**Rate Limit Documentation**

* **Understanding of Github API Rate limits**

Github API allows access to all the available public user profiles and repositories on Github through their API’s hosted at URL - https://api.github.com/

There are well defined APIs to access information on –

1. Github Users
2. Github User Repositories
3. User Commits and URLs

Any user accessing this Github API URL to fetch data needs to follow some basic rules limiting the number of requests made to the Github API in given time.

As mentioned in Github Developer Guide Reference as below, https://developer.github.com/v3/#rate-limiting

The defined rate Limit for –

1. Authenticated User – 5000 requests/per hour
2. Unauthorized User – 60 requests/per hour.

There are several parameters present in the response header of each request made by user to Github API URL that defines the Rate Limits, Remaining access per user, Reset Window Timestamp per user. These are as below,

| **Header Name** | **Description** |
| --- | --- |
| X-RateLimit-Limit | The maximum number of requests you're permitted to make per hour. |
| X-RateLimit-Remaining | The number of requests remaining in the current rate limit window. |
| X-RateLimit-Reset | The time at which the current rate limit window resets in UTC epoch seconds. |

Using these constraints, I came up with a solution to fetch the required details from Github API for public user profiles. My solution relies on a generic anonymous user to fetch data from Github, respecting the Rate Limits.

* **Solution Project : RateLimiter Approach**

I have tried to manage the rate limit constraint posed by Github API, by running an extra Monitor thread, along with a pool of Worker threads. **The application fetches multiple GitHub user’s data together without failing the rate limits using the below thread model.**

1. **Worker Threads**

These threads are used to hit Github API’s and fetch data in json format. Later this json data is parsed and the output object is created, to be written to output file. It waits for Monitor thread to notify it in order to acquire the Lock.

1. **Monitor Threads**

This is a single thread run at the start of the application and is invoked every 2 second to check whether the count of requests made to Github API has exceeded the limit or not.

* If limit is not reached, it will notify the waiting worker threads to fetch and process data.
* If limit is reached, this thread will sleep for 1 window time, after which the request limit for Github is reset for current user to 0. **Unless Monitor notifies, not a single request is made to Github, this is the main idea.**

1. **Main Thread**

This is the main thread of the Application from which the application starts running.

It is used to initialize the **RateLimitParam model object (LOCK object)** with start values and starts the Monitor thread also.

* **Execution Flow**

Below is the design Diagram, for the Rate Limiter Application.

It demonstrates –

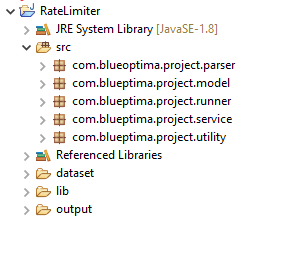
1. The Flow of control
2. The Model objects
3. The Execution logic
4. The Interfaces and Implementation classes design at a very High level.

A picture containing text, whiteboard

Description automatically generated

* **Detailed Explanation on Classes and Application Design**

At a high level, below is the structure of the project packages.



***Rate Limiter Project packages –***

* **/src**

It consists of the following subpackages –

1. com.blueoptima.project.parser - interfaces and classes for Json parsing of input and output.
2. com.blueoptima.project.model - classes for business data objects/POJO
3. com.blueoptima.project.runner - the application runner, monitor thread and worker thread classes.
4. com.blueoptima.project.service - interfaces and classes for the REST client used to read data from Github API.
5. com.blueoptima.project.utility - utility classes for writing output to file, validators for output from API and constants.

* **/dataset** It consists of the sample inputin a json file.
* **/lib** It consists of the library jars for consuming REST service , parsing json input/output using Jackson parser etc.
* **/output** It consists of the output generated for the sample input.

***Rate Limiter Project Classes and Interfaces –***

1. **ApplicationRunner**

This is the starting point of the application. It performs the following tasks in main method –

1. Setting the data object RateLimitParam with default values for –
2. X\_RateLimit\_Reset
3. X\_RateLimit\_Remaining
4. Timestamp
5. ResetIndicator

This object is used as a lock for the monitor thread and Worker thread to wait and notify when checking for rate limits at specified intervals.

1. Starting MonitorThread class. It is initialized with the data object RateLimitParam and the delay time interval of 2 seconds to check on the lock.
2. Invoking the InputParser passing it the lock object – RateLimitParam.
3. Sleep for 1 second and shutdown the MonitorThread and the end of the Application.

**2. MonitorThread**

This class is a thread to run as a checking monitor on the rate limits remaining after each API call made to Github. Hence this monitor is invoked at a very small interval of 2 seconds.

It does the following things in the method **public void run()** –

1. At first it sleeps for 1 second.
2. Next it checks for the condition –

githubRateLimit.getX\_RateLimit\_Remaining() > 5

If it is true, that is rate limit is not exceeded –

It notifies the waiting threads.

Otherwise,

It calculates the sleep time for worker threads and does not notifies them, for that much time. This sleep time is calculated based on two things –

1. github rate limit reset window = 1hour for anonymous user and ,
2. the timestamp of first request in a window.

Then it sleeps for the calculated sleep time.

1. **IInputParsingTool**

It is an interface for integrating any kind of input parser to the application.

At present the tool used as implementation for this interface is Jackson Parser Streaming Version to parse the input JSON file. This interface is implemented by InputParser class method **processJsonInput(filename).**

1. **InputParser**

This class uses Jackson Streaming parser to parse the input.json file not as a whole but chunk wise. This helps to incorporate the capability of reading large size input files for the application.

