Proposal for Building a Web-Based Code Editor

Introduction:

This proposal outlines the plan for developing a web-based code editor using the Monaco Editor and Flask. The editor will support multiple programming languages (C/C++, Java, Python, SQL) and provide functionalities such as code writing, syntax highlighting, code execution, and error reporting. The document will cover the tech stack, libraries, functions, and official documentation links.

Tech Stack:

1. Frontend:
   - HTML, CSS
   - JavaScript
   - Monaco Editor
   - Bootstrap (for UI components)

2. Backend:
   - Python
   - Flask (Web Framework)

- subprocess (to execute code)

3. Server:

- Flask development server for local testing

4. Version Control:

- Git

Approach

1. Frontend Development:

- Design the user interface using HTML, CSS, and Bootstrap.

- Integrate Monaco Editor for code writing and syntax highlighting.

- Implement functions to save and execute code.

- Display output or errors in a designated area.

2. Backend Development:

- Set up a Flask server to handle requests from the frontend.

- Implement routes to save code, run code, and return the output or errors.

- Use the subprocess library to compile and run code securely.

- Clean up temporary files after execution.

3. Integration:

- Connect the frontend and backend through RESTful APIs.

- Ensure seamless communication between the client and server.

4. Testing and Deployment:

- Test the application locally for functionality and performance.

Functions and Libraries:

Frontend Functions:

1. Monaco Editor:

- Description: Monaco is the code editor that powers VS Code, providing syntax highlighting, autocompletion, and other features.
- Libraries Used: Monaco Editor
- Official Documentation:

  https://microsoft.github.io/monaco-editor/

- Explanation:

  This script initializes the Monaco Editor within a `div` element with the ID `editor`. The editor is set

up to use the Python language for syntax highlighting.

```javascript
require(['vs/editor/editor.main'], function() {
    window.editor = monaco.editor.create(document.getElementById('editor'), {
        value: '',
        language: 'python'
    });
});
```

2. Save Code:

- Description: Sends the code to the backend to be saved.
- Libraries Used: Fetch API
- Official Documentation: https://developer.mozilla.org/enUS/docs/Web/API/Fetch_API

- Explanation:

  This function gets the current code from the Monaco Editor and sends it to the backend

using a POST request. The selected programming language is also sent along with the code.

javascript

```
function saveCode() {
    const code = window.editor.getValue();
    const language = document.getElementById('language').value;
    fetch('/save', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ code, language })
    }).then(response => response.json()).then(data => {
        console.log(data.message);
    });
}
```

3. Run Code:

- Description: Sends the code to the backend to be executed and displays the output.

- Libraries Used: Fetch API
- Official Documentation: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

- Explanation:

  This function gets the current code from the Monaco Editor and sends it to the backend for execution using a POST request. The backend response, which contains the execution output or errors, is then displayed in the `output` element.

javascript

```
function runCode() {
  const code = window.editor.getValue();
  const language = document.getElementById('language').value;
  fetch('/run', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ code, language })
  }).then(response => response.json()).then(data => {
```

```
        document.getElementById('output').innerText =
data.output;
    });
}
```

Backend Functions

1. Flask:

- Description: Flask is a lightweight web application framework in Python.
- Libraries Used: Flask
- Official Documentation:

  https://flask.palletsprojects.com/

- Explanation:

  This code initializes a Flask application. Flask is used to handle HTTP requests and provide routes for saving and running code.

```
python
from flask import Flask, render_template, request, jsonify
import subprocess
import os

app = Flask(__name__)
```

2. Save Code Route:

- Description: Saves the code received from the frontend to a file.
- Libraries Used: Flask
- Official Documentation: https://flask.palletsprojects.com/

- Explanation:

  This route handles saving the code to a file. It receives the code and language from the frontend, determines the appropriate file extension, and writes the code to a file.

