COMPSCI 326 Group 1 (Online)

Mini Terminal

(https://github.com/VicayoMua/326_Mini_Terminal)

Milestone 7

Team Member: Vicayo Zhang, Aryan Ghosh, and Stella Dey

5/7/2025

Project Name: Mini Terminal

Problem/Why This Project?

- Large universities, like UMass, usually provide ssh server environments, like EdLab, for students to get familiar with Linux-styled commands. However, most students only use the most basic features on EdLab, so the power assumption and the continuous maintenance cost are not worth it.
- Students, who are not enrolled in a university and have no previous experience in installing and using Linux, usually find it hard to install a Linux distribution on their PCs.

Solution:

- A terminal simulator, which can be accessed on web browsers and run on local devices (phones, tablets, and PCs), can perfectly solve this problem.

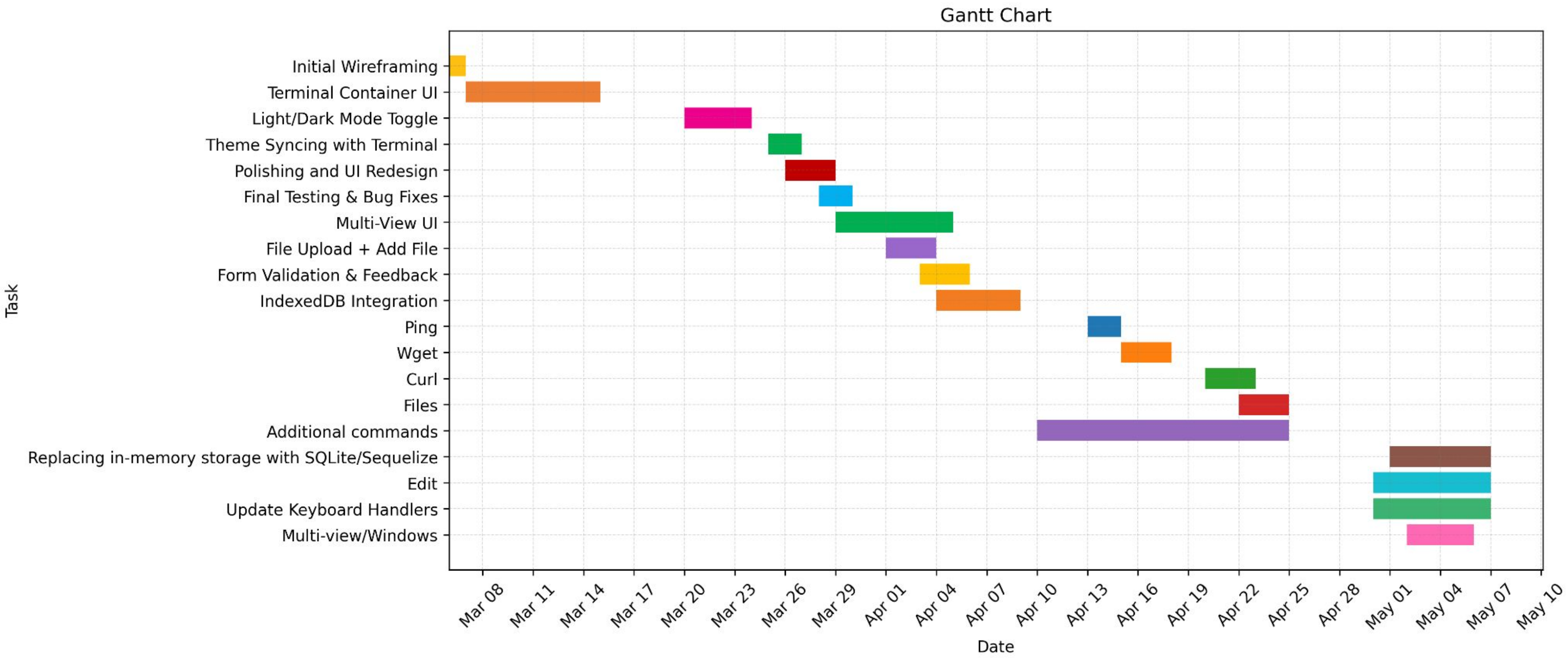
Key Features:

- The terminal simulator will support the most usual Linux commands, like ls, pwd, cd, mkdir, rename, touch, cp, edit. It has file managing abilities.
- Additionally, we can set up additional buttons outside the terminal window to support log-saving, file-system-importing, and file-system-exporting. So, people can easily save and resume their work efficiently.
- Finally, if time allows, the terminal will have supports for Web Assembly.

