Study and Design a Datalake Architecture for Management of Image and Voice Data

Le Nhu Chu Hiep
Supervisor TRAN Giang Son

University of Science and Technology Hanoi
ICT Department

October 13, 2020

Acknowledgement

Firstly, I would like to express my thanks to my research supervisor Dr. Tran Giang Son for giving me the opportunity to do research and providing invaluable guidance during this internship, hence I can freely explore new abilities of myself. Secondly, my sincere thanks also goes to Dr. Nghiem Thi Phuong for his encouragement, insightful comments and hard questions during the research. Thirdly, I thank my fellow labmates, friends in ICTLab for the stimulating discussions. I also very appreciate to have been a student at USTH where all professors and staffs are always willing to support their students. Last but not least, I express my deep and sincere gratitude to my parents who raise me up throughout my life by their patient and their love. Sincerely thank!

List of Figures

Contents

1. Introduction

1.1 Context and Motivation

While USTH in general and ICT lab in particular is an academic environment that performs many studying and researching activities that create a massive record of information and data, we lack an effective platform to store, public, and share these records.

In teaching, there are a bunch of lectures, slides, books, or practice exercises created each year, and most of them are still stored personally by lecturers or students which is difficult to approach and retrieve. Therefore, it is necessary to have central storage to store and manage all USTHs knowledge, so everyone could access and use them.

Besides the education knowledge, data collection is also an important part that needs to be considered. Unlike knowledge, the data is a very large raw collection that has various structures and poor quality but contains much hidden valuable information that is used a lot in the research activity of USTH, especially, their attributes are difficult to be handled in normal management and sharing software.

For example, most data collection of ICT lab team is collection of a large binary file (such as **medical image** (CTScan, DICOM, etc.) of Lung Cancer Team, **voice file** of Voice Viet Team or **Insert Wing image** of Bioinformatics Team) that are an obstacle to both filesystem and normal database management system (MariaDB, SQL Server, etc.) directly.

Therefore, we still have not had any effective software to collect, manage and share data collection, plus with the demand for central storage for teaching reason (mention above), so we decide to create a **data repositories** which is a central repository to collect, store, manage and public teaching and research data.

This internship scope focuses on **Data Repositories** for ICT lab only. Figure 1 presents the role of **Data Repositories**.

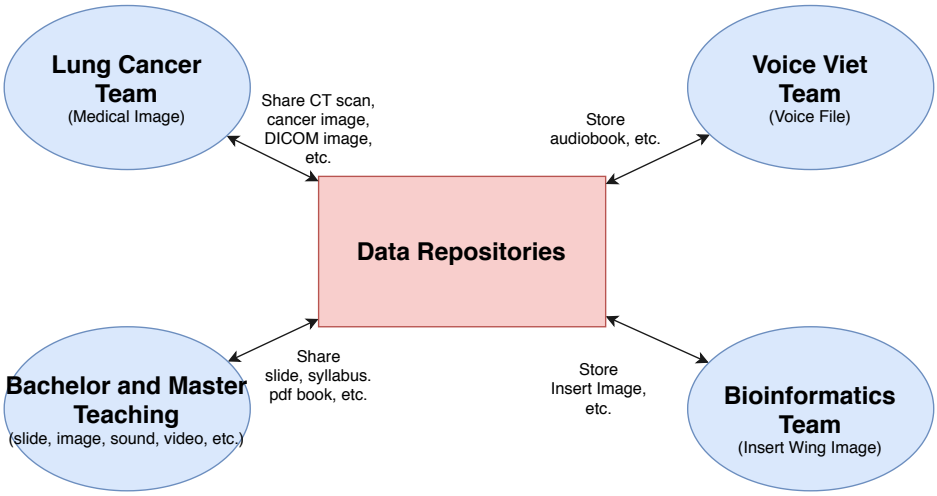

Figure 1: Role of Data Repositories in Teaching and Research Activities of ICT lab USTH

Although data warehouse is a common design applying on many centralized systems, this design bases on the relational database management system (RDBMS) that would not handle data of ICT lab (binary file is an obstacle to most of DBMS), so we end up with data lake concept which is a new idea of a centralized system for working with various type of data that includes large size binary file as image or voice.

1.2 Objective

The main objective is to design and develop a software system based on data lake architecture to collect, store, manage, and retrieve scientific data for ICTLab. Specific goals include:

- Study knowledge and background of data lake concept and architecture.
- Study existing open-source frameworks to design and implement data lake architecture.
- Design data lake architecture for ICTLab based on studied open-source frameworks.
- Implement designed data lake architecture to collect, store, manage, and retrieve ICTLab data using popular software development tools.

1.3 Thesis Organization

This thesis aims to introduce the concept and architecture design of data lake that could apply to develop ICT lab data repositories. We will introduce the reason that the traditional data management system could not deal with ICT lab data in chapter 2 - *Related Work*. Chapter 3 - *Background* will overview the data lake concept and our data lake design idea, and chapter 4 - *Building Data Lake For ICT Lab* is the detailed description of architecture design and simple

implementation derived from that idea. Finally in chapter 5 - *Result & Demo* and chapter 6 - *Conclusion & Future Work*, we will show our result in this internship and discuss future work to improve our data lake architecture.

2. Related Work

Nowadays, the modern data is not only about analysis report or statistic report which are structured data, but also lie on a large set of semi-structured (XML, JSON, etc.) or unstructured data (binary blob) data, for example as image and voice data of ICT lab, hence the modern data management software is also required to support both 3 type data. In this chapter, we will point out that the traditional management system could not satisfy the condition to manage modern data by themself.

2.1 Filesystem

The most universal data management system is the filesystem that is used to store daily life data. Although there is a huge amount of different types of the filesystem, they share many same characteristics.

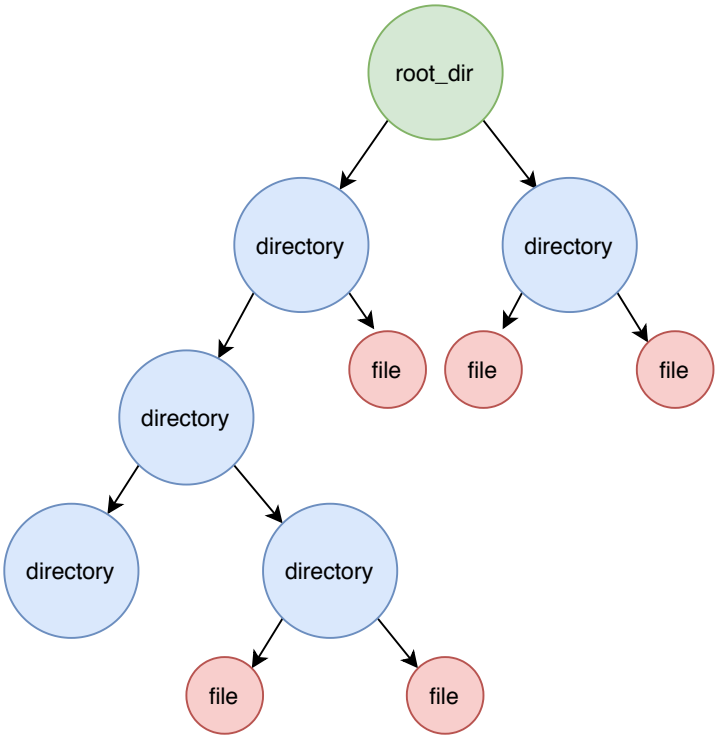

Figure 2: File System Structure

This kind of system constructs data into binary blob called file, and these blobs will be indexed by a specific path (name). Each binary blob contains several metadata that provide extra information about data that would be used in the discovery process.

Secondly, the filesystem applies a simple mechanism called "directory" / "folder" to manage the whole system as a tree, and this is a simple and effective way to store and manage any arbitrary data type.

However, the filesystem management idea is too simple and does not support any advance discover in native that is a weak point in managing a large dataset with complex structure, and file naming convention is another problem when data grow up.

2.2 Database Management System (DBMS)

DBMS is the software developed on top of the filesystem to fix its weak point and propose more power management mechanism that contains a stronger query toolset for data exploiting, and retrieving.

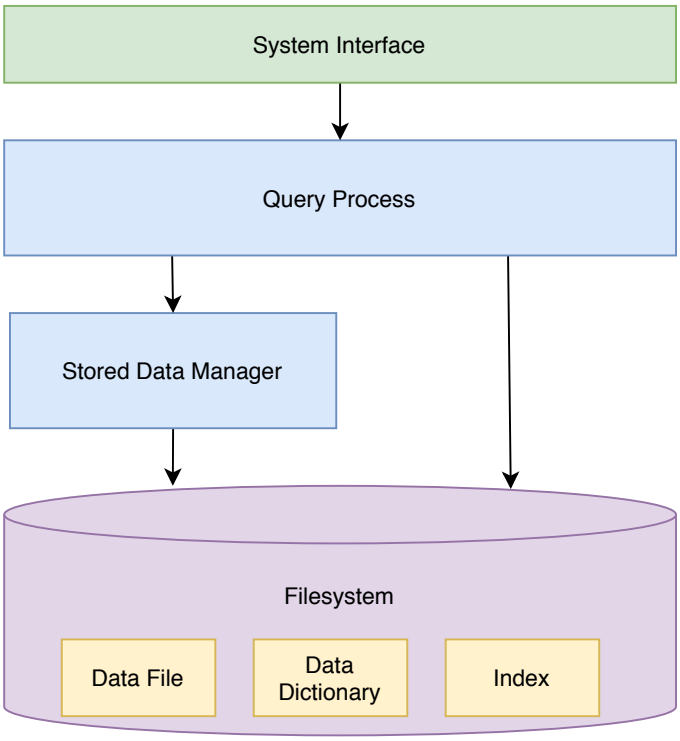

Figure 3: DBMS Structure

There are two types of DBMS which are Relational Database Management System (RDBMS) and NoSQL Database Management System (NDBMS). RDBMS was invented in 1974 to manage

structured data and typically applied SQL language as its interface (for example as MySql, MariaDB, SQL Server, etc.). On the other hand, NDBMS is commonly about non-relational or non-SQL interface DBMS that is designed to handle both structured and semi-structured data, and its famous implementation is MongoDB - document-oriented database program.

Although providing an effective solution to handle structured and semi-structured datasets, the traditional DBMS is seen not to work well with unstructured data (binary file). While it is possible to store binary blob in the database, these blobs will slow down the query performance a lot, especially if the blob is large in both size and number. Since mostly ICT lab data is the binary file, traditional DBMS is not a proper option to build Data Repositories.

3. Background

The modern dataset is hard to manage in the traditional management system and software, so it requires a new idea that applies modern view and technology to develop the data management system. Therefore in this chapter, we will introduce a modern data management concept - data lake that was born to trickly overcome the difficulty caused by modern dataset attributes, and also propose a design idea for this concept.

3.1 Data Lake Concept

Data lake was introduced by James Dixon in 2010 [1], he compares data lake to data mart and defines data lake like this:

If data mart is a store of bottled water that is cleansed and structured to easily consume, then data lake is a more natural state body of water. The contents are streamed from a source to fill up the lake and its user can come to do the examination, dive in, and take samples.

