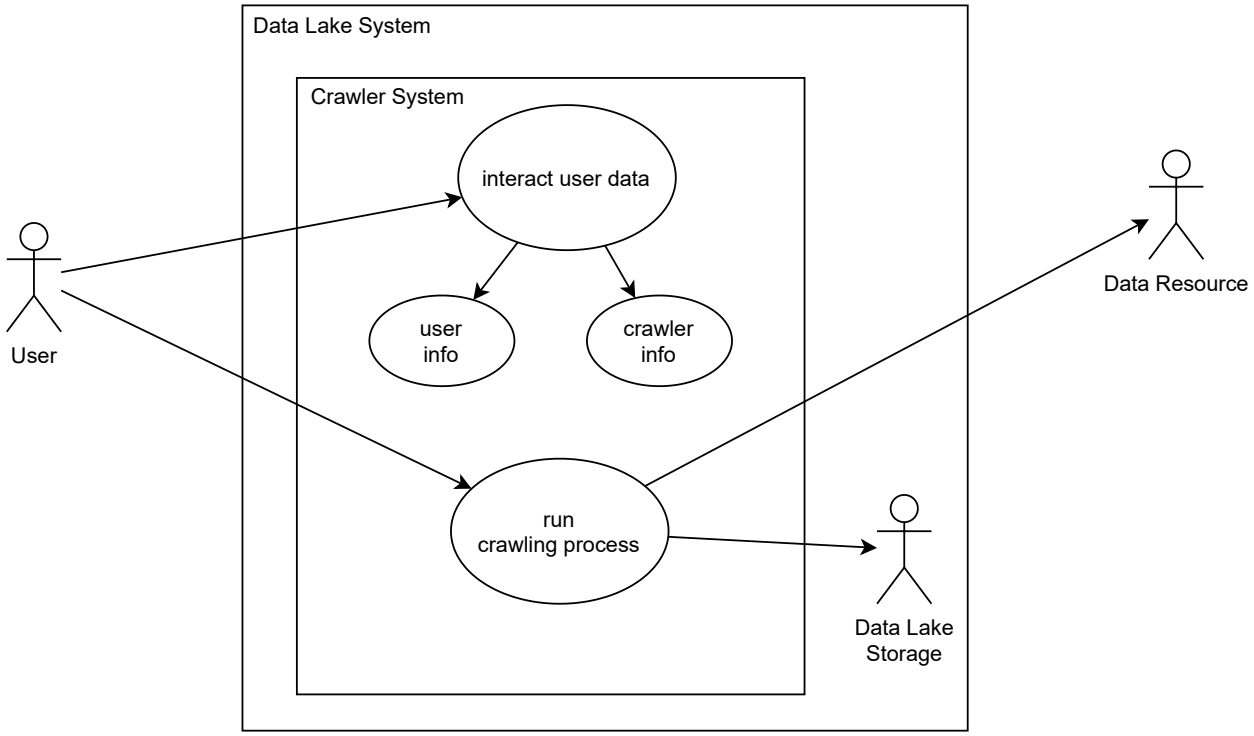

Figure 3.1: Crawler System Use-case Diagram

3.1.2 System Abstract Components

Deriving directly from system functional requirement, there will be, obviously, three components are Database, Crawler System and Service. Firstly, because the system hold user information, it is necessary to have a persistence space to hold user configuration data, the database keep this role. Besides of that, there will be needed to have a object performing any process relating to crawl activity, thus the Crawler System is created for this purpose. Finally, the service object acts as intermediate person who receives request from user interface, calls right method of right service and responses back result.

- **Database:** holds configuration data.
- **Crawler System:** performs crawl activity.
- **Service:** acts as an intermediate between user and system service, also controls system request and response data.

Figure 3.2: System Abstract Components

3.1.3 Sequence Diagram

This section will contains a brief description and minimum explanation together with each sequence diagram.

Interact User Data Sequence

This sequence represent the mechanism for user to interact and setup configuration on system. In this version, there are three action accepted by system and the diagram represents same mechanism for both user info and crawl info. In reality, user and crawl could have different API interface depending on implementation.

- **GET data:** allowing user to retrieve their configuration. The system method is simple, each time user request data, service will call database for corresponding result then return back to user.
- **POST data:** allowing user to add new configuration. The user will post data with body including new configuration, then service request database to update this new configuration to persistence space. The database recheck constraint rule and insert new info if everything satisfy and return response back to service. The service, based on status response, will response back to user.
- **DELETE data:** the mechanism is same to POST data but database action will be delete a configuration object and parameter that user send to system is configuration id instead of new configuration.

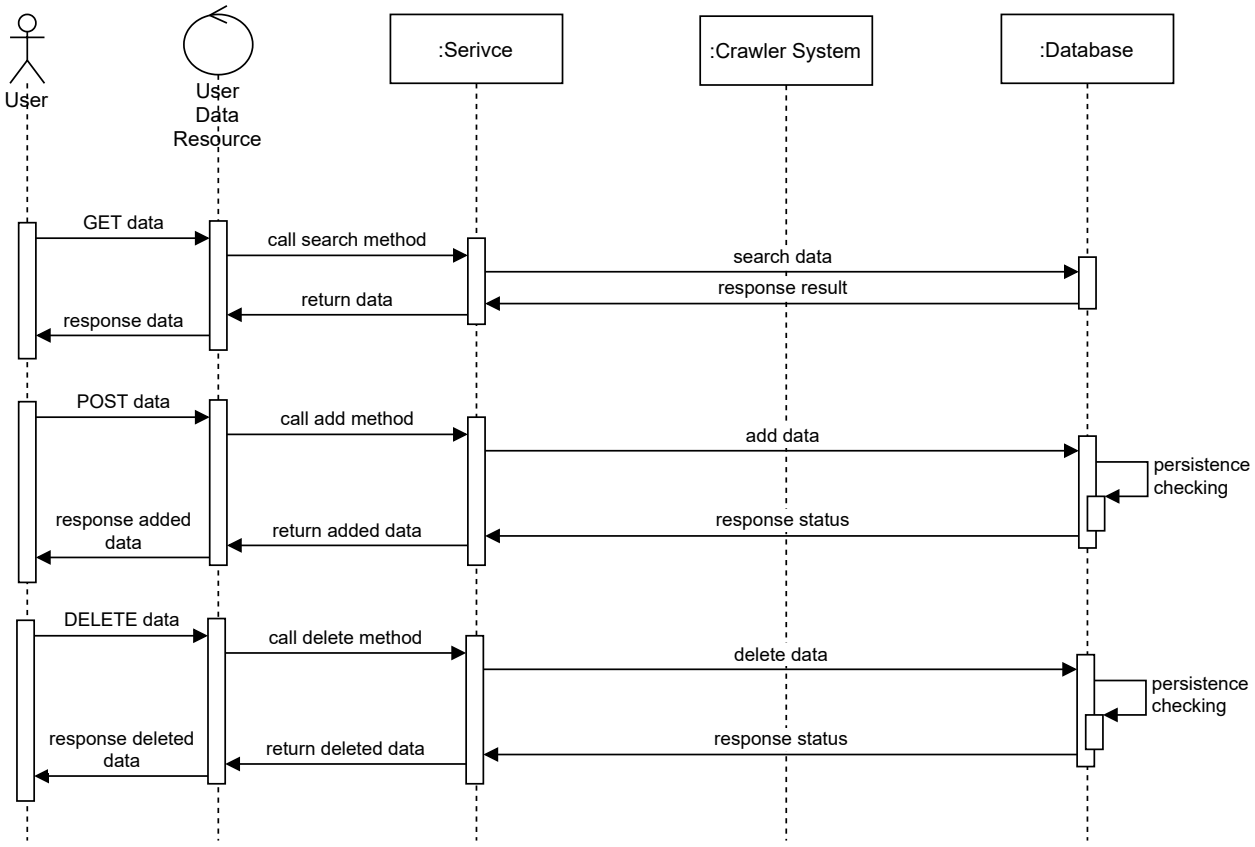

Figure 3.3: User Data Interaction Sequence

Run Crawler Process Sequence

The sequence proposes two usage of crawler system which is considered as two mode. First mode is FETCH and another is DOWNLOAD. The reason of these different mode because of correctness in crawling process. Since the crawling process based on user policy to run, then sometimes the user will want to check if the policy provide right result as expected before pushing them to lake. Hence, the fetch mode is created to serve as checking step when returning crawl result back to user while download mode, on the other hand, will push data directly to lake storage after crawling.

