**DEPARTMENT OF Mathematics and Astronomy**

A Technical Report on

Online Code Execution and Sharing Platform

**Submitted to:**

Dr. Puneet Misra

**Submitted by:**

Prakash Chaurasia

**Roll number:** 2110011016050

**Enrollment No:** 21101073

B.Sc. Semester – VI



CERTIFICATE

This is to certify that Prakash Chaurasia (2110011016050) has carried out the major project work whose technical report is presented in this study entitled “A Technical Report on an Online Code Execution and Sharing Platform” for the award of semester examinations in Computer Science at University of Lucknow under the supervision of Dr. Puneet Misra.

Dr. Puneet Misra

Project Guide

Head of Department,

Department of Computer Science

University of Lucknow,

Lucknow

**Dr. Puneet Misra**

**Project Guide**

**Department of Computer Science**

**University of Lucknow**

**Lucknow**

DECLARATION

I, *Prakash Chaurasia*, certify that the work embodied in this Technical Report is my original work carried out by me under the supervision of *Dr. Puneet Misra*, Associate Professor, Department of Computer Science, University of Lucknow.

This project is the result of my own independent work and research, and I have properly cited and referenced all the sources that I have consulted, whether they are books, articles, reports, lecture notes, or any other type of document, electronic or personal communication. I also declare that this project has not been submitted for evaluation in any other academic context. I confirm that I have disclosed and resolved any potential conflicts of interest that I may have.

Prakash Chaurasia

B.Sc., Semester VI

**Roll no:** 2110011016050

|  |
| --- |
|  |
| A Technical Report on an Online Code Execution and Sharing Platform, Snippet |

**Under the Guidance of** Dr. Puneet Misra

Associate Professor,

Department of Computer Science,

University of Lucknow

**By** Prakash Chaurasia

2110011016050

B.Sc. IIIrd year

Department of Computer Science,

University of Lucknow

|  |
| --- |
| CONTENTS  [1. INTRODUCTION 3](#_Toc167182350)  [2. Project Overview 4](#_Toc167182351)  [2.1. Motivation 4](#_Toc167182352)  [2.2. Potential Benefits 5](#_Toc167182353)  [3. Technology Stack 5](#_Toc167182354)  [3.1. Core Stack 6](#_Toc167182355)  [3.2. Other Dependencies 7](#_Toc167182356)  [3.2.1. Code Execution API: 7](#_Toc167182357)  [3.2.2. Access Control and Authentication API 7](#_Toc167182358)  [3.2.3. User Interface 7](#_Toc167182359)  [4. Key features and their implementations 8](#_Toc167182360)  [4.1. Client – Compiler Server Communication 8](#_Toc167182361)  [4.1.1. Socket.io 8](#_Toc167182362)  [4.2. Code Execution 10](#_Toc167182363)  [4.2.1. Code File Creation 10](#_Toc167182364)  [4.2.2. Code File Deletion 12](#_Toc167182365)  [4.2.3. Execution Command Mapping 14](#_Toc167182366)  [4.2.4. Code Compilation and Execution 17](#_Toc167182367)  [4.3. User Authentication 21](#_Toc167182368)  [4.3.1. User Registration 21](#_Toc167182369)  [4.3.2. User Login 23](#_Toc167182370)  [4.3.3. Resetting User Password 24](#_Toc167182371)  [4.3.4. Change Email 27](#_Toc167182372)  [4.4. Uploading Snippets 29](#_Toc167182373)  [4.5. Updating Snippet 30](#_Toc167182374)  [4.6. Deleting Snippet 31](#_Toc167182375)  [4.7. Retrieving Snippet 32](#_Toc167182376)  [4.7.1. Retrieving List of User Snippets 32](#_Toc167182377)  [4.7.2. Retrieving specific Snippet 34](#_Toc167182378)  [4.7.3. Searching for a specific Snippet 34](#_Toc167182379)  [5. USer Interface Design And Implementation 35](#_Toc167182380)  [5.1. Navigation Links 36](#_Toc167182381)  [5.1.1. Design 36](#_Toc167182382)  [5.1.2. Features 36](#_Toc167182383)  [5.1.3. Implementation 37](#_Toc167182384)  [5.2. Code Editor 40](#_Toc167182385)  [5.2.1. Design 40](#_Toc167182386)  [5.2.2. Implementation 41](#_Toc167182387)  [5.3. Terminal 43](#_Toc167182388)  [5.3.1. Design 43](#_Toc167182389)  [5.3.2. Implementation 44](#_Toc167182390)  [6. Conclusion 51](#_Toc167182391)  [7. Bibliography 53](#_Toc167182392) |

|  |
| --- |
|  |

Technical Report on an Online Code Execution and Sharing Platform, Snippet

# INTRODUCTION

Snippet is a innovative web-based coding platform which makes coding simple by providing an intuitive environment in which one can easily write, run and distribute code snippets. The goal of the project is to create a collaborative community where beginners and professionals can learn from each other as well as improve their programming skills. Not only does Snippet have a real-time code editor that supports various programming languages such as C, C++, C#, GoLang, Java, Python and Rust but also has a web environment for HTML, CSS and JS execution. Furthermore, it allows users to share any useful codes with like-minded individuals via its social coding platform.

The term *Snippet* in the context of programming refers to a small, reusable piece of source code or text. This term is originated from the general English word *snippet*, which means a *small piece snipped off*, dating back to the 17th century (1). The name *Snippet* reflects the small and functional nature of these code pieces, and their use is integral to efficient coding practices. For developers seeking assistance, providing a minimal reproducible snippet is a best practice that enables clear communication of issues and fosters collaborative problem-solving within the coding community. The ability to create and share these snippets on an online platform not only streamlines the development process but also builds a repository of knowledge that can benefit the wider developer ecosystem. In programming, snippets serve as concise, operative units that can be easily incorporated into larger code modules. They are particularly useful for repetitive tasks, where writing the same code multiple times would be inefficient. By using snippets, developers can save time and reduce errors by reusing code that has been proven to work.

The value of Snippet lies in its capacity to democratize access to computer programming education and practice. This aims at learning and working together among developers all over the world by making them use one single online tool. As a result, this could lead to an inclusive technology community where ideas and solutions are freely exchanged thus propelling innovation forward. Moreover, Snippet’s focus on collaboration and sharing can reduce the gap between developers of different abilities thereby allowing for mentorship within the community.

# Project Overview

Snippet is envisioned as an online coding platform designed to cater to the needs of developers across various skill levels, from beginners to seasoned professionals. The platform's scope extends beyond mere code execution; it aims to foster a collaborative environment where users can write, execute, and share their code snippets with a community of like-minded individuals. This communal aspect is poised to encourage peer learning and collective problem-solving, making it an invaluable resource for educational institutions, coding boot camps, and self-learners.

## Motivation

The motivation behind Snippet is rooted in the recognition of the challenges faced by developers in sharing and testing code efficiently. Traditional methods often involve cumbersome processes that hinder productivity and collaboration. Snippet addresses these issues by providing a streamlined, user-friendly interface that simplifies the coding experience. Its integrated tools and features are designed to facilitate real-time feedback and iteration, which are crucial for rapid development cycles.

Snippet was conceived with the vision of building a collaborative environment for developers. The platform's primary motivation is to streamline the process of seeking and offering help within the developer community. By enabling users to share minimal, reproducible code snippets, it addresses a common challenge faced by new developers: finding assistance from more experienced peers who may have previously encountered and resolved similar issues.

Moreover, Snippet alleviates the need for newcomers to install various compilers by providing a unified, browser-based interface where code can be written, executed, and shared seamlessly. This not only simplifies the initial learning curve but also shields users from the complexities associated with setting up a development environment.