For a more technical point of view which is widely accepted by the AWS team [2], the data lake is a centralized repository that allows users to store both structured and unstructured data at any scale, and users can run different types of analytics process on their raw data.

Even though the data lake concept is clearly defined through the example of James Dixon and the AWS team's opinion, after long time research, we decide to provide our customize explanation as a third viewpoint that will fit our concrete implementation of the data lake.

Fundamentally, if the data warehouse is a DBMS designed to support the analysis and reporting process (or ONAP) for structured data, then in purpose, the data lake is its upgrade version that handles the various type of data in any scale. Additionally, while the data warehouse is schema on write, and strictly obey refined schema in any process, the data lake is schema on read, and enable that data is stored in 1 schema but processed in other schemas.

Data Lake Definition

1. It allows store various types of data.
2. It has a mechanism to manage and keep track of its data.
3. It allows high-performance queries and process data from different viewpoints.
4. It could handle the exponential growth of data in the system.

During data lake maturity, the pioneer researchers propose many constrained properties for the lake, but not all of them are necessary to our **data repositories**, so we choose filtered properties that are important to our system only. Figure 4 presents the data lake properties.

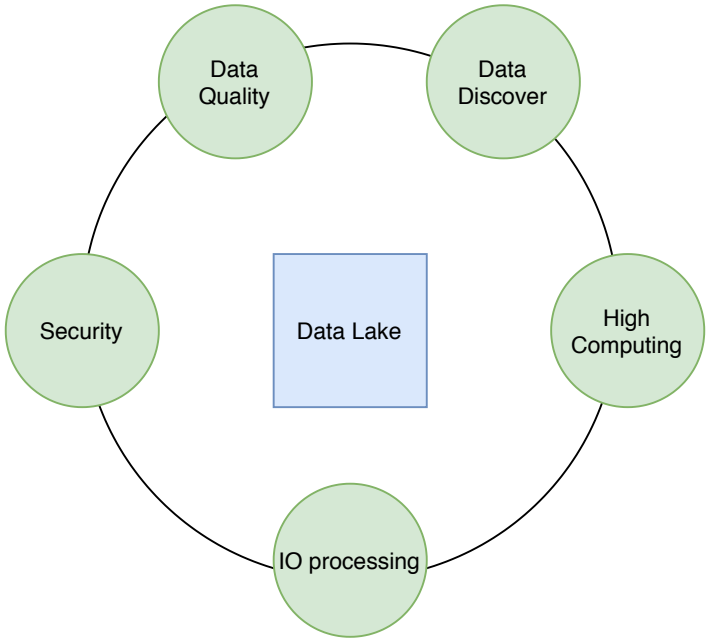

Figure 4: Data lake properties

Data lake properties explanation

- **IO processing**: high-throughput IO process with unlimited storing volume.
- **Security**: data lake sensitive information is protected but does not interfere data discovery process.
- **Data Quality**: expected data outcome equal to actual outcome.
- **Data Discover**: exploited and retrieved all data as long as they are stored in the lake.
- **High Computing**: effective querying data without interfering with size and number of data.

3.2 Kylo Frameworks For Building Data Lake

Despite fact that data lake is a top trend in the data science community, the actual lake design blueprint or open-source implementation project is pretty rare, and most of the common successful data lake providers have not yet opened their source or public community version of the lake, hence our researching process largely stopped in theory.

Thankfully, Thinkbig - a small company of Teradata corporation provided an open-source data lake project called Kylo which was also announced in Teradata site [3] in 2017. However, for some reason, this project runs in the inactive state after 2 years and most of the support and product link disappeared or not found, but the source code has remained in Github [4].

Since this is the only open-source data lake that existed on the Internet at that moment, and to avoid building a system on nothing, we spend lots of time on setup, rebuilding, and deploying Kylo. Although we could not wake up all functions of the system, we successfully run its basic mode with UI and gained some experience and the general concept of how this system works.

Kylo is a data management software platform that allows users to config and builds its processes pipeline to automatically drive the flow of data. It keeps tracking all activity and data status of each process, and use them to monitor, profile, and audit data repository. Moreover, Kylo also integrates SparkSQL - a power structured data processing module of Apache Spark to support data high-performance query on the system.

In architecture, Kylo is a typical micro-service program that contains Apache Hadoop (distributed filesystem), Hive (data warehouse based on Hadoop), and PostgreSQL (RDBMS) to store and manage data together with their meta while leverages Apache Nifi (data automate software) to allow the self-service process and Apache Spark (distributed cluster computing framework) plus Elastic Search (full-text search engine) to perform effective data discovering and exploiting jobs.

Even though applying many big data technology in the core and having an effective solution to collect, store, and manage large datasets, Kylo still contains some limitations in design. Firstly, it

only allows users to upload 50MB maximum file size, and while supporting structured data well, it lacks a regular method to collect the binary file as an image or voice. Secondly, the system still applies schema on write mechanism. Finally, this software depends too much on third-parties software, so its deployment process is complex and difficult including much incompatible trouble between each ingredient.

Depending on the Kylo framework, we develop other architecture to collect, store and manage data that leverages the benefit idea of Kylo as the data query interface or metadata tracking mechanism and also propose a fixed solution for its weak point that is file unlimited size and type, support multiple data viewpoints, and is platform-independent software.

4. Building Data Lake For ICT Lab

4.1 Requirement Analysis

4.1.1 Functional Requirement

This data lake is designed to provide a simple storage management solution to data repositories. It contains 4 main basic use-cases:

- **Manage System**: Allow users to control and monitor system configuration. Including 2 sub-use-cases:
  - **Manage System Metadata**: Control, register, and monitor system metadata.
  - **Monitor Log**: Allow users to keep tracking system activity.
- **Manage Repository**: Allow users to store data and metadata effectively. There are 3 sub-use-cases:
  - **Initialize Repository**: Allow users to create a new repository in the system where holds the data share the same attributes.
  - **Update Repository**: Allow user ingest data with its extra meta to the repository.
  - **Audit Repository**: validate the quality of the repository
- **Query**: Allow user query storage by SQL language and provide extra analysis toolset. It includes 2 sub-use-cases:
  - **Search Storage**: slow realtime query - the system performs query execution in storage then returns the result.
  - **Search Cache**: fast caching query - the system cache query results in the cache and return cache if the same query request is sent.
- **Download Data**: Allow users to collect and retrieve data content from the system.

4.1.2 Non-functional Requirement

- **Large Volume**: the data lake is centralized storage so its capability needs to store and manage a massive volume of data.
- **High-Performance Computing**: while the traditional storage management system will perform a poor performance in the large-volume data, the data lake is designed to reduce the affection of data size in query performance.
- **High Throughput**: the data lake is expected to allow many users to perform multiple data upload and download action at once time.
- **Compatible**: working as a data management solution, the system interface should be flexible enough to support different types of analysis programs to connect and collect data.
- **Upgradable**: With the fast development of big data, the system should enable the integration of new technology to improve its performance or provide new useful features with low or none impact on original system architecture.

4.2 Diagrams

4.2.1 Usecase Diagram

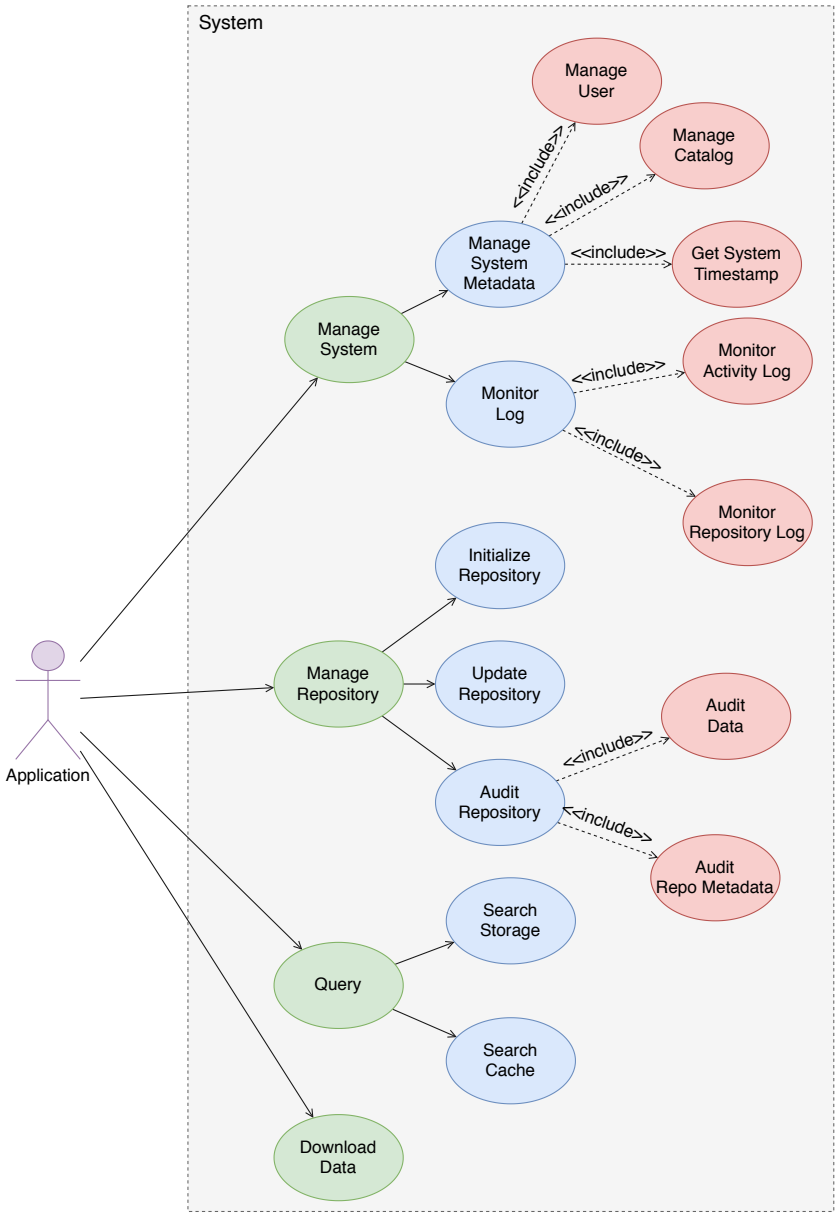

Figure 5: Usecase Diagram

Usecase explanation

- **Manage System**: Control and monitor system configuration. Including 2 sub-use-cases:

  - **Manage System Metadata**: register and monitor system metadata. These metadata are:
    * **Manage user**: register and monitor system users.
    * **Manage Catalog**: register and use catalog in data classification (used in query process).
  - **Monitor Log**: keep tracking system activity. There are 2 types of logging in the system:
    * **Monitor Activity Log**: Logging activity of each component in the system.
    * **Monitor Repository Log**: Centralizing metadata of each repository in the system.

