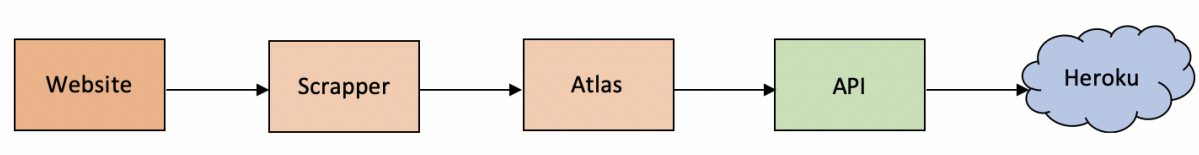
API Design Details Report



Scrapper Architecture:

- For our Scrapper we have used BS4 in order to extract data from the given website. Then we have modelled the data using JSON and placed it into Atlas MongoDB.

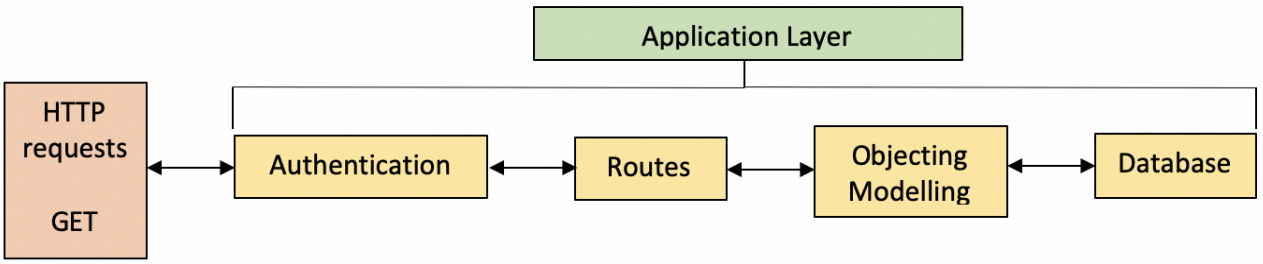
	Modules	Reason For Choice	Challenges	Shortcomings
Scrapper	BS4	- Used the html.parser as a parser, does not require external dependency hence is a fast parser.	- Not able to interact with JavaScript elements. - BS4's simplicity means that it's hard to differentiate content contained in the same tags. We had to outsource some scraping tasks to external libraries.	- We didn't push the scraper to the cloud and have it run and update the database automatically. - When scraping data we did not implement a way to make a distinction between old and new data. - We did not use NLP to extract information efficiently.
	Selenium	- Used to interact with JavaScript, since neither requests or BS4 could.		
	Threading	- Increases 1 request/second to approximately 150 requests/second. - Used concurrent.futures instead of threading since it bypasses pythons global interpreter lock and hence does real threading - concurrent.futures also has syntactic sugar which simplifies threading code.		
	JSON Output	- Directly compatible with MongoDB which converts it to an encodable format (BSON). - Faster parsing than its alternative XML		
	Requests	Used the request module to retrieve html source instead of BS4/Selenium whenever possible since it is native and doesn't require parsers or drivers.		

API Architecture:

- For our API architecture we have designed a single all purpose endpoint. It allows for sorting and projection on multiple fields including: disease, location, and date. Our API catches errors and returns them as messages when the field is missing. The API Requests are logged to the users with their request and saved in the backend in different suitable formats.

- We constructed our REST API on a Node.js server. This decision was based on:

    - Node.js having single threading means our API can fire off multiple concurrent requests without consuming extra memory and data and therefore has a high performance.
    - All long-running tasks (data access, input/output) are always executed asynchronously on top of worker threads.
    - The Node.js server uses the Express framework for creating and exposing our RESTful API to communicate as a client with our server application. The Express routes that we made are mapped to HTTP requests.



- The above diagram summarises the flow of our API. It starts with the user making a Get request then it starts by authenticating the accessibility of the user, then it accesses the routes where we have designed our end-point. Then using Mongoose we obtain the required data from the database that is stored in MongoDB Atlas. Then we retrieve required data, and it passes through again the same process in order to return it back to the user.

Below is a table of the reasons behind choosing our modules to design our API, challenges faced, and the shortcomings:

	Modules	Reason For Choice	Challenges	Shortcomings
API	NodeJS - (Express)	- Used Nodemon to optimise workflow, refresh the server on changes. - Used eslint and husky to create pre-commit hooks that would warn users about unsatisfactory code when pushing. - NodeJS architecture involves a single-thread event loop, which, when paired with the non-blocking nature of Node.js, allows it to handle thousands of concurrent connections without incurring thread context switching. - This is helpful to us since we need to instantiate the web crawler and fetch data in concurrent requests to increase performance and speed since our application would require frequent data transmission from the client to the server.	- Since Node.js is asynchronous, it uses callback functions for each queued task. The load that these tasks produce in the form of callback functions can result in 'callback hell' which means that there are a lot of nested callbacks several levels deep making the code unreadable. This makes it harder to cross-collaborate with other programmers and makes maintenance generally harder. - Learning about query expressions mongodb/mongoose.	- We did not add Pagination.
	MongoDB - (Mongoose)	- Optimized for write performance, and features a specific API for rapidly inserting data, prioritizing speed over transaction safety. - Performance is an imperative aspect to consider since we would need fast CRUD operations to take place on our app.		

Deployment:

- We decided to run our API on the Heroku Web Service as it offers easy integration with Node.js, has a simple log-checker and does not require users to create a start script.
- Our current API link is : https://crackedpepper.herokuapp.com

	Modules	Reason For Choice	Challenges	Shortcomings
Deployment	Heroku	- Contains premade buildpacks that easily prepare a node js for server deployment. -"Heroku Git CLI" allows for build versioning, and pushing code from main work branch in github to Heroku server which we found convenient and easy to use. - Config Vars for database access can be configured through a UI. - Built-in simple log checker, furthermore Heroku provides a wide range of Add-ons (Very valuable for Log analysis, however requires credentials). - Heroku automatically runs a start script and does not require users to create one. - Integrates nice with Github allowing for automatic deployment if tests made with github actions succeed.	- Learning how to use Heroku and trouble shooting the connection to the database was time consuming.	- So many useful add-ons which would have allowed us to better monitor our app.