On a personal note, I harbored a desire to share my programming achievements with others. Recognizing that sharing executable code with non-technical individuals can be daunting owing to the intricacies of running code in different environments—Snippet offers a solution. It allows developers to share their work in a manner that is accessible and executable, regardless of the recipient's technical expertise. This feature empowers even those with limited coding knowledge to engage with and appreciate the functionality of shared code, bridging the gap between developers and the wider community.

## Potential Benefits

The potential benefits of Snippet are manifold. For individual developers, it offers a sandbox environment to experiment with new languages without the need for complex setup. For teams, it serves as a platform for collaborative coding sessions, code reviews, and knowledge sharing, thereby enhancing the quality of the codebase and accelerating project timelines. For educators, Snippet provides a dynamic teaching aid that allows students to engage with code directly, promoting active learning and immediate application of theoretical concepts.

In essence, Snippet is poised to become a hub for coding excellence, where innovation is not just encouraged but actively pursued. Its target audience is as diverse as the programming community itself, ensuring that whether one is writing their first line of code or architecting a complex system, Snippet has something valuable to offer. By bridging gaps and connecting ideas, Snippet stands to revolutionize the way we think about coding and collaboration in the digital age.

# Technology Stack

The project is structured into three distinct repositories, each serving a unique purpose within the platform's ecosystem.

The first repository is dedicated to authentication and sharing API, ensuring secure access and efficient distribution of code among users. The second repository hosts compilers within isolated Docker containers, providing a robust and scalable environment for code compilation. The third repository focuses on the user interface (UI), offering an intuitive and user-friendly interaction layer for the platform's users.

Each repository employs its own technology stack, meticulously chosen to optimize performance and reliability. The authentication API utilizes technologies suited for secure data handling and network communication, while the compilers are supported by a stack that excels in processing speed and resource management. The UI repository, on the other hand, leverages a stack that emphasizes responsive design and user engagement.

These repositories work in concert to deliver a comprehensive coding platform, where each component's stack is described in detail in the sections below, highlighting the harmonious integration of diverse technologies to create a seamless user experience.

## Core Stack

* **Node.js:** Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts (2).
* **Express.js**: Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications (3). It provides middleware for handling requests, routing, and managing HTTP methods.
* **React.js:** React lets you build user interfaces out of individual pieces called components. Create your own React components. Then combine them into entire screens, pages, and apps (4).
* **Next.js:** Next.js is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations (5).
* **MongoDB**: MongoDB is an open-source document-oriented database that is designed to store a large scale of data and also allows you to work with that data very efficiently. It is categorized under the NoSQL (Not only SQL) database because the storage and retrieval of data in the MongoDB are not in the form of tables (6).
* **Mongoose.js**: Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box (7).
* **Docker**: Docker is an open-source containerization platform by which you can pack your application and all its dependencies into a standardized unit called a container (8). These Containers are used to host the required compilers on server.

## Other Dependencies

### Code Execution API:

* cors
* express
* socket.io
* uuid

### Access Control and Authentication API

* cookie-parser
* cors
* dotenv
* express
* express-async-handler
* jsonwebtoken
* mongoose
* multiparty
* nodemailer
* passport
* passport-jwt
* passport-local
* passport-local-mongoose

### User Interface

* @monaco-editor/react
* next
* next-themes
* qs
* react
* react-dom
* react-intersection-observer
* react-toastify
* sharp
* socket.io-client

# Key features and their implementations

In this section, we will explore the key features implemented in this app and the thoughtful process behind their development.

## Client – Compiler Server Communication

To connect the user to the API, I initially considered using the traditional method where the client sends a request to the server, which then processes the request and returns a response. However, this approach has several limitations. Fetch requests are inherently half-duplex, meaning they allow communication only in one direction at a time. This model works well when the user provides all inputs in advance, and the server executes the code accordingly.

For example, if the code requires two numbers to compute their sum, the user must supply these numbers beforehand. While this is straightforward for simple tasks, it becomes impractical for more complex scenarios that require interactive user inputs at different stages.

Consider a program with multiple conditions and loops that require user inputs dynamically based on previous inputs. In such cases, a half-duplex communication model is insufficient because it cannot handle real-time interactivity.

After extensive research, I discovered that using sockets could establish full-duplex communication between the client and the server. This allows data to be sent and received simultaneously, enabling real-time interaction. I found a library called socket.io that simplifies the implementation of socket communication.

### Socket.io

**Installation:**

npm i socket.io

**Setup:**

const server = require("node:http").createServer(app);

const socketIO = require("socket.io");

const io = socketIO(server, {

    cors: {

        origin: "\*" // To allow what domains can request from the server

    }

});

#### Usage:

Now, whenever the client establishes a connection, we will get a user connected in our logs.

io.on('connection', (socket) => {

    console.log('a user connected');

});

In sockets, we can listen for certain events. Using socket.io, we will listen for event code to know when the client is sending the code to be executed. After getting note of the event we would run that code on server and provide the input to the client.

io.on('connection', (socket) => {

    socket.on("code", async ({code, language, input = ""}) => {

await runCodeViaSocket(socket, {code, language, input});

    })

});

Here, runCodeViaSocket is a function that will be responsible for the execution of code and sending the code output to the client. Its execution process will be elaborated in the upcoming sections. While running the code it is possible that we might encounter an error. To prevent our server from crashing in that scenario, we’ll handle the errors using try-catch blocks. If we encounter any error during the code execution, we will return that error to the client. We will emit an error event from the server. Using socket.io we can use emit method to achieve the same.

io.on('connection', (socket) => {

    socket.on("code", async ({code, language, input = ""}) => {

        try {

            await runCodeViaSocket(socket, {code, language, input});

        } catch (err) {

// emit error event to the client

            socket.emit("error", {

                status: err?.status || 500,

                ...err

            })

        }

    })

});

Our server is now capable of receiving the code and sending error to the client in case something bad happens.

## Code Execution

When a user submits their code along with the chosen programming language and any input data, the server processes this request. This includes validation to ensure the code and language are provided and that the language is supported. We are dividing the process into following steps to help reader get better understanding of what is happening.

### Code File Creation

We will need to create a code file to execute the code on our server. This section will provide detailed analysis of how it was achieved in our project.

#### Imports

The server generates a unique identifier (UUID) for the code submission. This ensures that each code submission can be uniquely identified and managed.

const {v4: getUUID} = require("uuid");

const jobID = getUUID();

#### Creation of required Directories

The server checks if the necessary directories (codes and outputs) exist. If not, they are created. This ensures there is a designated space for storing code and output files. We take help of predefined methods like existsSync, mkdirSync, writeFileSync provide by fs and join provided by path module.

const {existsSync, mkdirSync, writeFileSync} = require("fs")

const {join} = require("path");

if (!existsSync(join(process.cwd(), "codes")))

    mkdirSync(join(process.cwd(), "codes"));

if (!existsSync(join(process.cwd(), "outputs")))

    mkdirSync(join(process.cwd(), "outputs"));

#### Code Creation Function

The user's code is then written to a file named using the unique identifier and the appropriate file extension for the chosen language. This file is stored in the codes directory.

const createCodeFile = async (language, code) => {

    const jobID = getUUID();

    const fileName = `${jobID}.${language}`;

    const filePath = join(process.cwd(), `codes/${fileName}`)

    await writeFileSync(filePath, code?.toString())

    return {

        fileName,

        filePath,

        jobID

    };

};

#### Export

We will export this function for further use.

module.exports = {

    createCodeFile,

};

### Code File Deletion

Upon successful execution of the code, the associated code file becomes redundant. Therefore, to optimize storage utilization, we will implement a function to automatically delete the code file. This process not only ensures efficient storage management but also maintains the system's organization by preventing the accumulation of unnecessary files.