- **Manage Repository**: store data and metadata in an effective way. There are 3 sub-use-cases:
  - **Initialize Repository**: create a new repository in the system.
  - **Update Repository**: ingest data with its extra meta to the repository.
  - **Audit Repository**: validate the quality of the repository, the quality includes:
    * **Audit Data**: validate the quality of data.
    * **Audit Repo Metadata**: report the current state of the repository.

- **Query**: query storage by SQL language and provide extra analysis toolset. It includes 2 sub-use-cases:
  - **Search Storage**: perform query execution in storage.
  - **Search Cache**: perform query as search storage but caching results in the cache and return while the same query request is sent.

- **Download Data**: collect and retrieve data content from the system.

4.2.2 Main components of data lake

To handle the use-case requirement, we define the set components as below figure. These key concepts identify set feature objects which used to describe the system behavior in each use-case.

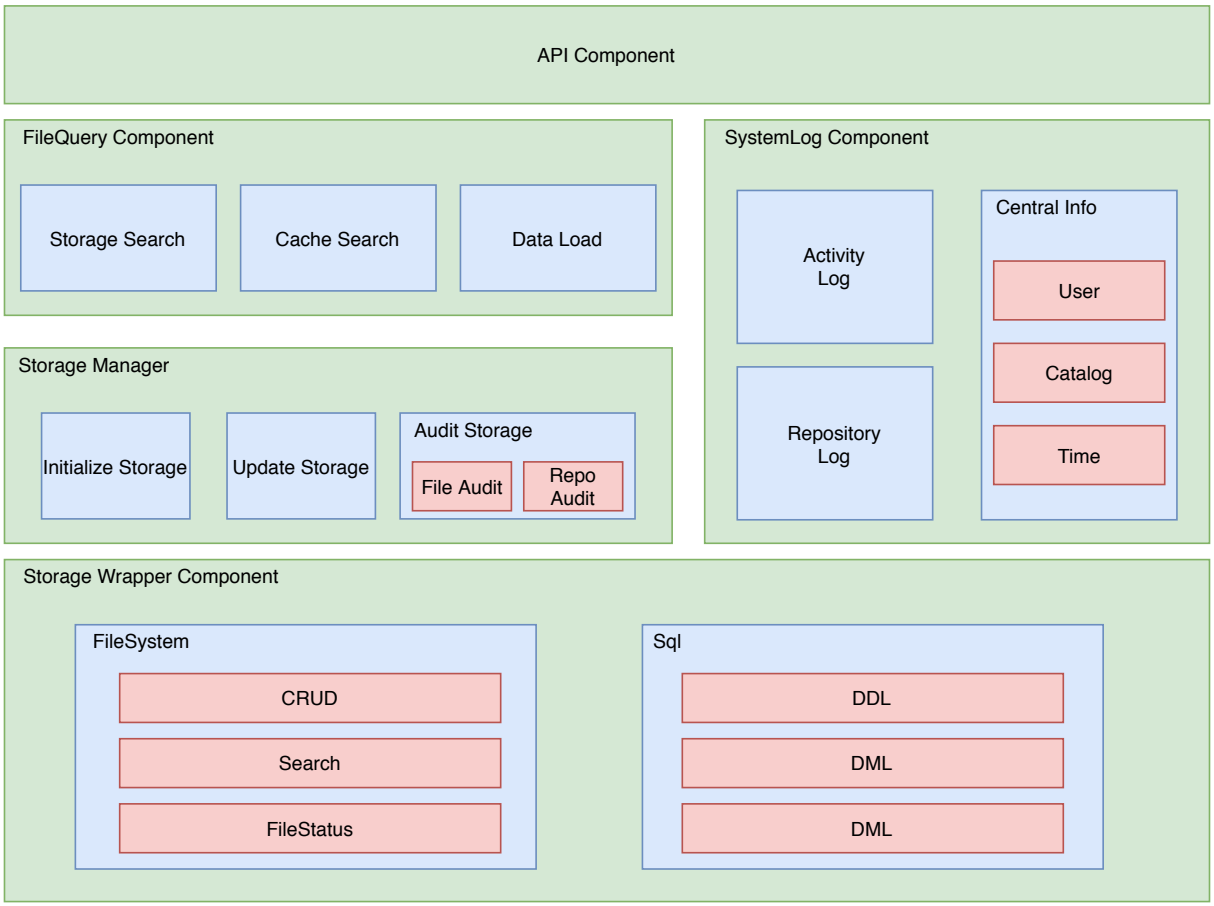

Figure 6: Component diagram of data lake

Components definition

1. **Api**: present the gateway interface between user and system.
2. **FileQuery**: a system functional block handles the data query and retrieval process. Including features:
    1. Storage Search: a realtime and slow query execution process.
    2. Cache Search: similar to storage search but caching search result to speed up the next repeat request.
    3. Data Load: simple stream process to retrieve file content in storage.
3. **StorageManager**: a system functional block handles data ingestion, management, and storage modification. Including features:
    1. Initialize Storage: generate new data repository with related meta info in storage.
    2. Update Storage: ingest data into the repository.

3. Audit Storage: validate the mechanism to guarantee the quality of the repository.
    - File Audit: validate the quality of data in the repository.
    - Repo Audit: repository healthy checking mechanism.

4. **SystemLog**: a system functional block track the system activity and global metadata.

    1. Activity Log: system activity log.
    2. Repository Log: track all repository in storage and their health.
    3. Central Info: store system global metadata.
        - User: track registered users.
        - Catalog: track data label which used to classify data type.
        - Time: a unified time source for the whole system.

5. **StorageWrapper**: present persistence store of the system providing a set storage interface. Include features:

    1. FileSystem: include the basic interface of a persistent filesystem
        - CRUD: Create, Read, Update, and Delete function.
        - Search: the simple pattern searching suck as "grep" in unix-like.
        - FileStatus: the mechanism for getting the metadata of a specific path.
    2. Sql: include the set query in SQL defined language
        - DDL: group query of data definition language.
        - DML: group query of data manipulation language.
        - TCL: group query of the transaction query language.

4.2.3 Sequence Diagrams

I. Manage System

a. Manage System Metadata

Usecase allows users to register or retrieve metadata from the system.

1. Register user/catalog: set metadata to the system

The user sends a JSON string includes request route and register information. The system parses JSON string, initialize route service, call register action, and pass the register parameter to service. The service opens the metafile, validates the parameter and new register, adds a new meta, and updates the metafile, then returns the JSON result which is converted to string passing as a response to the user.

2. Gather user/catalog: get metadata of system

Users send JSON string includes the requested route and gather information. The system parses JSON string, initialize route service, call gather information action, and pass the gather parameter to the service. The service opens metafile, validates parameter, and retrieves related information, then returns a JSON result which is converted to string passing as the response to the user.

3. Get System Timestamp: read clock of the system

The user sends a JSON string that includes the requested route. The system parses the JSON string, initialize route service, call get clock action of service. Service return system timestamp as a JSON result which is converted to string passing as a response to the user.

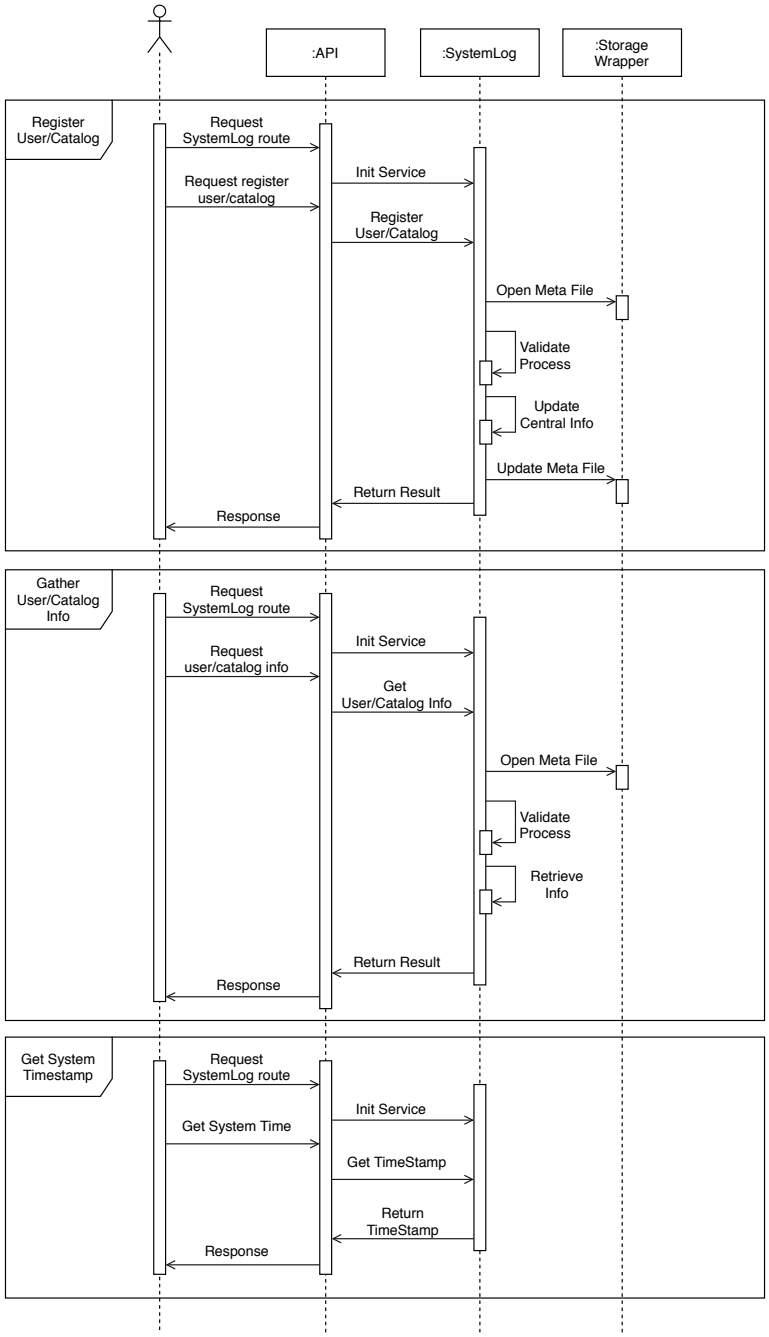

Figure 7: Manage System Metadata

b. Monitor Log

Usecase to gather the log of the system: activity tracking log or repository tracking log.

The user sends a JSON string that includes the requested route. The system parses the JSON

string, initialize route service, call log action of service. The service opens metafile, validates parameter and collects log data, then returns JSON result which is converted to string passing as a response to the user.

