Universiteit Utrecht

Documentation

SearchSECO: A Search Engine for the Worldwide Software Ecosystem

Part of SecureSECO

July 5, 2022

1 Introduction

This is the full documentation of the SearchSECO system. It was created by a group of students from Utrecht University as Software Project in 2021 and then further developed after the conclusion of the project. We first give a brief overview of the system, after which all components are documented in detail individually. Following this, we discuss several interesting observations we made. Finally, we provide improvements that could be made in the future. Some parts of this documentation might be slightly outdated compared to the current version of the software

Contents

2 Overview

The system is built up out of several pieces of software, which can be divided in client-side and server-side. The server-side consists of the Database, its API and the Jobqueue. The client side consists of the Controller and its subcomponents (Crawler,Spider and Parser). The servers are manager by trusted parties. A client can be run by anyone on the world who want to contribute to the SearchSECO system. All components (including how they work together) are documented in detail below.

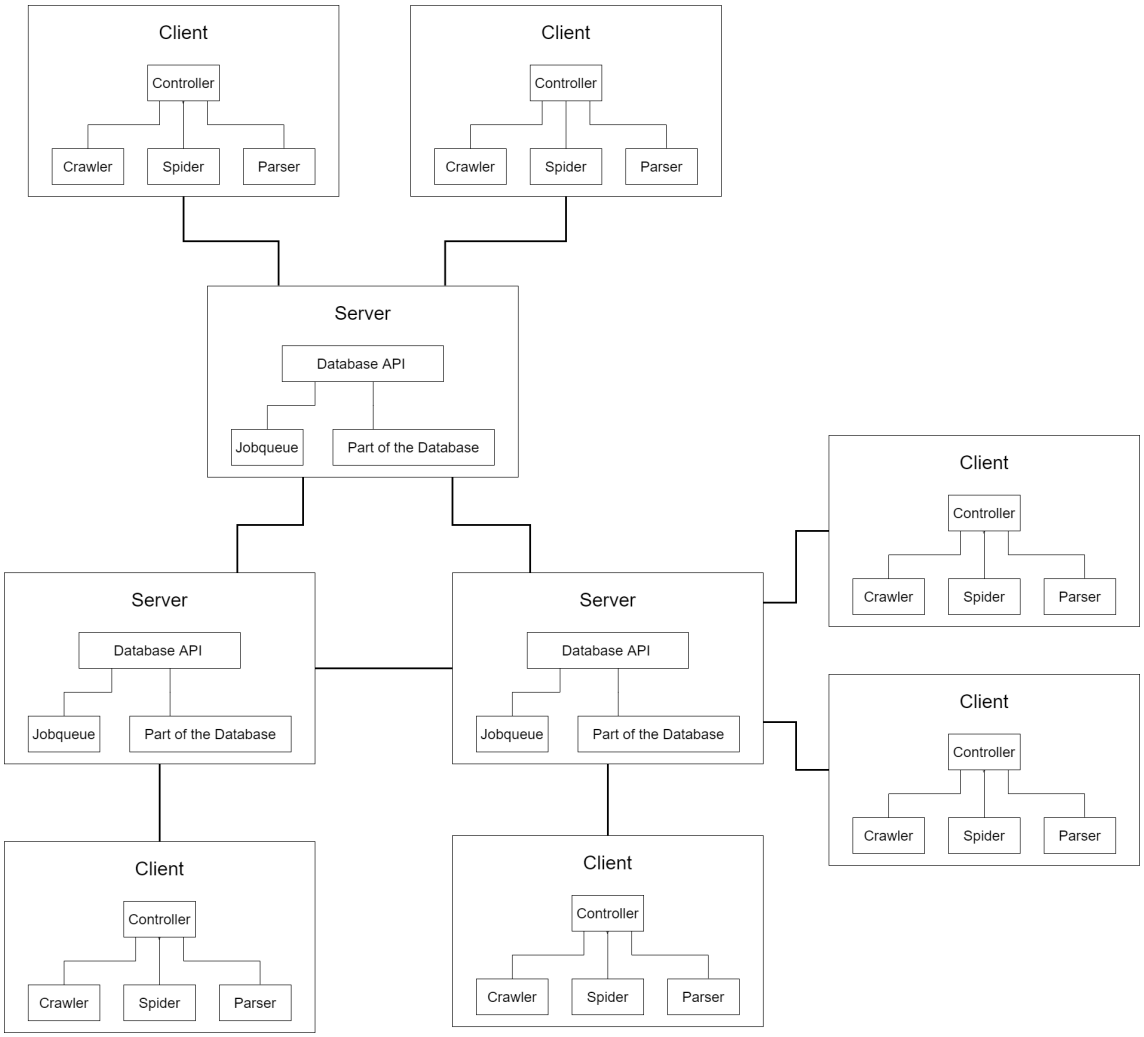

Figure 1: An overview of the complete system

3 Controller

3.1 General structure

The Controller is responsible for handling user input, executing commands and communicating with the rest of the system. All other submodules of our program only communicate with the Controller and not with each other. The Controller is also the program that the users are going to interact with. User commands will be given to the Controller and it is the Controllers job to decide what needs to be done with this user input. Most of what needs to be done with the user input will be handed of to the subcomponents of our program, as the Controller itself is not responsible for most of the logic. The Controller is also responsible for all network communication with the Database api.

User input
Our system does not expect a lot of user input. Almost all user input is done at startup, either through command line arguments or typed in after starting the program. The Controller expects the same input in both cases. The first thing the Controller expects is the command to be executed. Another argument might be expected right after the command, depending on which command is typed. After that is room to set additional flags. Each flag has a shorthand version and a long version. The shorthand version will always be in the format -(one character), e.g. -b for branch. The longhand version is the name of the flag with two dashes in front (e.g. --branch).
Flags can also get a default value from a config file. This config file will be read when parsing the user input. If the user input contains a certain flag, then that value will be used. If the input does not contain that flag but the config file does, then the value in the config file will be used. Otherwise a default value will be used.

Flags
There are two flags that work more like commands, these are the -v (version) and the -h (help) flags.
The version flag will print the version of the Controller and its submodules into the console.
The help message will print all available commands into the console. You can also give it a specific command. If you do, only the help message for that command will be printed. All flags that are valid for the command will also be printed to the console.
Here is a list of the other flags that can be set:

- -V verbosity: Sets the verbosity level of the console output. This can be used in combination with any command and can be set to 5 levels: Log all debug information (5), log everything except debug information (4), only log warnings and errors (3), only log errors (2) or log nothing at all (1).
- -c cores: Sets the amount of threads the program is allowed to use. This can be used in combination with any command.
- -b branch: Sets the branch that will be downloaded from the repository. Can be used with the Check, Upload and Checkupload commands. Without this flag, the default branch (indicated by GitHub) is used.

Executing commands
Once the user input is parsed, it is time to execute the command. We have created a Command super class, from which we derive a class per command. These derived classes override an `execute` function that defines the logic to execute the command. They also contain the help message string for the command it represents. We have created a Factory that will return an instance of the correct derived class based on the user input.

Commands
A list of commands that can be executed by the Controller:

- **Check:** The `check` command will parse the most recent version of a given git repository, and finds all matches between the methods found in that repository and the methods stored in the database. First, the given repository is cloned in the Spider. The Parser then extracts all methods from the cloned repository. Finally, the Spider extracts local author data by blaming the files in the repository. The only data that is sent to the Database API are the parsed method hashes. The Database API responds with all known methods that have the same hash as one of the sent hashes. To combine the data received from the Database API with the local authors and methods into understandable output, the dedicated `printMatches` class combines the Spider output, the Parser output and the Database output to give a summary of all matches found. To achieve this, two additional database requests are sent: One to get the names of the authors that the database returned, and another to get the metadata for the projects found in the matches.

- **Upload:** The `upload` command processes all tags of a given repository, and upload the methods found to the database. First, the Crawler is instructed to retrieve metadata for the repository, like the license and default branch. Next, as with every other command, the Spider is instructed to clone the repository. It also retrieves all tags of the repository. The Controller then instructs the Spider to revert to the first tag and begins looping through all tags. Per tag, the Parser is called to extract methods and the Spider retrieves authors per file. The Controller combines this data to derive which authors worked on a certain method. Together with the project data, all this data is sent to the database and stored. From tag to tag, the Controller checks which files remain unchanged. Since this means the methods in these files also were not changed, no processing needs to be done on these files. This list of unchanged files is sent to the database to update the project version for the methods in these files.

- **Checkupload:** The `checkupload` command both checks the most recent version for matches with the database, and uploads every version of the given repository. This is mostly an ease-of-use command and is implemented by simply running both the `check` and `upload` commands. The output of the `check` command gives an overview of the encountered matches.

- **Start:** The `start` command starts a worker node. The idea of a worker node is that it asks the Database API what it needs to do, and then attempts to complete the assigned job. Once it is done with its job, it will ask the Database API for another. This repeats until the user inputs `stop`, after which the worker node will finish its current job and then stop.
  There are 2 types of jobs the Database can give back to the Controller: A Crawl job and a Spider/Parse job. If the Controller gets a Crawl job, then the Crawler is instructed to crawl repositories. These repositories are then sent to the Database API. For the Spider/Parse job, the exact same function that is used for the `upload` command is called.
  There is one more valid response the Database can give to the Controller: A NoJob response. This means that the Jobqueue is empty and another worker is already running a Crawl job. In this case the worker waits for a while and then asks for a job again.

Error handling and logging
SearchSECO uses an integrated error handling and logging system, using the Loguru [9] library to handle logging. Several verbosity levels can be specified by the user when executing a command. Only the messages with a verbosity level lower than the threshold are displayed. The system generates two log files: one that contains all logs of every verbosity level from all runs, and a smaller file that only contains the warnings and errors from the last run.
To identify and keep track of errors, each submodule has its own range of error codes, which they can define and use independently. Considering the DRY development principle, the actual message logging and program termination are grouped in a single base function. This function has several facades, which are called throughout the program as specific error handlers.

The linking of error codes to their descriptions is set up in a very modular way, so new error messages can easily be added, inserted, or reordered.
The Controller retains the responsibility to terminate execution, so if a submodule runs into a non-fatal error, it is allowed to display its error message only. The fact that it ran into an error then has to be communicated back up to the Controller, so it can decide to retry, or to skip the project. If the state is corrupted such that continuing is actually impossible, only then will execution be terminated. The philosophy is that autonomous worker node should always remain running, and only crash when that is the only option left.

Using the submodules
The Controller uses the three submodules described below. These are all called via their own interfaces. Should someone want to replace one of these components, the only Controller code that has to change is that part. Also newer components with additional functionality can be easily added. The Controller is responsible for all communication between components and between components and the user. This includes throwing errors and warning the user if something is wrong.
Each submodule is only called in one place of the code. This is all done in a separate module facades class. In this class we have a separate functions for each submodule entry point that we need to call. This means that, if a submodule changes, that we only need to change it in one place in the code.

Client-side server-side communication
For sending network request and receiving them on the API we use the ASIO Boost library[10]. Each request to the database goes in the same way:

1. The Controller sends a header to the Database API. This header contains a four letter code, referring to what type of request. After this four letter code is a number corresponding to the number of bytes we are going to send for this request. There will be a end line character at the end signalling the end of the request header.

2. The body of the request. Here we send the actual data that we want to send to the database. The size of the body should be the same as the size that was given by the header of the request.

3. Receiving a response. The database will always respond with something. The response will always start with a http status code to signal if the request was handled correctly or not. The requested data will follow after the http status code.

Data sent and received will all be in a string format. In this string, a end line character is used to signal the end of an entry, and a question mark is used to signal the end of a parameter in an entry.
In the Controller, each Database request has its own method. However, because they are all largely the same, they all use a internal request methods to handle the actual logic of executing a request. The individual methods are to transform the data to the string format that the database expects, and to add the right four letter code. The transforming of the data to a string is done in a separate class `networkUtils`. The methods in this class take the data that will be send, and transform it into a character buffer that has to the correct format for a database request. The Database API will give back an error if the request sent by the Controller is not a valid request. This can occur for multiple reasons:

- The four letter code given is not a valid command.
- The length of the request that was send in the header is not a valid number.
- The length in the header is not the same as the length of the request received.
- A problem occurred while executing the request.

