Design

Google Sheets Table Design:

"GitHub Issue number" gives the user easy reference to the specific issue. Sorted in ascending order.

The state of the Issue, whether it be opened or closed, is represented graphically with conditional formatting for easy identification.

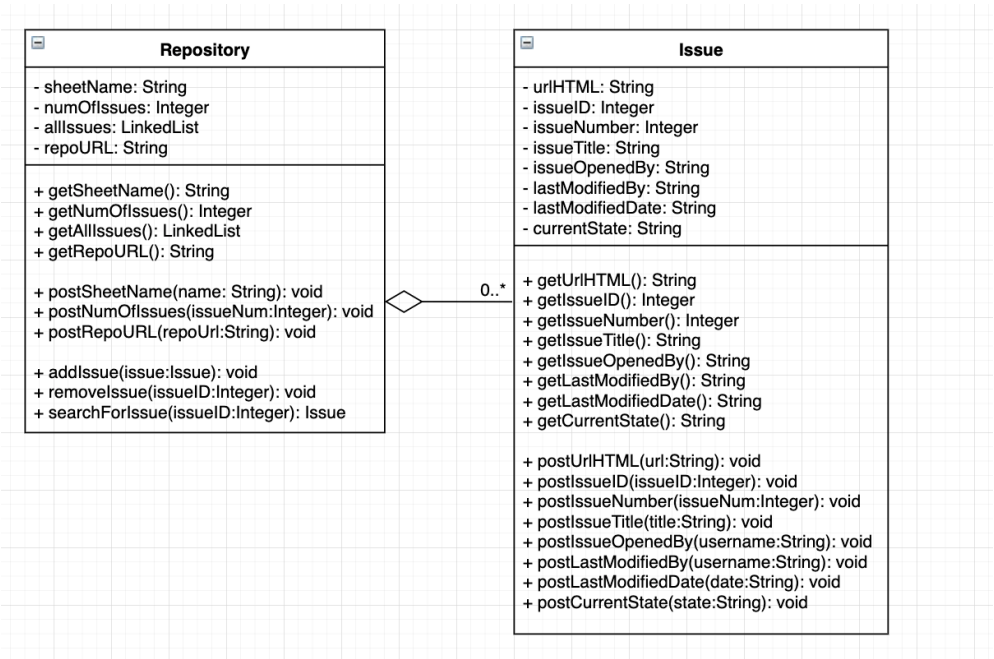
Num	Issue Title	Issue Opened By:	Last Modified By:	Last Modified Date:	State	URL
1	Issue Test 1	*GitHub Username*	*GitHub Username*	*DD/MM/YYYY HH:mm:ss*	Open	xxxxxx.com/issue#1
2	Issue Test 2	*GitHub Username*	*GitHub Username*	*DD/MM/YYYY HH:mm:ss*	Closed	xxxxxx.com/issue#2
3	Issue Test 3	*GitHub Username*	*GitHub Username*	*DD/MM/YYYY HH:mm:ss*	Closed	xxxxxx.com/issue#3
4	Issue Test 4	*GitHub Username*	*GitHub Username*	*DD/MM/YYYY HH:mm:ss*	Open	xxxxxx.com/issue#4

Repository-Name-1 | Repository-Name-2

The different tabs below the Google Sheets can be toggled to switch between different repository's Issue logs. They are automatically named to the name of the repository they represent.

Code Design:

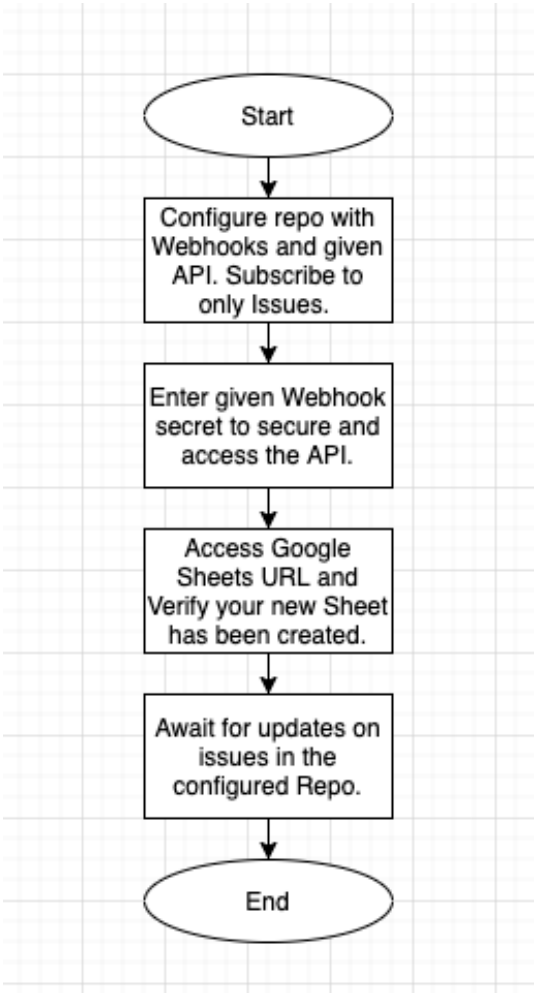
I've decided to develop this program in an object-oriented way on NodeJS. Hence, I've designed the code to be modular, splitting the code into two main classes: "Repository" and "Issue". The UML diagram can be found below.



