**Programming Manual**

**ANNOTATOR API**

**CSIRO -Group Members**

Bhyravajoshyula Sathvik Joshi

Sai Anurag Malyala

Sakshi Chaudhary

Harsha Sugandika Adikari

Syamala Phani Deepika Chittajallu

**Introduction**

This programming manual provides information for application developers about the architecture of the application. It provides full description about how the text annotating webservice connects to external web services.

The application provides the following features:

* User types the words or phrases and the system finds definitions from external vocabularies.
* Allow the user to select which definition applies to the usage of the term in the context.
* Allow the user to select an alternative or preferred term to replace the original.
* A RESTful API and a user-interface which accesses the service through the API.

**Contents**

**1 Architecture of the Application**

1.1Frontend

1.1.1 Loading Ontologies

1.1.2 Populating the table

1.2 Backend

1.2.1 Node JS Server

1.2.2 API Endpoints

1.2.3 Connecting to External APIs

1.2.4 Setting-up the URL

1.2.5 Adapter classes

1.3 External Vocabularies.

1.4 Saving Session

**2 Running the Application**

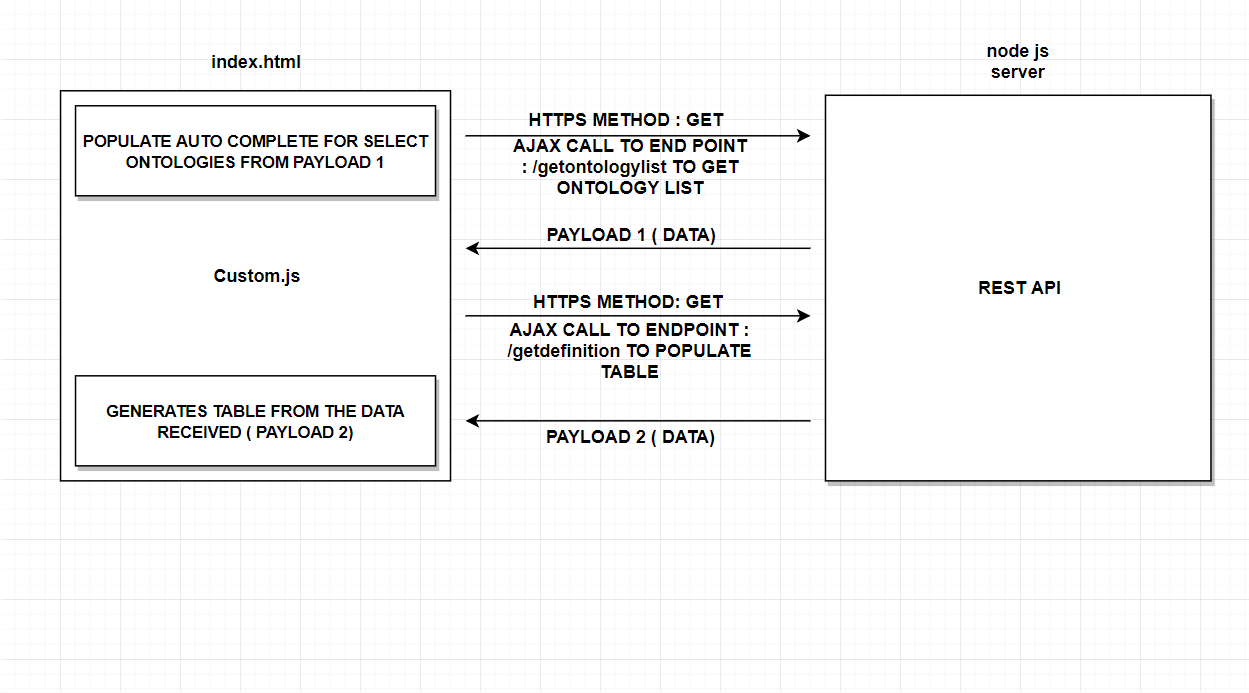
2.1 Setting-up local node JS server

**3 Debugging the Application**

3.1 Debugging the client-side part.

3.2 Debugging the server-side code.

**Architecture of the Application**



The architecture is segregated into Frontend and Backend parts.

**Frontend:**

* Used Bootstrap framework for CSS.
* Jquery for making Ajax calls to API.
* Custom.js script file is run which implements the Ajax call to the API.

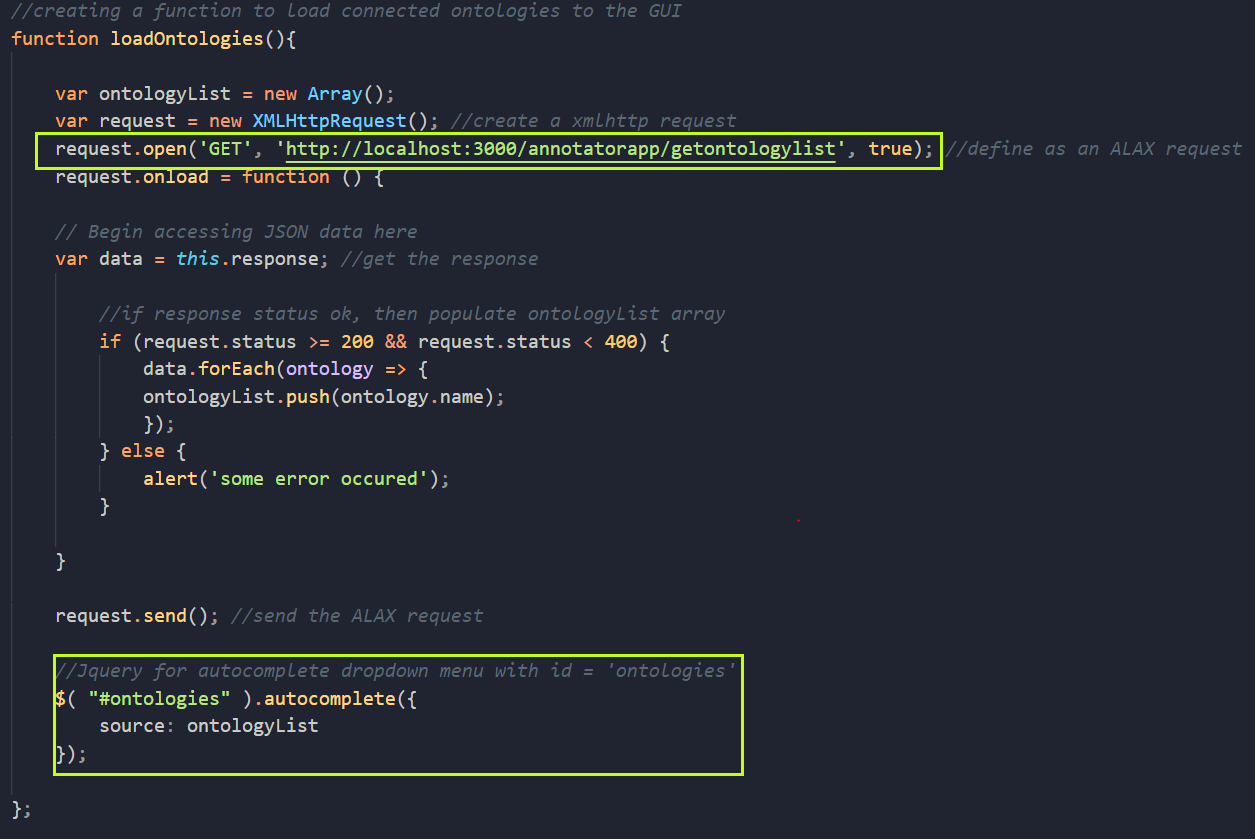
A screenshot of a computer screen

Description generated with very high confidence

**LOADING ONTOLOGIES:**

After entering the words, users must select the ontologies. An Ajax calls is made here to the server, to auto-complete the available ontologies to the user.