- The connection drops while processing the request, although in this case no response can be returned.

3.2 Class/Namespace Documentation

entrypoint
The `entrypoint` function in the `entrypoint` namespace is the first function to be executed. It is the entrypoint for our program. This function expects the command line input argc and argv. Other then that it expects the apiIP and apiPort. These will both be -1 by default. The apiIP and apiPort will both be read out from the .env file if both of them are set to -1 (This is not done by this function, but they are set by this function and passed on from there).
This function will parse the command line input by sending it to the `input` class. After that the requested command will be executed by with help of the `CommandFactory` class.
The `entrypoint` function will also setup the loguru library for logging.

input
`Input` is responsible for parsing the command line input. This class is used by creating an instance of this class. The constructor takes the command line input as arguments. The constructor will then parse this input with help of its local methods. The result of this will be stored in the member variables of this class. command will contain the command the user typed, executablePath contains the path to the executable and flags will contain the value assigned to each flag.

Here is a list of private functions used by the constructor to parse the user input:

- `parseCliInput`: Parses the command line input. Another line will be read if no command has been entered. That new line will be used as command line input in this case. This function will parse the executable path itself. It will then call `parseOptionals`.
- `parseOptionals`: Parses the command, mandatory arguments and flags of the input using regex. The result of this function will be stored in the flags member variable.
- `applyDefaults`: Reads the config file and sets the flags defined in that file as the defaults.
- `sanitizeArgmuments`: Is called for every flag that is found by either in the config file or the user input. Will call the corresponding `sanitize<Flag>Flag`. The sanitize functions will actually store the value found in the flag object.
- `validateInteger` and `requireNArguments`: Helper functions to make sure the input is in a correct format.

Flags
`Flags` is a class that is responsible for storing and passing on the value of each flag that is found by the input function. Flags has a member variable for each flag that this program uses. Flags has a few functions to help parsing the flags:

- `mapShortFlagToLong`: Maps the shorthand names of the flags to their longer versions. This is used to make it easier to parse.
- `isFlag`: Checks if a given flag is a valid flag.
- `isShortHandFlag`: Checks if a given flag is a shorthand version of a flag.
- `isLongFlag`: Checks if a given flag is a long version of flag.
- `parseConfig`: Reads the config file and returns flags found in the config file.

CommandFactory
`CommandFactories` responsiblity is to create the right command object for the given command. This is done by the `getCommand` function. This return a command object using a map that goes from a string to a command object.
The `CommandFactory` also has a function that print the help message.

Command
`Command` is a abstract super class from which each command inherits. This class has two functions: `execute` and `helpMessage`. `helpMessage` is not virtual and will always return the `helpMessageText` property which is set by each command. `execute` is an abstract function that will execute the command. Each individual will have to implement this logic themselves.

Start
Implements the `start` command as explained in the General structure of the controller. The execute function mainly loops forever until stop has been typed in the commandline. It will do a database request to get the next job and subsequently call one of its helper methods to handle the request:

- `handleCrawlRequest`: Will call the crawler to crawl urls.
- `versionProcessing`: Is called for a spider request. Will parse all tags of a given url by first downloading the repository and its metadata. If the repositry has tags, it will be passed on to the `loopThroughTags` method which will call `downloadTagged` for each tag in the repository.

Check, Upload and Checkupload
These classes all inherit from the Command super class and implent their respective command to perform the functionality as described in the previous section General structure. For this they will call the submodules through the `moduleFacades` class. All communication to the database api is done through the `DatabaseRequests` class.

`Check` and `Checkupload` will also call the `printHashMatches` from the `PrintHashMatches` class to handle the printing of the summary of the matches found.

moduleFacades
`ModuleFacades` is responsible for calling the submodules. This class exists to make it easier when submodules change, because they are only called in one spot. There is a function in `moduleFacades` for each entrypoint of each submodule.

- `downloadRepository`: Calls the the entry point of the spider to download a repository and get the author that worked on it.
- `getRepositorytags`: Calls the spider to get all tags that are present in the downloaded repository.
- `parseRepository`: Calls the Parser to parse the downloaded repository that is located at the given filepath.
- `getProjectMetadata`: Calls the crawler to get the metadata of the given url.
- `crawlRepositories`: Calls the crawler to crawl more jobs from a given startpoint.

DatabaseRequests
Database requests is responsible creating all requests to the database. The networking for this is done by the `Networking` class. The actual body of the request will for most request be generated by the `NetworkingUtils` class. `DatabaseRequests` has a function for each database request that can be sent to the Database API.

All these individual function will internally call the `execRequest` function which will actually handle the request and send it to the Database API. The database response will be send to the `checkResponseCode` function to check if the database returned an error or not. These request are done in the format that is explained in the General structure section of the controller.

NetworkHandler
The `NetworkHandler` is responsible for handeling the network communication with the database api. The `NetworkHandler` also reads out the .env file to get the ips that it needs to connect to.
A `NetworkHandler` is created by calling the `createHandler` method. Before you are able to send request with this handler however, you have to call the connect method first. The connect method asks for the server ip and port that you want to connect to. If you give for both of these -1, a random ip from the .env file will be chosen. If the system can not connect to the randomly chosen ip, it will move on to the next until the list is empty. If the system can not connect to any of these ip's or just the one you gave it, it will throw an error.
Once a connection has been established, data can be send and received. Data can be send using the `sendData` function. Data can be received with the `receiveData` function. This function will keep reading the data from the connection until the connection gets closed.

termination
The `termination` namespace is the only place that the program exits. When a class wants the program to exit, it will call a function in `termination` instead of exiting itself.

print
The `print` namespace handles all calls to the console. All logging calls will be done through this methods in this namespace.

- `loguruSetSilent` and `loguruResetThreadName`: External calls to the loguru library.
- `debug/log/warn`: All log a message to the console and to the log files but at different log levels.
- `printLine`: Uses `std::cout` to write to the console and adds a new line character at the end.
- `writelineToFile`: Writes a string to a file and adds a new line character at the end.
- `printAndWritelineToFile`: Uses `std::cout` to write to the console and adds a new line character at the end. Will also write this to a file.
- `versionFull`: Prints the version of this program and its subcomponents to the console. This function is used when the user types `-v` or `--version`.

error
The `error` namespace contains a function for every error that can be thrown by the system. Each of these individual error functions will call an internal `err` function. This `err` function calls the `terminate` function from the `terminate` class. The error will be logged to the console and the logs.

regex
The `regex` namespace handles all regex functions the system uses. The namespace contains one function for every regex expression the system uses.

Utils
The `Utils` class contains generic functions that are can be used by every class.

- `Contains`: Checks if a list contains a certain element.
- `split`: splits a string on a given character.
- `trim`: Gets rid of the given characters at the end and beginning of a given string.
- `trimWhiteSpaces`: calls the trim method for whitespace characters.
- `isNumber`: Checks if a given string is a number.
- `padLeft`: Adds a given character to a string until it is the given length.
- `getIntegerTimeFromString`: Converts a yyyy:mm::dd hh:mm:ss format to a long long which represents the time in milliseconds from 1970.
- `replace`: Replaces each occurrence of a certain character with a different character.
- `getExecutablePath`: Gets the absolute path to the executable.
- `shuffle`: randomly shuffles a list.
- `getIdFromPMD`: Generates an id for a given project.

4 Crawler

The Crawler is responsible for retrieving the GitHub projects. The Crawler gathers projects by their project ID in ascending order. The Crawler is also responsible for retrieving project metadata from GitHub projects. In practice, all communication with the GitHub API is done through the Crawler.
All data retrieved by the Crawler is only returned in code and not displayed in the console. The only things that will be printed to the console are error/warning/info messages and indications of how far and how much the crawler has crawled.

Crawling GitHub
Whenever the Crawler crawls, it crawls one page of the GitHub list. This one page contains 100 URLs. We communicate with GitHub via libcurl [8]. However, as a lot of URLs have become invalid over time (e.g. the project has been deleted or set to private), we do not always return 100 URLs to projects. We also do not crawl the URL of a project if the project contains zero bytes that we can parse in our parser. We check the amount of parseable bytes (bytes which are in files with extensions of languages that we can parse) a project has by doing an additional GitHub query asking for the project's languages list.

The function responsible for crawling GitHub is `crawlRepositories(url, start, username, token)`. The start parameter in the `crawlRepositories()` repositories function indicates from which project ID the Crawler will start crawling. It does this by going to the URL `https://api.github.com/repositories?since=start` where `start` is the initial project ID. For example, to start from project ID 9001, you would write `https://api.github.com/repositories?since=9001`. If you want to find the ID of a project given its URL, you can do so by replacing `github.com` with `api.github.com/repos/`. For example, `https://github.com/torvalds/linux` would become `https://api.github.com/repos/torvalds/linux`. The "`id`" entry indicates the project ID of this project (which is 2325298 in this case).

A standalone project of the crawler is delivered with the final product, and the `main` file in the standalone project calls the `crawlRepositories()` function, defining the username and token to use with the GitHub API. To execute the standalone with a working token you have to set the token variable to a GitHub API token.

GitHub API tokens
For crawling GitHub we use the standard GitHub API (the REST version) with GitHub API tokens. The default amount of GitHub API calls we can make per hour is 60, but by using a GitHub API token we can extend this number to 5000. If we were to crawl non stop we could crawl 24 pages of the GitHub list per hour.
When the Crawler is done crawling a page it will return a list of projects with their priority measures and it will also return a number indicating the last project ID we have found. As there are quite a lot of gaps between the project IDs, the number from which we have to start crawling next time needs to be returned by the worker node that just crawled to prevent duplicate projects in the job queue.

A GitHub token can be generated by going to `https://github.com/`, signing in, then clicking on your profile picture in the top right corner and then selecting "Settings". Then click on "Developer settings" on the left hand side and select "Personal access tokens" and finally "Generate new token". You don't have to tick any boxes, just clicking "Generate token" at the bottom of the page will do fine. Make sure to save the token you receive after this as it will not be shown to you again by GitHub. See this for additional information.

Priority measure
On every project we crawl we put a so called "priority measure". This is an integer which we return together with every URL we retrieve which indicates the priority it should get in the job queue: the lower its priority measure, the sooner it should be parsed. In order to calculate this priority measure we need the number of stars, the date at which we crawled this project, the amount of parseable bytes in the project and the percentage of the total amount of parseable bytes to the total amount of bytes of code in the project. This requires two GitHub API calls per project: one to retrieve the number of stars and one to retrieve the information about the languages. This means that per page of GitHub information we need 201 GitHub calls (we need one call for the page containing the projects). The formula with which we calculate the priority is the following.

$$T - 20.000.000 \cdot P \cdot \lg(S+1) \cdot \lg(\lg(B+1)+1).$$

Here, T represents the time of uploading the project, P indicates the fraction of bytes that are contained in files written in languages which our parser can parse, S is the number of stars the project has and B is the total number of parseable bytes. The crawler calculates $20.000.000 \cdot P \cdot \lg(S+1) \cdot \lg(\lg(B+1))$ and sets this as the priority. When the database API receives a new job, created by the crawler, it will subtract this priority from the current timestamp to arrive at the $T - 20.000.000 \cdot P \cdot \lg(S+1) \cdot \lg(\lg(B+1))$ priority. This is done server-side to make sure all timestamps are retrieved on synchronised machines. The lower the priority score of a project, the sooner it will be retrieved from the job queue.

We need to clone an entire repository before we can parse the parseable files, so a repository with a small fraction of parseable lines will take relatively longer to process per line, than a repository for which we can parse almost every line. Therefore, the priority measure we chose depends on P in a linear way. Since the number of stars and number of parseable bytes is unbounded (unlike P, which is bounded between 0 and 1), we scale these numbers by taking a base 2 logarithm. This is also because these are more a measure of popularity than parsing speed, unlike P. Since the number of stars is more of an indication of importance than the number of parseable lines, we take a nested double logarithm of B. We add 1 to prevent the logarithm to dip below 0. Finally, the priority depends on the time of uploading in a linear way, since we want unpopular projects which are in the queue for a while to eventually get moved to the front and processed. The constant of 20.000.000 was chosen so that a small, somewhat popular repository gets processed before Linux if Linux was uploaded a little over a week later than the smaller repository.