- **Fetch Data:** After request is sent to system, service calls Crawler System to perform crawling process with mode is FETCH. The crawler system checks on database for fully configuration and starts to executes policy for generating centralization table which will be discussed in 3.2. This table after filling up will be returned back to user through service.
- **Download Data:** The processing steps is similar to Fetch mode but after having centralization table (3.2), the system will use this table to generating final crawling contents and push them to lake storage. Lake storage will response back status to system which will be propagated to user.

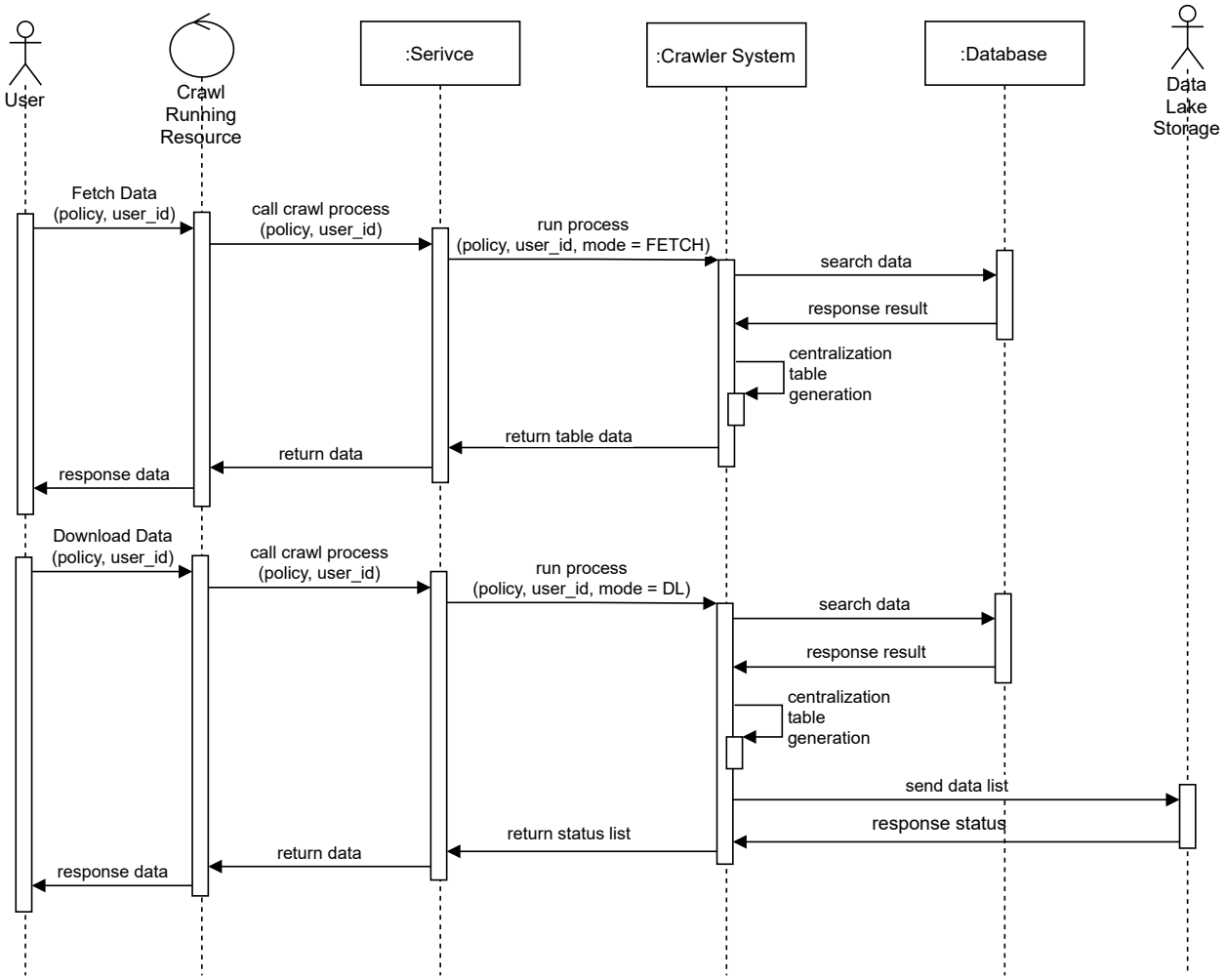

Figure 3.4: Crawler System Running Sequence

3.2 Crawling Strategy

Since crawling process is a bit complicate and difficult to perform for system target interface, this section is designed to discuss about the method and strategy used to perform crawling process.

When deciding API is main interface for crawling data on Internet, the traditional mechanism is not suitable anymore. While traditional crawling initial position is a hyper link pointing to a HTML document accessed by GET method and had full detail information for detecting next hyper link, the API interface, normally, has more complicated access mechanism with multiple type of variable and also includes only specific data relating to that API with less or no hint to next API needed. Moreover, API is design with preferring to spread information into multiple than a single API and the crawler need to perform several requesting call for collecting enough data of one topic target. In some case, the lack of information is not big deal since only specific data is concerned which could be collected from single API, Data Lake ingestion system, for its responsibility, need to collect as much as possible information relating to that specific data, hence single API crawling has just created a bad and useless data in storage.

The report proposes some simple solutions for these problems. Firstly, for work around

the scatter problem, a temporary buffer is used to store each piece of data from multiple API and only pushing those data together into Data Lake storage when full information is satisfied, detail structure will be in 3.2.1. Secondly, a API itself does not have enough information for a continuously crawling process, the basic way overcoming that is a detail instruction for guiding crawler should be provided as the crawling policy. For this, a simple crawling policy language is developed and more discussion will be in 3.2.2.

3.2.1 Centralization Table

Thinking on a temporary buffer which is used to hold full data collected from several API about targeted topic, a table like structure is pushed on top comparing to any other option because of several concerns. Firstly, although contents of a targeted topic could be spread on multiple API, their properties are same both structure or field type and only different by their data. For example, when crawling image file, there would be thousand of files, but always existed fields relating file format, file size and file name coming with file content. Secondly, a fixed schema is big strength of table structure allowing to process data easily, the semi-structure like document or key value would be alternative but designing mechanism on unpredictable attribute is not convenient as fixed one. Therefore, a 2D arrays with row and columns including ordered fields (keys) list is a clear imagination about this required buffer which combines with its role called **Centralization Table**.

size	filename	download link
10	dog	http://sample.link.only/dog
20	cat	http://sample.link.only/cat
15	human	http://sample.link.only/human
”string”	liza	http://sample.link.only/unformated

Table 3.1: Example of Centralization Table

Rule of table is pretty straightforward. Each row represents single instance of targeted data with full information relating to it, the column of table contains multiple fields (keys) present attributes of instance which could be list of metadata and a single download link to retrieve octet-stream content. The number of fields is not limited and could be variable depending on crawling policy. Finally, there are not any restriction on each field in table even the field type, the reason for it because the demand of field type is not yet necessary during system developing time. However, these rule is not yet matured and could be updated in future if needed.