It parses the input, reading Firstname, Lastname and Location for Users provided to search for their Github Developer profiles. It has below methods -

**initilizeParser()**

1. Initialize parser with the Json Parser factory objects.
2. Initialize the Input data POJO class UserProfileInput
3. Initialize the executor for worker threads with thread pool size of 5.

**processJsonInput (filename)**

1. Parse input.json.
2. Start Worker threads

**shutdownParser()**

1. Shutdown Parser, by checking if all the worker threads are finished now.
2. **WorkerThread**

This class initializes a thread with two objects –

1. UserProfileInput <input data object>
2. RateLimitParam <lock object>

It performs the basic task of fetching data from Github for users provided in input and parse the json response received from the REST API. It has below methods -

**run()**

1. It is called when the WorkerThread is started.

**processUserProfile()**

1. It instantiates and initializes the Rest Client class and the GithubResponseMapper class.
2. Fetches User profile. Checks if User exists on Github.
3. Extracts Repository URL for Given User.
4. Fetches data from the Repository URL for list of User Repos.
5. For Each Repo, parses to read the commits URL
6. Fetches each commit URL and counts the number of commits made the the given user.
7. Updates the output data object UserProfileOutput with the data for user public profile and User Repo names and Repo commit counts for each.
8. Instantiates a file handle and adds the data gathered to an output file created in the name of user.
9. **IClient**

This is an interface used to expose the methods for reading data from the Github REST API.

We can write different implementations for different URLs using this interface. It is implemented by the class RestClient.

1. **RestClient**

This class RestClient is used to make calls to different webservices provided by Github to read data about user’s public profile. It has the following methods-

1. **callWebservice(apiURL, acceptParam)** Creates a webResource with given inputs and records info on the lock object. This method before hitting any URL calls on wait() on lock object, waiting to get notified by the MonitorThread once monitor ensures ratelimit is not reached. It updates the lock object with rate-limit-remaining infor after hitting each request URL.
2. **createRequest(user)** Creates request with given user firstname, lastname and location.
3. **getUserProfile(user)**
4. **getUserRepoList(repo\_url)**
5. **getUserNoOfCommits(repo\_list)**
6. **GithubResponseMapper**

This class is useful in creating the specific output object as is the requirement of this application.. It has the following methods-

1. **extractRepoPath(jsonData)** From user profile reads user Repo URL
2. **extractRepoList(jsonData)** For each Repo Found from aboveURL, repo name is extracted and commits url is extracted and a HashMap of this data is created and returned.
3. **extractCommitCount(repoMap, user)** For each entry of the HahMap created above, commitURL fetches data on total commits. For each user’s each repo, all commits are checked for an exact or partial match in name. Those found are added to a result map with key as Repo Name and value as all commits made by this user.
4. **IPostUserToDataStore**

This is an interface to update the user github profile output object formed by the processing of WorkerThread, to some data store. In this application’s scope File is used as permanent data store. It has the following methods-

1. **PostUserToFile**

This class is used to add the UserProfileOutput object contents to a file created in the name of User’s name and location given as input. Output file has a JSON dictionary format.It has the following methods-

1. **addUserToDataStore(userProfile, user)**

Other Classes are –

Data Model for input, Output and Lock Objects-

1. **RateLimitParam**
2. **UserProfileInput**
3. **UserProfileOutput**

Utility Classes –

1. **Constants** Class storing all constant values used in the application.
2. **Validator** Class used to validate the input and output values for null and spaces.

* **Assumptions**

1. Github searches users based on our input for first name, last name and location. As the application is looking for a specific user only, the best match user returned in response with the maximum match score is picked for processing.
2. When user not found no record added Ex – *Sharvi Verma, India*
3. Did not remove Sysouts to explain code flow to the evaluator.
4. Input and Output both are JSON files.
5. Using anonymous account to access Github API.

* **Design Considerations**

1. Use of **JackSon JSON Parser for streaming very large inputs** for the application without causing a burden on the memory.
2. Came up with a design, having generic interfaces exposed methods, with implementation classes defining their specific behavior. This **ensured loose coupling of dependencies**.

Input, Output, Rest Client and Monitor threads all are individual plugin-plug out components.

1. Used the concept of Executor to **handle Threads in a pool, to reduce overhead** of thread creation and destroy.
2. **RateLimitParam** Model object used as the **LOCK object is instantiated** at the beginning of the Application **as a Singleton and is maintained all through to achieve consistency** in thread results.
3. No use of field variables or Global objects, to maintain thread accuracy of shared variables.
4. Used BufferedWriter to achieve fast performance while writing Strings output to files.

* **Additional Criteria for Accuracy**

1. As the Github [docs](https://developer.github.com/v3/repos/) indicate, these API methods take a **type parameter** as input that can filter the repositories returned based on what type of access the user has for the repository. In this way, we can fetch only directly owned repositories excluding the organization repositories, or repositories the user collaborates on via a team.
2. I tried to use the X-RateLimit-Reset variable from response header of Github API calls to calculate the wait time for next start window, but it was giving improper results. Using the correct value of this dynamically can help to improve performance of the application by avoiding unnecessary fixed wait time.

* **Improvements** *If time permitted* - 1.Application could have been scaled to use MongoDB as a store for output.

1. Application could have been scaled to parse the Json input from large files and fed the input records to a queue that could streamline the working of WorkerThreads in case of failures.

* **Overall Summary**

1. The application can be run by running the main class – ApplicationRunner.
2. Put the inputs in a json file, named input.json and place it in dataset folder in the application.
3. The outputs generated can be seen in the folder output.

* **Application Performance**

The application can run for all 13 users given in sample input with a total time of approximately 35 mins, given a wait time of 6 mins each.