Timeout measure
On every job there is also a timeout being calculated. This is the amount of time in milliseconds that the database api and controller will allow for this job to take. It is calculated as follows:

$$\min\left\{180000 + 5000 \cdot \sqrt{B}, 1800000 + 800000 \cdot \sqrt{S}\right\}$$

Here B is the total amount of parseable bytes and S the total number of stars.

Retrieving metadata
Besides the crawling of GitHub the Crawler is also responsible for retrieving metadata of a project. When given an URL of a project, the Crawler will retrieve the following information about the project:

- The current version of the project (the last time a commit was pushed to it);
- The license of the project (if available);
- The name of the project;
- The name of the author of the project;
- The e-mail address of the author of the project (if available);

- The default branch of the project.

It will return this information in a "ProjectMetadata" struct which contains the information specified above. In case of an error (e.g. the url was no longer available) an empty struct is returned.

The function responsible for crawling GitHub projects can be called in the `main` class by calling `RunCrawler::findMetadata(url, username, token)` where `url` is an URL to a GitHub project. We could for example again take `https://github.com/torvalds/linux` as input, and we would receive project metadata for this specific project.

Do note that in both cases of crawling and retrieving project metadata that we can only do this with GitHub. The structure for extending this does exist in the RunCrawler class, so it should not be too difficult to combine new code supporting other codebases in the future with the existing code.

JSON format and the JSON adapter
In almost every case where we use libcurl to communicate with the GitHub API, we receive data in JSON format. At the moment we use a JSON adapter which uses Nlohmann's JSON library, as we want to be able to easily switch our JSON library. This exists in case you would want a faster JSON library or a more memory efficient JSON library in the future.

The JSON adapter uses a lot of templates: as the Nlohmann's JSON library allows either strings or ints to be used to index on given keys, we have decided that we want to keep this same functionality by specifying what type they are through a template. Since a JSON variable can contain values of several different types (among which are string, integer, and boolean) there is also an "Output" template argument in several functions, which is the type we expect to receive as output. The most important functions in the JSON adapter are the `get()` and `branch()` functions, which allow one to retrieve a variable from a JSON structure and branch on a given index respectively. Besides this there are several other functions such as `isNull()`, `isEmpty()`, and `contains()` that allow the user to check if a variable exists in the current branch of the JSON variable, and there are also some helper functions such as `repeatedGet()` and `exists()` that prevent code duplication by combining the functions that already exist in the JSON adapter.

Error handling
In most parts of the component where error handling is needed we have some sort of error handling. In the case of our class that communicates with GitHub through the use of libcurl we check the code from GitHub that we received and check if that is an OK code or otherwise an error. In the case of the latter we return either a warning or an error to the user, and if it is a fatal error the Controller is informed. It then makes the decision whether to stop the program or not. Similarly in the JSON adapter we check if there was an error while trying to execute a function (for example, parsing can easily throw an error) and handle accordingly if there indeed was an error. This is most of the times just a fatal error, unless we expected it in which case we throw a warning instead.

This error handling works through the use of so called `ErrorHandlers`, which are classes that contain dictionaries that for each type of error contain an `IndividualErrorHandler` which has a single function called `execute()` that handles a given response. We have decided not to make these functions static as we do not want that each class can return an error of every type. For example, the class that handles communication with GitHub should not be able to give JSON errors, as those errors are reserved for the JSON class. However, this approach is sadly not completely without downsides: in the case of the JSON adapter we do not want to create a new JSON error handler for each JSON adapter, and we had to resort to a singleton JSON adapter.

Viewing data
As the crawler does not send its data to the console, the data can be viewed by either debugging the program or by adding code that prints the retrieved data itself out. Both the `crawlRepositories()` and the `getProjectMetadata()` return the data they've found directly and you can simply print the results you receive from these functions.

5 Spider

The Spider downloads repositories from a codebase (currently only GitLab and GitHub) and then gathers `AuthorData` from the downloaded repository. This `AuthorData` is then sent to the controller so other components can use this data. The downloaded repository stays locally on disk so the parser can parse the files of that repository.

5.1 RunSpider

Starting the process of downloading and gathering `AuthorData` is done in several steps. First, `RunSpider::setupSpider` sets up a Spider specific to a given codebase, with the specified number of threads. To start downloading, this same Spider object is needed.

The function `RunSpider::downloadRepo` clones a repository to the specified location. Further, the function `RunSpider::getAuthors` creates `.meta` files containing all the authors of the files currently in the downloads folder. To switch from one tag to another, `RunSpider::updateVersion` is used. This method returns a list of unchanged files and deletes these from the downloads folder (to prevent them from being parsed). Currently the program checks if the URL is either a GitHub or a GitLab URL and if that is the case, a `GitSpider` is created. The GitSpider class is a subclass of the Spider class. If you want to add a different type of Spider for a different type of website, you can add a new regex check to the `RunSpider::getSpider` method which returns a specific Spider subclass in charge of handling that website.

5.2 Spider class

The spider class is responsible for downloading projects from a codebase. It is an abstract class which has a few methods already predefined. Having a Spider superclass this way allows the `RunSpider` to work with arbitrary Spider implementations without having to worry about how the data is retrieved.

5.3 Git class

In the case of the GitSpider class, downloading the source is done with help of a Git class. To download a git repository a `git clone` command is called. The `git clone` command is set up to perform a sparse checkout which allows us to only download the files with certain extension types. This way we ignore all files that cannot be parsed by the parser.

5.4 Tags

The Git class can also get a specific tag from a git project. A tag is a commit that the developer tagged as important. These tags can be retrieved by calling `runSpider:getTags` after a repository has been downloaded. The Git download function is given a 'tag' argument and a 'nextTag' argument. The 'nextTag' argument indicates which tag to download. The 'tag' argument is used for comparison. The Spider expects the oldest tag to be downloaded first. Afterwards you can always give the next newest tag as 'nextTag' and the old tag as 'tag'. Then the spider checks which files changed between the tags and throws away the unchanged files as they have already been parsed previously.

5.5 Author data

The `AuthorData` data structure consists of a vector of `CodeBlock` data structures for every file in the repository. A `CodeBlock` consists of a starting line variable, a # of lines variable and a `CommitData` data structure. All lines from the starting line till the starting line plus the # of lines belong to that `CommitData`. The `CommitData` data structure contains data such as who the author is, what the email address of the author is and some other miscellaneous things. This data is useful to check which author wrote which parts of a file.

5.6 Branches

`RunSpider::runSpider` also allows has an argument to define which branch should be downloaded of the repository. By default, only the branch that is defined as the "Default branch" in Git in downloaded, but by using this argument it's possible to download different branches if that is desirable.

5.7 Collecting author data

In the GitSpider class, author data is collected using `git blame`. Every file in the downloading repository gets a `git blame` command called on it and the output is written to a .meta file. After every file has been blamed, the blame data of all the .meta files gets parsed by the Git class and then stored in an `AuthorData` data structure.

5.8 Error handling

There are a few errors that can happen during the spidering process, for example being unable to download any files because there is no internet connection or there being some strange data among the blame data. In some cases only a warning is given, for example if the spider is unable to parse certain blame data, then that blame data is skipped and the spider continues as usual. This approach is sometimes not possible, for example if the spider fails to download anything. In this case it throws an error and stops running, the controller has to decide how to proceed afterwards.

6 Parser

The Parser is responsible for taking a project (a folder containing a collection of code files) and extracting the methods from it. It does this by first identifying the methods inside a source file, then abstracting its content, before hashing this abstracted value. This hash is then combined with extra information concerning the method, for example the name and line number and stored in a large list. Once all files are analysed the Parser returns this list of found methods for the project. The Parser can currently parse C, C++, C#, and Java code using srcML [6], while parsing Python and Javascript code using a custom parser built using ANTLR [7].

The class structure of the Parser can be seen in figure Figure 6 at the end of the documentation. The entry point for the stand-alone application is the `main` method in Main. Furthermore, the Logger class and HashData struct are used a lot throughout the entire Parser, but for clarity the arrows are not included (as they are used in most of the other classes).

The Parser abstracts in such a way that it can recognise type 2 code clones [1, 11, 12]: meaning it removes comments and whitespaces, while abstracting variable names (including names of function calls).

To prevent the matching of trivial clones, any function which contain less than 50 characters (after abstraction) or six lines is excluded.

For hashing the Parser use the MD5 algorithm. MD5 is a very fast hash algorithm, so using MD5 as a hash algorithm allows the parser to waste little time on calculating hashes. For the implementation the ”RSA Data Security, Inc. MD5 Message-Digest Algorithm” was used..

6.1 SrcML

For C, C++, C#, and Java files the parser uses srcML to convert source code to an XML format. This allows for easy identification of methods, as well as variables and function calls. The parser currently calls srcML using a (hidden) command line interface, since srcML can also be used as a library, it should be possible to call it directly. There have been no problems with the current approach, but there might be advantages to calling srcML directly. This calling is done by the SrcMLCaller class which starts srcML in another thread, it also creates a StringStream object which will be connected to the output of srcML and sent to the rest of the program so the output can be handled while srcML is still processing input.
SrcML is build upon ANTLR but does not use a strict parsing method, generally going through source code once assuming the format is valid. If srcML happens to encounter something that it can’t understand it will simply group it under a general tag until it recognises the structure again. This makes it very fast (particularly in comparison to our own Parsers which were also made using ANTLR).
In figure 2 an example of the parsing using srcML is given, as well as what format data is stored between steps. The output of srcML is first parsed by the XMLParser class (XMLParser.cpp/.h) which parses the output of srcML linearly, identifying the start and end of files and methods and sending found methods to the AbstractSyntaxToHashable class. The found methods are send with the same structure received from srcML (A tree like structure where nodes identify code parts, for example a code block node, containing a number of statements, containing an assignment, containing a variable name and a value which is assigned to it, see ”Recognised function” in figure 2). The AbstractSyntaxToHashable then turns this tree into a single string, applying abstraction where necessary, and collecting any extra data from the method (it’s name for instance), see ”Data extracted” in figure 2. This string is then hashed using the MD5 hashing algorithm, and saved together with any extra info (method name, file, starting/ending line).

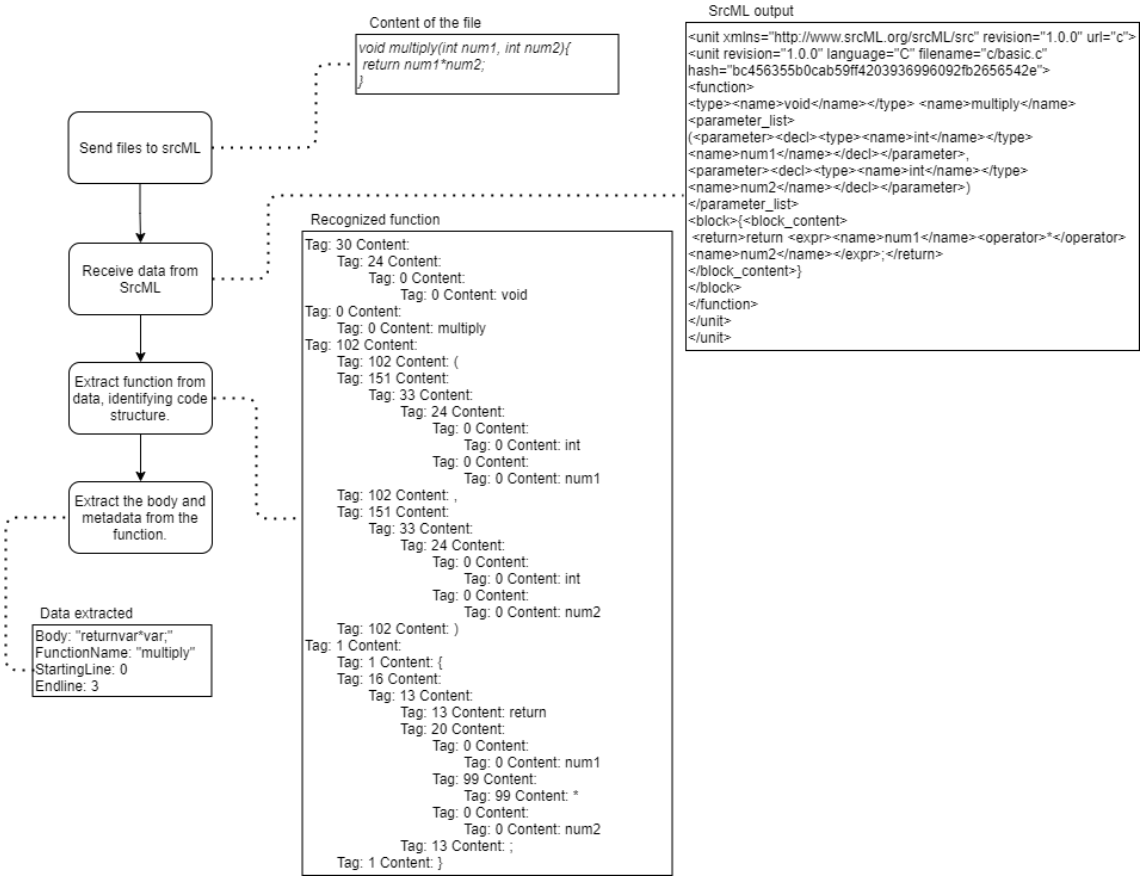

Figure 2: Flow of the Parser when using srcML.