Beside of usage as a data set, the system could use some reserved fields in table for driving processing system, such as if resource need accessing token (depending on crawling support feature), **token** field could be added to provide necessary information. In such case, the table could contains some predefined fields which is have data defined by user.

Finally, the question about the mechanism to build up centralization table is actually a headache, mostly because of table fields problem. Since each field is independent and possible coming from different API call, sometimes, a API call, which is expected to return single result for one instance topic, returns two instead. It could be caused because that instance topic have many different information in such domain field. For example, in file

sharing system, a user owns one or many files, if one instance presents one user then API call for file info could return more than one response data. Although it is only problem of defining which is real instance topic (file in example case) but the problem of conflict result size for different API call is still existed. After long analyzing process, the solution for this issue, thankfully, is sketched out and presented in below proposal table processing method. It contains 3 steps:

1. Generate empty table with predefined fields

token	size	filename	download link
null	null	null	null

2. Fill up system reserved fields or any predefined fields

token	size	filename	download link
ax1yz	null	null	null

3. Process API call and fill up another fields

token	size	filename	download link
ax1yz	10	dog	http://sample.link.only/dog
ax1yz	20	cat	http://sample.link.only/cat
ax1yz	15	human	http://sample.link.only/human
ax1yz	”string”	liza	http://sample.link.only/unformated

Describing process steps, the table is not filled row by row but column by column following left to right precedence. For a specific column, same functional process is run with every rows and is driven by that row data. Example, token column hold accessing token defined by user would be used by fetching API to get download link, size and filename of data when corresponding columns are executed. After execution, response data will be updated into right column of right row. In here, the way to solve size conflict problem is simply duplicate method. During processing the column of a row, if response data include n result, that row will be cloned into n row, each row will be same to each other but processing column will contain only one in n results set. This method is showed clearly in example, the table has the single row in step 2 but cloned into 4 rows in step 3 because API call return 4 different results for same token, the token column itself is duplicated for each result data. Although it is not best practical solution, this idea is acceptable since the unique attribute of each row is still guaranteed as long as API call does not return 2 duplicate response data.

3.2.2 Crawl Policy Language (CPL)

If 3.2.1 provided a hint on a mechanism to collect and concentrate data into buffer, this section will introduce a typeless language used to define table field, column precedence and column processing instruction. In other words, this is a policy guideline for driving **centralization table** generation process including API call instruction. For convenient

convention, this language called Crawl Policy Language and will be mentioned, from now on, with short name CPL.

Behind the scene, the idea of CPL is derived directly from 3.2.1 method, the user will define a set of column name coming with their definition way to retrieve data. There will be two types of definition, declaration allows to add predefined data to table and processing data, shortly called data, contains detail method to collect data from some source. The processing data itself has several different way to treat and adjust data. The detail description will discussed together with language syntax in below.

The design and syntax of CPL, somehow, is similar to Pascal but have its own reserved words. That is because CPL building knowledge both in language design and system implementation absorbing from a online series which teaches reader the way to build a simple interpreter for Pascal in Python [4]. Although it is not an official knowledge source, this series introduces many useful technique to describe language, one of them is context-free grammar following a soft of Backus normal form (BNF) [3]. This method introduced in part 5 of series and is a great way to describe and explain language mechanism.

Let start the CPL syntax explanation with the fully BNF description first, then detail will follow after that.

```
[exec   ]: EXEC declare (data)* (return | NOP) END
[pair   ]: KEY VALUE
[var    ]: VALUE (VAR)*
[declare]: DECLARE pair (pair)* END
[data   ]: DATA (req | pattern) map END
[pattern]: PATTERN var END
[req    ]: REQ METHOD path (HEAD | NOP) (BODY | NOP) END
[path   ]: PATH var END
[map    ]: MAP (MAP)*
[return ]: RETURN
```

CPL language is a combination of multiple sentence, each sentence has its own functional. Each sentence, itself, is the group of multiple token (or terminal) which actually a type of one lexical unit in CPL and/or sentence reference which is reference word point to another sentence. Example of Token are EXEC, KEY, VALUE, etc. which will be presented by uppercase word and sentence reference will be lowercase word like "data", "return", "pare", etc. Since token is considered as unit block of language, they are a pair of TYPE and VALUE looking similar to a key value structure such as "VALUE" token could be showed as below.

```
{
    type: "VALUE",
    value: "random string"
}
```

The main role of sentence serving as a command that will perform some action, however, the sentence is not always necessary to be a command but stay as a part of another sentence

to group up the command. While CPL sentence will start with a functional token (EXEC, DECLARE, etc.) and stop with END token. The list of sentences like "pare", "map" and "return" are not a command sentence so but only a part of another sentence.

For fully understanding CPL, some special token need to be discussed in detail. Firstly, the functional token is a list of token represent a initial point for command and contains: EXEC, DECLARE, DATA, PATTERN, REQ, PATH while END token as known used to finish a command. VALUE is a most straightforward token in here since it is representation of value in language holding a constant data and because CPL does not care about data type, it could be string, map or list. NOP token is actually not a token, it is combined with a arbitrary token with meaning that the arbitrary token is only optional in sentence. For instance, below clause mean BODY token could be exist or not in sentence.

```
(BODY | NOP)
```

Any another token which is not mentioned in here will be discussed when analyzing sentence syntax. And also note that CPL is still in development circle without a mature version, so the sentence or command of CPL could be added, updated or changed in future.

Main

Like any other language, CPL has the starting sentence which will be run first with name is EXEC command. This command is compounded by 3 path: declare reference, data reference and return reference. As discussed above, both declare and data are used to define the column in centralization table. On the other hand, return is an optional with usability to filter final table data. However, in this CPL version, it act as reserved keyword only.

```
[exec   ]: EXEC declare (data)* (return | NOP) END
```

Additional information, while the data statement could be call multiple times, the declare statement is designed to be defined once and must be in the first position. The reason is because of logical sequence in column processing. The column precedence is from left to right so first columns running process must finish before second run. Therefore, all predefined data which is considered to be highest precedence in table must be added before any further action start. And the explanation for single declare statement because one is enough to declare all data.

Variable

Variable is an important path in CPL language, it allow to pass data between column in table. However, the variable, for now, has several restriction. Firstly, one row processing have access limited to that row data and only processed columns are available. Secondly, the usage of variable is still poor since only replacement action is supported that have variable enrolling.

VAR is specific token designed to represent a variable which have structure as below.

```
{
    type: VAR,
    value: {variable_name: column_name}
}
```

In here, nothing special about type of token, but the value has big impart on the way data is assigned into variable. VAR value actually is a pair with key is variable declare name and value is column field in table for data look up. Besides, the row position is not necessary because limitation of variable scope mention above and only data of current processing row returned. Below example will provide better view on these assumptions.