**Custom.js file**



**POPULATING THE TABLE:**

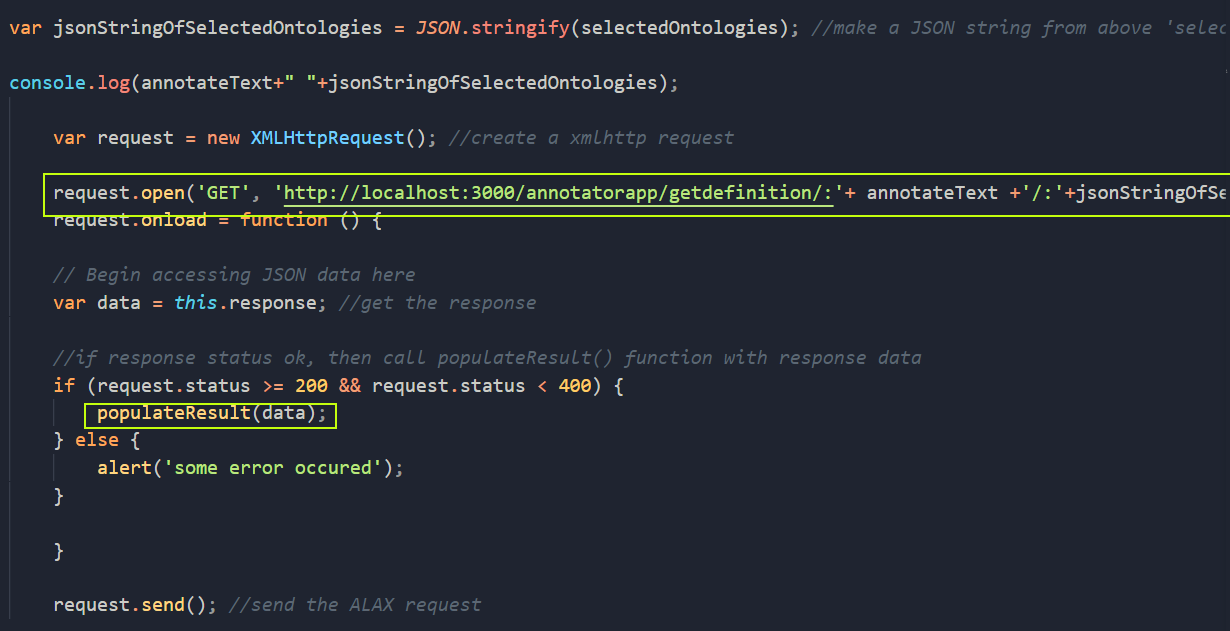
A screenshot of a cell phone

Description generated with very high confidence

For populating the table as shown in the mockup, the ontologies which are selected are taken from DOM.