#### Imports

This function uses Node.js's fs (filesystem) and path modules to achieve its functionality.

const {unlinkSync} = require("fs");

const {join} = require("path");

The above modules provide following functionalities:

* unlinkSync: This function from the fs module is used to synchronously remove a file or symbolic link from the filesystem.
* join: This function from the path module is used to join all given path segments together using the platform-specific separator as a delimiter.

#### Constructing File Paths

const codeFile = join(process.cwd(), `codes/${uuid}.${lang}`);

const outputFile = join(process.cwd(), `outputs/${uuid}.${outputExt}`);

In above lines, process.cwd() is used to get the current working directory of the Node.js process. join(process.cwd(), `outputs/${uuid}.${outputExt}`) constructs the path to the code file and join(process.cwd(), outputs/${uuid}.${outputExt}) constructs the path to the output file.

#### Deleting the Code File

await unlinkSync(codeFile);

We, then, synchronously delete the code file using above lines.

#### Deleting the Output File

if (outputExt)

  await unlinkSync(outputFile);

It deletes the output code file if it was outputExt (output extension) was provided.

#### Function Implementation

The removeCodeFile function is designed to delete both a code file and its corresponding output file from the filesystem. By using Node.js's fs module for file operations and path module for constructing file paths, the function ensures that the specified files are properly located and removed based on the provided unique identifier, language extension, and optional output extension. This function is then exported as part of the module, making it accessible for use elsewhere in the application. Below is the full implementation of this module.

const {unlinkSync} = require("fs");

const {join} = require("path");

const removeCodeFile = async (uuid, lang, outputExt) => {

    const codeFile = join(process.cwd(), `codes/${uuid}.${lang}`);

    const outputFile = join(process.cwd(), `outputs/${uuid}.${outputExt}`);

    await unlinkSync(codeFile);

    if (outputExt)

        await unlinkSync(outputFile);

};

module.exports = {

    removeCodeFile,

};

### Execution Command Mapping

We create a commandMap function which maps programming languages to their respective compilation and execution commands

#### Parameters:

* jobID: A unique identifier for the job.
* language: The programming language in which the code is written.

#### Returns:

An object containing:

* compileCodeCommand: Command to compile the code (e.g., g++, gcc, mcs).
* compilationArgs: Arguments for the compilation command.
* executeCodeCommand: Command to execute the compiled code or script (e.g., java, node, python3).
* executionArgs: Arguments for the execution command.
* outputExt: The extension of the compiled output file (e.g., out, exe).
* compilerInfoCommand: Command to get the version information of the compiler/interpreter.

#### Language Cases

* Java: Executes the .java file using the java command.
* C++: Compiles the .cpp file with g++ and executes the output file.
* Python: Executes the .py file using python3.
* C: Compiles the .c file with gcc and executes the output file.
* JavaScript: Executes the .js file using node.
* Go: Runs the .go file using the go run command.
* C#: Compiles the .cs file with mcs and executes it with mono.

#### Final Implementation

Following is the complete implementation of this function:

const commandMap = (jobID, language) => {

    switch (language) {

        case 'java':

            return {

                executeCodeCommand: 'java',

                executionArgs: [

                    join(process.cwd(), `codes/${jobID}.java`)

                ],

                compilerInfoCommand: 'java --version'

            };

        case 'cpp':

            return {

                compileCodeCommand: 'g++',

                compilationArgs: [

                    join(process.cwd(), `codes/${jobID}.cpp`),

                    '-o',

                    join(process.cwd(), `outputs/${jobID}.out`)

                ],

                executeCodeCommand:

join(process.cwd(),`outputs/${jobID}.out`),

                outputExt: 'out',

                compilerInfoCommand: 'g++ --version'

            };

        case 'py':

            return {

                executeCodeCommand: 'python3',

                executionArgs: [

                    join(process.cwd(), `codes/${jobID}.py`)

                ],

                compilerInfoCommand: 'python3 --version'

            }

        case 'c':

            return {

                compileCodeCommand: 'gcc',

                compilationArgs: [

                    join(process.cwd(), `codes/${jobID}.c`),

                    '-o',

                    join(process.cwd(), `outputs/${jobID}.out`)

                ],

                executeCodeCommand: join(process.cwd(),

`outputs/${jobID}.out`),

                outputExt: 'out',

                compilerInfoCommand: 'gcc --version'

            }

        case 'js':

            return {

                executeCodeCommand: 'node',

                executionArgs: [

                    join(process.cwd(), `codes/${jobID}.js`)

                ],

                compilerInfoCommand: 'node --version'

            }

        case 'go':

            return {

                executeCodeCommand: 'go',

                executionArgs: [

                    'run',

                    join(process.cwd(), `codes/${jobID}.go`)

                ],

                compilerInfoCommand: 'go version'

            }

        case 'cs':

            return {

                compileCodeCommand: 'mcs',

                compilationArgs: [

                    `-out:${join(

                        process.cwd(),

                        `outputs/${jobID}`

                    )}.exe`,

                    `${join(process.cwd(), `codes/${jobID}.cs`)}`,

                ],

                executeCodeCommand: 'mono',

                executionArgs: [

                    `${join(process.cwd(), `outputs/${jobID}`)}.exe`

                ],

                outputExt: 'exe',

                compilerInfoCommand: 'mcs --version'

            }

    }

}

### Code Compilation and Execution

In the [previous](#_Socket.io) section we used runCodeViaSocket function to execute code. We will discuss its implementation in this section. It dynamically compiles and executes code in various programming languages. This function handles code execution requests, manages temporary file creation and deletion, and handles errors and timeouts. Below is a detailed breakdown of each part of the code.

#### Imports and Dependencies

const {commandMap, supportedLanguages} = require("./instructions")

const {createCodeFile} = require("../file-system/createCodeFile")

const {removeCodeFile} = require("../file-system/removeCodeFile")

const {info} = require("./info")

const {spawn} = require("child\_process");

* commandMap and supportedLanguages are imported from the instructions module. commandMap provides the necessary commands and arguments for compiling and executing code based on the job ID and language. supportedLanguages is an array of languages that are supported by the system.
* createCodeFile and removeCodeFile are functions from the file-system module that handle the creation and deletion of code files, respectively.
* info is imported from the info module and provides additional information about the language being executed.
* spawn is a function from Node.js's child\_process module used to execute shell commands.

#### Input Validation

We will ensure that the parameters passed aren’t faulty to prevent our server from facing unexpected errors.

if (!socket) {

    throw {

        status: 500,

        error: "Could not establish a socket connection."

    }

}

if (code === "")

    throw {

        status: 400,

        error: "No Code found to execute."

    }

if (!supportedLanguages.includes(language))

    throw {

        status: 400,

        error: `Please enter a valid language. The languages currently supported are: ${supportedLanguages.join(', ')}.`

    }

#### Code File Creation and Command Retrieval

We utilize previously defined functions createCodeFile and commandMap functions to achieve the same.

const {jobID} = await createCodeFile(language, code);

A code file is created using the createCodeFile function, and a unique jobID is returned.

const {compileCodeCommand, compilationArgs, executeCodeCommand, executionArgs, outputExt} = commandMap(jobID, language);

The commandMap function is used to retrieve the necessary commands and arguments for compiling and executing the code based on the jobID and language.

#### Code Compilation

If a compileCodeCommand is provided, the code needs to be compiled. The spawn function executes the compile command with the provided arguments. If any errors occur during compilation, they are captured and the promise is rejected with the error details. On successful compilation, the promise is resolved.

if (compileCodeCommand) {

    await new Promise((resolve, reject) => {

        const compileCode = spawn(compileCodeCommand, compilationArgs || [])

        compileCode.stderr.on('data', (error) => {

            reject({

                status: 200,

                output: '',

                error: error.toString(),

                language

            })

        });

        compileCode.on('exit', () => {

            resolve()

        })

    })

}

#### Code Execution

The spawn function executes the code with the provided execution arguments. The output and error string are initialized to capture the standard output and standard error from the execution process. A timer is set to enforce the timeout limit. If the execution exceeds the timeout, the process is killed, the code file is removed, and an error is returned. If input is provided, it is written to the process's stdin. Data from stdout and stderr are collected and sent to the client via socket.emit.