6.2 Custom Parsers

The custom parsers for Python 3 and JavaScript make use of the ANTLR4 library. Most of the ANTLR-specific information in this document will be based on the documentation from the ANTLR website. Here, ANTLR is described as follows.

> *"ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees."*

An ANTLR grammar is stored with the `.g4` file extension. In essence, a grammar file is split up into a Lexer and a Parser grammar. Such a grammar consists of a declaration, followed by a list of rules. In the grammar declaration, the grammar name must be specified:

```
grammar <name>;
```

The name of the grammar must match the name of the `.g4` file: the grammar file `X.g4` must contain the declaration `grammar X;`. Such a grammar can contain both lexer and parser rules. Similarly, a grammar with declaration `lexer grammar <name>;` can contain only lexer rules. Changing the prefix to `parser` allows only for parser rules.

An example of a lexer rule is the following:

```
WHILE : 'while';
```

This simply implies that whenever we encounter the string `'while'` in the source, this should be tokenized as WHILE. Tokenizing can be seen as preprocessing the file to a list of tokens, from which a parse tree can be constructed based on the parser rules. A parser rule starts with a lowercase letter, for example:

```
file_input: (NEWLINE | stmt)* EOF;
```

Here, the NEWLINE is defined in a lexer rule, and the stmt is defined in another parser rule. The expression `(NEWLINE | stmt)` matches is either of these rules match and the expression `(NEWLINE | stmt)* EOF` matches if there is any number of instances matching `(NEWLINE | stmt)`, followed by an end-of-file. Like this parser rule, more complex lexer rules are also possible:

```
DECIMAL_INTEGER
 : NON_ZERO_DIGIT DIGIT*
 | '0'+
 ;
```

These rules are all examples of rules defined in the Python3 grammar from the ANTLR4 grammars on https://github.com/antlr/grammars-v4. We use this grammar, together with the JavaScript lexer and parser grammar from the same source, as a basis for our custom parser. We have made several small changes to these grammars to better suit our needs. These changes will be explained in detail later in this section.

To generate C++ code from the grammar file, we use the following bash script:

```
java -jar antlr-4.9.2-complete.jar -visitor -Dlanguage=Cpp
    -o generated/ Python3.g4;
```

We will now shortly look into how the parse tree is structured, to get an idea of how we can apply abstractions like those described in the previous section. We will look at a small Python3 code example to illustrate the process. This example is simply a method that doubles any input it is given:

```
def double(n):
    return n * 2
```

Using our altered version of the provided Python3 grammar, the parse tree for the code above has the following structure:

```
(file_input
  (stmt
   (compound_stmt
    (funcdef def
     (name double)
     (parameters (
      (typedargslist
       (tfpdef
        (name n))) )) :
     (funcbody
```

```
(suite \n indent
 (stmt
  (simple_stmt
   (small_stmt
    (flow_stmt
     (return_stmt return
      (testlist
       (test
        (or_test
         (and_test
          (not_test
           (comparison
            (expr
             (xor_expr
              (and_expr
               (shift_expr
                (arith_expr
                 (term
                  (factor
                   (power
                    (atom_expr
                     (atom
                      (name n))))) *
                  (factor
                   (power
                    (atom_expr
                     (atom 2))))))))))))))))) \n)) dedent))))) <EOF>)
```

This is a lot of data, considering the size of the input code. This is because the parser needs to specify the outer rule, before moving on to the smaller inner rules. Important things to note are that the method name `double` is contained in the first `name` tag in the bigger `funcdef` tag, and that `name` tags in the `funcbody` tag correspond to variables and function calls.

We can use these observations to construct an abstracted version of the method using a custom listener. This is a piece of code that can be included when walking through the parse tree, as follows:

```
antlr4::tree::ParseTreeWalker::DEFAULT.walk(&cl, t);
```

Here `cl` is the custom listener and `t` is the parse tree.

The construction of the parse tree consists of several steps. First, we tokenize the input:

```
antlr4::ANTLRInputStream input(data);
Python3Lexer l(&input);
antlr4::CommonTokenStream tokens(&l);
tokens.fill();
```

Here, `data` is a string containing the input file. Now, we construct a parser and extract the parse tree:

```
    Python3Parser p(&tokens);
antlr4::tree::ParseTree* t = p.file_input();
```

The method `file_input()` is generated by ANTLR, based on the parser rule for `file_input` as seen before. This is the encapsulating rule for the entire file, so we use this to extract the entire parse tree.

The custom listener can overwrite methods such as `enterFuncdef` and `exitFuncdef`, which can be found in the base listener generated by ANTLR. Like the name suggests, these methods are called upon entering and exiting a `funcdef` tag. We override several of these functions to extract and abstract method bodies. Since function definitions can be nested, we store all important data in stacks: starting line numbers, function names and function bodies. Furthermore, we also store a stack of so-called `TokenStreamRewriter` objects. These are used to dynamically store changes to the parse tree. The stored changes are only applied when we retrieve the abstracted method as a string, which is why we chose to use a `TokenStreamRewriter` per method, instead of a single `TokenStreamRewriter` for the entire file. This massively speeds up the program, since the changes stored in processed methods can be discarded after the method has been abstracted, by deleting the `TokenStreamRewriter` object.

The custom listener for Python 3 overrides the following methods:

- `enterFuncdef`: Called when a new method definition is entered. We prepare for computing the abstracted hash-data of this method by pushing a new `TokenStreamRewriter`, an empty function name and function body and the current line as a starting line to their respective stacks.
- `exitFuncdef`: Called when a method definition is exited. We pop all method data from the stacks and store the current line as the ending line. Check if an abstracted method longer than 6 lines and 50 characters and if so, hash the method body with MD5 and store all data in the `output` vector.

  If this method definition was given in another method definition, we abstract the inner method to the empty string. This is because a method defined in another method could just as well have been defined outside of the outer method.
- `enterFuncbody`: Called when a method body is entered. Store this by setting the `inFunction` boolean to `true`. This is used to only abstract terms inside the method body.
- `exitFuncbody`: Called when a method body is exited. Here, we extract the text of the entire `funcbody` tag from the top `TokenStreamRewriter`, which was used to abstract the current method body:

  ```
  functionBodies.top() = tsrs.top()->getText(ctx->getSourceInterval());
  ```
- `enterName`: Called when an unknown name is entered. A name is a string which is used in an expression, but is not a function call. In general, this consists mostly of variables. We thus abstract all names in a function body to "var":

  ```
  tsrs.top()->replace(ctx->start, "var");
  ```

  Like we saw above, the name of the method is the first name tag in a funcdef tag. We also check this in this method.
- `enterFunccallname`: This is similar to the `enterName` method, but the funccallname tag is solely for function calls. These are abstracted to the string "funccall":

  ```
  tsrs.top()->replace(ctx->start, "funccall");
  ```
- `enterExpr_stmt_single`: Called when an expression with only a single statement is entered. We store this by setting the `inSingleStatement` boolean to `true`.

- `exitExpr_stmt_single`: Called when an expression with only a single statement is exited. We store this by setting the `inSingleStatement` boolean to `false`.
- `enterString`: Called when a string is entered. This is a string in Python, so `'abc'` or `"abc"`, or even a multiline string (`'''abc'''` or `"""abc"""`). Generally, we do not abstract these strings, and simply keep them as-is. However, an expression solely consisting of a single string is ignored by Python. This behaviour is often used to create multiline comments. Since these have no impact on the functionality of the method, we want to ignore it. Therefore, we replace any string by the empty string if the `inSingleStatement` is set to `true`.

Using our Python3 parser, the `double` method is abstracted as follows:

```
indentreturnvar*2
dedent
```

This is then hashed with MD5 and returned together with the starting and ending line of the method (including header) and its name, which in this case is `double`.

The second custom parser we created is the JavaScript parser. This is based on the same ideas as the Python parser, and is implemented in a similar way. A change in how the JavaScript parser abstracts methods, is that function calls are also abstracted as ”var”. This is possible, since all function calls in JavaScript are immediately followed by an opening bracket and variables are never followed by an opening bracket. The fragment ”var(” thus uniquely identifies a function call. Like mentioned before, the JavaScript grammar is split up into a lexer and parser grammar, but the code can be generated in the same way, by simply generating the lexer code, followed by generating the parser code with the following bash script:

```
java -jar antlr-4.9.2-complete.jar -visitor -Dlanguage=Cpp
    -o generated/ JavaScriptLexer.g4;
java -jar antlr-4.9.2-complete.jar -visitor -Dlanguage=Cpp
    -o generated/ JavaScriptParser.g4;
```

The general structure of the JavaScript custom listener is the same, but we override different methods:

- `enterAnonymousFunctionDecl`: Called when an anonymous function declaration is entered. JavaScript has different types of functions, including anonymous functions. These are functions without a name. Like with the Python 3 parser, we push a new `TokenStreamRewriter`, an empty function name and function body and the current line as a starting line to their respective stacks. Furthermore, we set the `inNonAbsFuncDef` boolean to `false`, indicating that we are in an anonymous function definition. This is mainly used to indicate that we do not need to bother with finding a function name, since there is none.
- `enterFunctionDeclaration`: Called when a non-anonymous function declaration is entered. This mostly does the same as `enterAnonymousFunctionDecl`, but it sets the `inNonAbsFuncDef` boolean to `true`.
- `exitAnonymousFunctionDecl`: Called when an anonymous function declaration is exited. We do the same as with the Python 3 parser and pop all stacks, get the current line as ending line and store the MD5 hash of the method if it is longer than 6 lines and 50 characters. An anonymous function is mostly used as a variable in an outer method, so we abstract the entire method to ”var” after storing the abstracted method.
- `exitFunctionDeclaration`: Called when a non-anonymous function declaration is exited. This mostly does the same as `exitAnonymousFunctionDecl`, but it abstracts the entire method to the empty string after storing the abstracted method. By the same reasoning as in the Python 3 parser, a non-abstract method could also have been declared outside of the current method.

- `enterParseFunctionBody`: Called when the body of a method declaration is entered. This implies that we exited the method header, so we set the `inNonAbsFuncDef` boolean to `false`.
- `exitParseFunctionBody`: Called when the body of a method declaration is exited. Here, we store the abstracted function body in exactly the same way as in the Python 3 parser:

```
functionBodies.top() = tsrs.top()->getText(ctx->getSourceInterval());
```

- `enterIdentifier`: Called when an identifier is entered. This can be both a variable or function call, but in both cases, we abstract this to "var". The single exception to this is the extraction of the method name. Similar to the Python 3 parser, this is the first `identifier` tag in the `functionDeclaration` block. Therefore, if `inNonAbsFuncDef` is `true` and the function name is not yet known, we store the current `identifier`:

```
if (inNonAbsFuncDef && functionNames.top() == "")
        {
                functionNames.top() = ctx->start->getText();
        }
```

To improve the custom parsers, we first studied the parse tree of a given piece of source code and looked for potentially interesting patterns, such as the strings in single expressions as a comment in Python 3. We then changed the grammar to create a specific rule (tag) for this case and regenerated the generated C++ code. The custom listener was then changed to include this new tag, and handle it as we desired. A potential improvement would definitely be to write a grammar from sratch, with fewer rules. This is probably possible, since we do not use all of the intricacies of a language and apply quite a basic abstraction. This has the potential to greatly speed up the parser, depending on how minimal the resulting grammar is.

