PROJECT VALIDATOR

Installation Instructions

To run this application you need to have nodejs installed and have npm or its equivalent.

First clone the repository using the git clone command: -

git clone https://github.com/Kuber144/Project_Validator.git

After cloning open a terminal inside the backend folder (if you wish to run the project on localhost then you need to replace the url in frontend for api calls with localhost).
After opening the terminal run the following commands in the same directory as index.js to start the backend: -

npm install
node index.js

To run the frontend, open another terminal in the Frontend folder this time and run the following commands: -

npm install --force
npm start

We have to use npm install --force in this case because of dependency issue with codemirror.

About the project

In this application, I've developed a convenient way for users to test HTML, CSS, and JavaScript code directly within the browser. The primary functionality revolves around allowing users to input code snippets, create tests to evaluate their code, and determine whether their code meets the specified criteria.

Frontend Development:

I've built the frontend using React, providing users with a user-friendly interface to interact with the application. On the homepage, users can input their code and preview its output in real-time. They also have access to navigation buttons to create new tests and manage existing ones. Additionally, users can open a sidebar to select and run specific tests, visually indicating whether the test passes or fails. If a test fails, users can click on the failed test to view detailed information about why it failed.

In the create tests section, the user has to enter the test name and after this add a test and add the description of the test. Then they select the type of test.

If the user selects the html type then they have to fill what type of element they want to check. For example if they want to check the id of the welcome-section then they fill the comparison type as id and value as welcome-section. Then they fill in whether they want to check if it exists, equals something or some other condition like just after another element or just before and so on. Then they enter the details for the element to compare to if applicable.

Similarly for CSS first the user enters the selector type, then the value they want to select. Then the property they want to test and whether they want to test if the value will be equal to or less than or more than, etc other value that the user enters.

For javascript it currently only supports if the user declares a function using the function keyword. The user enters the function name and can choose whether they want to check if the function exists or not. Otherwise they can enter the number of arguments, the arguments and the expected output to check if the function satisfies the conditions given.

All of this value is taken and then sent to the backend where it is stored in a mongodb database.

Backend Development:

When testing, individual elements (html, css and js) and the whole code formatted is sent to the backend along with that the title of the test is sent. A json array is maintained to keep track of each test and then send back as the response.

For processing the test, first the test with the matching test title is fetched then it is iterated over. For each test, the test type is checked if it is either html, css or js.
If it is html then we use a library called cheerio to efficiently iterate over the html code. We send the cheerio element and the test to a function where it is computed. First using the htmlOption, the element is fetched from the code which matches the type and value. Example: htmlOption is id and value is ‘welcome-section’ then that will be selected. Then it is checked if it exists. If it does not, an error is thrown and sent back as response that element does not exist. Moving further on we then fetch the element that we have to check with in a similar manner. Then we take the htmlCondition and check what the condition was. The test can check if the values are equal, contains a certain value, is the first element in the html doc, is the last element in the html doc, simply exists, is just after or just before any other element. More checks can be included in a similar way. Additionally if the user cannot select a html attribute like id or classname, they can select custom and enter their own value and it will be checked directly.

Thus the response is generated if it passes and fails with corresponding reason and sent back to append into the response array.

For css testing, we use JSDOM as it directly provides functionality to select css elements. We send the whole document containing the html, css and javascript code together and create a parse using JSDOM to parse it. Then we use the selectorType to again select the element with the value and type from the given doc. Then we check if it exists or not. After that, we get the cssProperty using the getComputedStyle function of JSDOM. In this case JSDOM is better than cheerio as for css elements we need to create a dummy window element which can be easily created by JSDOM to actually compute the css value for each element. Then similarly as html testing we check if the obtained value is equal to less than, above or between certain value(s). And then we return the result with the corresponding result.

For javascript testing we send only the js code and the test. Then we use a regex to find wherever the word function with the functionName is contained. If it exists and the user asked to only check if the function exists then we return the appropriate response. Otherwise if the function exists, we get the index of “(” and “)” after where the function is declared to find where the arguments are declared. Then we use tokenization using the “,” to split into different arguments. This is important because we later recreate the function and need to pass the same arguments. We then check whether the number of arguments is equal to the number the creator of the test specified; if not then an appropriate response is sent. After this we take the index of “{“ and “}” and take the content contained within this index to get the actual content of the function. Now that we have the arguments and the function we recreate the function and then we create a new function and pass the array of arguments that was in the test and execute the function and check if the output received from it is equal to the expected output in the test and return the appropriate answer.

Now when all test cases are tested, the response is sent back and then accordingly displayed for each test whether it passed or failed.

Conclusion:

Project Validator offers a robust solution for code testing, enabling developers to ensure the quality and functionality of their code effortlessly. By providing a seamless integration of frontend and backend components, along with intuitive test creation and processing capabilities, the application empowers users to validate their code with confidence.

Future Enhancements:

Integration of additional testing functionalities for comprehensive code validation.
Implementation of user authentication and access control features to enhance security.
Integration of user feedback mechanisms to continually improve the user experience.