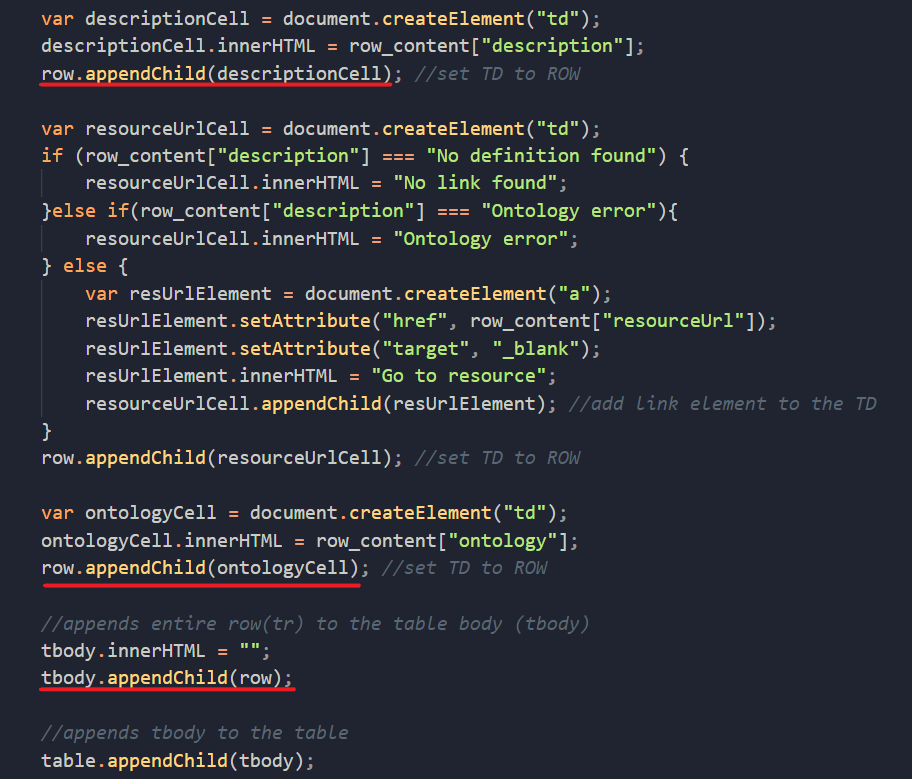


**Custom.js file**



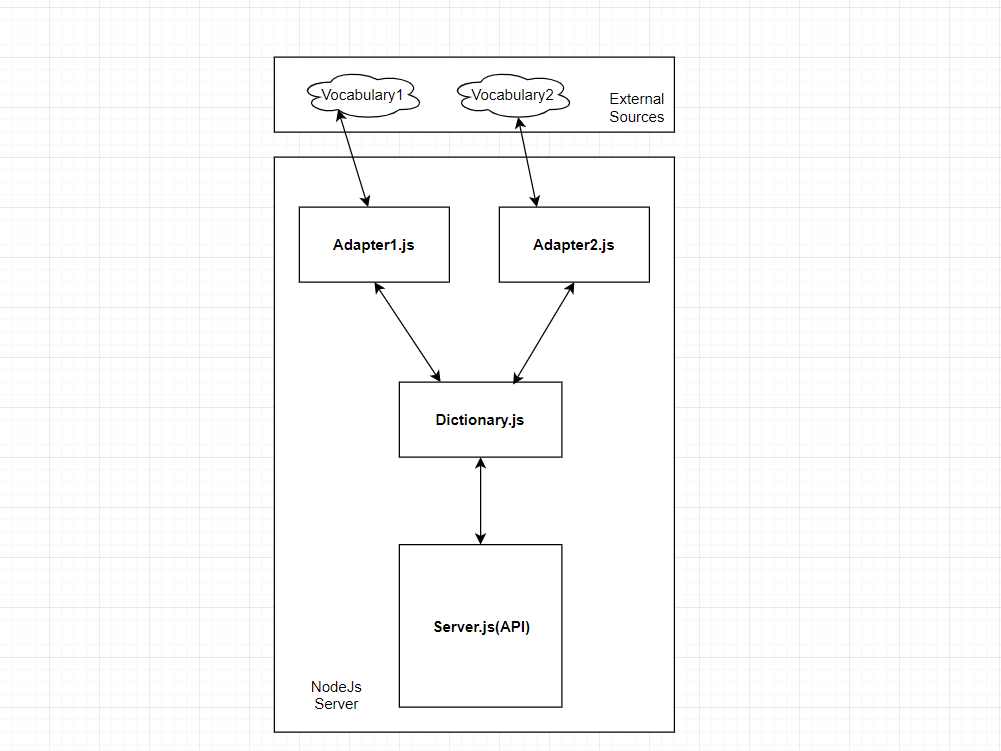
After the payload is received, to inflate the new html elements(table) into the index.html, populateResult() function is invoked.

**populateResult function:**



Inflating new html table into the DOM programmatically.

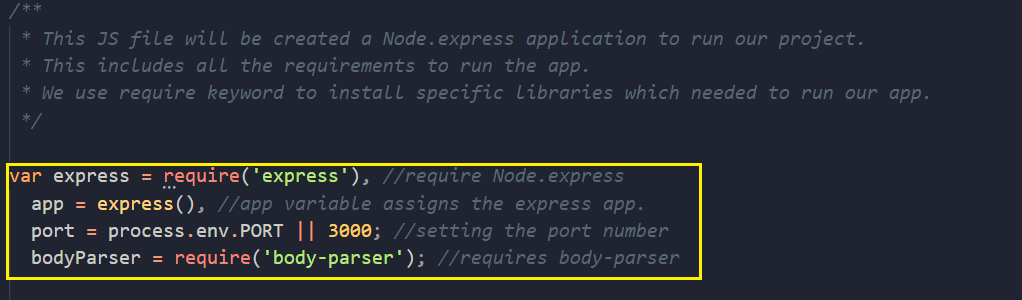
**BACKEND**



* Any route in the node JS server connects to the dictionary.js file which then in turn connects to available adapters to the external API’s.

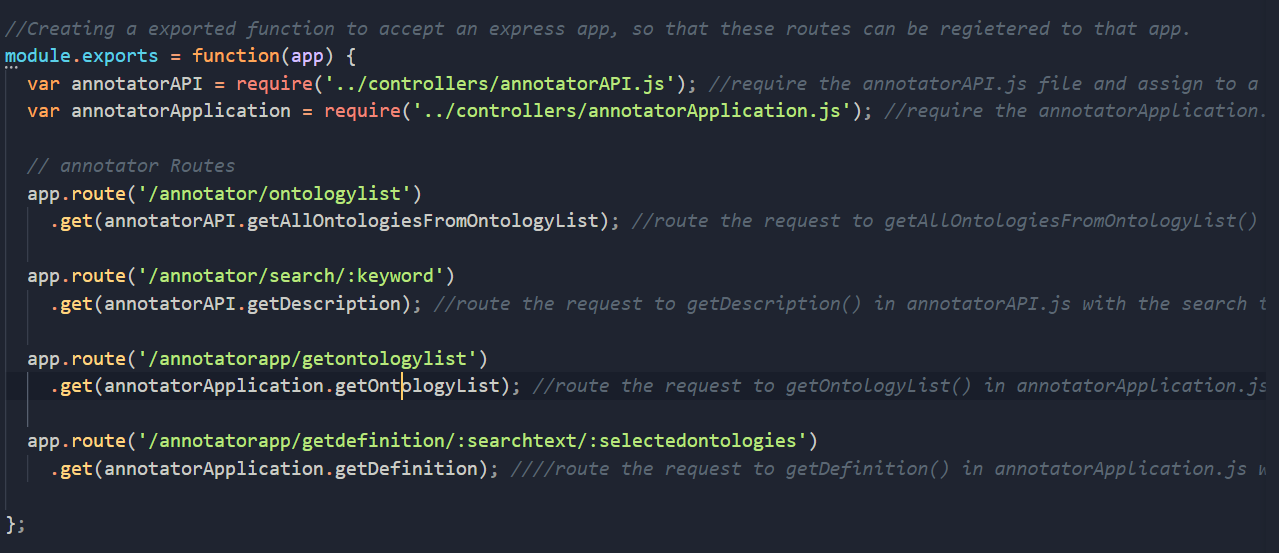
**NodeJS Server:**

* The Node.js framework is mostly used to create server-based applications.
* The framework can easily be used to create web servers which can serve content to users.
* There are a variety of modules such as the "http" and "request" module, which helps in processing server related requests in the web server space.
* The express framework is built on top of the node.js framework and helps in fast-tracking development of server-based applications. Routes are used to divert users to different parts of the web applications based on the request made.



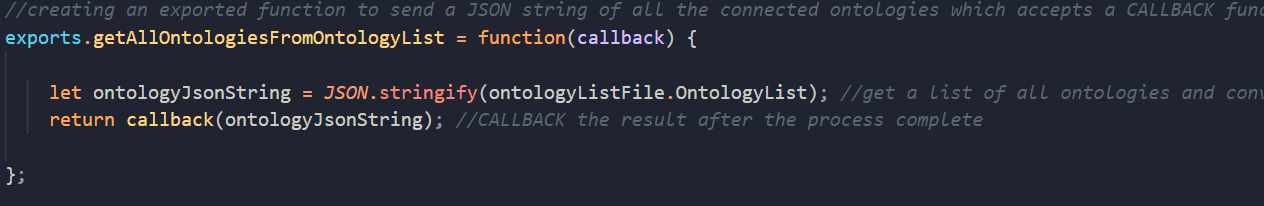
* Setting up express JS in the server.js file, Here the server will run on 3000 port locally.

**API ENDPOINTS:**

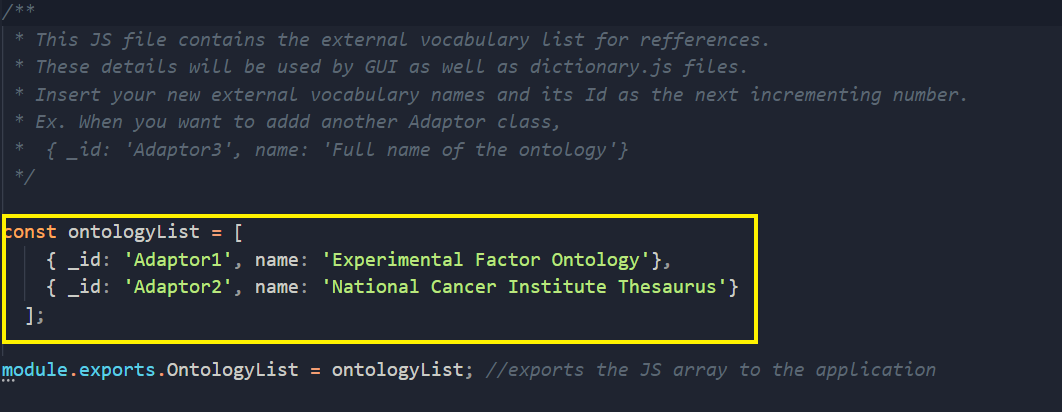


* Front end of the application calls two of the API routes.