Upon process exit, the timer is cleared, and the result is resolved with the collected output and error. We will limit the maximum time taken by a process to execute to be 120 seconds.

const timeout = 120;

const result = await new Promise((resolve, reject) => {

    const executeCode = spawn(executeCodeCommand, executionArgs || []);

    let output = "", error = "";

    const timer = setTimeout(async () => {

        executeCode.kill("SIGHUP");

        await removeCodeFile(jobID, language, outputExt);

        reject({

            status: 408,

            error: `CodeX API Timed Out. Your code took too long to execute, over ${timeout} seconds. Make sure you are sending input as payload if your code expects an input.`

        })

    }, timeout \* 1000);

    if (input !== "") {

        input.split('\n').forEach((line) => {

            executeCode.stdin.write(`${line}\n`);

        });

    }

    // Allow more data through socket emit

    socket.on("input", (input) => {

        executeCode.stdin.write(`${input}\n`)

    })

    executeCode.stdin.on('error', (err) => {

        console.log('stdin err', err);

    });

    executeCode.stdout.on('data', (data) => {

        output += data.toString();

        socket.emit('output', {

            output: data.toString(),

            executionStatus: true

        })

    });

    executeCode.stderr.on('data', (data) => {

        error += data.toString();

        socket.emit('error', {

            error: data.toString(),

            executionStatus: true

        })

    });

    executeCode.on('exit', (err) => {

        clearTimeout(timer);

        socket.emit('exit', {

            output,

            error,

            executionStatus: false

        })

        resolve({output, error});

    });

})

## User Authentication

In our application, we provide the functionality for users to save their code snippets. To ensure the security of these snippets, authentication is necessary. Without proper authentication, there is a risk that individuals could alter someone else's code. This security measure also aids in maintaining the integrity of the user's work. Therefore, we have securely implemented user authentication utilizing Passport-Local-Mongoose, a reliable and efficient solution for managing user sessions and data protection.

### User Registration

New users can register themselves on the application by providing a unique username and their email address.

#### API Endpoint

We will use a POST endpoint /api/users/register for user login, linked to the register controller function.

#### Function Implementation

export const register = asyncHandler(async (req, res) => {

    //Get the username, email, and password from the form.

    try {

        var {

            username: [username],

            email: [email],

            password: [password],

        } = await asyncFormParser(req).catch((err) => {

            throw err;

        });

    } catch (error) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("All fields are mandatory!");

    }

    // Credit: https://www.tutorialspoint.com/How-to-validate-email-address-in-JavaScript

    const emailFormat = /^\w+([**\.**-]?\w+)\*@\w+([**\.**-]?\w+)\*(**\.**\w{2,3})+$/;

    // Reject Form Data if required details not available

    if (!(username && email && password)) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("All fields are mandatory!");

    }

    if (username.includes(" ")) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Username Can't have spaces!");

    }

    if (!email.match(emailFormat)) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Invalid Email Address!");

    }

    const user = await createUser(username, email, password).catch((err) => {

        if (err.code === 11000) res.status(400);

        throw err;

    });

    if (user) {

        sendVerificationMail(user);

        sendAuthDetails(res, user, statusCode.CREATED);

    } else throw new Error("Something went Wrong!");

});

* This function is an asynchronous handler (asyncHandler) for handling user registration requests.
* It parses the form data for username, email, and password.
* It validates the email format and rejects the request if any required field is missing.
* It creates a new user in the database.
* It sends a verification email to the user's email address.
* Finally, it returns an access token and a refresh token.

### User Login

Existing users can login to their account by providing their username or email along with their password.

#### API Endpoint

We will use a POST endpoint /api/users/login for user login, linked to the login controller function.

#### Function Implementation

export const login = asyncHandler(async (req, res) => {

    try {

        var {

            username: [username],

            password: [password],

        } = await asyncFormParser(req).catch((err) => {

            throw err;

        });

    } catch (error) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("All fields are mandatory!");

    }

    const { user } = await User.authenticate()(username, password);

    if (!user) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Invalid Credentials!");

    } else sendAuthDetails(res, user, statusCode.ACCEPTED);

});

* This function is also an asynchronous handler for handling user login requests.
* It parses the form data for username and password.
* It authenticates the user by checking the credentials against the database.
* Finally, it returns an access token and a refresh token.

### Resetting User Password

If a user has forgotten their password or wishes to reset it, the application facilitates this by dispatching a password reset link to their registered email address. Upon receiving the email, the user is required to verify the link which then grants them access to create a new password. This process ensures that only the rightful owner of the account can modify the password, thereby maintaining the security and integrity of the user's data.

#### Request Password Reset

To allow users to get a reset password link to their registered email.

##### API Endpoint

We define a POST endpoint /api/users/req-password for initiating a password reset request. It's linked to the reqPassword controller function.

##### Function Implementation

export const reqPassword = asyncHandler(async (req, res) => {

    try {

        var { username } = req.body;

    } catch (error) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Please Provide a username!");

    }

    const user = await User.findByUsername(username, false);

    if (!user) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Invalid Credentials!");

    }

    const reset = generateResetToken(user.id);

    sendMail({

        to: user.email,

        subject: "Regarding Changing Your Password.",

        text: `Hey ${

            user.username

        }! We got to know You forgot your Password. Click on the following link to create a new password.\n Link: ${

            process.env.CHANGE\_PASSWORD\_LINK + reset

        } If this was not you Please ghost this email.\n Regards,\n Snippet `,

    });

    res.send({

        message: "Reset Password Link sent to the email successfully!",

    });

});

* It parses the form data for the username.
* It finds the user by username.
* It generates and sends a password reset link to the user's email.
* Finally, it returns a success message.

#### Reset Password

We permit users to initiate a password reset by clicking on a valid link dispatched to their registered email address. The integrity of this process is safeguarded by our canReqPassword middleware function, which verifies the legitimacy of the password reset request.

##### API Endpoint

We define a POST endpoint /api/users/reset-password for resetting password. It's linked to the resetForgottenPassword controller function and protected by the canReqPassword middleware.

##### Middleware Implementation

export async function canReqPassword(req, res, next) {

    const reset = tokenResetExtractor(req);

    if (!reset)

        res.status(statusCode.VALIDATION\_ERROR).json({

            error: "Invalid Link!",

        });

    else {

        const token = getDetailsFromJWT(reset, "reset");

        if (!token) {

            res.status(statusCode.UNAUTHORIZED).json({

                error: "Invalid Link!",

            });

            return;

        }

        const user = await User.findById(token.sub);

        if (!user) {

            res.status(statusCode.UNAUTHORIZED).json({

                error: "Invalid Link!",

            });

            return;

        } else {

            req.user = user;

            next();

        }

    }

}

* It extracts the reset token from the request using the tokenResetExtractor function.
* If no reset token is found, it responds with a validation error indicating an "Invalid Link".
* If a reset token is found, it decodes the token and retrieves the user's ID.
* It attempts to find the user in the database using the retrieved ID.
* If the user is not found, it responds with an unauthorized error indicating an "Invalid Link".
* If the user is found, it attaches the user object to the request (req.user) and calls the next() function to proceed to the next middleware in the stack.

##### Main Function Implementation

export const resetForgottenPassword = asyncHandler(async (req, res) => {

    try {

        var {

            password: [password],

        } = await asyncFormParser(req).catch((err) => {

            throw err;

        });

    } catch (error) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Please Provide a new Password!");

    }

    const user = req.user;

    await user.setPassword(password);

    await user.save();

    res.status(statusCode.ACCEPTED).send({

        message: "Password Changed Successfully",

    });

});

* It parses the form data for the new password.
* It sets the new password for the user.
* It saves the user's updated information in the database.
* Finally, it returns a success message.