Team Members

- **Vicayo Zhang – Project Manager & Core Service Developer**
  - **First Issue To Work ON: Basic terminal input and output logics**
  - **Second Issue To Work ON: Log recording logics**
  - **Third Issue To Work ON: file system simulation logics**
  - **Fourth Issue To Work ON: some demo on-terminal applications/commands (hello, help, man, and echo)**
- **Aryan Ghosh – Software Developer & Admin Monitoring**
  - **First Issue To Work ON: Ping command setup**
  - **Second Issue To Work ON: User Session data**
  - **Third Issue To Work ON: Project Documentation**
- **Stella Dey – Debugging & Testing Coordinator**
  - **First Issue To Work ON: Customizable Theme - (Light/Dark Mode) #17**
  - **Second Issue To Work ON: File System Structure Data #18**
  - **Third Issue To Work ON: Curl and files command setup**
  - **Fourth Issue To Work ON: Relacing in-memory storage system with SQLite/Serialize**

Historical Development Timeline



Vicayo Zhang - Assigned Work Summary

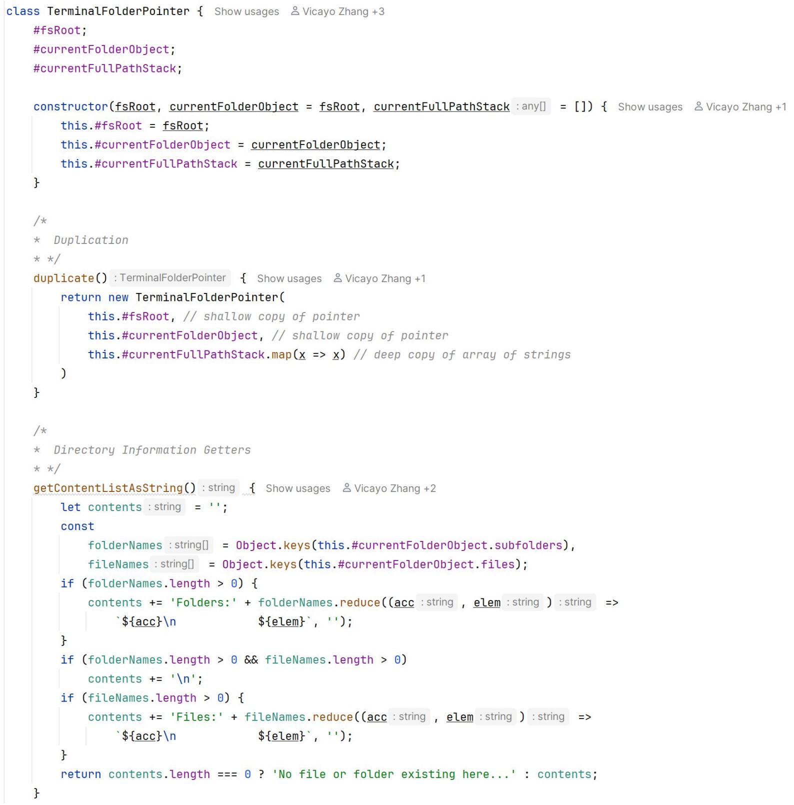
In the JS files:

- Re-implement the file system managing APIs (Feature A. Screenshot 1.)
  - Re-implement the file system pointer using a class
  - Make the pointer even more easy-to-use
- Implement the terminal commands (Feature B.Screenshots 2-8.)
  - Touch (Screenshot 2.)
  - cd (Screenshot 3.)
  - mv (Screenshot 4.)
  - cp (Screenshot 5.)
  - rm (Screenshot 6.)
  - download (Screenshot 7.)
  - print (Screenshot 8.)
- Update the keyboard handlers (Feature C. Screenshot 9.)
- Design the button to open multiple terminal windows and the switch for the multi-view (Feature D. Screenshot 10.)
- Solve all the branch conflicts in the pull requests (Screenshot 11.)

Link to related issues:

- https://github.com/VicayoMua/326_Mini_Terminal/issues/18 (Feature A.)
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/56
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/57
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/63
- https://github.com/VicayoMua/326_Mini_Terminal/issues/54 (Feature B.)
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/58
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/61
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/62
- https://github.com/VicayoMua/326_Mini_Terminal/issues/64 (Feature C.)
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/65
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/66
  - https://github.com/VicayoMua/326_Mini_Terminal/issues/67
- https://github.com/VicayoMua/326_Mini_Terminal/issues/70 (Feature D.)