**1. route(‘/annotatorapp/getontologylist’):**



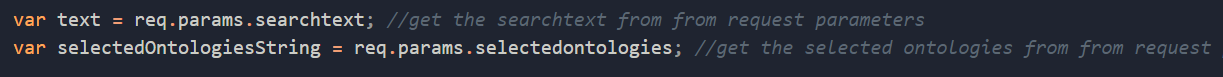
* This end point is invoked to get the list of present ontologies. As mentioned before any route connects to the dictionary.js file.
* Since node JS works asynchronously, a callback is used for response from the endpoint.
* *ontologyListFile* contains the available ontologyList which is passed to the callback function.



* The ontologies that are supported are initialized in the ontologyList.

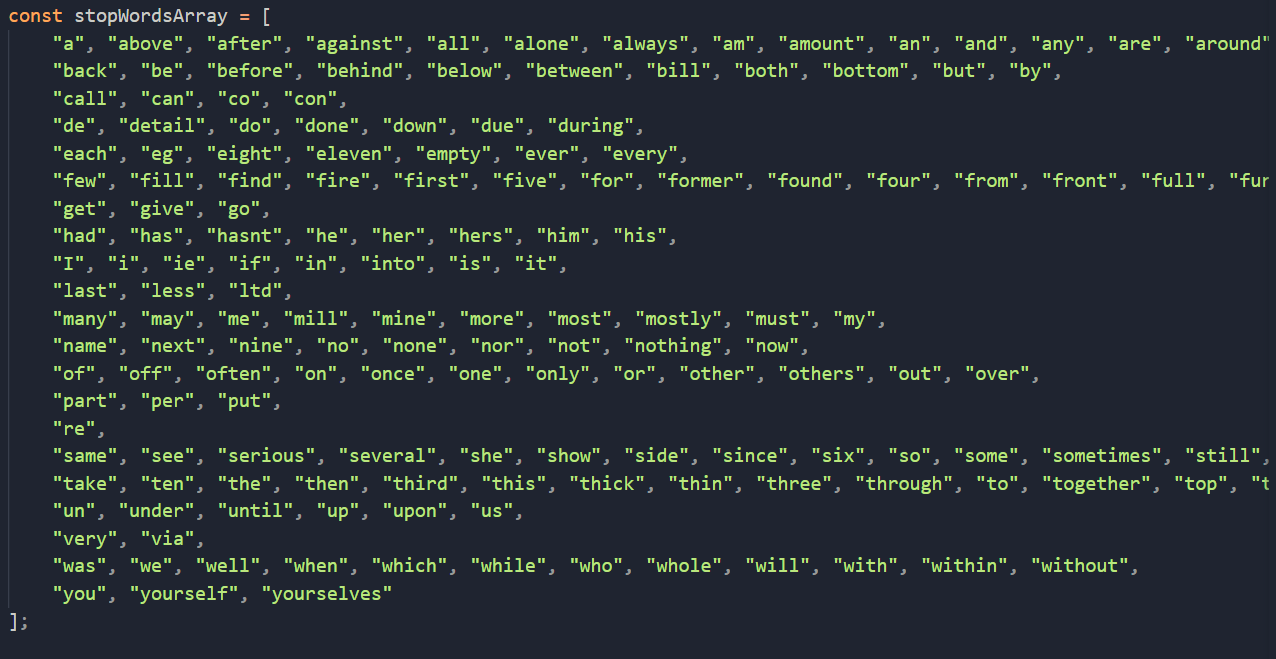
**2. route (‘/annotatorapp/getdefinition/: searchtext/: selectedontologies’):**

* Here **searchtext, selectedontologies** are the request parameters which are passed with the request object.

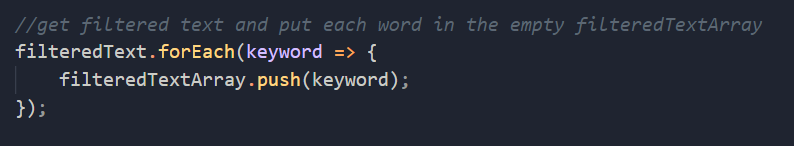


* Before making API calls to the external vocabularies, stop words are filtered out from the search text.
* Stop words don’t carry any specific meaning, hence they can be avoided.

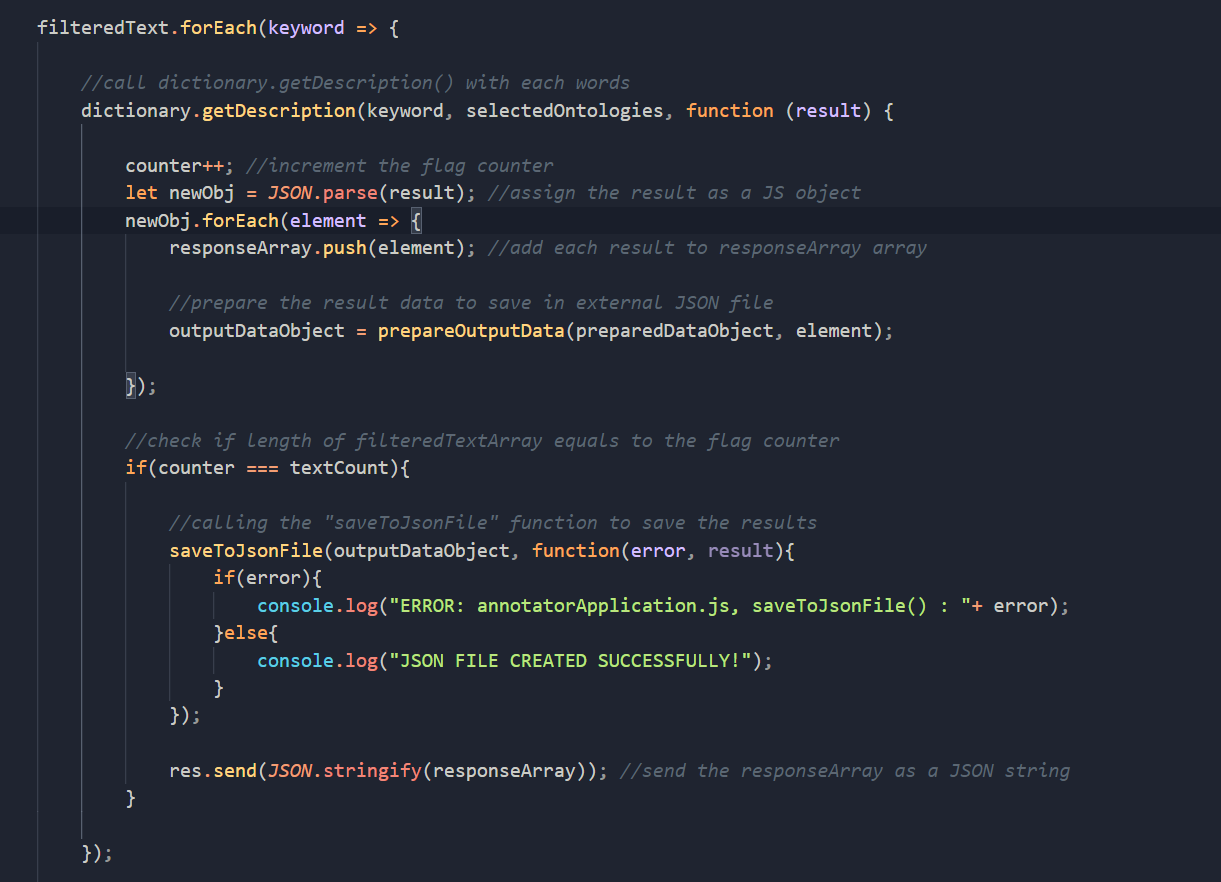
These are the some of the stop words that were used.