### Change Email

There might be a case that user entered wrong email address while registering to the app. It will result him in not getting the verification email. To address this issue, the application provides a feature that allows users to update their email address. This functionality is not only crucial for rectifying entry errors but also serves as a convenience for users who may need to change their email address for various reasons, such as losing access to their previous email address.

#### API Endpoint

We define a POST endpoint /api/users/change-email for changing a user's email address, protected by authentication middleware. It's linked to the changeEmail controller function.

#### Function Implementation

export const changeEmail = asyncHandler(async (req, res) => {

    try {

        var { email } = req.body;

    } catch (error) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Please Provide a valid Email");

    }

    // Credit: https://www.tutorialspoint.com/How-to-validate-email-address-in-JavaScript

    const emailFormat = /^\w+([**\.**-]?\w+)\*@\w+([**\.**-]?\w+)\*(**\.**\w{2,3})+$/;

    if (!email || !email.match(emailFormat)) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Invalid Email Address!");

    }

    const user = req.user;

    if (email === user?.email) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw new Error("Provide A different Email than the existing one.");

    }

    user.set("email", email);

    await user.save().catch((err) => {

        if (err.code === 11000) {

            res.status(statusCode.VALIDATION\_ERROR);

            throw new Error("Email already Exists!");

        }

        throw err;

    });

    res.status(statusCode.ACCEPTED).send({

        message:

            "Email Changed Successfully! Please check your mailbox for verification link.",

    });

});

* It parses the form data for the new email.
* It validates the format of the new email.
* It checks if the new email is different from the existing one.
* It updates the user's email address.
* It saves the user's updated information in the database.
* Finally, it returns a success message.

## Uploading Snippets

After creating Code snippet user can upload their snippets on the application. This is one of the most important features on this application.

#### API Endpoint

We define a POST endpoint /api/snippet/add for uploading Snippet, protected by authentication middleware. It's linked to the addSnippet controller function.

#### Function Implementation

export const addSnippet = asyncHandler(async (req, res) => {

    const { title, files } = req.body;

    if (!files || !files.length) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw Error("Please Provide Code File");

    }

    for (const file of files) {

        if (!file || !file.language) {

            res.status(statusCode.VALIDATION\_ERROR);

            throw Error("Please Provide Language of Code");

        }

    }

    const snippetId = new Types.ObjectId();

    const newSnippet = new Snippet({

        \_id: snippetId,

        user: req.user.\_id,

        title,

        files,

    });

    await newSnippet.save();

    res.send({

        message: "Saved Snippet Successfully",

        id: snippetId,

        user: req.user.\_id,

    });

});

* Extracts title and files from the request body.
* Validates the presence of files and their languages.
* Creates a new snippet with a generated ID and the authenticated user's ID.
* Saves the snippet to the database and responds with a success message.

## Updating Snippet

The user might need to make some changes to the uploaded code snippet. To make that feature available the app allows the user to update the code snippet.

#### API Endpoint

We define a POST endpoint /api/snippet/update for uploading Snippet, protected by authentication middleware. It's linked to the updateSnippet controller function.

#### Function Implementation

export const updateSnippet = asyncHandler(async (req, res) => {

    const { title, files, \_id } = req.body;

    if (!files || !files.length) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw Error("Please Provide Code File");

    }

    for (const file of files) {

        if (!file || !file.language) {

            res.status(statusCode.VALIDATION\_ERROR);

            throw Error("Please Provide Language of Code");

        }

    }

    const result = await Snippet.updateOne(

        {

            \_id,

            user: req.user.\_id,

        },

        {

            title,

            files,

        }

    );

    if (result.matchedCount === 1)

        res.send({ message: "Snippet updated Successfully" });

    else {

        res.status(statusCode.VALIDATION\_ERROR);

        throw Error("Snippet Not Found");

    }

});

* Extracts title, files, and \_id from the request body.
* Validates the presence of files and their languages.
* Updates the snippet if it belongs to the authenticated user.
* Responds with a success message if the update is successful or a validation error if the snippet is not found.

## Deleting Snippet

If the snippet does not serve any purpose anymore app provides a feature to delete the snippet. This allows user to have full control over what he wants to be uploaded and what not.

#### API Endpoint

We define a DELETE endpoint /api/snippet/delete for updating Snippet, protected by authentication middleware. It's linked to the deleteSnippet controller function.

#### Function Implementation

export const deleteSnippet = asyncHandler(async (req, res) => {

    const { id } = req.body;

    const result = await Snippet.deleteOne({

        \_id: id,

        user: req.user.\_id,

    });

    res.send(result);

});

* Extracts the snippet ID from the request body.
* Deletes the snippet if it belongs to the authenticated user.
* Responds with the result of the deletion operation.

## Retrieving Snippet

User will need to retrieve snippet for further modification or another user might need the snippet considering it was shared with him. User might need list of all the snippets posted by him or another user or the user might need to get a specific snippet. The app has endpoints for both of them.

### Retrieving List of User Snippets

#### API Endpoint

We define a GET endpoint /api/snippet/user/[username] for retrieving user Snippets. It's linked to the getUserSnippet controller function

#### Function Implementation

export const getUserSnippets = asyncHandler(async (req, res) => {

    const { username } = req.params;

    const { q = "", offset = 0, limit = 12 } = req.query;

    const user = await User.findOne({

        username,

    });

    if (!user) {

        res.status(statusCode.NOT\_FOUND);

        throw Error("User not Found!");

    }

    const snippets = await Snippet.find(

        {

            user: user.\_id,

        },

        "\_id title files.language user"

    )

        .and({

            $or: [

                { title: { $regex: q, $options: "i" } },

                { "files.language": { $regex: q, $options: "i" } },

            ],

        })

        .skip(offset)

        .limit(limit);

    res.send(snippets);

});

* Extracts username from URL parameters and q, offset, and limit from query parameters.
* Finds the user by username.
* If the user is not found, responds with a 404 error.
* Searches for snippets belonging to the user, filtered by title or file language, and paginates the results.
* Responds with the found snippets

### Retrieving specific Snippet

#### API Endpoint

We define a GET endpoint /api/snippet/get/[snippetid] for retrieving Snippet. It's linked to the getSnippet controller function.

#### Function Implementation

export const getSnippet = asyncHandler(async (req, res) => {

    const { id } = req.params;

    try {

        const snippet = await Snippet.findById(id);

        if (!snippet) throw Error();

        res.send(snippet);

    } catch (error) {

        res.status(statusCode.VALIDATION\_ERROR);

        throw Error("Invalid Id");

    }

});

* Extracts the snippet ID from URL parameters.
* Tries to find the snippet by its ID.
* If the snippet is found, responds with the snippet data.
* If an error occurs (e.g., invalid ID), responds with a validation error.

### Searching for a specific Snippet

Another important feature is to allow the user to search for the code snippets of certain language or with certain title from large number of snippets available.

#### API Endpoint

We define a GET endpoint /api/snippet/ for retrieving all available Snippets. The result can be filtered by providing query using parameter q. It's linked to the findAll controller function.

#### Function Implementation

export const findAll = asyncHandler(async (req, res) => {

    const { q = "", offset = 0, limit = 12 } = req.query;

    const snippets = await Snippet.find(

        {

            $or: [

                { title: { $regex: q, $options: "i" } },

                { "files.language": { $regex: q, $options: "i" } },

            ],

        },

        "\_id title files.language"

    )

        .skip(offset)

        .limit(limit);

    res.send(snippets);

});

* + Extracts query parameters q, offset, and limit.
  + Searches for snippets by title or file language using a case-insensitive regex.
  + Paginates the results using skip and limit.
  + Responds with the found snippets.