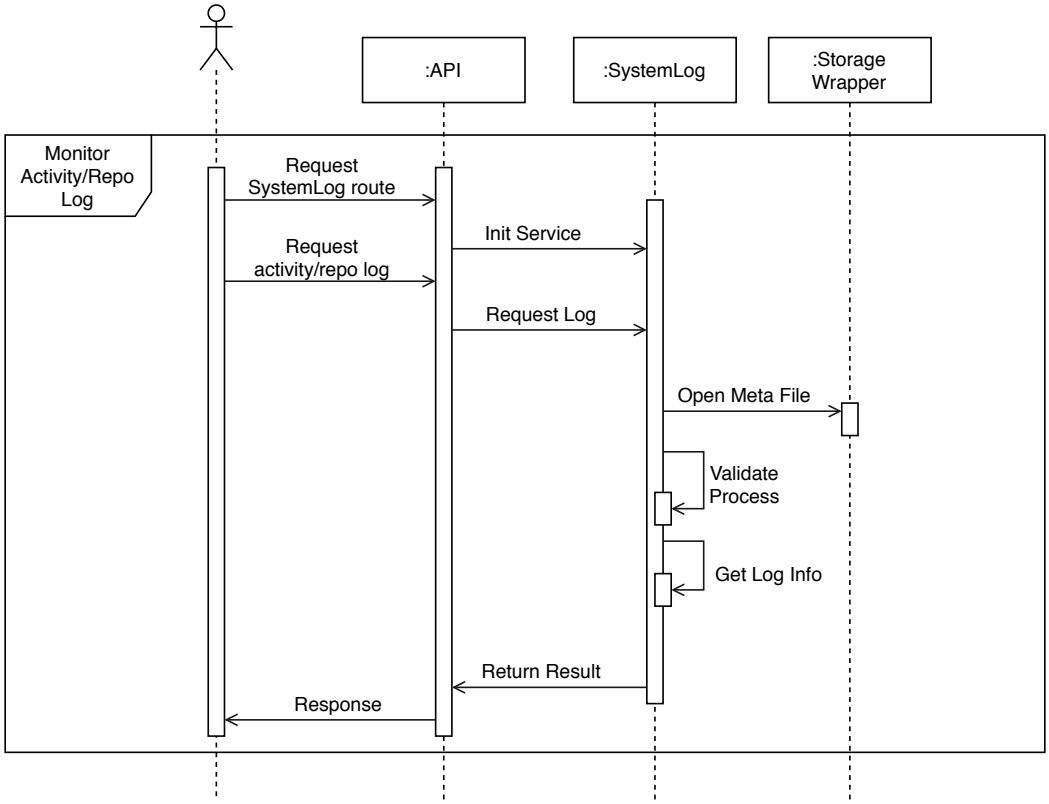

Figure 8: Monitor Log

II. Manage Repository

a. Initialize Repository

Usecase allows users to create a new repository in the system.

The user sends a JSON string includes request route and repository initialization information. The system parses JSON string, initialize route service, call initialize storage action, and pass the parameter to the service. Service validate parameter, generate repository, log its activity, and new repository to SystemLog, then return JSON result which is converted to string passing as a response to the user.

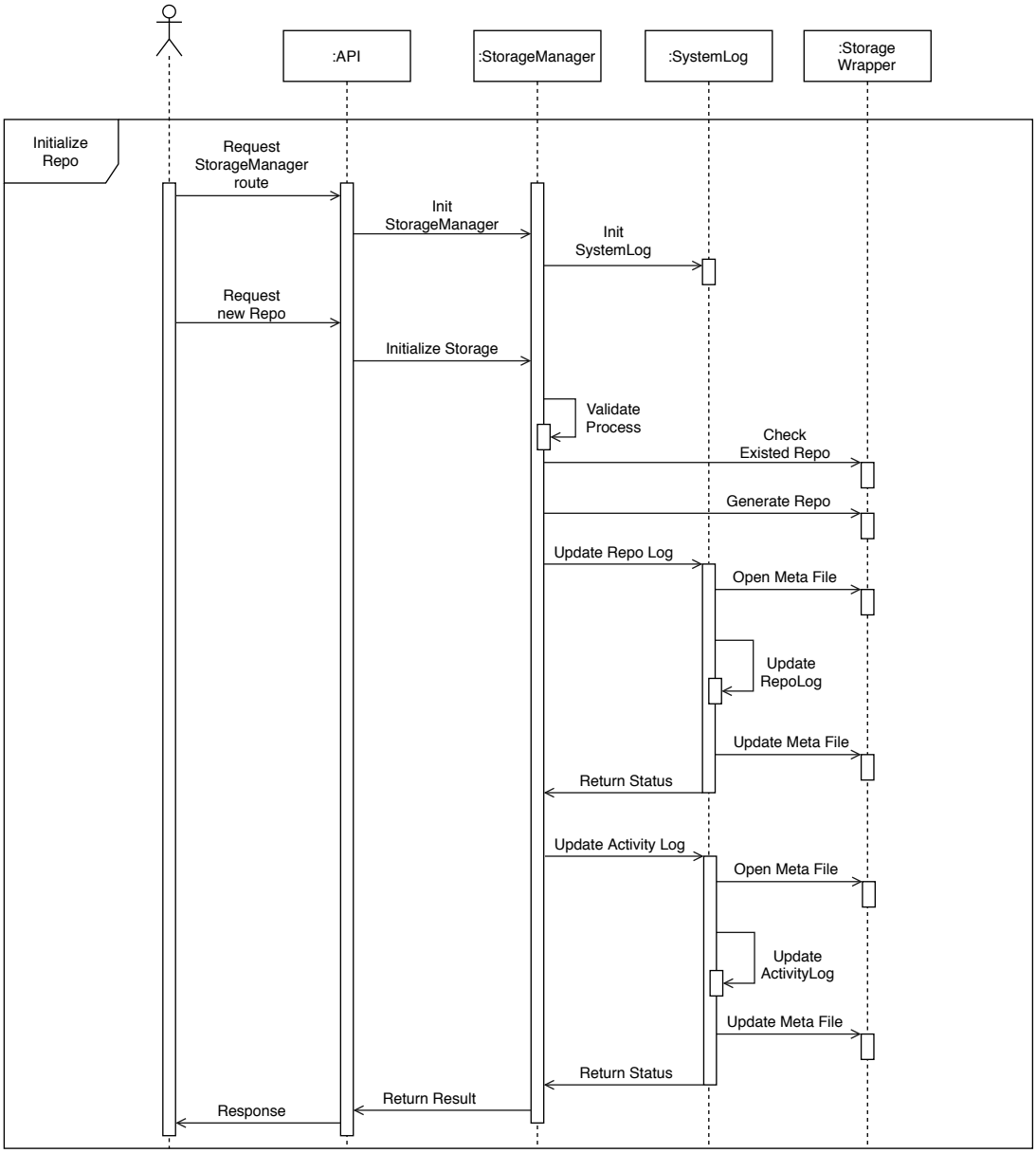

Figure 9: Initialize Repository

b. Update Repository

Usecase is used to upload data to the specific repository.

The user and system open a communication stream. The stream could be split into 3 states:

- State 0 (Route define): User sends route request, the system begins related to service.
- State 1 (Action perform): User sends JSON string includes update request and related

parameter, the system parses JSON, validate action and parameter, then update repo metafile of new data update, setup the stored space for new data, after that, the system sends confirm the status response.

- State 2 (Data upload): User gets a confirmed response, begins to stream data content to the system then call the end stream when finished, system flow stream into new data space until reaching the end stream, next, close new data space, update repository metadata about new data content, logs activity and update changing of repository then returns the response as well as closes stream.

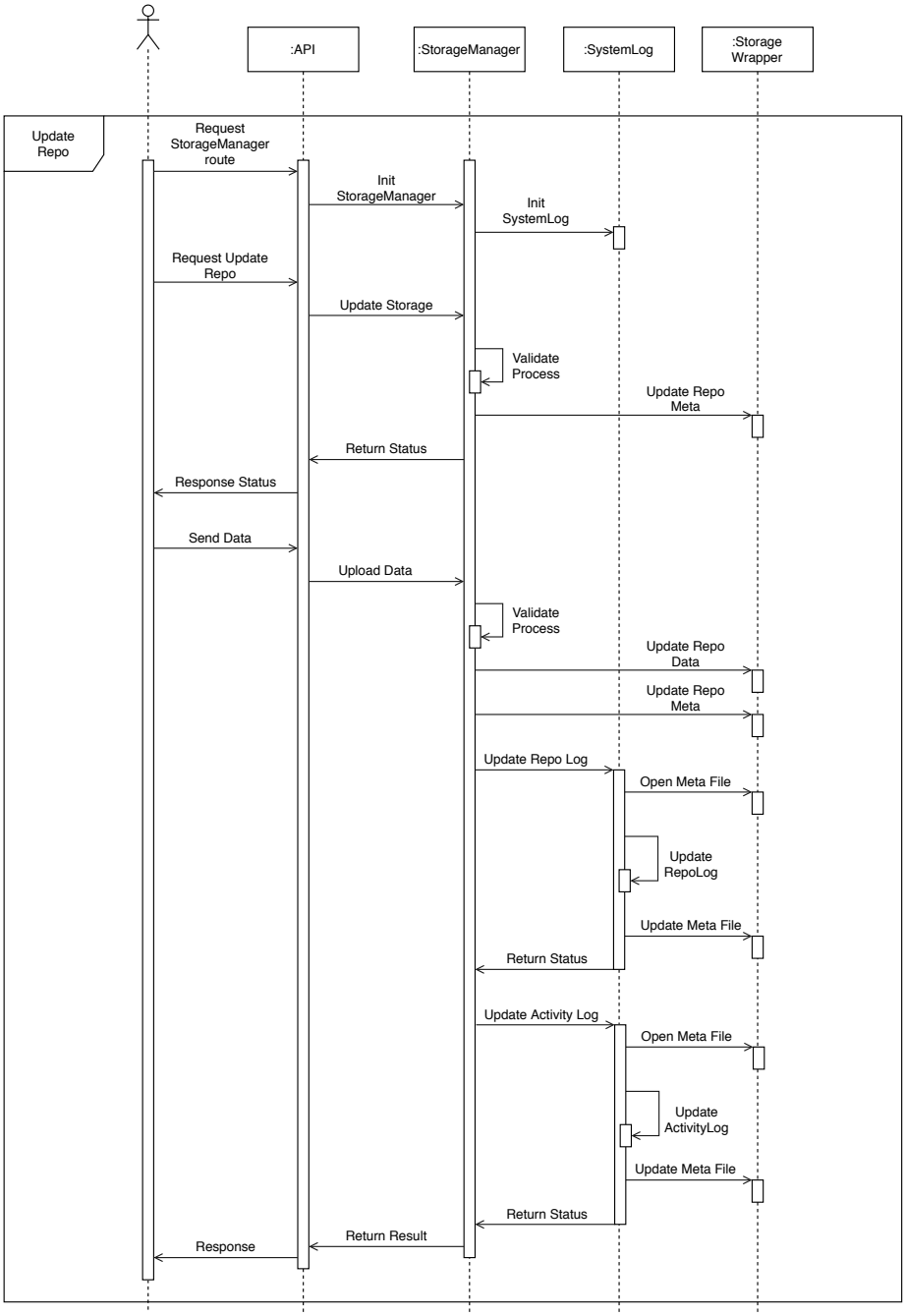

Figure 10: Update Repository

c. Audit Repository

Usecase allows users to validate the quality of a specific repository.

The user sends a JSON string includes request route and audit information. The system parses

JSON string, initialize route service, call audit action, and pass audit parameter to the service. The service validates parameter, opens repository metadata and data, performs a comparison process between metadata and real data content, update activity and repository log, then returns JSON result which is converted to string passing as a response to the user.