The general structure of the custom parsers can be seen in the UML diagram in Appendix A. In essence, both custom parsers are subclasses of the `LanguageBase` class, as defined in LanguageBase.h. The main method this class describes is the `parseData` method. This parses an entire source file to a vector of HashData structs, containing all information required of a hashed method. It also contains the `ClearCache` method, required for garbage collection purposes. These two methods are implemented in the `Python3AntlrImplementation` and `JavaScriptAntlrImplementation` subclasses (as the names suggest, one class for the Python 3 parser and another for the JavaScript parser). These classes handle the construction and traversing of parse trees, a procedure described above. While traversing the parse tree, they make use of the custom listeners defined in the `CustomPython3Listener` and `CustomJavaScriptListener` classes. These are subclasses of the ANTLR-generated `Python3BaseListener` and `JavaScriptParserBaseListener` classes and contain the method overrides as listed above.

7 Database

7.1 Database structure

For our database we use Cassandra [5]. Cassandra makes use of a structure where the top level of your data are keyspaces. This means all data tables are distributed throughout keyspaces. You can specify the distribution per keyspace. A keyspace then contains different tables.

Each table has a partition key which is used for the distribution, since a single partition is always on a single server. In the tables below, we will refer to a partition key by 'K'. The rest of the primary key is defined with clustering keys, which have a sorting order. We will refer to a clustering key by 'C'. The sorting order is indicated by means of an arrow. More specifically, an arrow pointing up, ↑, is used to indicate a column in a table that is sorted in ascending order (i.e., from lowest (in the topmost row) to highest (in the bottom-most row)). Similarly, an arrow pointing down, ↓, next to a column is used to indicate that the table is sorted on the column in descending order.

The structure of our database consists of two keyspaces. One keyspace for the data about the jobs and one with all the data about the methods and projects in the database.

7.1.1 Jobs

The keyspace with the jobdata, called `jobs`, contains the jobs queue table:

jobsqueue		
Constant	Int	K
Priority	Bigint	C↑
Jobid	UUID	C↓
Url	Text	
Retries	Int	
Timeout	Bigint	

As we can see, the jobs queue is sorted with ascending order on the `Priority` and descending on the `Jobid`. The `Constant` is included to make sure that the whole `Jobqueue` is part of a single partition and can therefore be sorted on the priority. The priority is a 64-bit integer which is calculated as the current time minus the priority score we get from the crawler. The priority is sorted from lowest to highest so the job with the lowest priority will be done first. The job id is a Universally Unique Identifier (or UUID) so it becomes easier to keep track of the different jobs. The `Url` is the url that should be parsed when this job is done. The `Retries` is the amount of times this job has bee retried. Finally the `Timeout` is the timeout calculated by the crawler.

Next there is also a table containing the jobs that are currently processing:

currentjobs		
Jobid	UUID	K
Time	Timestamp	
Timeout	Bigint	
Priority	Bigint	
Url	Text	
Retries	Int	

The contents is very similar to the jobsqueue table. The main difference is the `Time`, this is the time at which this job has been given out. This is used for both identifying this job and seeing when this job has timed out.

Another table in this keyspace is the table containing the failed jobs:

failedjobs		
Jobid	UUID	K
Time	Timestamp	C↓
Timeout	Bigint	
Priority	Bigint	
Url	Text	
Retries	Int	
ReasonID	Int	
ReasonData	Text	

This is again similar to the currentjobs table. The differences her are the `ReasonID` and `ReasonData`. These two columns represent the reason this job failed, where the id is an identifier for the reason and the data is some extra explanation about this specific failure.

The final table in this keyspace is the table containing the current crawlid:

variables		
Name	Text	K
Value	Int	

Here the `Name` is the name of the variable and `Value` is the value.

7.1.2 Projectdata

The other keyspace, called `projectdata`, contains the data for the methods and projects that we store. The first table is the table containing the methods:

methods		
Method_hash	UUID	K
ProjectID	Bigint	C↑
StartVersionTime	Timestamp	C↓
File	Text	C↓
StartVersionHash	Text	
EndVersionTime	TimeStamp	
EndVersionHash	Text	
Name	Text	
LineNumber	Int	
Authors	{UUID}	
Parserversion	Bigint	
VulnCode	Text	

Here the `Method_hash` is the hash of the method obtained by the parser. The `ProjectID` is the identifier of the repository this method was found in. The `StartVersionTime` is the timestamp of commit that is the first version this method was found in. The `File` is the name of the file this method was found in including the path of the file. The `StartVersionHash` is the hash of the commit from the first version of this method. The `EndVersionTime` and `EndVersionHash` are respectively the timestamp and the hash of the commit for the latest version this method was found in. The `Name` is the name of the method and the `LineNumber` is the number of the line this method starts in the file. The `Authors` is a set of UUIDs for the authors that have worked on this method. Finally the `Parserversion` is the version of the parser used to parse this method.

The next table is for the data about the projects:

projects		
ProjectID	Bigint	K
VersionTime	Timestamp	C↓
VersionHash	Text	
License	Text	
Name	Text	
URL	Text	
OwnerID	UUID	
Hashes	{UUID}	
Parserversion	bigint	

In this table the `ProjectID` is the identifier for the repository. The `VersionTime` and `VersionHash` are respectively the timestamp and hash of the commit for this version of the project. The `License` is the license of the project retrieved from git. The `Name` is the name of the project. The `URL` is the url where the project can be found. The `OwnerID` is the UUID for the owner of the project. The `Hashes` is a set of hashes of the methods in this project. Finally the `Parserversion` is the version of the parser used to parse this project.

Since Cassandra is a non-relational database, a relation between a method and its authors is not directly available. A new table, `method_by_author`, has to be introduced to relate the authors to methods. More precisely, the following table is used to find the methods created by a certain author:

method_by_author		
AuthorID	UUID	K
Hash	Text	C↑
ProjectID	Bigint	C↑
StartVersionTime	Timestamp	C↓
File	Text	C↑

Here the `AuthorID` is the ID of the author. The `Hash` is the hash of the method. The `ProjectID`, `StartVersionTime` and `File` correspond to the other parts of the primary key for the methods table.

Finally we have the table containing the authors:

author_by_id		
AuthorID	UUID	K
Name	Text	
Mail	Text	

In this table the `AuthorID` is the id of the author which is determined by hashing the name and mail. The `Name` and `Mail` are respectively the name and mail of the author.

7.2 Distributed database

Distributing with Apache Cassandra
For the distribution of the database we used an existing database system: Apache Cassandra [5]. Apache Cassandra automatically distributes the data in the database over the database servers connected to the network, we will refer to these servers as nodes from here on out. Cassandra has an ingenious internal system to correctly distribute the data over the connected nodes. We can specify how many copies of each piece of data should be stored in the database. In Cassandra we specified each server as a datacenter, this means that we can even specify how many copies of each piece of data should be stored on each server. We currently have three servers on which data is

stored, when we hand over the project the system will keep running on two of those servers. In our current settings we have set the number of copies of each piece of data that should be stored in the database to two, this can be changed at any time. Also, Cassandra automatically stores these copies on different nodes, so every piece of data is stored on two different database nodes. When we upload data to the database, Cassandra will automatically store it on a node and store duplicates of all the data on a different node.

7.2.1 Node failure

When Cassandra notices that a database node has gone down, it will automatically try to redistribute the data over the rest of the nodes. We also have a dashboard which shows if a database is up or not. When a node goes down, the maintainer is able to see this in this dashboard. The maintainer can then restart the node and when Cassandra notices that the node has reconnected it will once again redistribute the existing data.

7.2.2 Data redundancy

As mentioned before, with the current Cassandra settings every piece of data is stored in the database on two different nodes. This means that one node can go down without any data loss. After the node goes down, Cassandra will redistribute the data, so now there are once again two copies of each piece of data. As a result, it is again made possible for a node to go down without any data loss.

It should be mentioned, however, that when all nodes are down, all data will be lost. Also, when two nodes go down in such quick succession that Cassandra does not have time to redistribute the data, we may also suffer data loss. This latter risk can be reduced by storing more than two copies of each piece of data, which is not hard to set up.

7.3 Maintainer dashboard

We also have a website with information about the database displayed in graphs. Information we display here is real-time and is about disk usage per server, number of partitions per table in the database and disk usage per table. This website is designed for maintainers and can only be accessed with a username and password. These should only be known by maintainers of the system or people trusted by the maintainers who need this information for research purposes, for example. A screenshot can be seen in Figure 3.

7.3.1 How it works

The dashboard first uses cassandra-exporter to show the statistics to other programs. Next we use Prometheus to collect the data. Finally we use Grafana to nicely visualise the statistics and host the website.

8 Database API

The Database API is responsible for multiple tasks. It handles the requests from the Controller to add data to the database and retrieve information about this data from the database. It is also responsible for the job distribution, for which it will give jobs to the worker nodes that ask for it. The class structure can be seen in Figure 8.

The `Database-API` class is the entrypoint for the program. The `ConnectionHandler` is responsible for handling the requests from the Controller, other servers or other clients. It will listen for requests on a specified port and retrieve the given data. It then passes this data to the `RequestHandler`, which parses the request to know what has to be done. If it is a request directed to the data in the database it is passed to the `DatabaseRequestHandler`. If the request is for the job distribution the

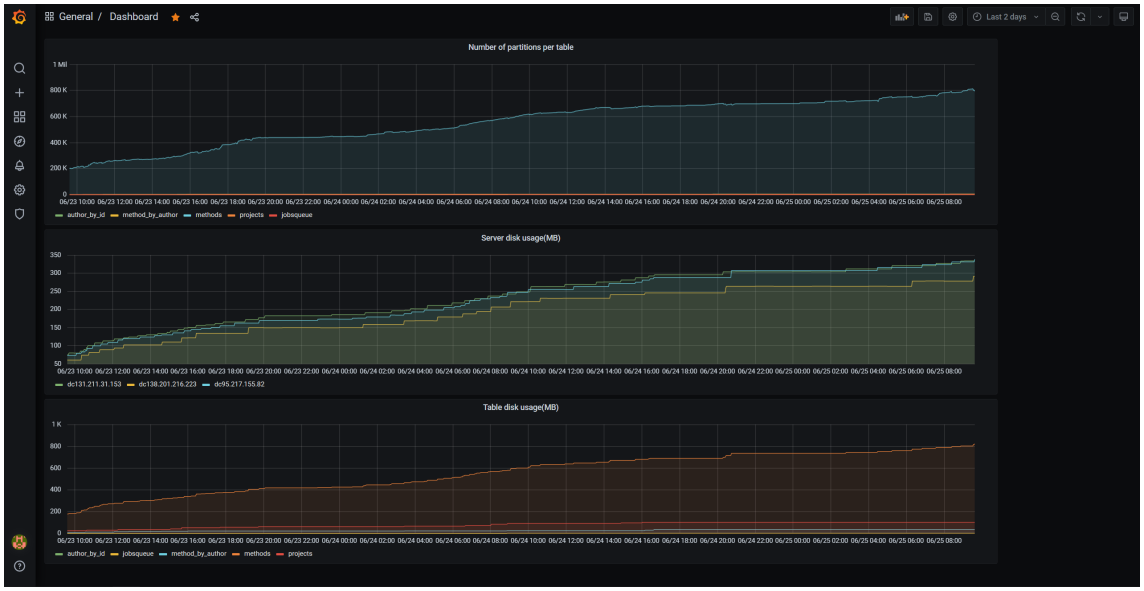

Figure 3: A screenshot of the maintainer dashboard

request is passed to the `JobRequestHandler`. The `DatabaseConnection` and `DatabaseHandler` are responsible for the connection with the database and the `RAFTConsensus` is responsible for the implementation of the RAFT system, which will be explained in more detail later.

8.1 Database Handling

For requests that have something to do with the data corresponding to the methods in the database there is a method called in the `DatabaseRequestHandler` from the general `RequestHandler`. In most cases this method will first parse the input by splitting the input and if necessary parsing it to the correct objects. After this multiple threads are started and a queue is created. Each thread will take an element from the queue and pass this to method in the `DatabaseHandler` which will actually perform the database queries. After all queries are completed the threads are stopped and the result is parsed to the resulting string which is then returned.