No	column 1	column 2	column 3
1	data 1	null	null
2	data 2	data 3	null

Table 3.2: Example of Variable accessing scope

In example 3.2, during processing column 2 of row 1, if the VAR token be like

```
{
    type: VAR,
    value: {variable_abc: column 1}
}
```

Then the variable declared will hold data: "data 1". Besides, with the process lookup data, both column 3 and row 2 are not visible.

Finally, talking about the var sentence which is representation of a bunch of replacement action. This action have pretty simple mechanism. The first token is VALUE that will setup a seed data (must be a string), then followed by list of VAR token with each VAR present a unique variable name and table data (discussed above).

```
[var    ]: VALUE (VAR)*
```

Then during process running time, each variable will search in seed string corresponding substring which is similar to variable name, after that replace them with corresponding variable value. Since there are multiple VAR token, the precedence will be left to right and new string returned by left variable will be input of right variable but VALUE token is input for most left variable. For example, following example 3.2 above, if the seed string as below.

```
"this is {variable_abc}"
```

then output of replacement process will be string as below.

```
"this is data 1"
```

Although the sentence is limited only to string data, it proposes a way to treat data depending on another data and is a hint on the data-driven process. The usage of this sentence will be addressed more in Data sentence path.

Declaration

Declaration is an important sentence in CPL, it provides a method to define and add predefined data to table. For defining a column, the pair sentence is developed. It is a well illustration of a key value pair with key is table name while value must be a list constant holding predefined data. Since CPL is typeless, the list is not necessary to be unique or same type but commonly relates to string list.

```
[pair   ]: KEY VALUE
```

The pair is not designed to standalone but group together in declare sentence. The declare sentence, recognized by initial function token DECLARE, is a command built to be a group of pair sentence to generate a well predefined data table. As showed in syntax, this statement contains one or many pair and hence, support user to define more than one columns.

```
[declare]: DECLARE pair (pair)* END
```

Since the statement allows multiple columns to generate, those columns order is equal to order of pair statement in sentence. Moreover, cross join is the method to merge multiple columns into single table. For example, if there are 2 columns, each have 2 row then table result contains 4 rows which is all possible combination of 2 columns as illustrated in table 3.3

Column 1
d1
d2

Column 2
d3
d4

Column 1	Column 2
d1	d3
d1	d4
d2	d3
d2	d4

Table 3.3: Example of Declaration Combination

Data

Data or processing data provide a mechanism to define the way to retrieve information from a resource. This sentence is the command starting with DATA token, then followed by req or pattern statement then end by map statement. Each data sentence contains a well defined process method for single column and is itself a representation of one column in table.

```
[data   ]: DATA (req | pattern) map END
```

In here, DATA token hold value is column name, after it, the processing method used to define mechanism to retrieve data for column which could be req or pattern statement

(supported only now). Map statement which be stay near end of sentence is a string used to filter response data from process step to get final data of column. Since the response of an arbitrary resource is not predictable, the map statement is required in sentence.

Discussing in detail, map statement actually is a sentence combining by list of MAP token and at least one MAP token must be existed. With this, data is not filtered once but through a filter pipeline to archive final stage. However, the value and mechanism of MAP token is not well defined in CPL but depending on implementation.

```
[map    ]: MAP (MAP)*
```

pattern and req statement are, nearly, the core idea of data sentence. They are the definition of method to derive data from resource into table that, as mentioned in 3.2.1, is depended on to create a real process for each specific column of any row in table. In current version, because of demand, only 2 method is developed only, but any future update would be possible to adapt situation requirement.

Firstly, pattern statement is the simplest method to process data. It actually a wrapper method to variable statement. The variable statement as discussed in variable part, allows to retrieve data from another column and process a seed into final new data. The pattern statement uses this final data to put into column as new data in table. The pattern syntax will be PATTERN token come with a var statement as below.

```
[pattern]: PATTERN var END
```

On the other hand, the req, shortly call of request statement, is used to define an API call through HTTP/HTTPS protocol. The req design is also straightforward. The components of one request includes request method, path, headers and body. Then this statement will have METHOD token to define calling method, path statement to define request path, HEAD and BODY token used for extra information if needed but is optional in a request.

In here, METHOD token has value is the string of HTTP method, HEAD will be a JSON like data structure with key and value. On the other hand, BODY is very flexible, it could be string or json-like structure.

while most of path in statement are terminal, path is a statement which actually another wrapper of variable statement. Like pattern, path statement uses another column to form up its value which then become path of main API call. The reason of this statement because most of current API system is REST which actually defines query parameter in their path. Therefore, the path statement leverages variable to control its own API parameter.

```
[req    ]: REQ METHOD path (HEAD | NOP) (BODY | NOP) END
[path   ]: PATH var END
```

Below is a example of common seed string of the path statement which will have its final form after variable processing.

```
"http://api.github.com/users/{username}/repos
```

The example is also the initial purpose and target development of variable statement and be explanation for its strictly mechanism: Variable is not designed for general purpose but only this usecase.

3.3 Crawling System

Leveraging all knowledge introduced in previous sections on system architecture and crawling method, this section provides a clear design closing to real implementation system. Particularly, the concrete design of crawling system will be main focus of section.

3.3.1 Design System

The system design is representation of a concrete view about system components and their dependency relationship. In here, the design is derived directly from architecture view but with a small modification.

- **API Server module:** contains implementation for both Service and Database components and run as a standalone HTTP server.
- **Crawler System module:** holds whole crawling logical code which is imported and used by API Server.
- **Database and Data Lake Storage:** represent third party program and interact to system through network.

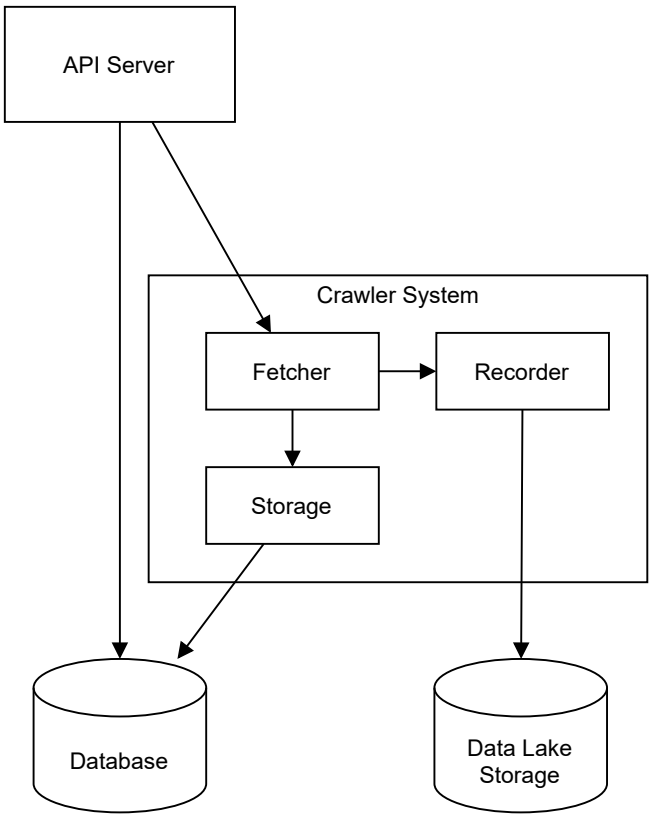

Figure 3.5: Crawler System Design

While API Server is similar to any others CRUD system containing an API interface exported with a basic logical implementation to interact with database, its detail elements will not be discussed in detail. Instead, the "crawler system" module which implements and leverages the crawler strategy solution on its logic is more interesting part.