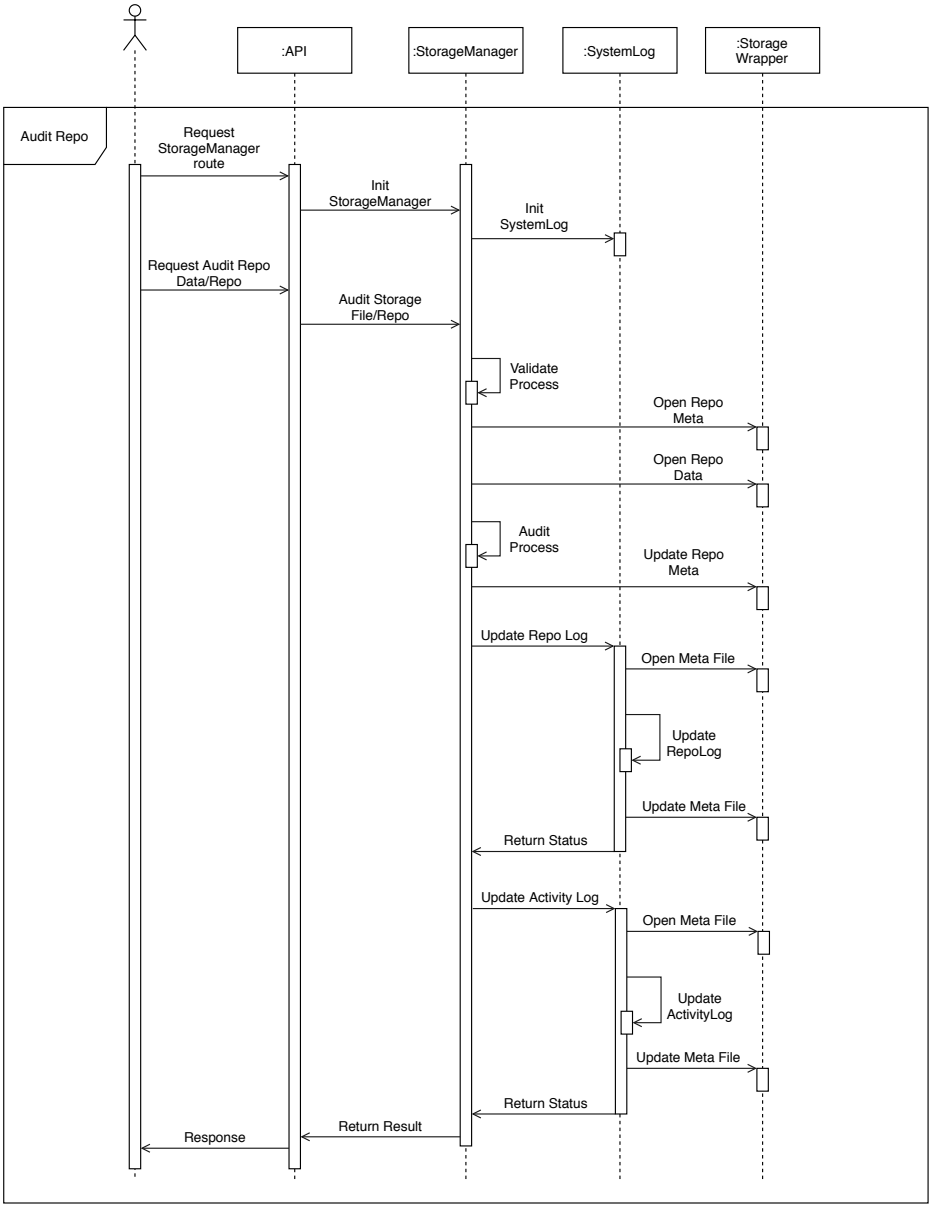

Figure 11: Audit Repository

III. Query

a. Search Storage

Usecase allows users to perform SQL queries in storage.

The user sends JSON string includes request route and SQL query. The system parses JSON string, initialize route service, call storage search action, and pass query parameter to service. The service validates the parameter, opens the repository metafile, builds the actual query action, and send the query to Storage Wrapper to execute it, after that, the system gets back table result, converts it back to JSON format, update the activity log, then return JSON result which is converted to string passing as a response to the user.

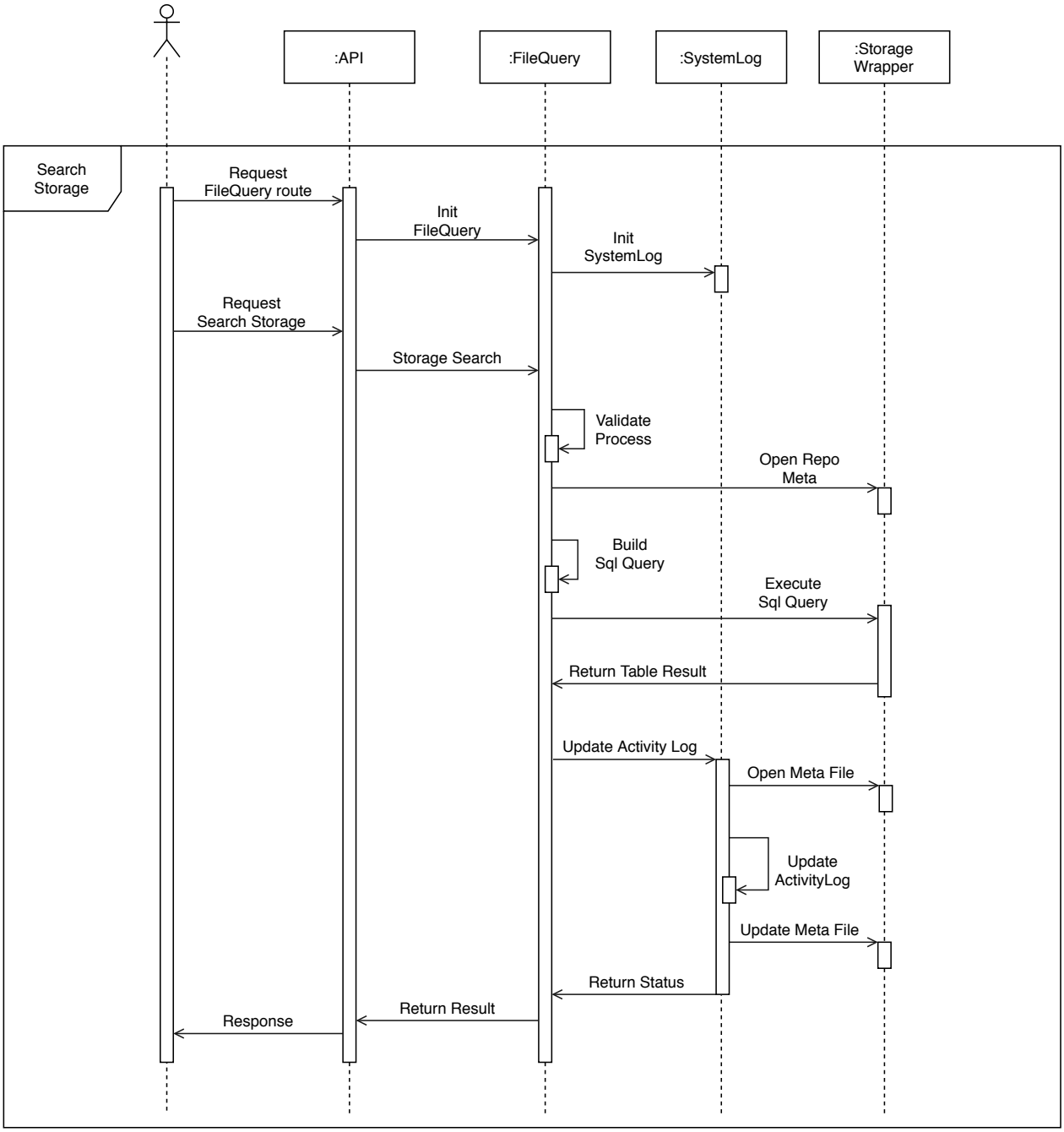

Figure 12: Search Storage

b. Search Cache

Usecase performs the same action as Search Storage but caching results for the next repeated query.

The user sends JSON string includes request route and SQL query. The system parses the JSON string, initialize the route service, call cache search action, and pass the query parameter to the service. Service validates parameter, opens repository metafile, builds actual query action, now, there are 2 sub-use-cases:

1. If the user does not set the refresh option in the parameter, then open the query dictionary and 2 cases could happen:
    1. Search Cache With New Query: the system could not find out a query in the dictionary, and send the query to Storage Wrapper to execute it, after that, the system get back the table result, converts it back to JSON format, saves table result, add the query to query dictionary and update activity log, then return JSON result which is converted to string passing as a response to the user.
    2. Search Cache With Cached Query: the system could find out a query in the dictionary, load the result from the cache and update the activity log, then return the JSON result which is converted to string passing as a response to the user.
2. If the user sets the refresh option in the parameter, then the system clears the query dictionary and cached table result, after that, the system loopback sub-use-case 1.