* The filtered text along with the selected ontologies is then sent to the dictionary.js file which then calls the external vocabularies.



* After the words are filtered, it is copied to a filteredTextArray.



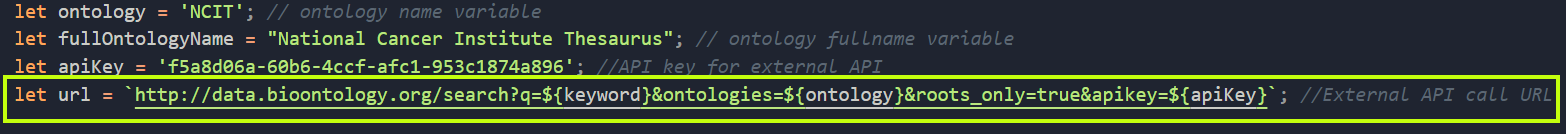
* The **filteredTextArray** is iterated through and **dictionary.getDescription()** method is invoked and response is added to the **outputDataObject.**

**CONNECTING TO EXTERNAL API:**

* The external API has protected routes which can be only accessed if the user is authenticated using the API key.
* Whenever a request is made the external vocabulary, API key needs to be appended with the URL parameter.

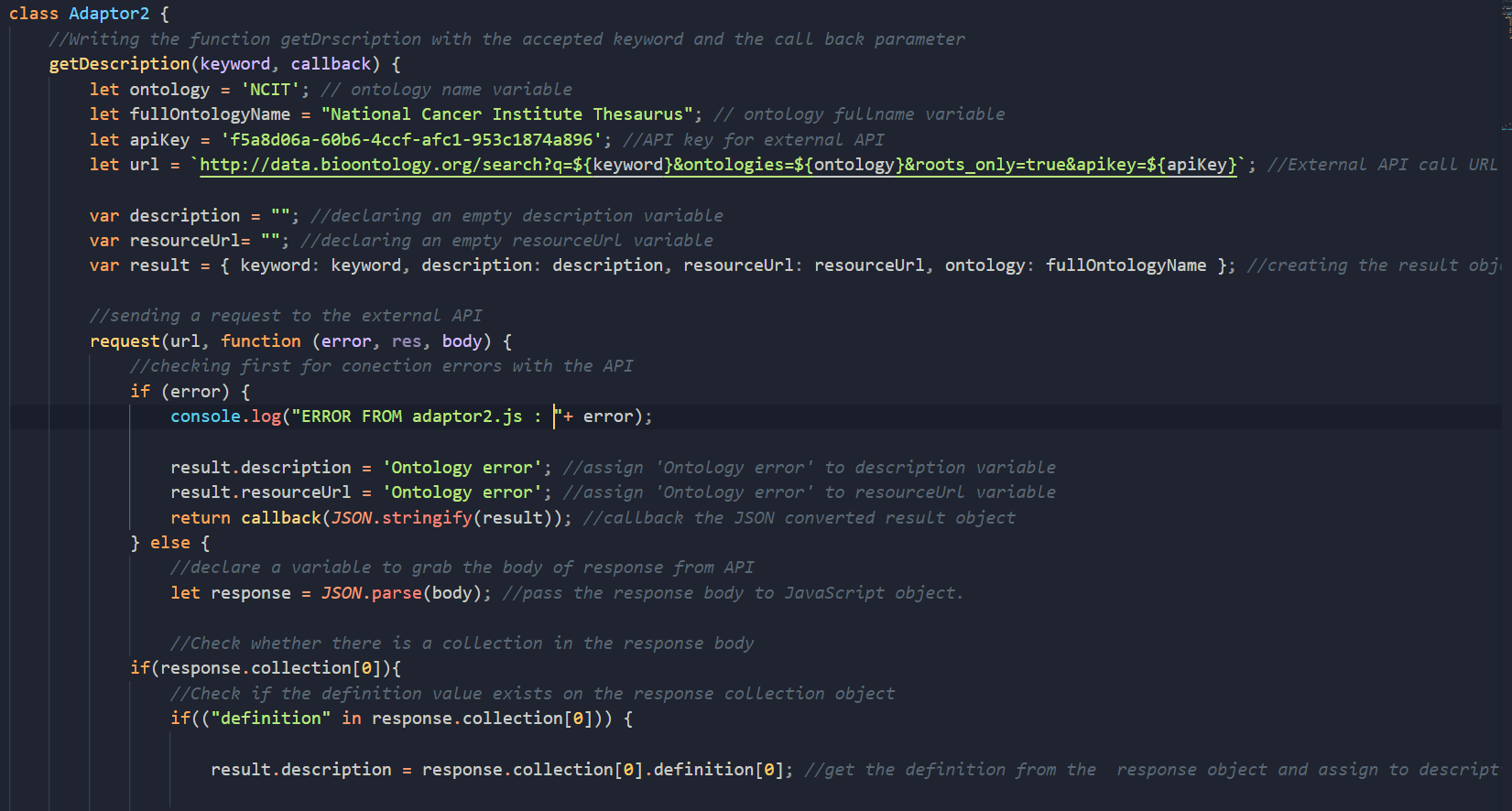


**SETTING-UP THE URL:**



* The URL here is ES6’s template literal feature.
* The keyword, ontology, api key vary with each adapter.