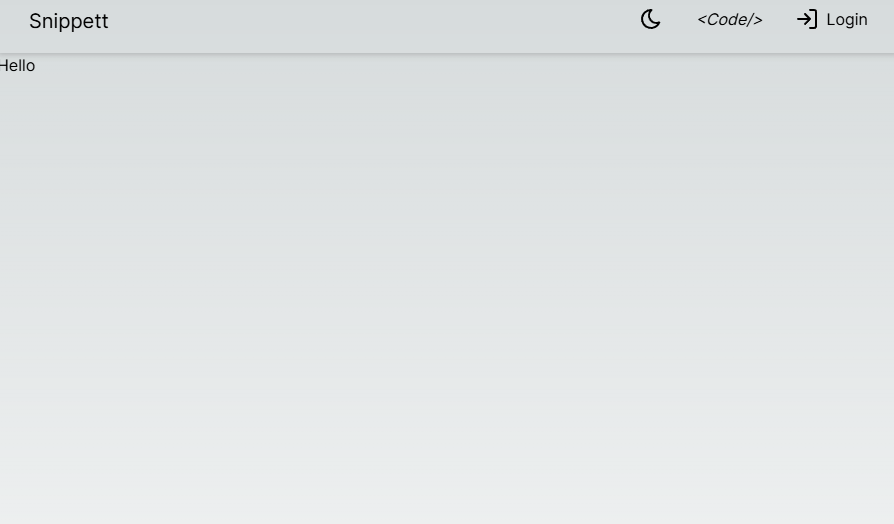
# USer Interface Design And Implementation

I have designed the user interface (UI) with simplicity in mind, ensuring it is intuitive for users to navigate. A straightforward UI streamlines the coding process, enhancing productivity and reducing the learning curve for new users. By focusing on a clean and uncluttered layout, we facilitate a more efficient coding environment that supports user concentration and minimizes distractions. This approach not only aids in quick comprehension of the interface's functionality but also promotes a more enjoyable coding experience.

## Navigation Links

Navigation links in the main layout allows user to navigate from one page to another seamlessly. This component is a versatile and interactive part of the snippet project. It handles critical functionalities such as theme toggling and navigation based on user authentication status. This implementation enhances user experience by providing intuitive controls and personalized navigation options.

### Design



### Features

The main navigation bar in the layout allows following functionalities:

* Switching between Light and Dark Mode
* Navigating to the Code Compiler
* Navigating to appropriate links based on user authentication status.

### Implementation

This component dynamically manages theme switching and user authentication links in the navigation bar. It integrates with the Next.js theme and navigation systems and leverages React context for authentication state. The component ensures a responsive UI by showing appropriate icons and links based on the user's state and current URL.

#### Import Statements

'use client';

import {

  LoginSvg,

  MoonSvg,

  SunSvg,

  ThemeLoader,

  UserSvg

} from '@/assets/icons';

import { useTheme } from 'next-themes';

import { useContext, useEffect, useState } from 'react';

import { getUserAuthDetails } from './Auth';

import { usePathname } from 'next/navigation';

import { AuthContext } from './AuthContext';

import Link from 'next/link';

* The use client directive specifies that this component should be rendered on the client side.
* LoginSvg, MoonSvg, SunSvg, ThemeLoader, and UserSvg are custom icons imported from a local assets directory.
* useTheme from next-themes is used to manage and apply theme settings.
* useContext, useEffect, and useState are hooks from React for state and lifecycle management.
* getUserAuthDetails is a function to fetch user authentication details.
* usePathname from next/navigation helps in getting the current pathname in a Next.js application.
* AuthContext provides authentication context, which includes user information.
* Link from next/link is used for client-side navigation.

#### Theme Management

const { theme, setTheme, systemTheme } = useTheme();

useTheme provides the current theme (theme), a function to change the theme (setTheme), and the system theme (systemTheme).

#### Component State and Context

const [mounted, setMounted] = useState(false);

const [{ username, access }, setUser] = useContext(AuthContext);

* mounted state indicates if the component has been mounted.
* AuthContext provides the user details (e.g., username and access).

#### Fetching User Authentication Details

  useEffect(() => {

    setMounted(true);

    (async () => setUser(await getUserAuthDetails()))();

  }, [setUser]);

useEffect runs after the component mounts to set the mounted state to true and fetch user authentication details asynchronously.

#### Theme Toggle Function

function toggleTheme() {

    const currTheme = theme === 'system' ? systemTheme : theme;

    if (currTheme === 'dark') setTheme('light');

    else setTheme('dark');

}

toggleTheme function switches between light and dark themes based on the current theme.

#### Rendering the Component

return (

    <nav>

      <ul className="flex h-full justify-evenly gap-8">

        <li>

          <button

            aria-label="Switch theme"

            title="Switch theme"

            onClick={toggleTheme}

          >

            {!mounted ? (

              <ThemeLoader />

            ) : theme === 'dark' ||

              (theme === 'system' && systemTheme === 'dark') ? (

              <SunSvg />

            ) : (

              <MoonSvg />

            )}

          </button>

        </li>

        {usePathname().includes('snippet') || (

          <li className="flex gap-2">

            <Link href={'/snippet'}>

              <i title="Write Code" aria-label="Write Code">

                &lt;<span className="hidden md:inline">Code</span>/&gt;

              </i>

            </Link>

          </li>

        )}

        <li>

          <Link

            href={access ? `/profile/${username}` : '/login'}

            className="flex gap-2"

          >

            {access ? (

              <UserSvg />

            ) : (

              <>

                <LoginSvg title="Login" />{' '}

                <span className="hidden md:inline">Login</span>

              </>

            )}

          </Link>

        </li>

      </ul>

    </nav>

);

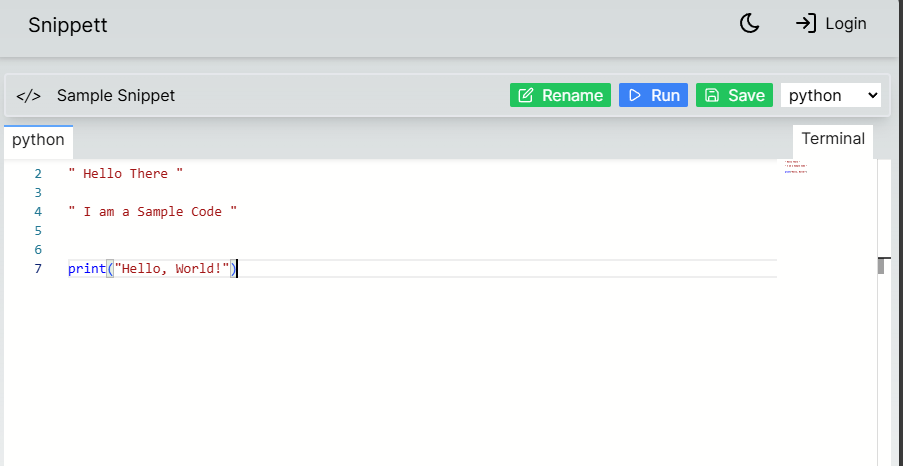
The nav element contains a ul element with three li elements:

* **Theme Toggle Button:** Changes the theme and displays either SunSvg or MoonSvg based on the current theme. If the component isn't mounted yet, it shows a ThemeLoader.
* **Snippet Link:** Conditionally renders a link to the /snippet page if the current pathname does not include snippet.
* **Profile/Login Link**: Links to the user's profile if they are authenticated, otherwise links to the login page. Displays either UserSvg or LoginSvg with a "Login" text based on the user's authentication status.

## Code Editor

If you have a good code editor while coding it adds to the coding experience. In this app I integrated Monaco Editor to provide better user experience while writing code.

### Design



### Implementation

The Code Editor component is designed to provide a rich code editing experience using the Monaco Editor. It includes theme support, handles conditional visibility, and listens for Ctrl+S to trigger a save function. It also updates the snippet state dynamically as the content changes.