The possible request are:

- Upload project: Will upload all data for a project to the database. This includes the metadata of the project and all methods in the project. Optionally, a project can be updated based on a previous version of a project (which should already be in the database, of course). This is done by providing the previous version of the project together with a list of unchanged files. The methods in these unchanged files are then updated accordingly by updating their end version, and other, non-key, attributes.
- Check methods: Will take in a list of hashes and match these against the hashes in the database. It will then return all matches found in the database.
- CheckUpload project: Will perform both previous request by both adding the project to the database and match the methods in this project to the methods in the database.
- Extract project: Returns the metadata corresponding to a project.
- Get author: Retrieves the author corresponding to the passed ID.
- Get method by author: Returns the methods know to be worked on by the given author.

8.2 Job distribution

The Job queue itself is part of the Database, and is hence also distributed. However, for the Job queue we wanted to go a step further and make sure we are not handing out the same job multiple times. To accomplish this, we implemented a RAFT Consensus system.

In a RAFT system[1], there is one leader node. The rest of the nodes are all non leader nodes. Once in a while, the leader node sends a heartbeat to all other nodes in the network. This heartbeat contains any information that needs to be synced between the nodes. The other benefit of this heartbeat is that we know for sure the leader is still running if we get one. Non-leader nodes do only one thing, and that is passing request on to the leader node. The leader node is the only node that actually executes the requests.

In this system, there is no problem when a non-leader node drops out. However, if the leader node drops out it becomes a bit harder. In our system, we made sure to sync the list of non leader nodes between all nodes in the network using the heartbeat. We make sure that the list is in the same order on all nodes. When the non leader nodes detect that the leader has dropped out, the new leader will be chosen. The new leader will always be the next node in the list of non leader nodes.

The Job request handler is the class that will actually handle the incoming requests for the Job queue. These requests include:

- Get top job: Returns the next job in the job queue. If the job queue is under a specified threshold (which can be edited in `JobRequestHandler.h`), it will return a crawl job to refill the job queue. It will not return a crawl job if there is currently another worker node crawling. A crawl job will also be returned if the previous crawl job has not returned jet and has taken longer than a specified timeout time (which also can be edited in `JobRequestHandler.h`).

  We manually keep track of how many jobs are in the job queue. This way we do not have to do a database query each time to get the number of jobs in the queue. When the system starts, we do this query one time to know the initial number of jobs in the database. From there we subtract one each time we take on out, and add one each time we put one in.

- Add Jobs: Adds one or more jobs to the job queue. This request is not meant for the crawl jobs, but specifically for if you want to add extra jobs manually.

- Add Crawl Jobs: Does the same thing as Add Jobs, but will mark the crawl job as done. It also requires a crawl ID. This is the ID from where the next crawl job will start crawling. The Database API will remember this ID and give it the the next worker node that is assigned a crawl job.

- Update job: Updates the given job to be started at the new current time. Also returns the new time.

- Finish job: Finishes the given job. It will be removed from the current jobs table and if it failed it will be added to the failed jobs table.

There is one more request the job request handler will handle: the connect request. The connect request is not called from a worker node, but rather from a different Database API. This request is to connect to the RAFT system. If the node that receives the request is the leader, we will return the IP and port of the leader node. If this node is the leader node, then we will accept the connection and send the initial data that the new node needs.

The other request will first ask the raft system if this node is the leader. If we are not the leader, we will pass the request on to the leader and return the output that the leader gives. If we are the leader, we will handle the request ourselves.

[1] https://raft.github.io/

9 Code coverage

This section contains some code coverage results. Some of these might be a bit outdated.

9.1 Controller

Trying to analyse the code coverage for the Controller Poses a small problem. When the code coverage is ran on the controller, all submodules will also be included. This makes it hard to get a concrete number for the total coverage.
The system tests are also not included in the code coverage, because they do not run in visual studio, which we use to analyse code coverage. Because of this, the command classes Start, Check, Upload and Checkupload are all not covered.

Here an overview of the code coverage when all tests except the system tests are ran:

Class	Blocks	Coverage	Notes
Check	65	0%	Is tested in the system tests, which is not included in the code coverage.
CheckUpload	64	0%	Is tested in the system tests, which is not included in the code coverage.
Command	3	0%	helpMessage not tested.
CommandFactory	76	0%	Not enough time to write good tests for this part.
DatabaseRequests	237	45%	Has a lot of similar trivial functions of which we only tested two. The main execRequest function is also only covered for 59% because we did not tests the retry functionality of the function.
EnvironmentDTO	11	100%	
ExecuteCommand	6	100%	
ExecuteCommandObj	6	78%	Some error cases are not tested.
Flags	203	6%	Not enough time to write good tests for this part.
NetworkHandler	215	46%	Networking part of the network handler is tested. Tests to see if the .env file is read correctly are not written.
NetworkUtils	408	96%	Only a test for getProjectRequest is missing.
PrintMatches	373	0%	Not enough time to write good tests for this part because of the complexity involved.
ProjectMetaData	51	100%	
Start	391	0%	Is tested in the system tests, which is not included in the code coverage.
Upload	82	0%	Is tested in the system tests, which is not included in the code coverage.
Utils	188	75%	No tests written for getExecutablePath and shuffle.

9.2 Parser

9.2.1 Unit tests

An overview of the coverage of the unit tests can be seen in Table 1. The SrcMLCaller class was not tested using unit tests, as most of its functionality simply calls srcML which we do not want to cover with unit tests.
No unit tests where made to test our custom parser, for large parts this would not be feasible as the code uses generated ANTLR files which we do not want to test using unit tests. There are, however, parts of the AntlrParsing class which could be unit tested, but due to time constraints we opted not to do this.

Class	Blocks	Coverage	Notes
AbstractSyntaxToHashable	117	78%	getHashable method has been updated without adapting unit tests, should be improved.
Logger	29	10%	Most functionality is trivial, and also uses an external library (loguru).
Node	90	96%	
StringStream	78	82%	
TagData	11	100%	
XmlParser	311	77%	One edgecase missing in handleUnitTag (unit tag for start of srcML output rather than start of file), handleClosingTag calls other parts of the program and is better tested using integration tests.

Table 1: Coverage of unit tests for the SearchSECO Parser.

9.2.2 Integration tests

The integration tests are divided up in 2 tests for each language we support. One of them tests whether the returned meta data for the methods is correct (filename, line numbers). The other checks whether the actual hashes found are correct. This division was made since any changes that would affect the hash should at least still get the same meta data, allowing a check before adapting the hashes in the tests to the new method. An overview of the coverage of the integration tests can be seen in Table 2

Class	Blocks	Coverage	Notes
AbstractSyntaxToHashable	117	95%	
AntlrParsing	272	82%	Some edge cases are not being hit, it might be best to tests these using unit tests.
CustomJavaScriptListener	247	95%	
CustomPython3Listener	160	95%	
JavaScriptAntlrImplementation	131	53%	A lot of error catching code is not being tested, it would be a good idea to add this to make sure they work.
Logger	29	21%	Most functionality is trivial, and also uses an external library (loguru).
Node	90	72%	
Python3AntlrImplementation	117	53%	A lot of error catching code is not being tested, it would be a good idea to add this to make sure they work.
SrcMLCaller	67	76%	
StringStream	78	91%	
TagData	11	100%	
XmlParser	311	83%	

Table 2: Coverage of integration tests for the SearchSECO Parser.

9.3 Crawler

The following table is an overview of the code coverage for all the classes of the crawler including both the unit tests and the integration tests.

Class	Blocks	Coverage	Notes
EmptyHandler	2	0%	Nothing to test, really.
ErrorHandler<enum JSONError>	20	100%	
ErrorHandler<enum githubAPIResponse>	22	95%	
GithubClientErrorConverter	31	35%	Testing the only function in this class would mostly come down to re-implementing that function as the function is a switch that sends a number to a githubAPIResponse.
GithubCrawler	375	93%	
GithubErrorThrowHandler	34	94 %	
GithubInterface	47	100%	
IndividualErrorHandler	1	0%	Abstract class, can't be tested.
JSON	460	79%	Not tested functions are mainly private functions or things that are hard to test or are (almost) never used.
JSONErrorHandler	34	97%	
JSONSingletonErrorHandler	10	100%	
LogHandler	20	50%	This function logs directly to the console and is therefore hard to test.
LogThrowHandler	20	55 %	Similar notes as LogHandler. It is tested as far as it can be tested but testing if the text it sends to console is the right text is rather hard.
LoggerCrawler	38	100%	
RunCrawler	150	73%	Can be tested further using more integration tests.
Utility	31	77%	This is code that is mainly copied from Stack Overflow. It could be tested more in the future.

9.4 Spider

An overview of the code coverage for the spider can be seen in tables 3 and 4.

Class	Blocks	Coverage	Notes
Error	3	0%	
ExecuteCommand	5	100%	
ExecuteCommandObj	50	50%	Contains all functions of ExecuteCommand, but not all of those are used.
Filesystem	14	64%	Remove function isn't tested, but it simply contains a call to a builtin function.
FilesystemImp	62	15%	Contains all functions of FileSystem, but not all of those are used.
Git	407	77%	
GitSpider	170	85%	
Logger	32	38%	
RunSpider	148	38%	Functions requiring a git repository to test aren't tested
Spider	19	68%	

Table 3: Overview of coverage of unit tests for Spider

Class	Blocks	Coverage	Notes
Error	3	0%	
ExecuteCommand	5	100%	
ExecuteCommandObj	50	78%	
Filesystem	14	86%	
FilesystemImp	62	89%	
Git	407	65%	Certain settings of the Git class aren't tested, as a single test requires a download of a repository which takes time.
GitSpider	170	99%	
Logger	32	25%	Errors and crashes not all covered as reproducing certain errors is hard in integration tests.
RunSpider	148	55%	Only runSpider method tested.
Spider	19	68%	

Table 4: Overview of coverage of integration tests for Spider

9.5 Database API

An overview of the coverage of the code can be found below in Table 5. We will discuss unit tests and integration tests separately afterwards.

Class	Lines	Coverage	Notes
DatabaseHandler	439	88%	Some edge cases are not handled.
DatabaseRequestHandler	562	93%	Most edge cases are not tested.
ConnectionHandler	75	0%	The tests did not work properly, so we removed them.
Database-API	8	0%	Nothing to test, really.
HTTPStatus	21	95%	A single edge case is not handled.
RequestHandler	76	88%	The connect request for the job request handler is not tested.
Utility	49	100%	Fully tested.
DatabaseConnection	107	76%	Some edge cases are not handled.
JobRequestHandler	102	72%	Mostly unused retry functionality is not tested. Additionally, some edge cases are not handled.
Networking	29	21%	Sending and receiving data is not tested.
RaftConsensus	188	46%	Some of the tests did not work properly, so we removed them.

Table 5: Code coverage for the SearchSECO Database API.

9.5.1 Unit tests

The functions in the DatabaseRequestHandler, RequestHandler and the JobRequestHandler are almost completely unit tested. The main issue with the tests is that a lot of edge cases are not tested. This could be done using the Database mock. This clarifies percentages like 88% and 93% for the DatabaseHandler and the DatabaseRequestHandler respectively.

9.5.2 Integration tests

The classes ConnectionHandler, Networking and RAFTConsensus are not tested properly. We had originally introduced tests to test their functionality, but the tests mostly did not work consistently, so we decided to remove most of the tests. Other parts of the programming, primarily the integration of the request handlers with the database, are tested properly, as far as can be tested.

10 Unsolved Known Issues

10.1 Controller

Fatal error with checkupload request
If a user sends a checkupload request without arguments, a fatal error occurs. Preferably, we would want to provide the user with a message on how to use the command properly.

10.2 Database API

Connection from server without open ports result in crash
When the database API gets a connect request from a server without open ports the database API crashes. Preferably, the database API should not crash but just continue.

11 Future improvements

Here, we discuss observations we made that could be improved in the future.