A modular design is advantageous for both development and maintenance as each module can be developed individually and put together once they're complete. It also decreases code maintenance by keeping debugging focussed and quick as you are only ever working with a small section of code at a time. Furthermore, it allows the code to be extensible and futureproof. Each part of the program doesn't necessarily depend on the other to work; new "modules" of code can be added to expand the program easily.

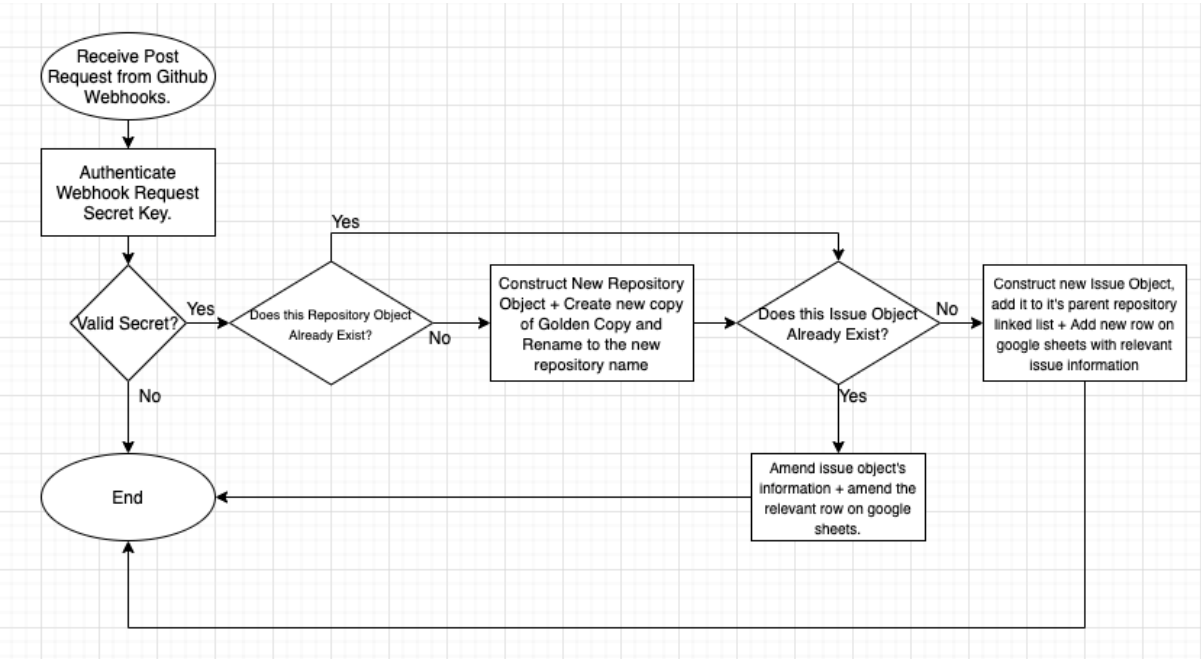
To contain the data, I'll be using multi-dimensional linked lists. "Issues" will be sorted into "Issue linked lists". This linked list, as seen in the UML diagram, is sorted into a repository object. Finally, all repositories are sorted together in a singular repository linked list. I've chosen to use link lists over static array structures because it's more memory efficient. Linked lists are also, by nature, dynamic, a feature we require due to the given scenario; we don't know how many repositories may be added. However, an argument could be made to use arrays or JSON to store data due to their faster memory access, neglecting the need for a search algorithm to access data.

Frontend Flowchart



This "frontend flowchart" depicts the logic flow for any typical user that wants to interact with our API. Only authenticated users should use the API. Hence, they will be required to enter the Webhook "secret" on the user end. The user should not share this secret publicly as it'll compromise the safety of the API and server. Only with the correct Webhook "secret" will the API work.

Backend Flowchart



The backend flowchart illustrates the logic flow after the API endpoint receives a Webhook from GitHub.

Test Plan

ACTION TO TEST	METHOD OF TESTING	EXPECTED RESULT
Any GitHub repository with the correct authentication and API URL should be able to connect to the server and update the Google Sheets.	Test updating issues with a non-authenticated API call vs an authenticated API call.	The non-authenticated API call will fail whereas the authenticated API will work as expected.
The Google Sheets document should have the ability to contain multiple GitHub repository "Issues" logs. Every sheet within a document should represent one repository log.	Try to add multiple GitHub repositories to the Google Sheets by creating several dummy repositories with the proper Webhook secret and API URL.	Multiple sheets should be created on that singular document such that each sheet represents a new repository.
To test whether the implementation of the Webhooks API works, the Webhooks "Ping" command should automatically initiate a new sheet for the repository, given that it doesn't already exist.	Set up Webhooks with the correct secret and API URL on a new repository. The initial ping request should then be sent. Afterwards redeliver your ping. Check your Google Document.	The initial ping request should create a new Sheet on the Google document. Any subsequent ping requests from the same repository will be ignored.
New issues should get their own row on the Google Sheet. However, updates to an old issue should be modified on an already-existing row.	Create a new Issue. Then close this issue.	The new Issue should be populated on a new line. However, you close the issue it should update the "state" of that same line instead of making a new row.
Issues should be sorted by "Sheets", with each "Sheet" representing a repository.	Create new Issues in multiple repositories using our API.	Each new issue should be populated under the correct tab in their own respective sheet.
The Google Sheets will contain the headers with the correct information for *"Number, Title, URL, Username, State".*	Check Google sheets to ensure these headers exist.	You should have the following / equivalent headers: *"Number, Title, URL, Username, State".*