Firstly, talking about this module elements, it is actually combined by three sub-modules, each keep a specific role and work together to form up module function.

- **Fetcher:** provides an implementation of CPL language inside and support all crawling function. Most of time, it is considered as main of whole module which used to fetch data from resource.
- **Storage:** this, in reality, is a interface helping fetcher to retrieve crawl configuration data. Its implementation is depended on each project and passed to Fetcher during running time.
- **Recorder:** this module provide implementation allowing Fetcher to interact and push crawled data to Data Lake storage.

Secondly, about CPL implementation in Fetcher sub-module, it is based on the pascal interpreter building tutorial on internet [4] and have processing pipeline as below.

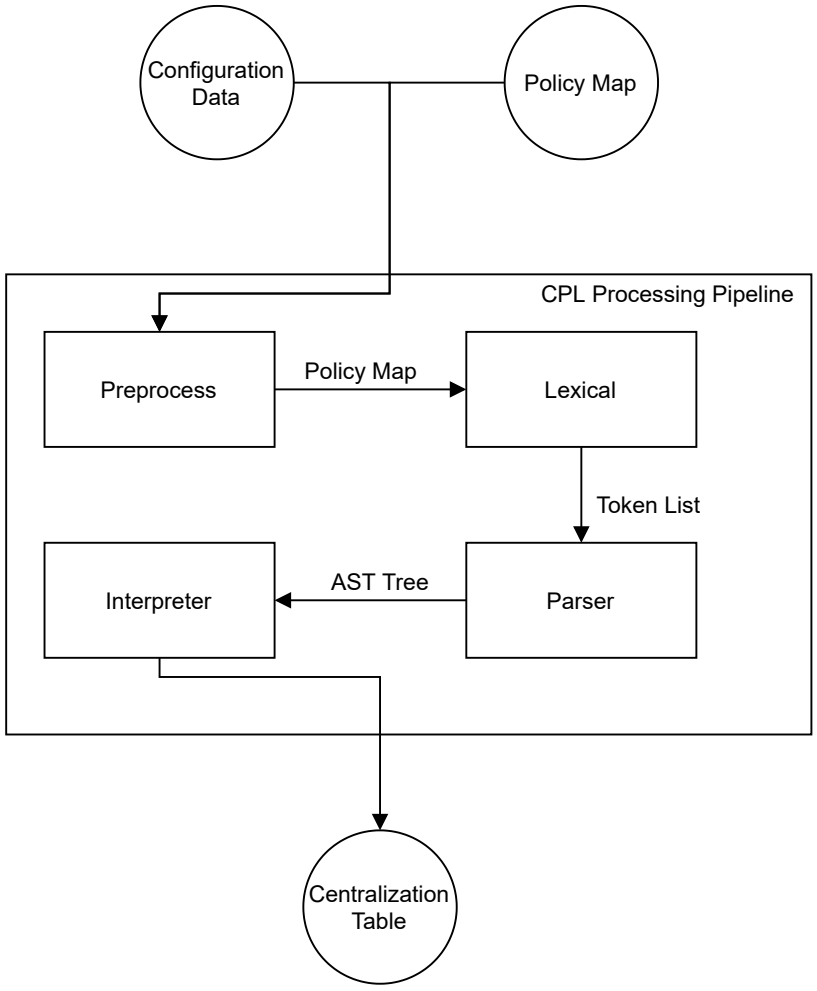

Figure 3.6: CPL Processing Pipeline

In here, the policy map, which is not real CPL but a modified version following JSON format (because JSON like structure is easier to produce and have highlight supported) passing by user will be processed by preprocessor based on Configuration Data getting from Database to generate final complete policy map. Then this policy map is passed to Lexical, this class converts JSON like format CPL to original CPL token list. After that, Token List will be read by Parser class to generate an AST Tree [1]. AST Tree will be final guideline for running Centralization Table generation process which is role of Interpreter class. When all generation processes finished, Interpreter return Centralization Table back to Fetcher for performing further action which could be download process or just sent back to user.

3.3.2 Design Database

The main target of Database is for storing user information and crawl information. Although the purpose is quiet small and underestimate its power, because of limitation on internship time, it is considered to be acceptable. But that means database design is not mature enough and could be updated in future. For current version, below design is real database design of crawling system.

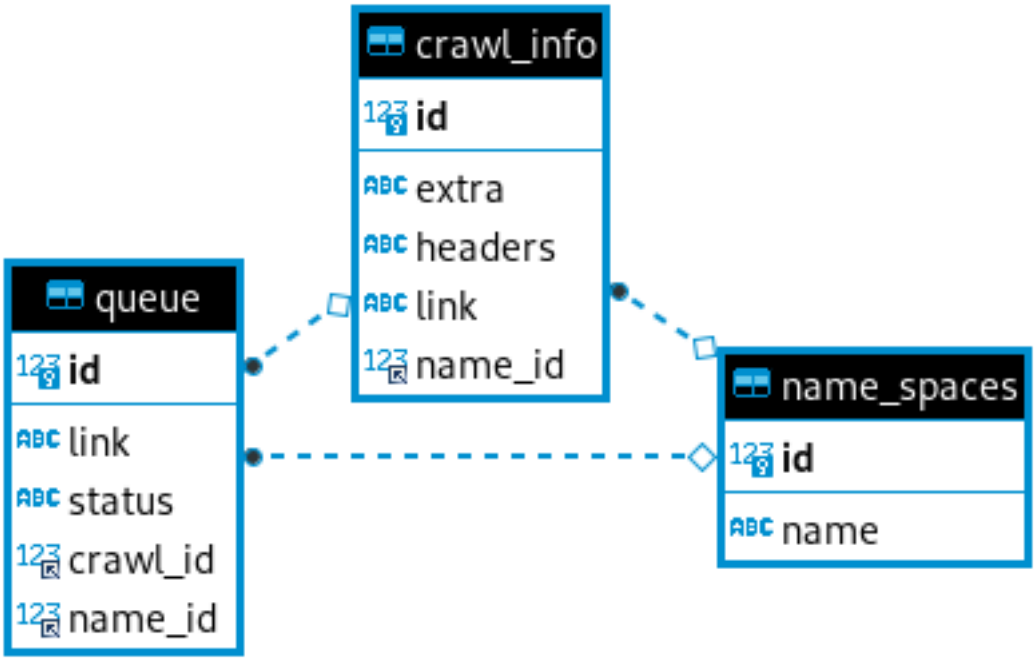

Figure 3.7: Database design

For explanation of tables plus their columns meaning, the brief description will be:

- **name spaces:** holding user information. The name field presents user name.
- **crawl info:** holding crawl configuration data. The nameId field links to owner of configuration while link and headers fields provide default configuration for crawling source destination. Extra is a special field used to hold policy default configuration which could be used by preprocessor in CPL Processing Pipeline.

- **queue:** this is interesting table designed for future feature which is not yet finished, therefore, it could be considered as redundancy table.

Chapter 4

Libraries

For working with this project, there are many tools and frameworks used both in development and deployment tasks. However, this chapter will be concentrated on two main libraries only. Although java is main project language, it will not be discussed in here since it is actually pretty old and familiar language. Both of these libraries supporting pretty well in writing API server module while crawler system module is mostly implemented by base java libraries with small path leveraging package from quarkus for specific function performing such as network interaction or JSON converting process.