Python:

```
@app.route('/save', methods=['POST'])
def save_code():
    data = request.get_json()
    language = data['language']
    code = data['code']
    filename = f'program.{language_extension(language)}'

    with open(filename, 'w') as f:
        f.write(code)
```

```
return jsonify({'message': 'Code saved successfully!'})
```

3. Run Code Route:

- Description: Compiles and runs the code received from the frontend and returns the output or errors.
- Libraries Used: Flask, subprocess, os
- Official Documentation: https://flask.palletsprojects.com/ , https://docs.python.org/3/library/subprocess.html , https://docs.python.org/3/library/os.html

- Explanation:

  This route handles the compilation and execution of the code. Depending on the language, it uses different commands to compile and run the code. The output or errors are captured and returned to the frontend.

Python:

```
@app.route('/run', methods=['POST'])
def run_code():
    data = request.get_json()
    language = data['language']
```

```
    code = data['code']
    filename = f'program.{language_extension(language)}'

    with open(filename, 'w') as f:
        f.write(code)

    output = ''
    try:
        if language == 'c':
            executable = filename.split('.')[0]
            compile_command = f"gcc {filename} -o {executable}"
            compile_process = subprocess.run(compile_command.split(), stdout=subprocess.PIPE, stderr=subprocess.PIPE)

            if compile_process.returncode != 0:
                return jsonify({'output': compile_process.stderr.decode()})

            run_process = subprocess.run(f'./{executable}', stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

```
            output = run_process.stdout.decode() +
run_process.stderr.decode()
        elif language == 'cpp':
            executable = filename.split('.')[0]
            compile_command = f"g++ {filename} -o {executable}"
            compile_process =
subprocess.run(compile_command.split(), stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

            if compile_process.returncode != 0:
                return jsonify({'output':
compile_process.stderr.decode()})

            run_process = subprocess.run(f'./{executable}',
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
            output = run_process.stdout.decode() +
run_process.stderr.decode()
        elif language == 'java':
            compile_process = subprocess.run(f"javac
{filename}".split(), stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

            if compile_process.returncode != 0:
```

```
                return jsonify({'output':
compile_process.stderr.decode()})


            run_process = subprocess.run(f"java
{filename.split('.')[0]}".split(), stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
            output = run_process.stdout.decode() +
run_process.stderr.decode()
        elif language == 'python':
            run_process = subprocess.run(f"python {filename}".split(),
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
            output = run_process.stdout.decode() +
run_process.stderr.decode()
        elif language == 'sql':
            output = 'SQL execution not supported in this example'
        else:
            output = 'Unsupported language'
    finally:
        if os.path.exists(filename):
            os.remove(filename)
        if language in ['c', 'cpp', 'java'] and
os.path.exists(filename.split('.')[0]):
```

```
        os.remove(filename.split('.')[0])

    return jsonify({'output': output})
```

4. Helper Function for Language Extension:

- Description: Maps the programming language to its file extension.
- Libraries Used: None
- Official Documentation: None

- Explanation:

  This helper function returns the appropriate file extension for the given programming language.

Python:

```
def language_extension(language):
    return {

'c': 'c',
        'cpp': 'cpp',
```

```
        'java': 'java',
        'python': 'py',
        'sql': 'sql'
    }.get(language, 'txt')
```

Documentation Links:

- Monaco Editor Documentation: https://microsoft.github.io/monaco-editor/
- Flask Documentation: https://flask.palletsprojects.com/
- Fetch API Documentation: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- subprocess Documentation: https://docs.python.org/3/library/subprocess.html
- OS Documentation: https://docs.python.org/3/library/os.html

Conclusion:

This proposal outlines the plan to build a web-based code editor using Monaco Editor and Flask. The editor will support multiple programming languages and provide functionalities like code writing, syntax highlighting, code execution, and error reporting. By leveraging the outlined tech stack and libraries, the development process will be streamlined, resulting in a robust and user-friendly code editor.