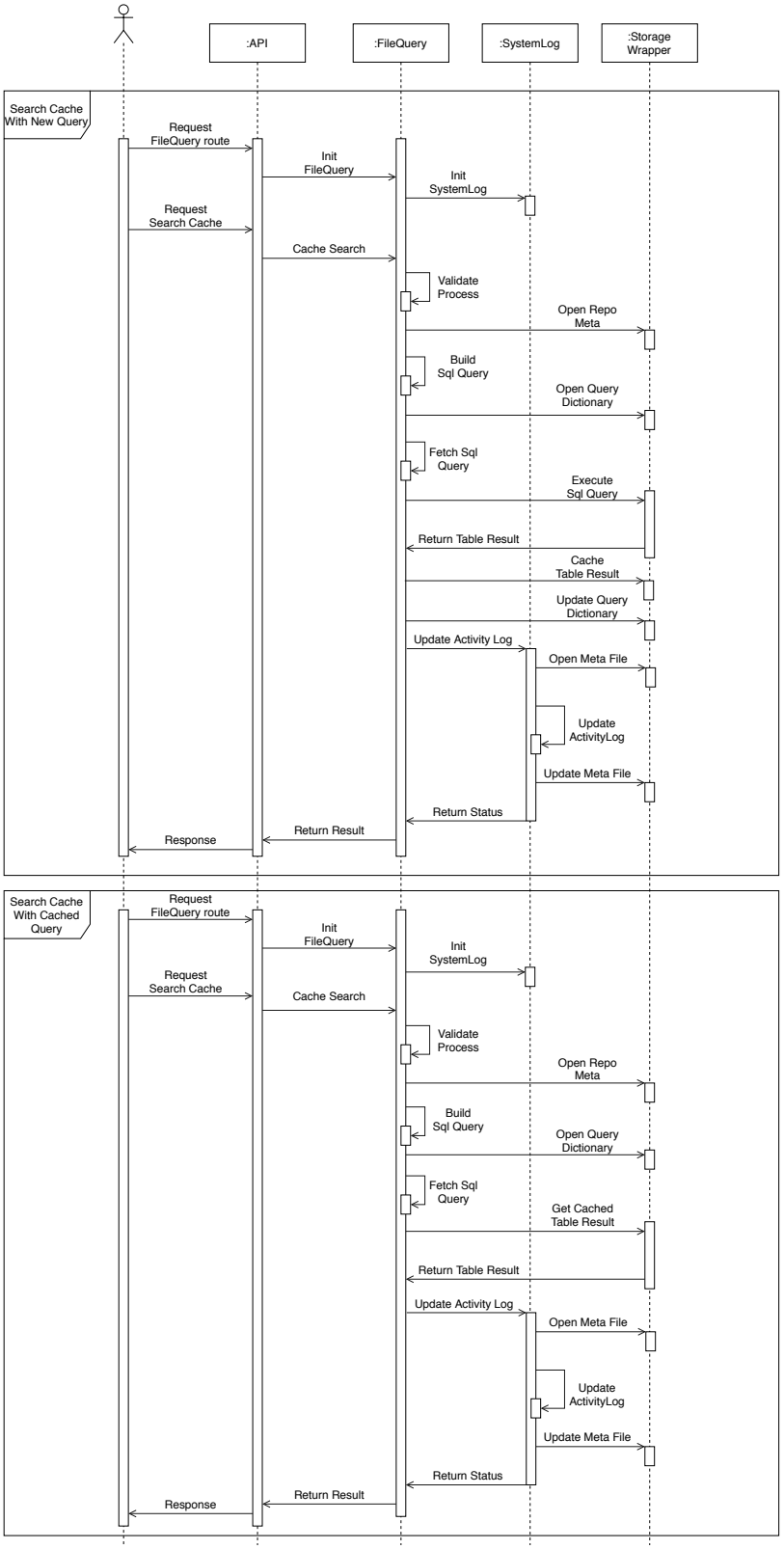

Figure 13: Search Cache

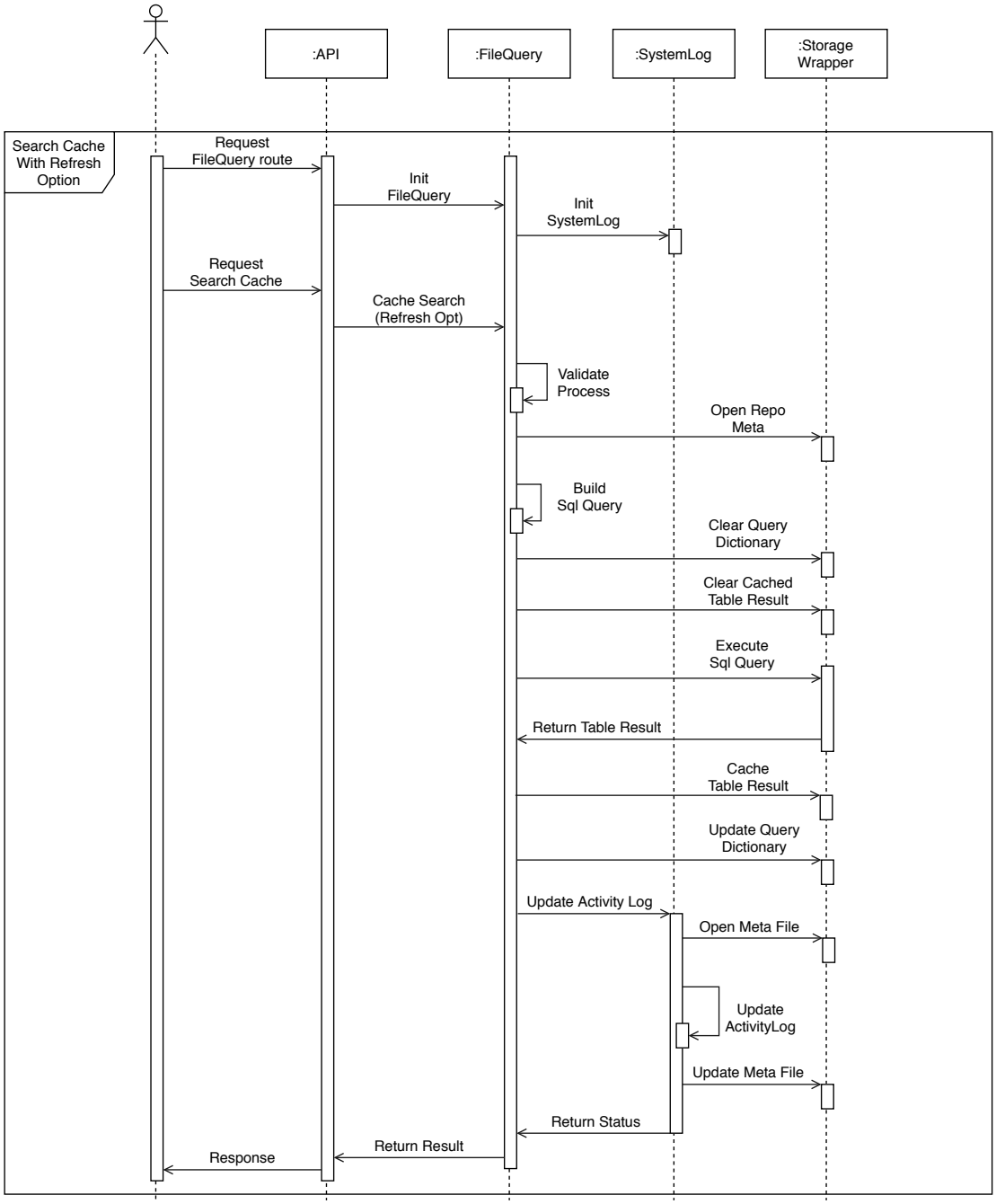

Figure 14: Search Cache With Refresh Option

IV. Download Data

A simple use-case to collect data content from the system.

The user and system open a communication stream. The stream could be split into 3 states:

- State 0 (Route define): User sends route request, the system begins related to service.
- State 1 (Action perform): User sends JSON string which includes download request and related parameter, the system parses JSON, validate action and parameter, opens stream output from data content, update activity log, then send confirm the response.
- State 2 (Data download): User gets a confirmed response, sends "ready to receive data content" signal, system chunks data, and sends to the user after the data sending stopped signal, the system closes data output stream and call end stream.

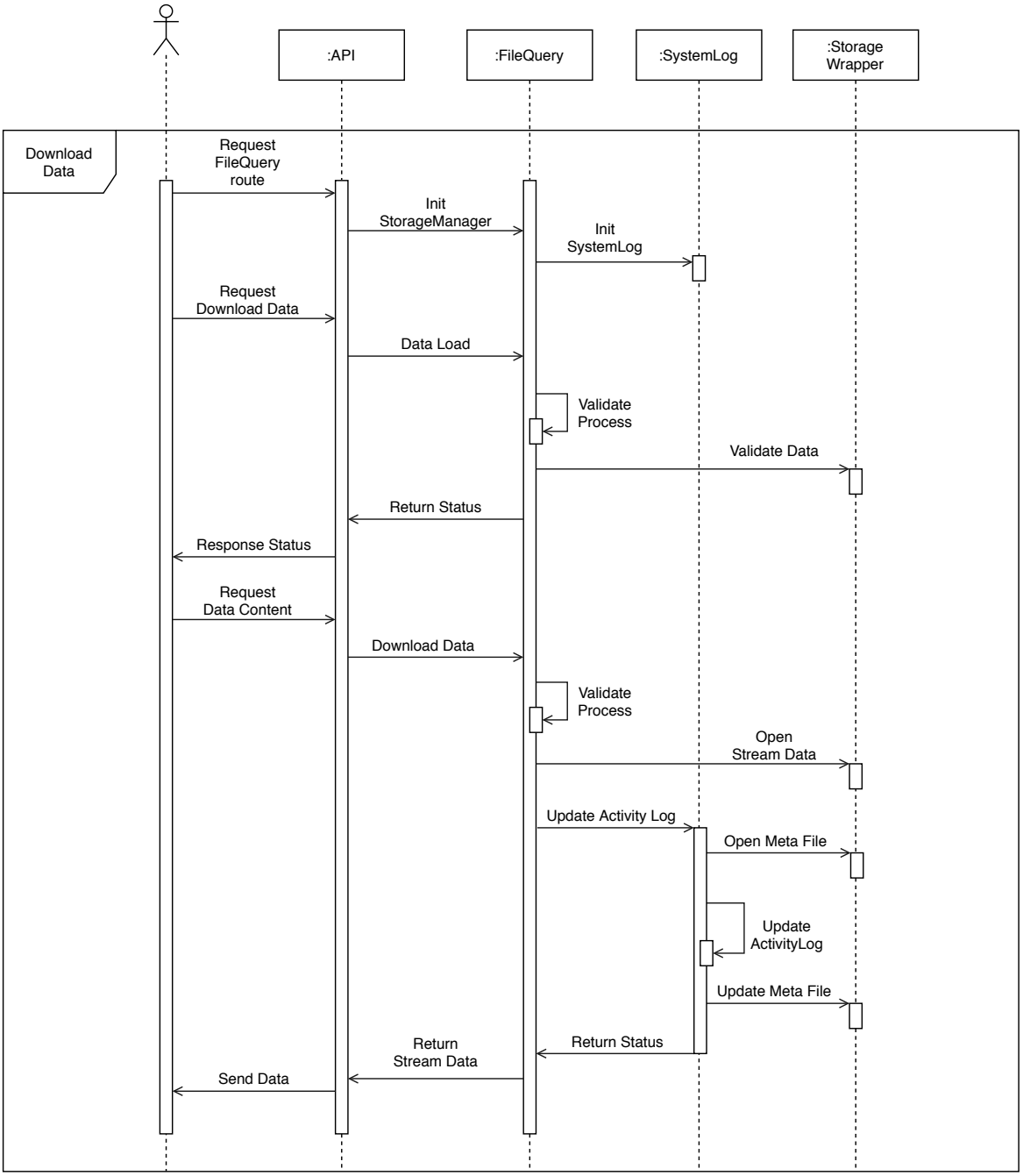

Figure 15: Download Data

4.3 System Architecture Design

4.3.1 Architecture Layout

The data lake belongs to the storage management system class dealing with the large and diverse dataset. Handling this type of dataset is difficult and complicated which could lead to a messy system.

We overcome the problem by splitting the system into 3 abstraction layers: storage, utility and API. The storage abstraction handles the effectiveness in data storing and retrieving while utility abstraction runs on top of storage to propose management and query solution, and API abstraction work as a system interface connecting to another application. The figure below presents the architecture layout of data lake.

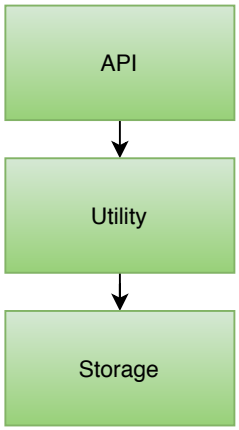

Figure 16: Architecture Layout

4.3.2 System Design

By mapping the system component into the architectural layer, the design diagram displays a concrete view of the system and describes a detailed system mechanism to handle the required features.