4.1 Quarkus Framework

Quarkus is a Kubernetes native java tailored for OpenJDK HotSpot and GraalVM [8]. This framework is designed for a fast deployment of a java project on the container base system and also supports large number of libraries for designing and implementing a web base application. There are also many another convenient features provided by quarkus like real time development (the developing application could be updated in real time when source code is changed) or simple configuration and cached file supporting for faster re-compiling. It is difficult to describe what actually this framework be, but if coming from experience, this framework could be considered as this: "developing a normal web base application by your preference libraries which is supported by quarkus, then building them together, the quarkus will make sure it run in most optimization way". Quarkus acts as all in one framework for both developing, testing and deploying with highest puspose is for making application best suitable to deploy on a container.

4.2 Hibernate Framework

Hibernate Framework is simple an object-relational mapping tool for java programming language [6] which is used to map an object-oriented domain model to a relational database. Allowing to decouple the relationship between system design and database specific type is one of most advance of this framework. By providing a wrapper interface outside of any database, the application applying hibernate to access into data system could change type of database without rewriting source code. Moreover, HQL (hibernate query language) is

simple and easy to use in programmatically way if comparing to traditional SQL. Besides of that, Quarkus supports Hibernate pretty well with a great minimize configuration file which is a bit messy in traditional hibernate framework.

Chapter 5

Result

In this chapter, the screen shot of real deployed system will be used to illustrate the implementation. Postman will be main platform showed in screenshot since this is an universal tool to work with API interface and 100% used during project development time. The results is also presented following system functional requirement to show out the implemented part comparing to objective list.

5.0.1 Interact User Data

There are a lot of API is implemented for this function, but only some of them will be showed out in here to illustrate result. Another are pretty similar in mechanism.