#### Import Statements

import { Editor } from '@monaco-editor/react';

import { useTheme } from 'next-themes';

import loading from '@/assets/loading.gif';

import Image from 'next/image';

import { handleSaveButton } from './SnippetHeader';

* Editor is imported from @monaco-editor/react, which is a React wrapper for the Monaco Editor.
* useTheme from next-themes is used to access the current theme (light or dark).
* loading imports a loading GIF to display while the editor is loading.
* Image from next/image is used for optimized image rendering.
* handleSaveButton is a function imported from SnippetHeader to handle saving the snippet.

#### Function Signature

export default function CodeEditor({

  snippet,

  setSnippet,

  fileIndex,

  isVisible

}) {

  const { theme } = useTheme();

* The CodeEditor function component accepts four props:
  + snippet: The snippet object containing code files and metadata.
  + setSnippet: A function to update the snippet state.
  + fileIndex: The index of the currently active file within the snippet.
  + isVisible: A boolean indicating whether the editor should be visible.
* useTheme hook is used to retrieve the current theme (light or dark).

#### Conditional Visibility of Code Editor

return (

    <div

      className={`w-full ${isVisible ? 'block' : 'hidden'} lg:block`}

      onKeyDown={(e) => {

        if (e.ctrlKey && e.key === 's') {

          e.preventDefault();

          handleSaveButton();

        }

      }}

    >

      // Editor Component Here

    </div>

  );

}

* The outer div element's className conditionally applies the block class if isVisible is true, otherwise applies hidden. It always applies lg:block for larger screens.
* onKeyDown event handler checks for Ctrl+S (or Cmd+S on Mac) to prevent the default browser save behaviour and instead call the handleSaveButton function.

#### Editor Component

<Editor

  height="80vh"

  language={snippet.files[fileIndex].language}

  value={snippet.files[fileIndex].content}

  onChange={(value) =>

    setSnippet({

      ...snippet,

      files: snippet.files.map((file, index) =>

        index === fileIndex ? { ...file, content: value } : { ...file }

      )

    })

  }

  theme={theme === 'light' ? 'light' : 'vs-dark'}

  loading={<Image src={loading} alt="Loading" />}

  path={fileIndex}

/>

Editor component from @monaco-editor/react is configured with several props:

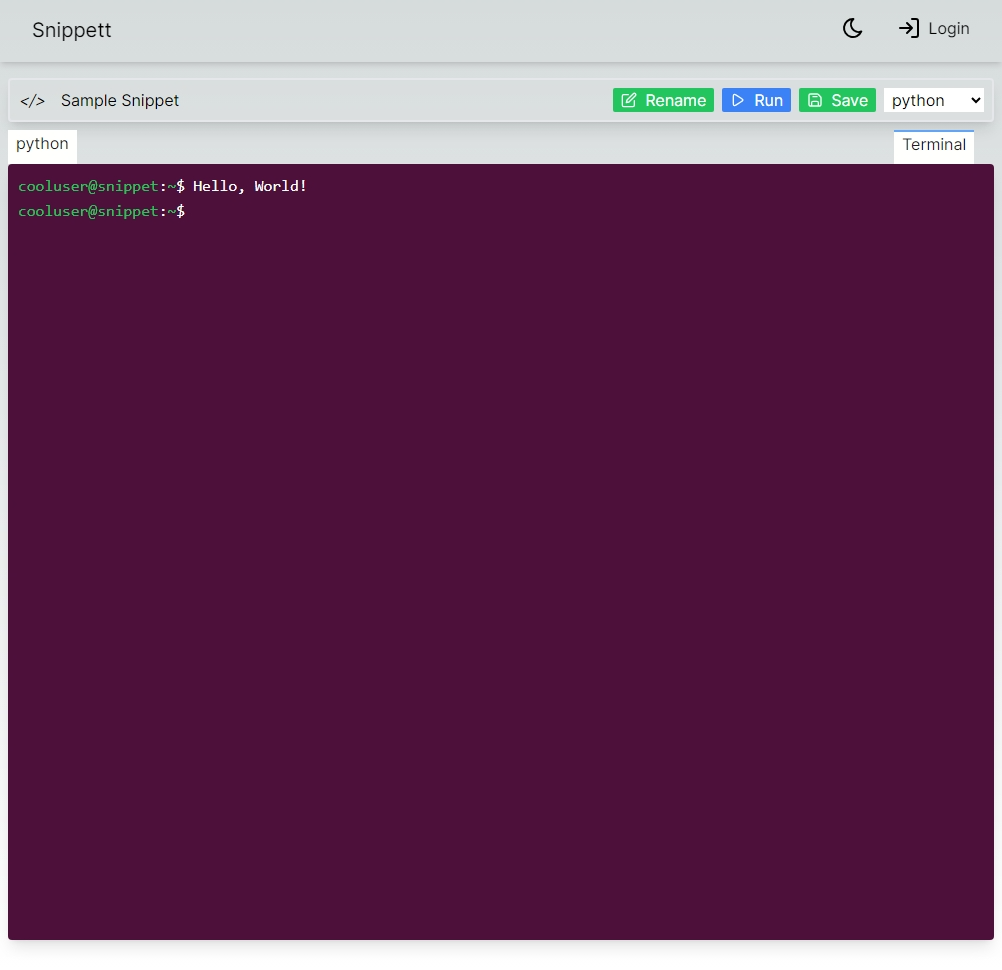
* height: Sets the height of the editor to 80vh (80% of the viewport height).
* language: Sets the language mode of the editor based on the currently active file's language.
* value: Sets the initial content of the editor to the currently active file's content.
* onChange: Event handler to update the snippet state when the content changes. It creates a new snippet object with the updated content for the currently active file.
* theme: Sets the theme of the editor based on the current theme (light or dark).
* loading: Displays a loading spinner (an image) while the editor is loading.
* path: Sets a unique path for the editor instance, which can be useful for tracking changes.

## Terminal

Upon writing their code inside the code editor provided by the application, users can effortlessly execute their programs. The execution process is designed to be intuitive, mirroring the simplicity of traditional desktop coding environments. The output, including any errors or exceptions, is displayed within the terminal interface. This immediate feedback loop is crucial for debugging and iterative development.

### Design

The terminal within Snippet is designed to replicate the look and feel of a Linux terminal. This design choice is not merely aesthetic; it serves a practical purpose by providing a familiar environment for developers accustomed to Linux-based systems. Moreover, it fosters a learning atmosphere for new programmers, who can benefit from the robustness and flexibility of a Linux-like terminal while operating within a web-based platform.



### Implementation

The Terminal component provides a fully interactive terminal interface. It integrates with a WebSocket server to handle real-time code execution and input. The component manages terminal data, user input, and WebSocket events to offer a smooth and responsive terminal-like experience within a web application.

#### Import Statements and Setup

'use client';

import { AuthContext } from '@/app/AuthContext';

import { useContext, useEffect, useRef, useState } from 'react';

import { sendInput, socket } from './data/socket';

* The 'use client' directive indicates that this code should be executed on the client side.
* AuthContext provides the authentication context, giving access to user details.
* useContext, useEffect, useRef, and useState are hooks from React for state and lifecycle management.
* sendInput and socket are imported from a socket module, which handles real-time communication with a server.

#### Global Functions Export

export let runCode = async () => {};

export let addDataInTerminal = () => {};

These global functions will be assigned within the component to enable external triggering of terminal actions like running code and adding data to the terminal.