- The custom parser is quite a bit (30 times) slower than the srcML parser. We could improve this by constructing a more simplified grammar than the general one provided by ANTLR.
- In the database API, multiple methods use similar code leading to much code duplication. Think for example of the single query thread methods. This could be fixed by adding a bit of abstraction to the code. We have also partially applied abstraction already by means of template functions, but other approaches may also be possible.
- In the database API, some components (i.e., ConnectionHandler, Networking and RAFTConsensus) are not (fully) tested due to the fact that the tests did not pass consistently. It would be useful to fix these tests to obtain a better code coverage.
- The unit tests in general do not cover all edge cases. It would be good to improve this.
- The custom parser needs to convert files to UTF-8 since that is the format ANTLR requires, currently there is some code which converts ANSI to UTF-8 and seems to also work on some other formats. It is not known what happens when the parser encounters a format/character that it can't work with, hopefully this will result in an error in the conversion which would be catched and handled. It could however also be send to ANTLR causing a crash.
- Communication between controller and database-api, as well as between multiple database-api's, is currently unencrypted in plain text. It would be trivial to send data to the database-api from any system, and possibly to find and exploit vulnerabilities in the database-api to hack a system running it.
- The contact between the database-api and the database as well as the different databases is not encrypted.
- The contact between the database-api and the database is not authenticated, so anyone would be able to execute queries on the database quite easily.
- SrcML seems to not recognise all valid code, when encountering something it doesn't recognise there is a chance that the rest of the file won't be parsed properly either. Exact cases that srcML doesn't recognise are not known to us.
- The job queue is currently saved in a Cassandra database, this seems to work fine at the moment, but since the Cassandra database handles deletes by inserting tombstones (leading to longer lookup times since all tombstones will be found first) this could lead to problems when scaling up the system.
- Sometimes a job can get stuck. There is a timeout implemented to combat this, but even with this timeout it can get stuck.
- Some of the timeouts are still a bit short or long, so it would be good to take another good look into tweaking the timeouts.
- To check whether a project has already been uploaded the controller asks for the most recent version of the project. This however does not mean that all previous versions have been uploaded, or that this is a complete version, as it might be a vulnerability. It would be good to implement a better check by retrieving all versions and checking whether the exact version is already in the database, instead of a newer one.
- When a job times out and the database-api removes the job, the controller will try to finish the job for a total of 6 times, because it fails every time. It would be better to not do this.

12 Licenses

Our project will be licensed under GNU AGPLv3, some of the code/libraries used have individual licenses which will be summarised here.

12.1 ANTLR

Is licensed under the following BSD license, which is included in all files and/or subdirectories containing files covered by the license.

12.2 Cassandra, Datastax, and cassandra-exporter-agent

These three are licensed under the Apache license, version 2.0 (https://www.apache.org/licenses/LICENSE-2.0).

12.3 BOOST

Boost has its own license, which can be found here.

12.3.1 ANTLR python3 and javascript grammar, curlcpp, and nlohmann-json

These are covered by the MIT license, which is included in all files and/or subdirectories containing files covered by the license.

12.4 curl

Curl uses a custom license, which is included in all files and/or subdirectories containing files covered by the license.

12.5 RSA Data Security, Inc. MD5 Message-Digest Algorithm

For Md5 we use RSA Data Security, Inc. MD5 Message-Digest Algorithm, this has a license included in the files containing relevant code.

12.6 Other things used by the project

Our project also uses a number of other projects, our project does, however, not derive from these and their license does not have to be included. SrcML is licensed under the GNU general public license, version 3. Git is licensed under the GNU general public license, version 2.0. Grafana is licensed under GNU general public license, version 3. Prometheus is licensed under Apache license, version 2.0.

References

[1] Kim, S., Woo, S., Lee, H. and Oh, H., 2017, May. Vuddy: A scalable approach for vulnerable code clone discovery. In 2017 IEEE Symposium on Security and Privacy (SP) (pp. 595-614). IEEE.

[2] GitHub Inc. (https://github.com): a provider of Internet hosting for software development and version control using Git.

[3] GitLab (https://gitlab.com): a web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking and continuous integration and deployment pipeline features, using an open-source license, developed by GitLab Inc.

[4] Software Heritage Graph (https://softwareheritage.org): the largest existing public archive of software source code and accompanying development history.

[5] Apache Cassandra (https://cassandra.apache.org/): a free and open-source, distributed, wide-column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Documentation can be found at https://cassandra.apache.org/doc/latest/

[6] srcML: https://www.srcml.org/

[7] ANTLR: https://www.antlr.org/

[8] libcurl (https://curl.se/libcurl/) is a free and easy-to-use client-side URL transfer library, supporting DICT, FILE, FTP, FTPS, GOPHER, GOPHERS, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, MQTT, POP3, POP3S, RTMP, RTMPS, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET and TFTP.

[9] Loguru: https://github.com/emilk/loguru

[10] Boost: https://www.boost.org/

[11] Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K. and Mia, M.M., 2014, September. Towards a big data curated benchmark of inter-project code clones. In 2014 IEEE International Conference on Software Maintenance and Evolution (pp. 476-480). IEEE.

[12] Farhadi, M.R., Fung, B.C., Charland, P. and Debbabi, M., 2014, June. Binclone: Detecting code clones in malware. In 2014 Eighth International Conference on Software Security and Reliability (SERE) (pp. 78-87). IEEE.

Appendix A: UML Diagrams

This appendix contains the UML diagrams for the system. Some of these can be partially outdated or incomplete.