**ADAPTER CLASS:**

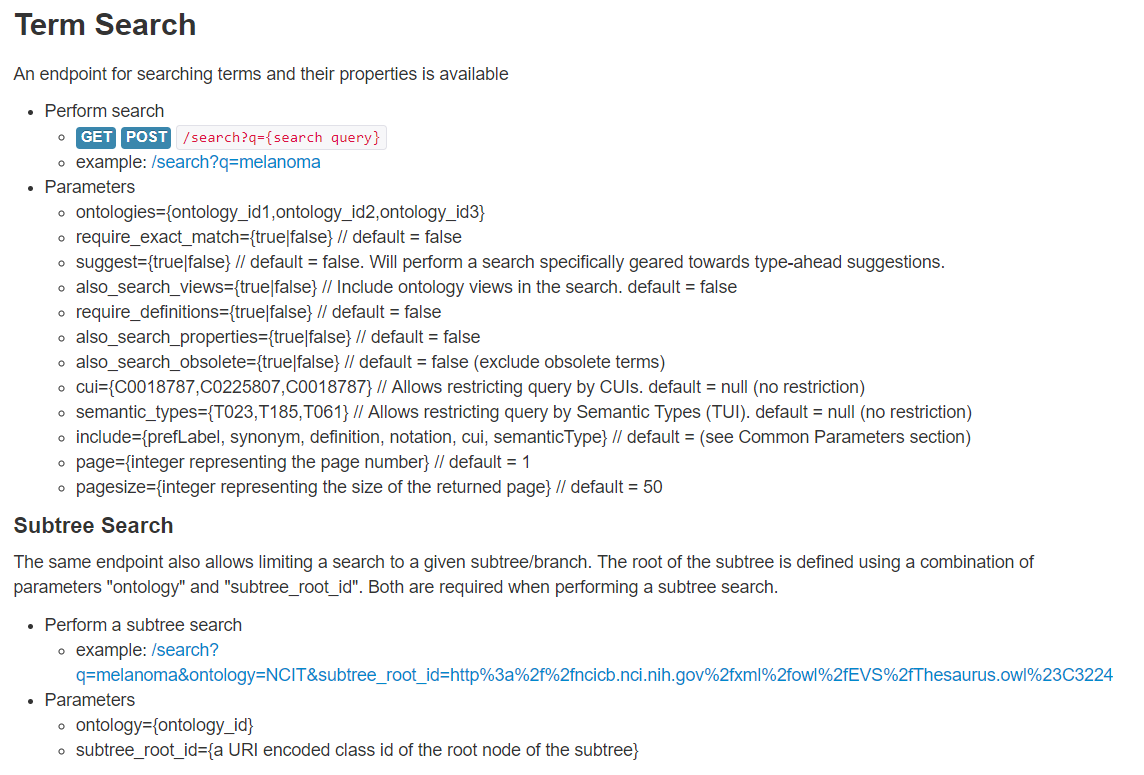


* All the Adapters follow the same abstract i.e **Setting-up the URL,** make a fetch request to the external vocabulary and attach a callback to it.
* Then get a payload or error and send the appropriate response via the response object.
* The call back function of the request method takes 3 parameters error, res, body.
* **Error** parameters takes any errors occurred when request is made.
* The actual payload of the request is in the body parameter.

**EXTERNAL VOCABULARIES:**

* Bio Portal’s annotation service is a useful precedent, with both a human UI and an API.
* However, the Bio Portal service is limited to vocabularies loaded locally and formalized as OWL Technologies.

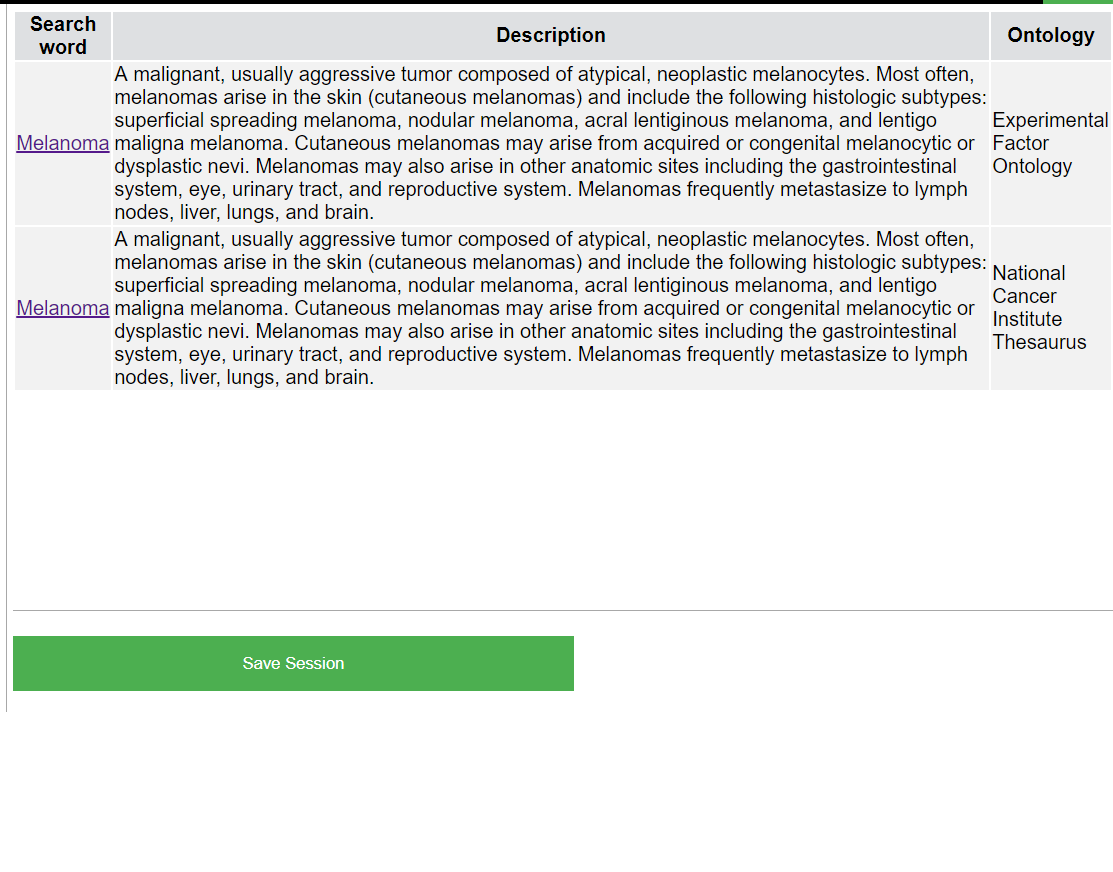
This the end point accessed by the adapter files.