#### Custom Hook for handling Terminal Data

export function useTerminal(username) {

  const initSpan = (

    <>

      <span className="pr-2 break-none">

        <span className="text-green-500">{username || 'cooluser'}@snippet</span>

        <span>:</span>

        <span className="text-green-500">~</span>

        <span>$</span>

      </span>

    </>

  );

  const [terminalData, setTerminalData] = useState(initSpan);

  const addTerminalData = (data, newLine = false, isError = false) => {

    setTerminalData(

      <>

        {terminalData}

        {isError ? (

          <span

            style={{

              color: 'red'

            }}

          >

            {data}

          </span>

        ) : (

          <span>{data}</span>

        )}

        {newLine && initSpan}

      </>

    );

  };

  const clearTerminalData = () => setTerminalData(initSpan);

  return { terminalData, addTerminalData, clearTerminalData };

}

* useTerminal is a custom hook that manages the terminal's state.
* initSpan initializes the terminal prompt.
* terminalData holds the current content of the terminal.
* addTerminalData appends new data to the terminal, optionally in red for errors and with a new line if specified.
* clearTerminalData resets the terminal to its initial state.

#### Main Terminal Component Function Props and Context

export default function Terminal({

  visibilityClass,

  executing,

  setExecStatus

}) {

  const [{ username }, \_] = useContext(AuthContext);

  const tARef = useRef(null);

  const [data, setData] = useState('');

  const { terminalData, addTerminalData, clearTerminalData } =

    useTerminal(username);

  addDataInTerminal = addTerminalData;

  const [isSocketConnected, setIsSocketConnected] = useState(false);

// More Code Here

}

* visibilityClass, executing, and setExecStatus are props to control the terminal's visibility, execution status, and a function to set the execution status, respectively.
* username is retrieved from the AuthContext.
* tARef is a reference to the textarea element.
* data holds the current input from the user.
* useTerminal provides terminal data and functions to manipulate it.
* isSocketConnected tracks the connection status of the WebSocket.

#### Web Socket Handling

  useEffect(() => {

    const onConnect = () => {

      console.log('User Connected');

      setIsSocketConnected(true);

    };

    const onDisconnect = (reason) => {

      console.log('Disconnected due to ' + reason);

      setIsSocketConnected(false);

      setExecStatus(false);

      socket.connect();

    };

    const onError = ({ error, executionStatus }) => {

      addTerminalData(error + '\n', false, true);

      tARef.current?.scrollIntoView(true);

    };

    const onOutput = ({ output, executionStatus }) => {

      addTerminalData(output, !executionStatus);

      tARef.current?.scrollIntoView(true);

    };

    const onExit = () => {

      addTerminalData('', true);

      console.log('process over');

      setExecStatus(false);

    };

    if (!isSocketConnected) socket.connect();

    socket.on('connect', onConnect);

    socket.on('disconnect', onDisconnect);

    socket.on('error', onError);

    socket.on('output', onOutput);

    socket.on('exit', onExit);

    return () => {

      socket.off('connect', onConnect);

      socket.off('disconnect', onDisconnect);

      socket.off('error', onError);

      socket.off('output', onOutput);

      socket.off('exit', onExit);

    };

  }, [addTerminalData, isSocketConnected, setExecStatus]);

* useEffect sets up WebSocket event listeners for connect, disconnect, error, output, and exit events.
* It handles reconnection attempts and updates the terminal based on the WebSocket events.
* The cleanup function removes the event listeners to avoid memory leaks.

#### Executing the User Code

  runCode = async (code, language) => {

    if (!code) return;

    socket.emit('code', {

      code,

      language

    });

    setExecStatus(true);

  };

runCode emits a code event through the WebSocket, sending the code and language to the server for execution.

#### Handling Keyboard Input

  const handleKeyDown = (e) => {

    if (e.key === 'Enter') {

      addTerminalData(data + '\n');

      if (executing && data)

        sendInput(data.trim());

      setData('');

    } else if (e.ctrlKey && e.key === 'c') {

      e.preventDefault();

      addTerminalData(data + '^C\n', true);

      setExecStatus(false);

      setData('');

    } else if (e.ctrlKey && e.key === 'l') {

      e.preventDefault();

      if (executing) return;

      clearTerminalData();

      setData('');

    } else if (e.key === 'ArrowRight' || e.key === 'ArrowLeft')

      // Don't allow user to move cursor position

      e.preventDefault();

  };

handleKeyDown handles various keypresses:

* Enter submits the current data.
* Ctrl+C stops the execution.
* Ctrl+L clears the terminal.
* Arrow keys are disabled to prevent cursor movement to prevent bad user experience due to implementation complexity of a terminal like interface.

#### Rendering the Component

  return (

    <div

      className={` ${visibilityClass} tailwind-css-classes-here`}

      onClick={() => tARef.current?.focus()}

    >

      <span>{terminalData}</span>

      <span className="break-words whitespace-pre-wrap">{data}</span>

      <span>

        <textarea

          ref={tARef}

          className="tailwind-css-classes-here"

          value={data || ''}

          onKeyDown={handleKeyDown}

          onChange={(e) => setData(e.target.value)}

        ></textarea>

      </span>

    </div>

  );

* The terminal interface is rendered inside a div with several classes for styling and functionality.
* The textarea element captures user input and is hidden with CSS styles to simulate a terminal experience.
* Clicking on the terminal focuses the hidden textarea.

# Conclusion

This project successfully implements an online compiler interface that allows users to write, execute, and share code seamlessly. By leveraging modern web technologies and integrating with a WebSocket server, the application provides a responsive and interactive coding environment. The code editor, built using the @monaco-editor/react library, supports syntax highlighting and customizable themes based on user preferences. The terminal interface replicates a traditional command-line experience, supporting real-time code execution, error handling, and user input. User authentication is implemented via passport-local-mongoose, ensuring secure access and personalized user experiences. The application also supports dynamic theming, allowing users to switch between light and dark modes. Real-time interaction with the server for code execution and terminal updates is managed through WebSocket connections, providing a smooth and efficient user experience.

# Bibliography

1. **Douglas Harper.** snippet | Etymology of snippet by etymonline. *Online Etymology Dictionary.* [Online] https://www.etymonline.com/word/snippet.

2. Node JS - Run JavaScript Everywhere. *Node JS.* [Online] https://nodejs.org/.

3. **Express.** Fast, unopinionated, minimalist web framework for Node.js. *Express JS.* [Online] https://expressjs.com/.

4. React - The library for web and native user interfaces. *React.* [Online] https://react.dev/.

5. Introduction. *Next JS.* [Online] https://nextjs.org/docs.

6. What is MongoDB – Working and Features. *Geeks For Geeks.* [Online] https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/.

7. Mongoose. *Mongoose JS.* [Online] https://mongoosejs.com/.

8. What is Docker? *Geeks For Geeks.* [Online] https://www.geeksforgeeks.org/introduction-to-docker/.

9. cookie-parser. *NPM JS.* [Online] https://www.npmjs.com/package/cookie-parser.

10. cors. *NPM JS.* [Online] https://www.npmjs.com/package/cors.

11. dotenv. *NPM JS.* [Online] https://www.npmjs.com/package/dotenv.

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to my mentor, *Dr. Puneet Misra*, for their invaluable guidance, expertise, and support throughout the course of this study. Their extensive knowledge, insightful feedback, and unwavering encouragement have been instrumental in shaping the direction and quality of this study.

I would like to thank the various sources that have contributed to my research and knowledge. I have used different information from books, articles, websites, and other media, either directly by quoting or paraphrasing, or indirectly by referring to their ideas and arguments. I acknowledge the value and importance of these sources, and I appreciate their authors and creators for their work and expertise.

I would also like to extend my gratitude to my dear friends who have been a source of inspiration and motivation. Your unwavering belief in my abilities, your listening ear, and your words of encouragement has given me the strength and confidence to overcome obstacles and pursue this study with dedication.

Finally, I want to express my heartfelt appreciation to all the participants and the resources that have contributed directly or indirectly to this research paper. Without their contribution, this study would not have been possible.

Thanking You,

Prakash Chaurasia