Despite fact that the microservice pattern is more popular nowadays, we design the data lake toward a much more traditional monolithic pattern to simplify system interconnection and keep components communication in low latency.

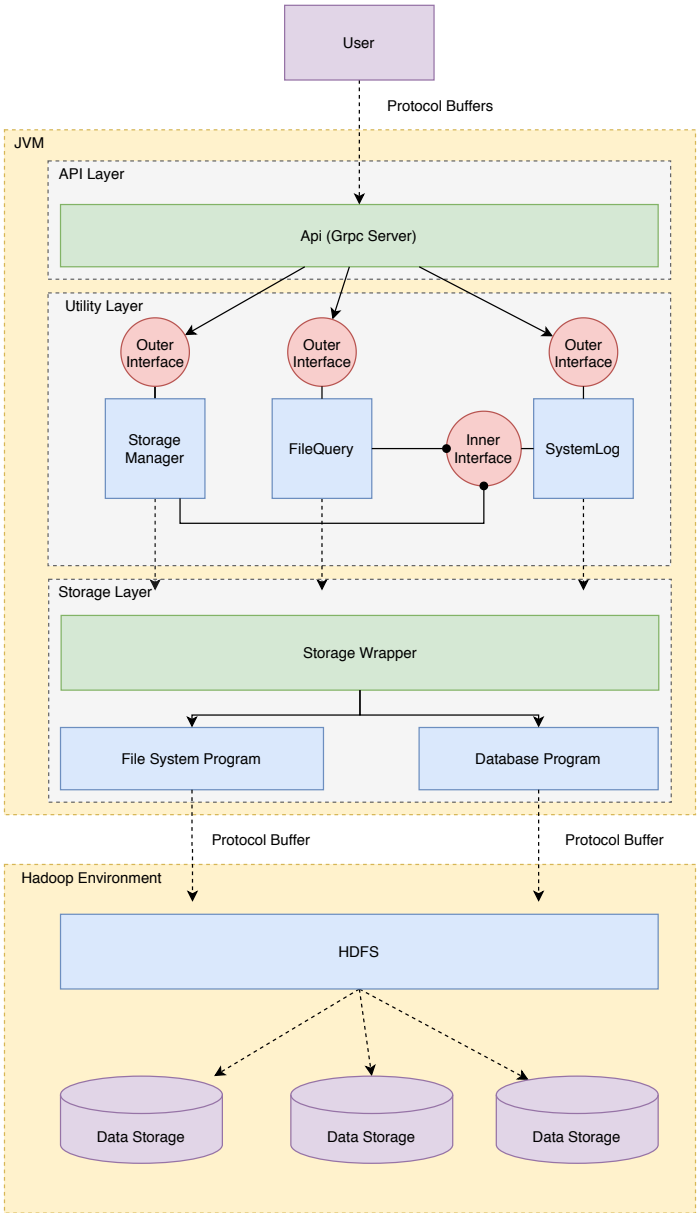

Figure 17: Hivilake System Design Diagram

a. Storage Layer

The storage layer provides a solution to access, store and retrieve data on storage, and handle most of IO issues like a file locking mechanism to simplify storage system call in a higher layer. By providing the data query, this layer could be confused with a simple data lake, however, it does not include any managing strategy or query other than data retrieval, hence this layer is not a data lake but a part of the lake only.

The figure below presents the detailed design of the storage layer.

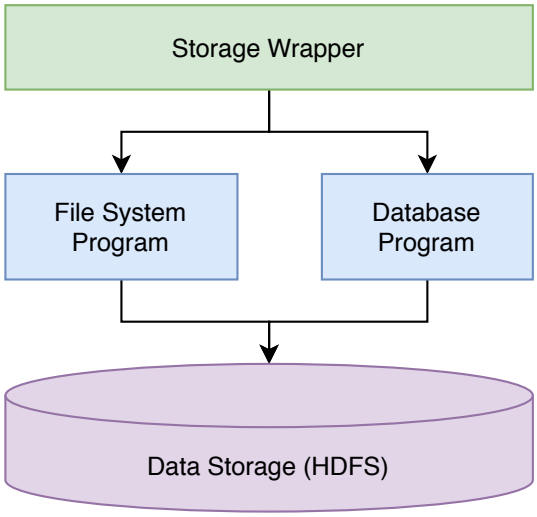

Figure 18: Storage Layer Design

The **data storage** is the software or physical device managing disk and supplying the basic system call to store and retrieve data. The data lake does not consider it being a system part but a platform that the system runs on, and we choose HDFS as data storage for the implementation.

Designed to handle various types of data, and the data lake needs an effective way to handle all stored data. As mention above, the filesystem view is great to access and handle binary files while the DBMS view is pretty suitable to work with tabular data, so the data lake leverage both point of view and propose 2 programs wrap on data storage: File System Program and Database Program.

File System Program views data in the point of filesystem and maps data to binary file or directory, by contrast, **Database Program** maps any data into a table and support SQL action on data. These programs allow a higher layer to process data in a different view.

Finally, **Storage Wrapper** is the wrapper interface that handles both the file system program and database program. Instead of working directly to low-level program as filesystem program or database program, the higher layer call set simple service of Storage Wrapper that, in turn, interact to lower program and handle low-level issue automatically. This mechanism removes the IO complicate and allows a higher layer to concentrate on logical design only.

b. Utility Layer

The utility layer is a logical part of the data lake to provide the management solution and data processing. By applying the storage layer to handle IO process, this layer is purely about

functional block design and how they communicate with each other.

The below figure presents layer design.

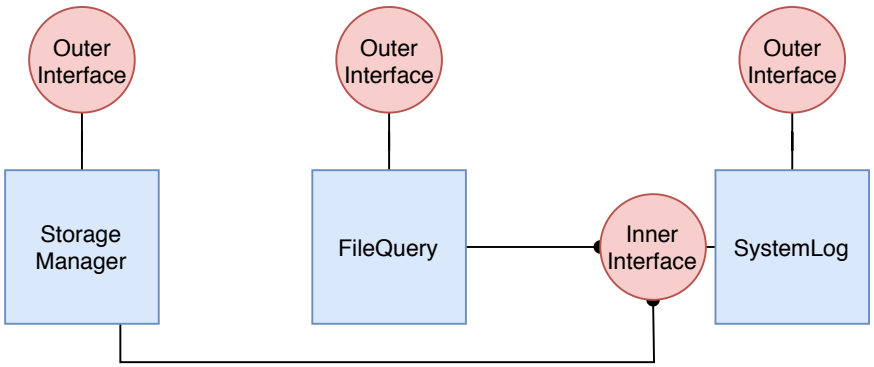

Figure 19: Utility Layer Design

Utility layer is a group of multiple functional units that itself performs a group of features to handle system service. For example, **Storage Manager**, **FileQuery** and **SystemLog** are concreted functional units of system main components that have behaviors unified to their same name key features.

The system does not constraint the inner architecture of the functional unit as long as it performs right expected behavior, however, each functional unit is required to have a single main that represents the unit block in concept. This unit point contains Inner Interface and Outer Interface allowing another unit or outer layer element to interact with that unit.

In semantic, **Inner Interface** used for inner layer interaction while **Outer Interface** could be called in anywhere. Although these interfaces are interchangeable, they should be used as their semantic meaning.

c. API Layer

The application programming interface (API) defines the interactions interface between multiple intermediary software that could be used to linking any arbitrary program on unrelated processes or even unrelated machines.

Since many users do not only want to fetch data from the lake but also feed those data to their software (such as BI tool, ML model or Analysis tool, etc.), and obviously, they would like to automate these task by building a pipeline from the data lake to their software which request a connector between a data lake and outer software. Therefore, we build a simple API layer as the main interface of the system to leverage its high compatible attribute on simplifying the connector development process.

The below figure is an example of API layer design applying Grpc technology.

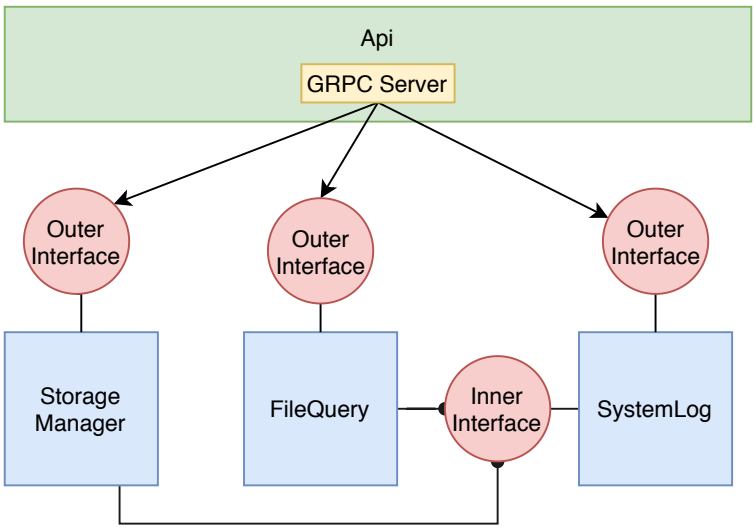

Figure 20: API Layer Design

In this design, the Grpc server works as a router that receives the request, passes it to the right utility unit through the outer interface, waits for a response, and sends back to the user. To make sure the system compatibility, the design is kept to be short and simple.

4.4 Implementation

In this sub-section, we propose a simple java implementation of our system that could perform basic functional features. However, the actual implementation classes are quite big and complex, so in this sub-section, we will only show the Class and Interface with their connection that builds up the system without its detail.