* This API limits the requests to 15 per minute.
* There are several vocabulary services are available which provide definitions for technical terminology, organized into 'controlled-vocabularies', published as ‘linked data’ with both human user interfaces, and semi-standardized APIs:

|  |  |
| --- | --- |
| Research Vocabularies Australia | https://vocabs.ands.org.au/ |
| CSIRO's Linked Data Registry | http://registry.it.csiro.au/ |
| AusGIN Geoscience Vocabularies | <http://www.geoscience.gov.au/data-standards> |
| NERC Vocabulary Service | http://vocab.nerc.ac.uk/collection/ |
| Bioportal | https://bioportal.bioontology.org/ |
| Library of Congress Subject Headings | <https://id.loc.gov/authorities/subjects.html> |
| DBpedia | https://wiki.dbpedia.org/ |
|  |  |

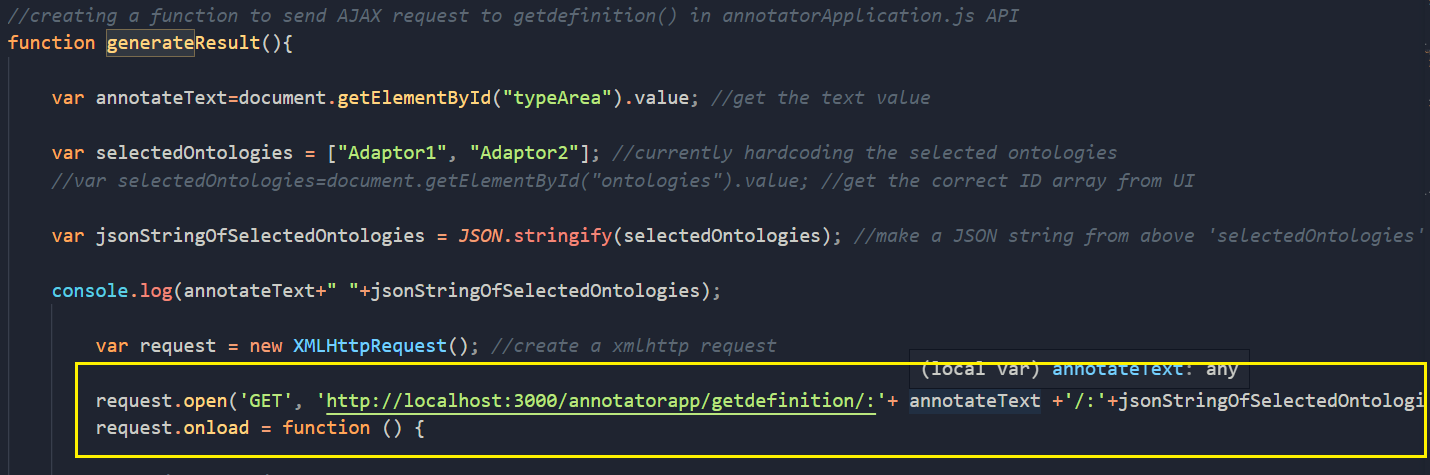
**SAVING SESSION:**



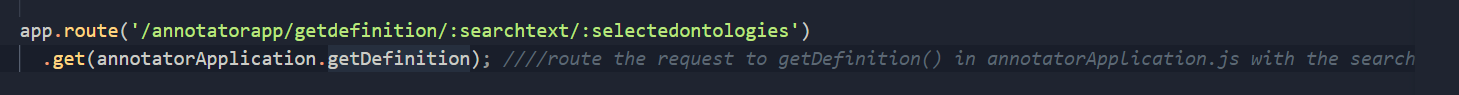
* When you search for a word, after the results are displayed, the web application provides an option to save the session of the user.
* When user clicks on the save session, an event listener for the same is called i.e **generateResult()**



* The Custom.js file contains generateResult() method which gets on-click event.
* The generateResult() function in return makes an api call to our backend endpoint.



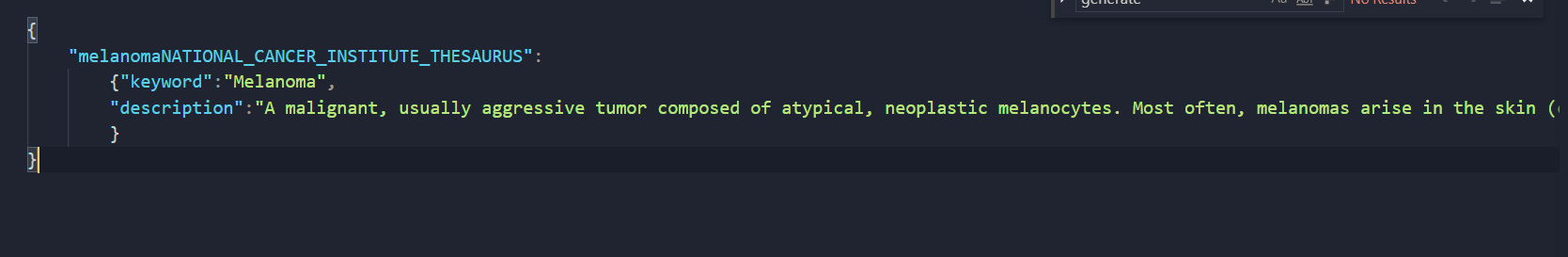
* The api route /getdefinition/:annotator/:selectedOntologies gets called in this function.



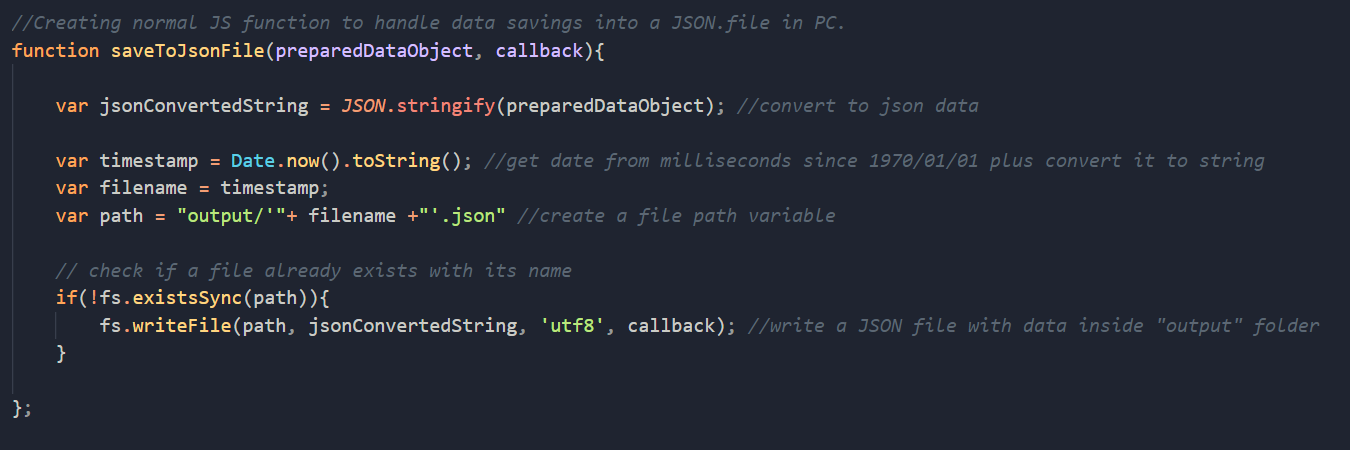
* The getDefinition controller deals with the logic of handle the get route of /annotatorapp/getdefinition/:searchtext/:selectedontologies.



* The getDefinition function along with making calls to external vocabularies, also creates a json file and stores the relevant content inside it.
* Below, is the json file created in the server side, when a new session is stored.