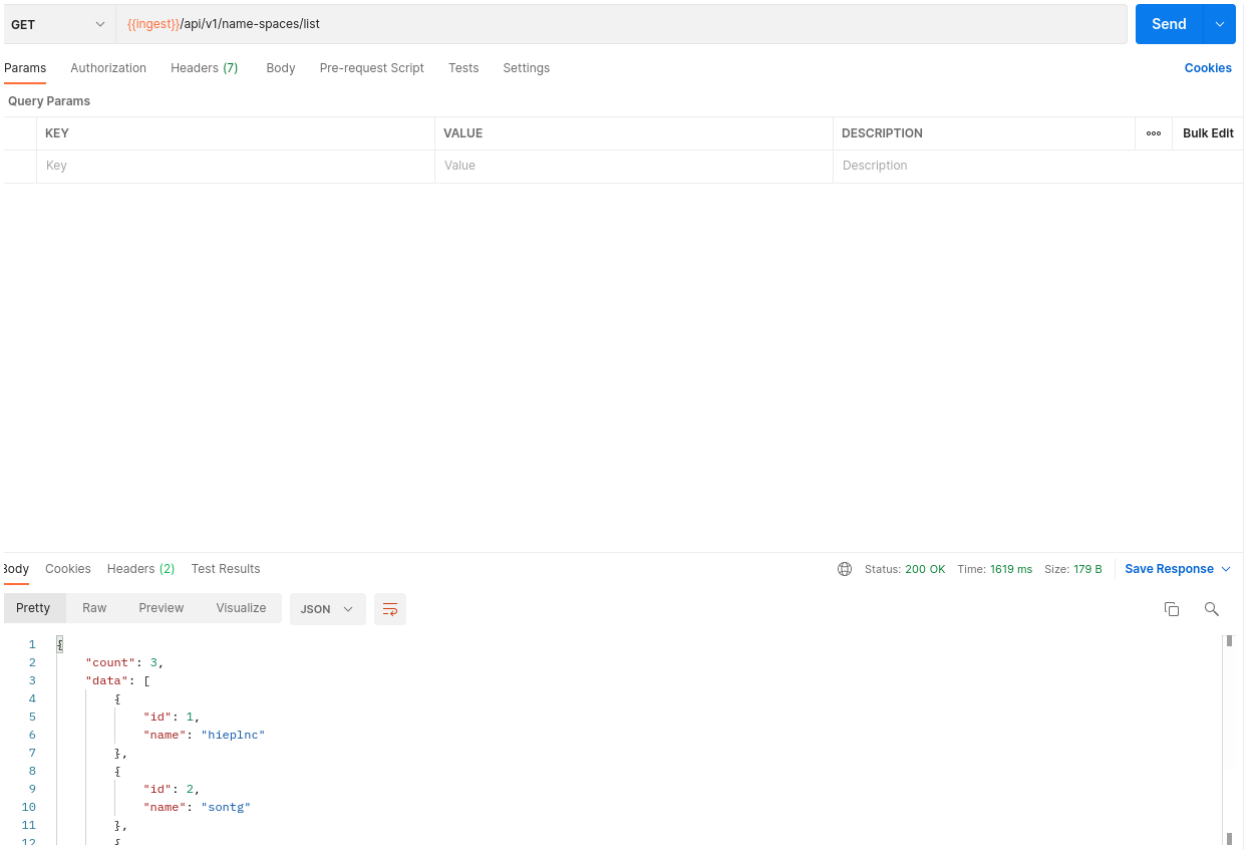

Figure 5.1: Result of get list data from name space table

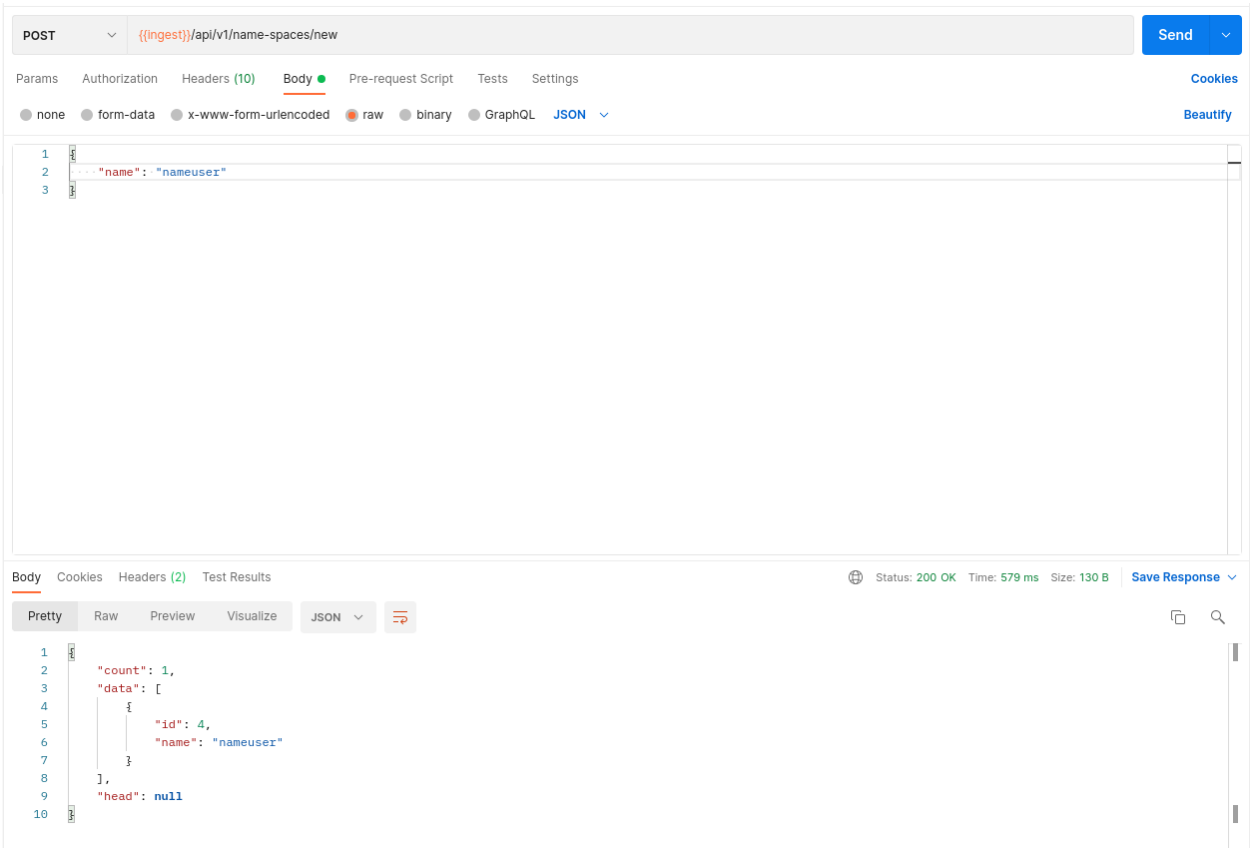

Figure 5.2: Result of create new user to name space table

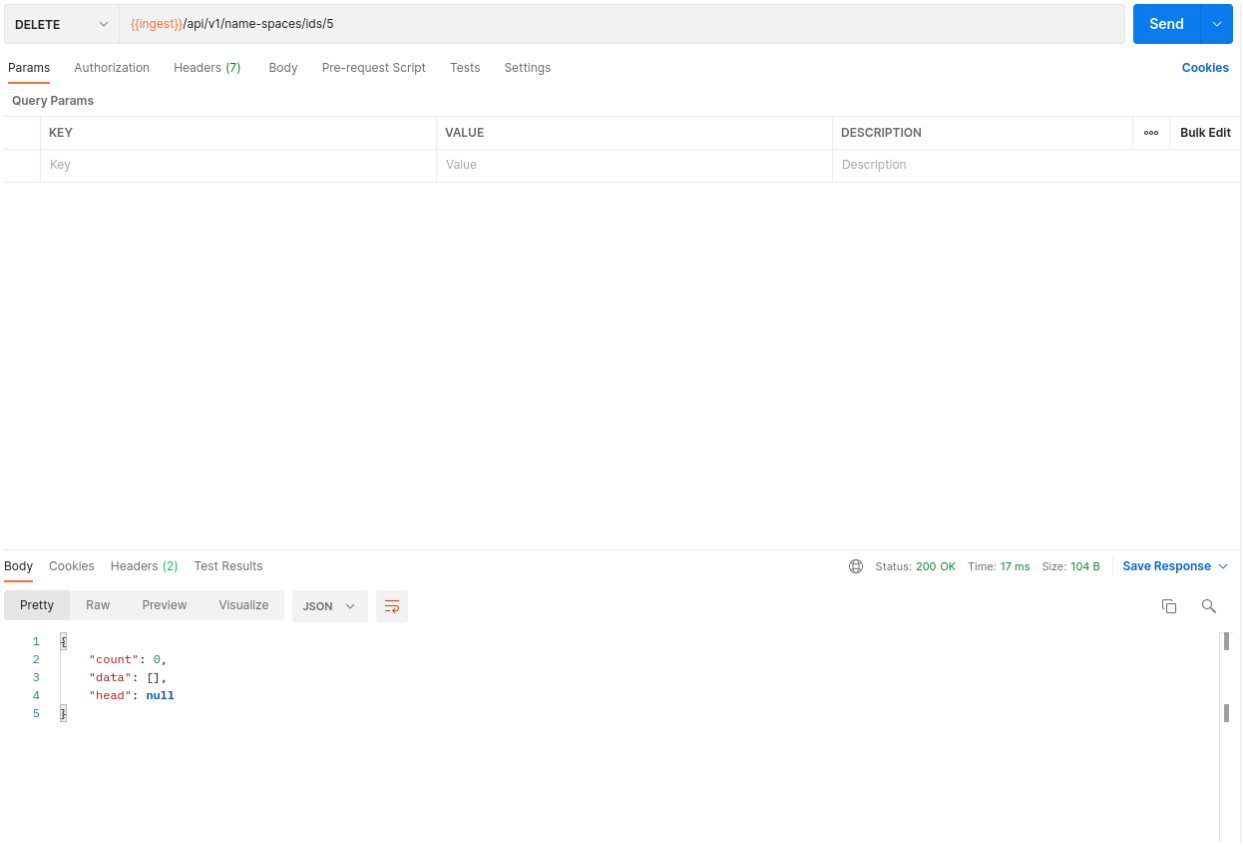

Figure 5.3: Result of delete a user by id of name space table

5.0.2 Run Crawling Process

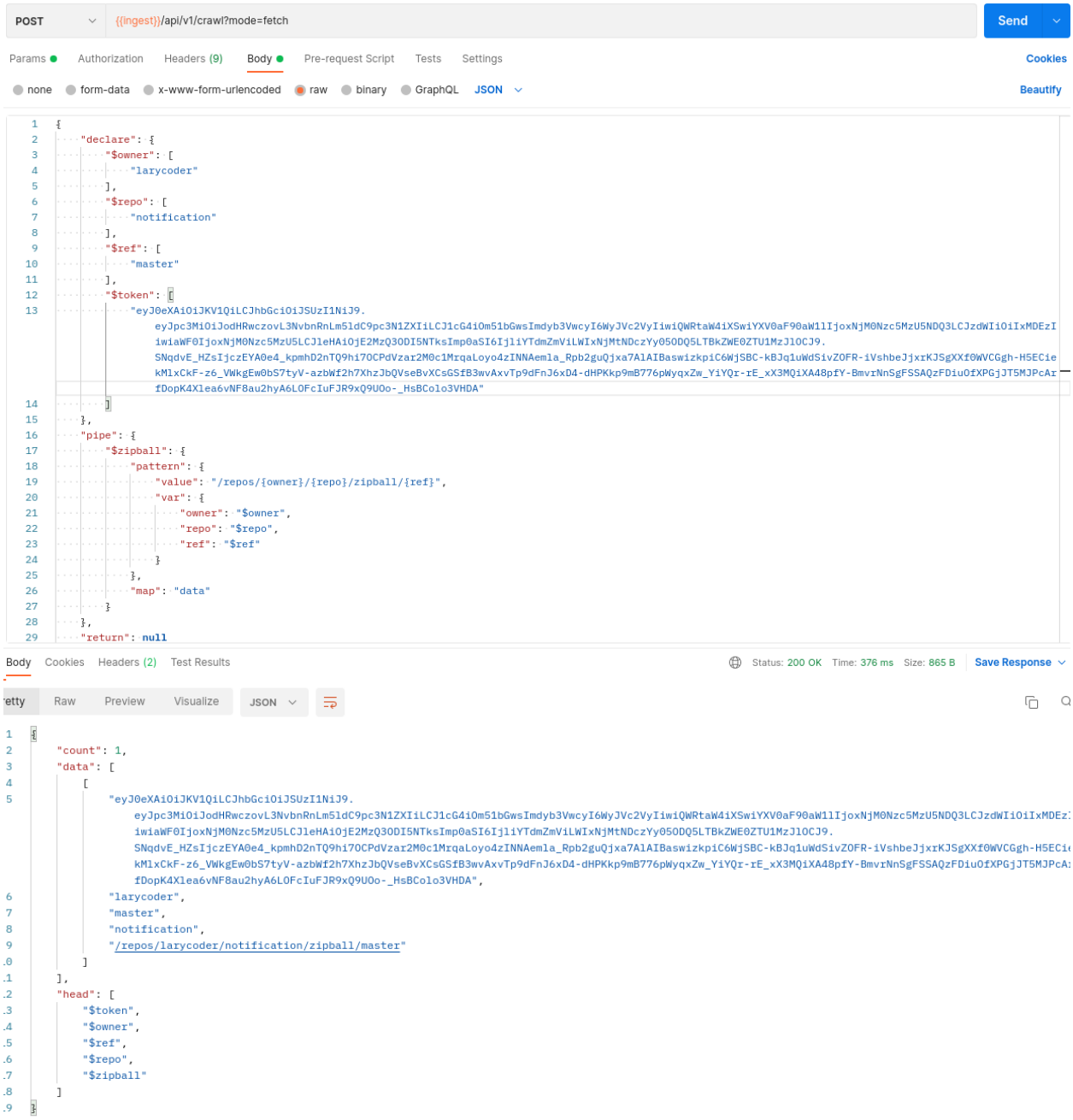

Figure 5.4: Call Crawl in FETCH mode

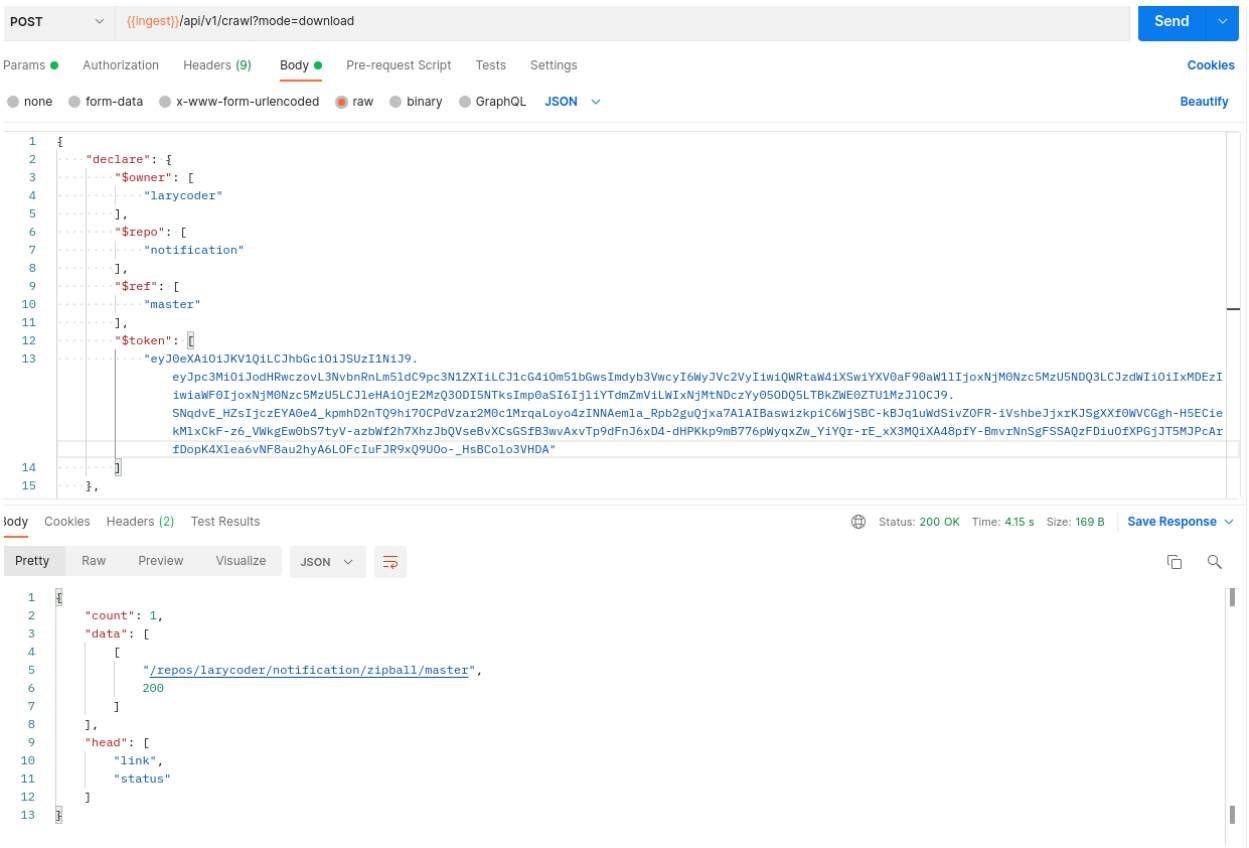

Figure 5.5: Call Crawl in DOWNLOAD mode

Chapter 6

Conclusion

In this internship, from requirement of designing a crawler system that need to satisfy two factors: It is well design to crawl resource through API interface and provide a mechanism to provide a data with as much as possible metadata for enriching Data Lake storage. For those issues, by providing centralization table, we propose a temporary buffer allowing user to add infinity metadata to data before pushing into lake storage although it could be limited by running resource and user current metadata. And for crawling through API interface which is a bit different from traditional HTML document, we create CPL language that give the user opportunity to design their own policy to guideline system the way to crawl any type of API. Finally, we actually propose a simple crawler architecture to put all above solution into reality.

However, there are many weakness in system that we will need to update and improve in future. Firstly, the CPL is still not yet mature and have poor variable mechanism. Its implementation is slow because of lacking parallel technique in Interpreter design of CPL implementation although the language itself supporting parallel in many way. The database system is not leveraged in proper way and usage interface of system is still difficult to use.