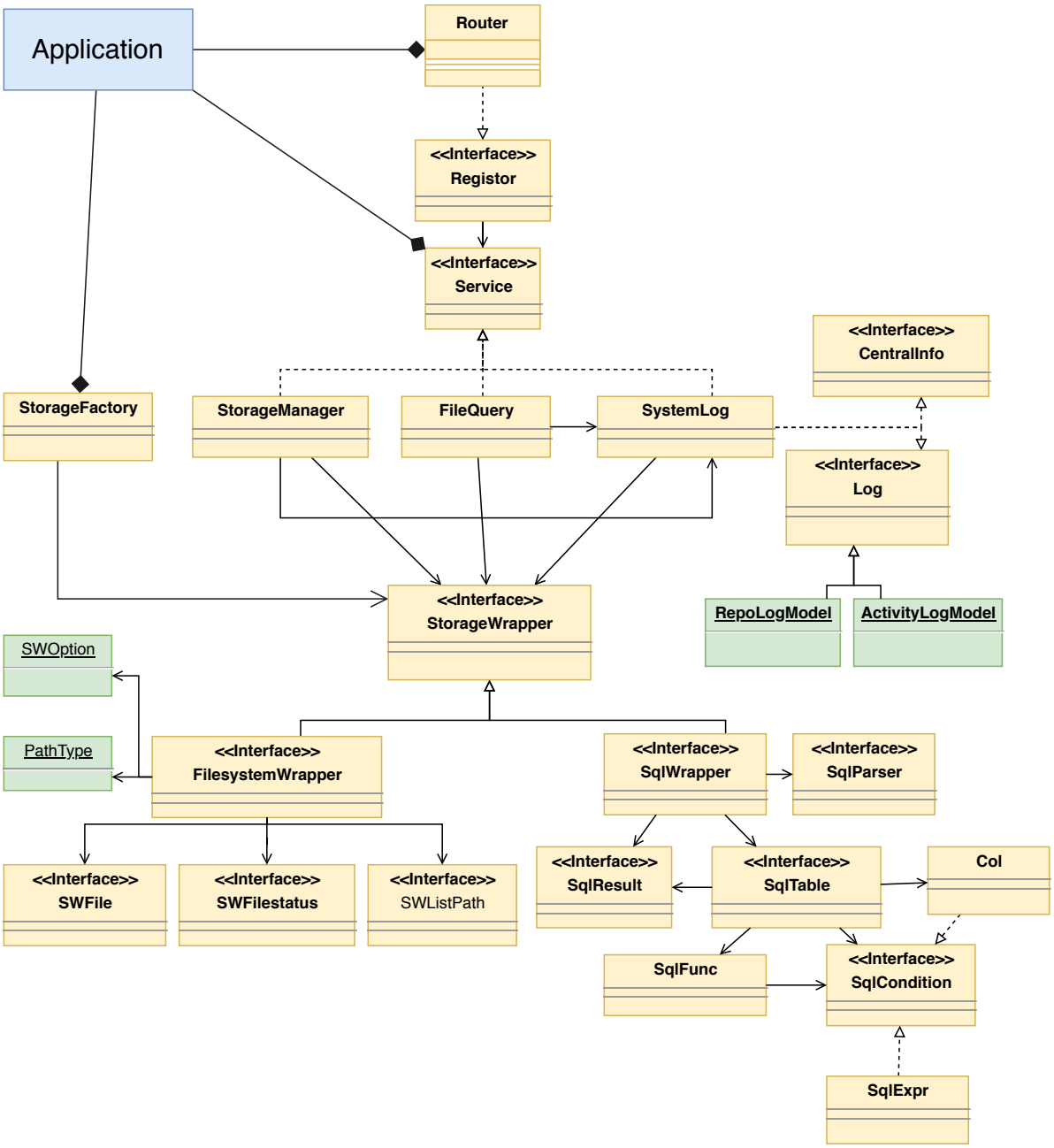

Figure 21: Class Diagram

Class Diagram Explanation

1. **Storage Layer**:
    1. **Storage Wrapper Interface**: represents the storage wrapper in design. It contains FilesystemWrapper and SqlWrapper Interface.
        1. **FilesystemWrapper Interface**: constructs the filesystem program in design. It contains:
            1. **SWFile Interface**: presents a file in the file system.
            2. **SWFilestatus Interface**: presents file metadata.
            3. **SWListPath Interface**: presents a list of SWFilestatus.
        2. **SqlWrapper Interface**: defines our SQL engine interface that could parse and execute some simple SQL queries. Although many components are included in this interface, they are too technique thus we do not provide their definition.
    2. **StorageFactory Class**: defines factory class to create the instance of StorageWrapper.
2. **Utility Layer**:
    1. **StorageManager Class**: implements StorageWrapper component in design.
    2. **FileQuery Class**: implements FileQuery components in design.
    3. **SystemLog Class**: implements SystemLog components in design.
    4. **Service Interface**: presents the outer interface of the utility unit block in the design.
3. **API Layer**:
    1. **Router Class**: implement the Grpc Server in design.
    2. **Registor Interface**: defines an interface that allows developers to register utility outer interface to the Grpc server so that it could be accessed from outside users.
4. **Application Class**: the main class contains the "main" method to initialize the system.

4.5 Tools & Technical Choices

4.5.1 Java (language)

There are many option programming language, but the java is suited for implementation in this internship. The java is an object-oriented programming language (OOP), hence it handles well to some object-oriented design in the system, especially about utility functional unit concept.

Additionally, by providing a garbage collector and class locking mechanism which is lacked in another low-level language (like C/C++), java support multi-process and resource management in native, so the scalability in java system is better. Besides, unlike many high-level languages

(for example as Kotlin or Python) with a poor document, JavaDoc is useful and straightforward, hence simplify the development process a lot.

4.5.2 Maven (project management)

Maven is a build automation tool for java project which is normally used as project management program that simplify java building process and support many useful utility on managing jar package. One critical strengths of maven is maven repository [5] which is a centralized repostiory of many pre-compiled jar packages, plugin and dependencies, hence saving a lot of compilation time for project dependencies.

4.5.3 Hadoop Distributed Filesystem (HDFS)

Apache Hadoop[8] is a collection of many open-source software utilities the power of cluster to solve a massive amounts of data and computation. HDFS is a part of Hadoop environment that provides a distributed filesystem which have a large and extensible volume and high-throughput IO processing. We build our architecture on top HDFS platform in order to satisfy non-functional requirement, but our architecture itself is platform independent system.

4.5.4 Json-simple

During implementation time, we need a refined structural protocol to share the complicate data inside system, and JSON is our final choice, hence we apply json-simple which is a simple java library for JSON processsing that includes read and write JSON data that follows JSON specification(RFC4627)[6] to handle JSON data in the system.

4.5.5 Grpc

gRPC[7] is an open source remote proceduce call system that is developed by Google applying protocol buffer method to define its transfer data type. This is a high performance RPC framework support multiple programming languages that can connect different program together. We choose this framework because of its simple definition protocol and fast configuration.

5. Result

5.1 Grpc Protocol

We defines 3 services in Grpc Protocol that present 3 types of request: single request-response, server stream response and client stream request. These requests allow outside uers could access and interacts to our implementation system.

```
message ActionRequest{
  string route = 1;
  string jsonAction = 2;
  string jsonParam = 3;
}

message StatusResponse{
  string system = 1;
  string action = 2;
  string result = 3;
}

message Chunk{ // chunks of data
  bytes data = 1;
}

service StorageManager{
  // single request-response service
  rpc setup (ActionRequest)returns(StatusResponse){}

  // client stream request service
  rpc upload_file(stream Chunk)returns(stream Chunk){}

  // server stream response service
  rpc download_file(stream Chunk)returns(stream Chunk){}
}
```

Detail explanation:

1. **Services**:
    1. **setup** - for normal 1 request-repsonse connection.
    2. **upload_file** - for streaming connection to upload long data.
    3. **download_file** - for streaming connection return long data.
2. **Messages**:
    1. **ActionRequest** - request message of normal connection
    2. **StatusResponse** - response message of normal connection
    3. **Chunk** - main message in streaming connection

5.2 API

To demonstrate that our design work, we propose a list of API request and result. These API will be constructed to follow the system components.

Warning

```
Util point of view:
For version 0.1.0-SNAPSHOT, it has 3 basic utilities:
- StorageManager
- FileQuery
- SystemLog

(2020-09-22 | WARNNING: this doc ver was not built from release ver
    -> it is not completed)
```

SystemLog API

- route: “SystemLog”

Action

```
1. countActivity
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{}"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: "{
            row_count: String
        }"

2. listActivity
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{}"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
```

```
        - result: Json String

3. listRepo
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{}"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: Json String

4. listUser
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{}"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: Json String

5. listCatalog
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{}"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: Json String

6. getUserInfo
```

```
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{
            name: String
        }"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: "{
            user_info: String
        }"

7. getCatalogInfo
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{
            name: String
        }"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: "{
            catalog_info: String
        }"

8. registerUser
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{
            name: String,
            describe: String
        }"
    - StatusResponse:
        - system: "{
            error: []
        }"
```

```
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: ""

9. registerCatalog
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{
            name: String,
            describe: String
        }"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: ""
```

StorageManager API

- route: “StorageManager”

Action

```
1. createRepo
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{
            path: String,
            schema: {
                fields: ["extra_field_1", "extra_field_2", v.v.],
                describe: String,
                type: String (FILE | DIR),
                status: String,
                notes: String
            }
        }"
    - StatusResponse:
        - system: "{
            error: []
```

```
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: ""

2. updateRepo
    - Service: upload_file
    - State:
        0. define route:
            - request: String (route name)
            - response: "{
                action: {
                    code: String
                }
            }"
        1. setup paramter
            - request: "{
                action: String,
                parameter: {
                    repoId: String,
                    meta: {
                        user: String,
                        name: String (path name),
                        type: String (FILE/PATH),
                        format: String (exp: csv),
                        label: String (catalog),
                        extra_field_1: String,
                        extra_field_2: String,
                        v.v.
                    }
                }
            }"
            - response: "{
                action: {
                    code: String
                }
            }"
        2. upload file raw data in chunk (512 kb)
```

FileQuery API

- route: “FileQuery”

Action

```
1. searchStorage
```

```
    - Service: setup
    - ActionRequest:
        - jsonAction: String
        - jsonParam: "{
            repoId: String,
            expr: "{
                path: String (option),
                select: ["field_1", "field_2", ...] (option),
                where: [
                    {
                        field: String,
                        operator: String ("=", ">", "<");
                        value: String
                    },
                    {
                        field: String,
                        operator: String ("=", ">", "<");
                        value: String
                    },
                    v.v.
                ] (option),
                order_by: {
                    field: String,
                    value: String ["ASC", "DESC"]
                } (option),
                limit: String (option)
            }"
        }"
    - StatusResponse:
        - system: "{
            error: []
        }"
        - action: "{
            status: String,
            message: String,
            code: String
        }"
        - result: Json String

2. loadData
    - Service: download_file
    - State:
        0. define route:
            - request: String (route name)
            - response: "{
                action: {
                    code: String
                }
            }"
        1. setup paramter
            - request: "{
```

```
53                  action: String,
54                  parameter: {
55                      path: String (path of file)
56                  }
57              }"
58              - response: "{
59                  action: {
60                      code: String
61                  }
62              }"
63          2. download file
64              - request: String (200 - READY)
65              - response: stream file raw data in chunk (512 kb)
```

6. Conclusion & Future Work

In this project, we study, design and implement a data lake architecture that could collect, store, manage, and retrieve the various type of data which is applied to managing ICT lab scientific data. However, our research is still in an early state that is required a lot of effort to be better both in design and performance. In near future, we have some point of goals:

- Implementing all features which is not yet done.
- Testing and fixing all bugs and errors in the system.
- Updating Storage Layer to support more data viewpoints.

Reference

[1] **Data lake definition source - James Dixon**: https://www.thalesgroup.com/en/markets/digital-identity-and-security/magazine/beginners-guide-data-lakes

[2] **Data lake definition source - AWS**: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake

[3] **Teradata announcement about Kylo project**: https://www.teradata.com/Press-Releases/2017/An-Industry-First-Teradata-Debuts-Open-Sourc

[4] **Kylo homepage**: https://kylo.io

[5] **Maven repository**: https://mvnrepository.com

[6] **Simple-json tutorial**: https://mkyong.com/java/json-simple-example-read-and-write-json/

[7] **gRPC main page**: https://grpc.io

[8] **apache hadoop main page**: https://hadoop.apache.org