* This function takes care of save the file to json and then exporting it.



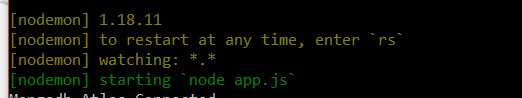
**RUNNING THE APPLICATION**

**Setting-up local node JS server:**

* By default, the server runs on localhost port 3000.
* **Download** the file and open command line inside the directory.



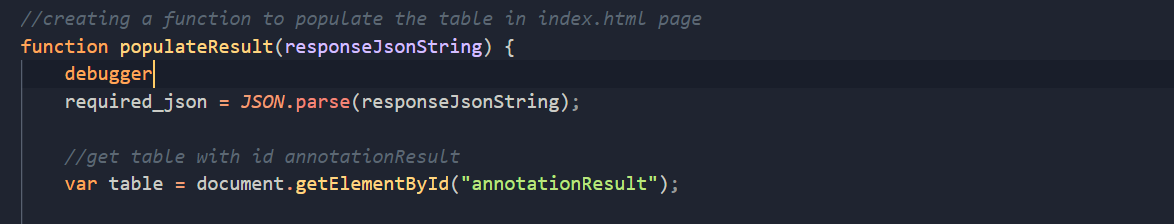
* This will install all the dependencies specified inside the json file.
* Now type “npm start”
* We are using nodemon module which will automatically restart the node JS server without starting and stopping the server manually.



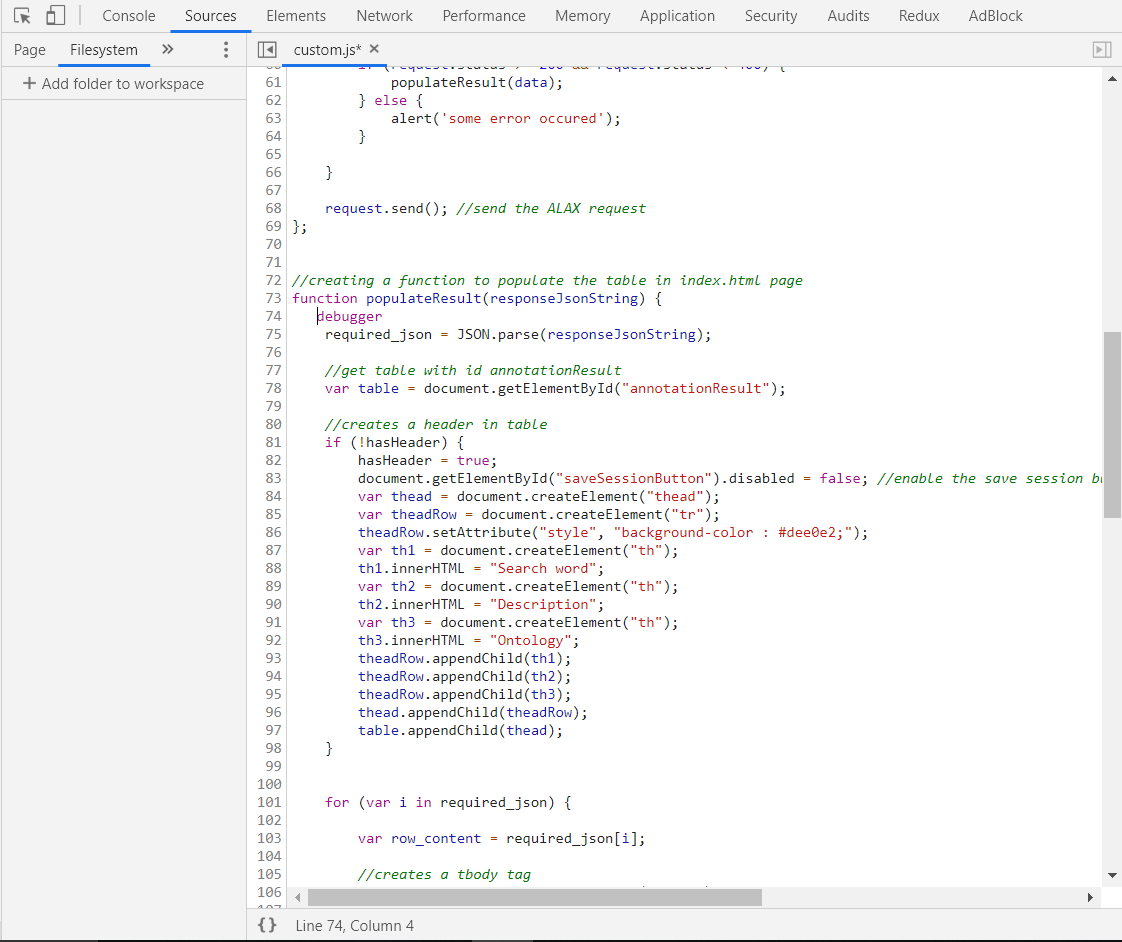
**DEBUGGING:**

**DEBUGGING THE CLIENT-SIDE CODE**

* Instead of debugging the client side of the code using console.log messages, it is more desirable to use a debugger.
* The application makes use of chrome’s developer tools to debug the application.

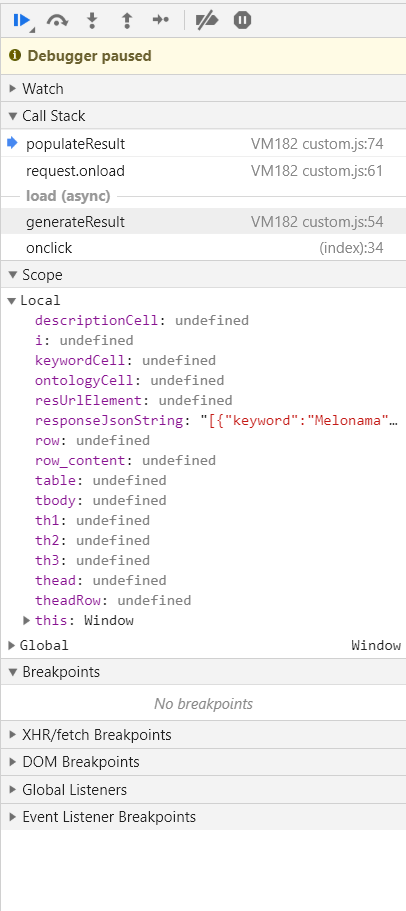


* The application stops at this debugger point, and we can set breakpoints to perform various operations likes step over function call etc.



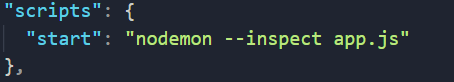
* As shown above, using the chrome developer tools debugging the code.
* Here we can perform various operations likes step over next function call,

Step into next function call, step out of current function.



**DEBUGGING SERVER-SIDE CODE:**

* Debugging server-side code is not that straight forward, we have to make use of the nodemon --inspect directive and chrome tools.
* Here first we set the --inspect directive in the script file.



* Open chrome://inspect/#devices
* We can find the localhost in the remote target, once it is added. We can debug the server-side code as well.