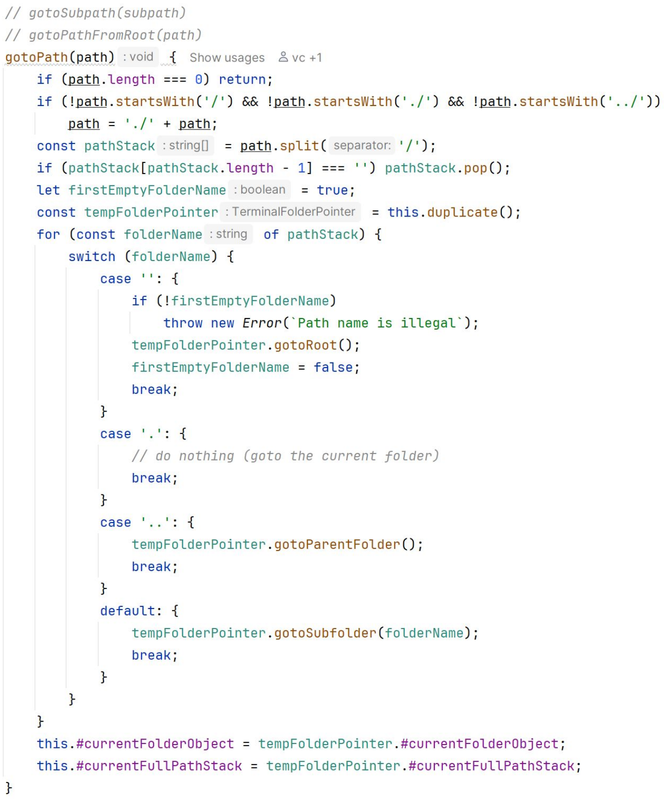
Vicayo Zhang - Screenshot 1 (Code & UI Explanation)



```
class TerminalFolderPointer {  Show usages  Vicayo Zhang +3
    #fsRoot;
    #currentFolderObject;
    #currentFullPathStack;

    constructor(fsRoot, currentFolderObject = fsRoot, currentFullPathStack : any[]  = []) {  Show usages  Vicayo Zhang +1
        this.#fsRoot = fsRoot;
        this.#currentFolderObject = currentFolderObject;
        this.#currentFullPathStack = currentFullPathStack;
    }

    /*
    *  Duplication
    * */
    duplicate() : TerminalFolderPointer  {  Show usages  Vicayo Zhang +1
        return new TerminalFolderPointer(
            this.#fsRoot, // shallow copy of pointer
            this.#currentFolderObject, // shallow copy of pointer
            this.#currentFullPathStack.map(x => x) // deep copy of array of strings
        )
    }

    /*
    *  Directory Information Getters
    * */
    getContentListAsString() : string  {  Show usages  Vicayo Zhang +2
        let contents : string  = '';
        const
            folderNames : string[]  = Object.keys(this.#currentFolderObject.subfolders),
            fileNames : string[]  = Object.keys(this.#currentFolderObject.files);
        if (folderNames.length > 0) {
            contents += 'Folders:' + folderNames.reduce((acc : string , elem : string ) : string  =>
                `${acc}\n            ${elem}`, '');
        }
        if (folderNames.length > 0 && fileNames.length > 0)
            contents += '\n';
        if (fileNames.length > 0) {
            contents += 'Files:' + fileNames.reduce((acc : string , elem : string ) : string  =>
                `${acc}\n            ${elem}`, '');
        }
        return contents.length === 0 ? 'No file or folder existing here...' : contents;
    }
```

Now, the terminal file system folder pointer is implemented using a class so that there is a prototype (template) of the object, and people can easily duplicate the current folder pointer.