Besides of all above weakness, the most concern is about CPL itself since the development of this language is totally based on github api. Although designed with ambition to serve an arbitrary type of text based API, most idea of CPL statement come to specificly solve github crawling problem (variable mechanism is a big example in here). Therefore, this language need to apply on more different API to prove itself as well as update more feature or fix issues if existed.

On the future, if having time continue with this project, the first thing will be update API interface to improve usability and make it more friendly. Then we could start to think on database and the way it could be leveraged better, such as for converting system from synchronous to asynchronous, the database or message broker tool need to be used and in more advance way. Asynchronous system will allow system to perform multiple crawl at same time and not require user to wait process finished, it is most necessary attribute for performing a long crawling process. Parallel to all of these improvement, we will find more API to test with CPL language to sketch out its strength and weakness, then from that updating current CPL version.

Bibliography

[1] *Abstract Syntax Tree*. URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree.

[2] *API History*. URL: https://en.wikipedia.org/wiki/API#History_of_the_term.

[3] *Backus–Naur form*. URL: https://en.wikipedia.org/wiki/Backus-Naur_form.

[4] *Build A Simple Interpreter Series*. URL: https://ruslanspivak.com.

[5] *Data Lake*. URL: https://en.wikipedia.org/wiki/Data_lake.

[6] *Hibernate ORM*. URL: https://en.wikipedia.org/wiki/Hibernate_(framework).

[7] *ICTLab*. URL: https://ictlab.usth.edu.vn.

[8] *Quarkus*. URL: https://quarkus.io.

[9] *Web Crawler*. URL: https://en.wikipedia.org/wiki/Web_crawler.