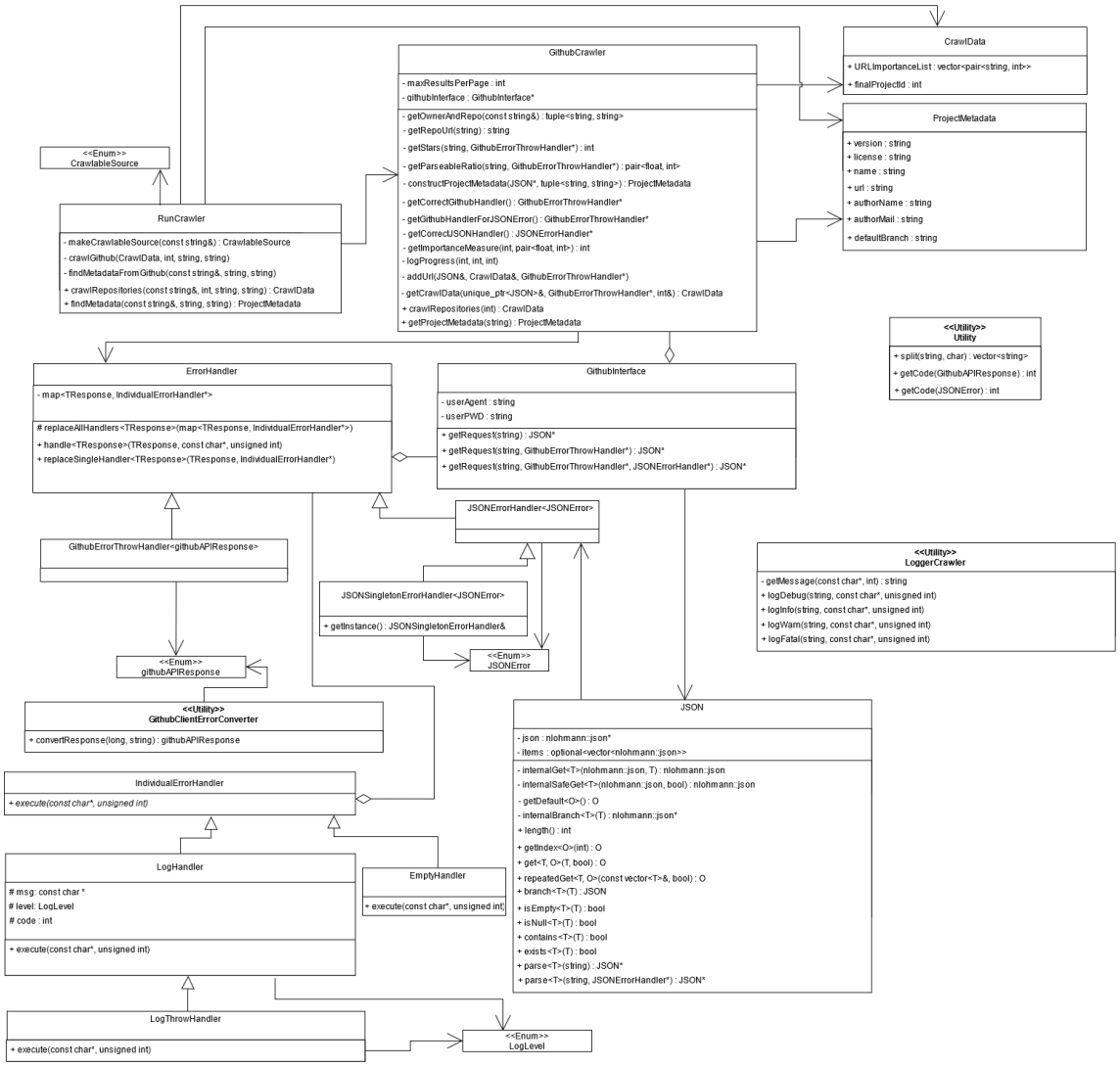

Figure 4: Crawler UML class diagram

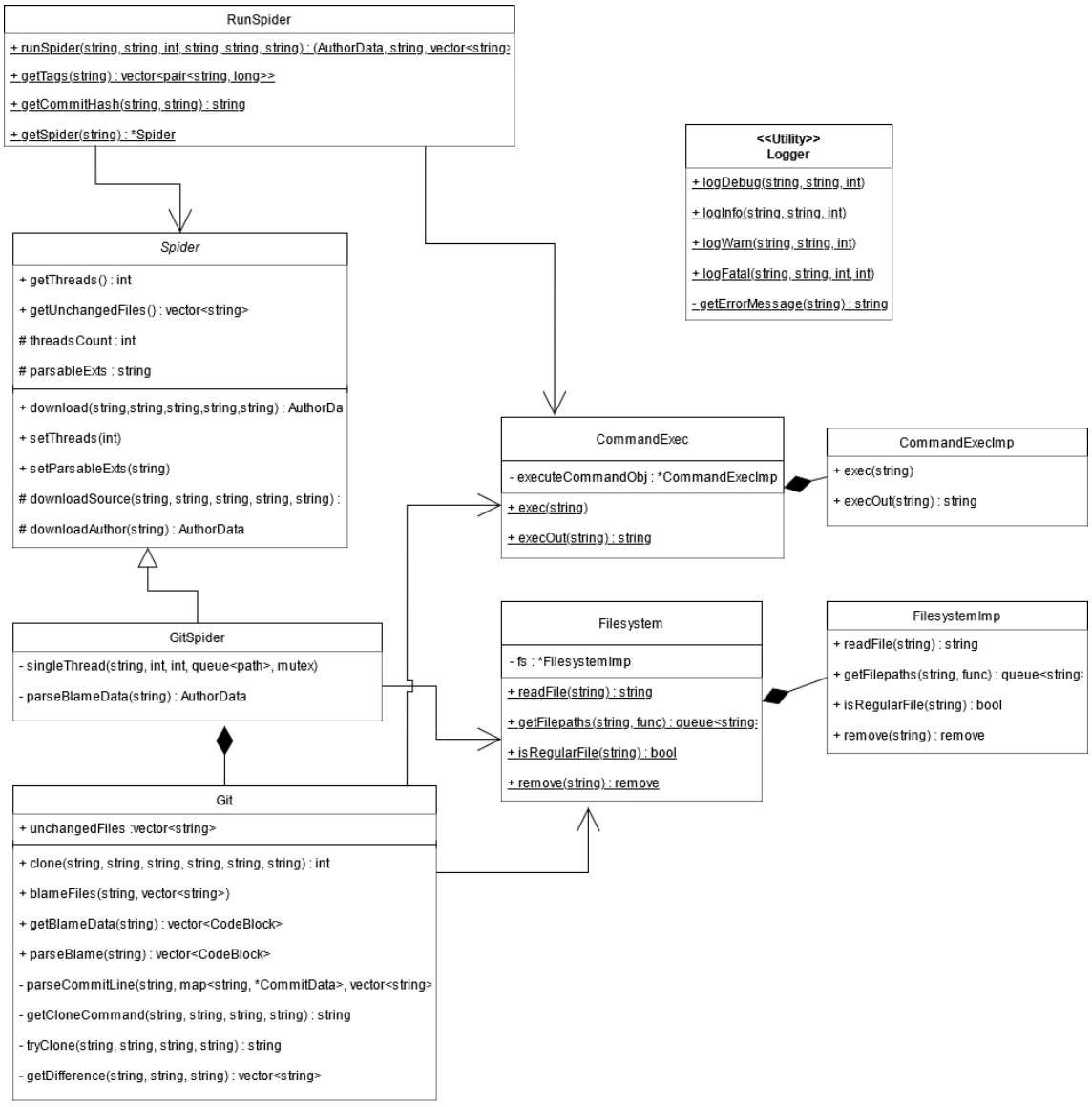

Figure 5: Spider UML class diagram

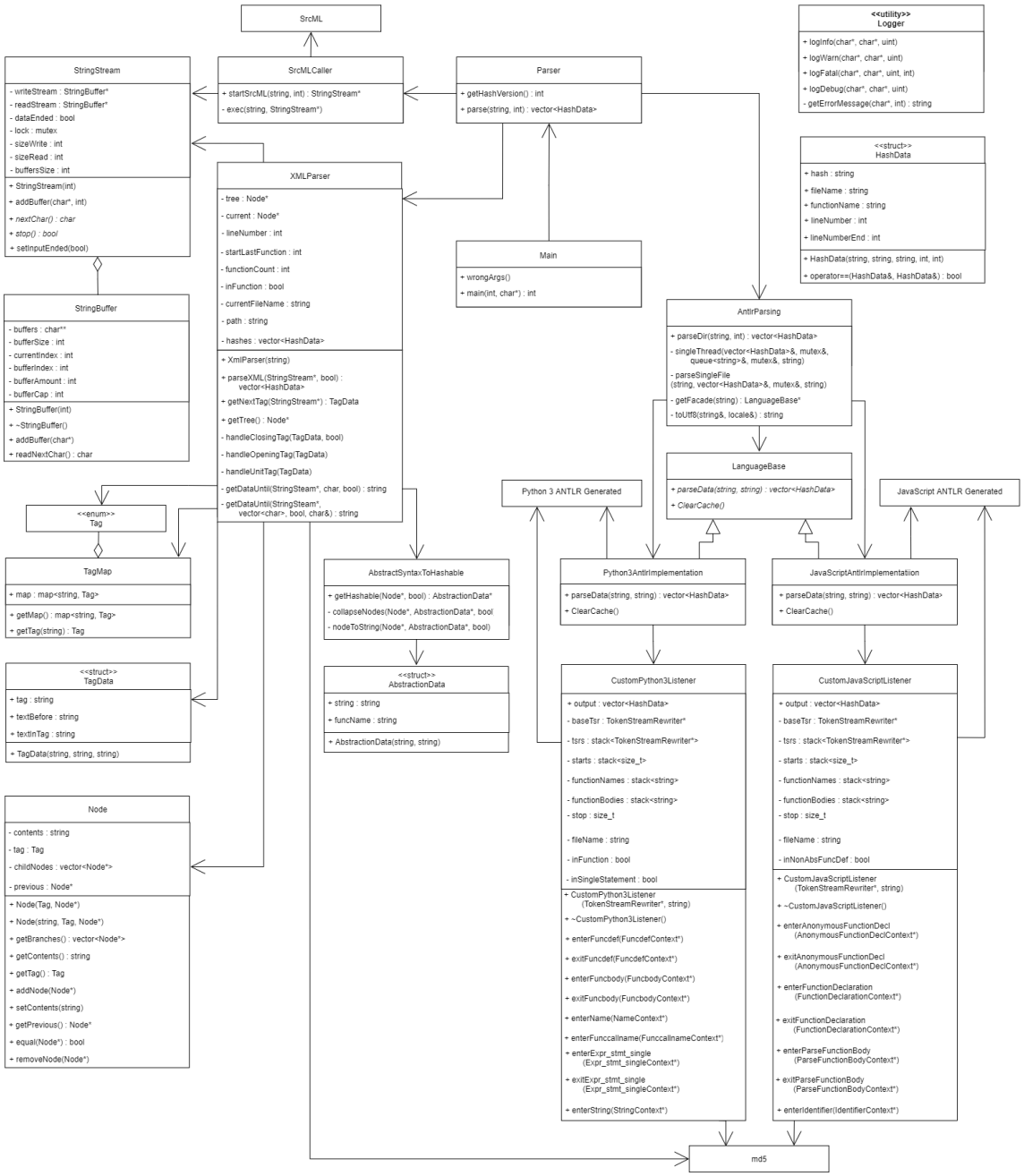

Figure 6: Parser UML class diagram

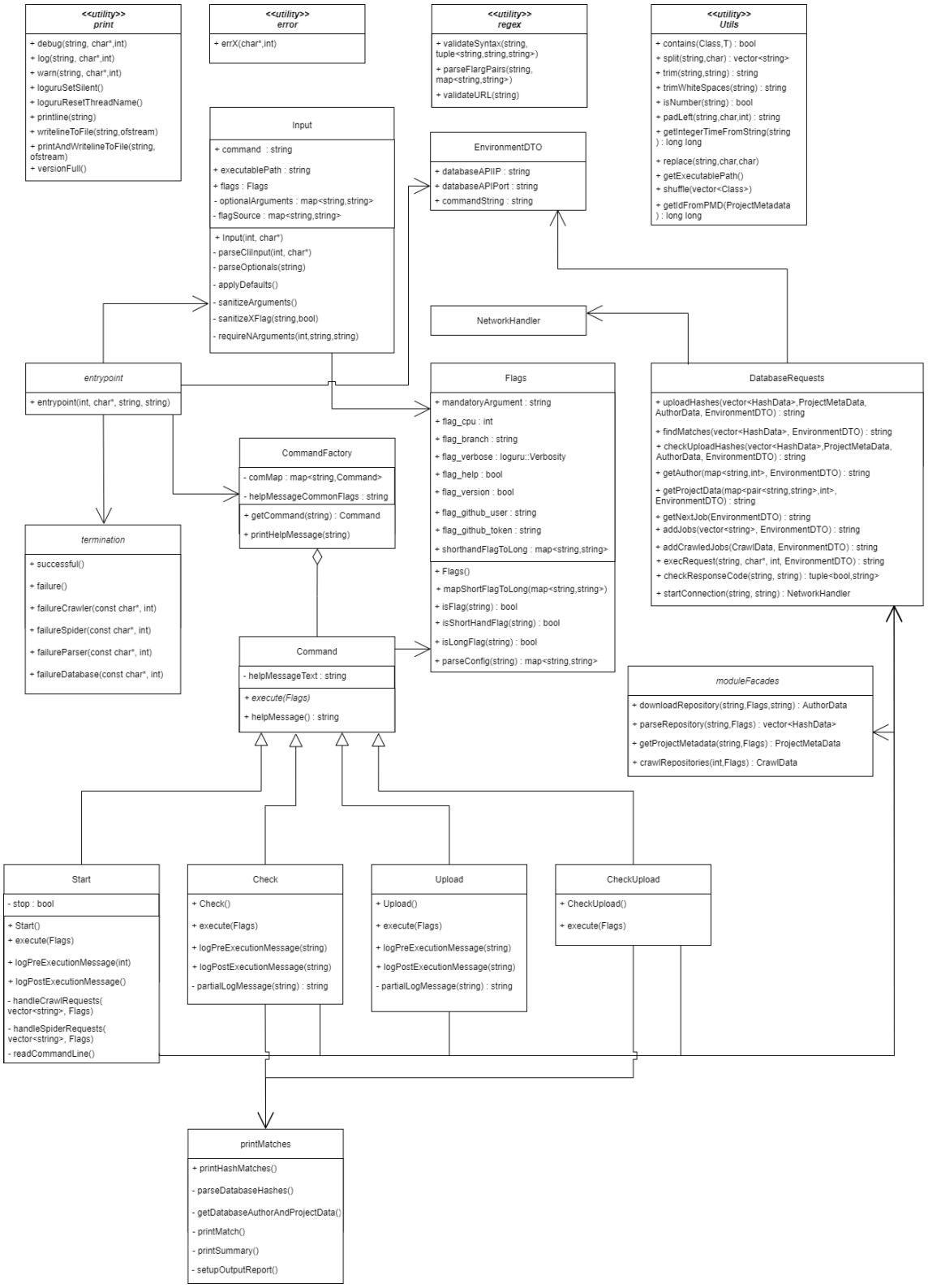

Figure 7: Controller UML class diagram

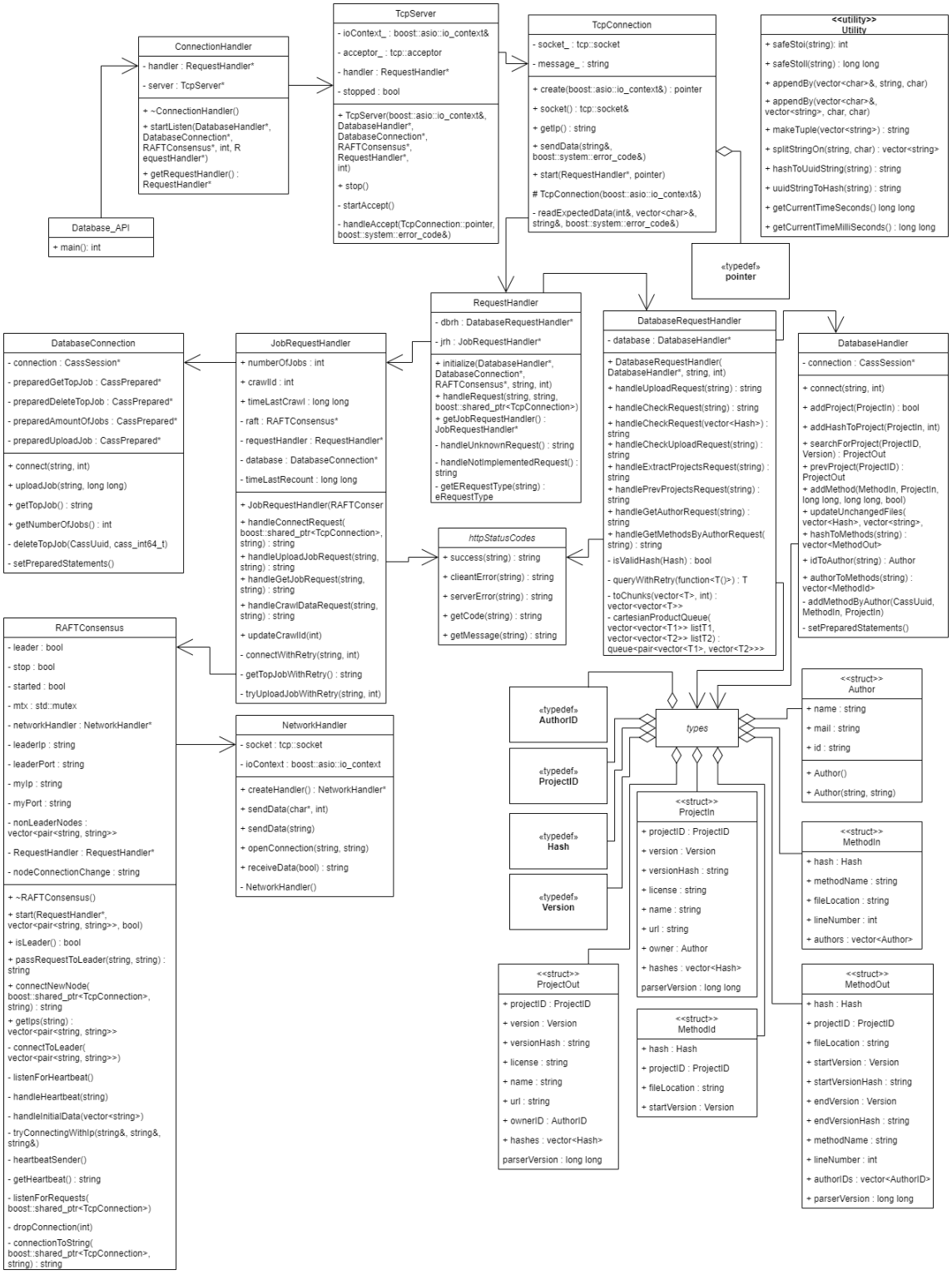

Figure 8: Database API UML class diagram