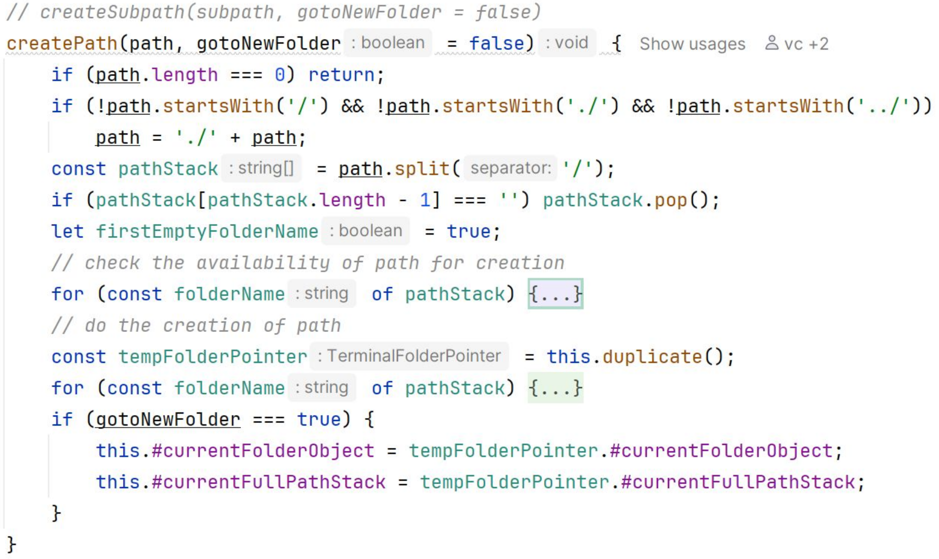
Vicayo Zhang - Screenshot 1 (Code & UI Explanation)



```
// gotoSubpath(subpath)
// gotoPathFromRoot(path)
gotoPath(path) : void  {  Show usages  vc +1
    if (path.length === 0) return;
    if (!path.startsWith('/') && !path.startsWith('./') && !path.startsWith('../'))
        path = './' + path;
    const pathStack : string[]  = path.split( separator: '/');
    if (pathStack[pathStack.length - 1] === '') pathStack.pop();
    let firstEmptyFolderName : boolean  = true;
    const tempFolderPointer : TerminalFolderPointer  = this.duplicate();
    for (const folderName : string  of pathStack) {
        switch (folderName) {
            case '': {
                if (!firstEmptyFolderName)
                    throw new Error(`Path name is illegal`);
                tempFolderPointer.gotoRoot();
                firstEmptyFolderName = false;
                break;
            }
            case '.': {
                // do nothing (goto the current folder)
                break;
            }
            case '..': {
                tempFolderPointer.gotoParentFolder();
                break;
            }
            default: {
                tempFolderPointer.gotoSubfolder(folderName);
                break;
            }
        }
    }
    this.#currentFolderObject = tempFolderPointer.#currentFolderObject;
    this.#currentFullPathStack = tempFolderPointer.#currentFullPathStack;
}
```

Previously provided methods, gotoSubpath(subpath) and gotoPathFromRoot(absolutePath), are upgraded to a unique powerful method gotoPath(path): this method can handle relative path like “abc”, “./abc”, and “/abc/.././”

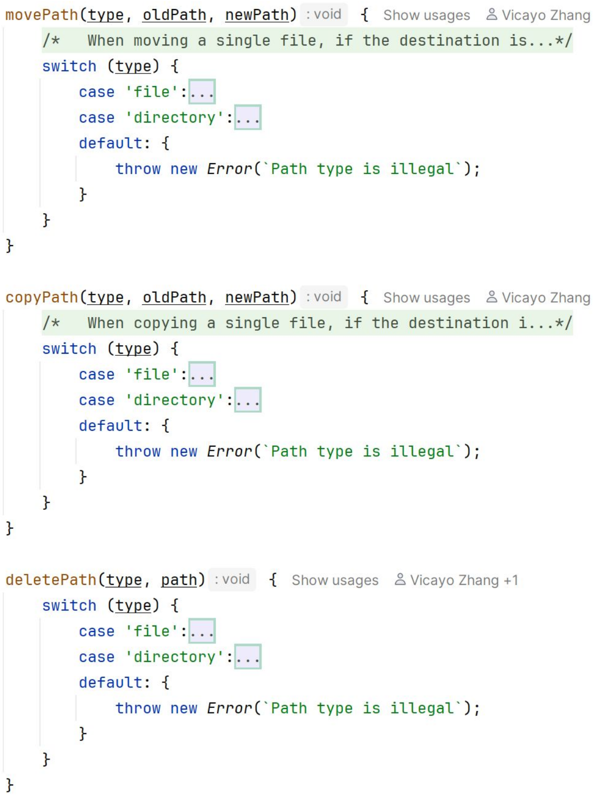
Vicayo Zhang - Screenshot 1 (Code & UI Explanation)



```
// createSubpath(subpath, gotoNewFolder = false)
createPath(path, gotoNewFolder : boolean  = false) : void  {  Show usages  vc +2
    if (path.length === 0) return;
    if (!path.startsWith('/') && !path.startsWith('./') && !path.startsWith('../'))
        path = './' + path;
    const pathStack : string[]  = path.split( separator: '/');
    if (pathStack[pathStack.length - 1] === '') pathStack.pop();
    let firstEmptyFolderName : boolean  = true;
    // check the availability of path for creation
    for (const folderName : string  of pathStack) {...}
    // do the creation of path
    const tempFolderPointer : TerminalFolderPointer  = this.duplicate();
    for (const folderName : string  of pathStack) {...}
    if (gotoNewFolder === true) {
        this.#currentFolderObject = tempFolderPointer.#currentFolderObject;
        this.#currentFullPathStack = tempFolderPointer.#currentFullPathStack;
    }
}
```

Previously provided method, createSubpath(subpath), is upgraded to a unique powerful method createPath(path): this method can handle relative path like “abc”, “./abc”, and “/abc/../.”

Vicayo Zhang - Screenshot 1 (Code & UI Explanation)

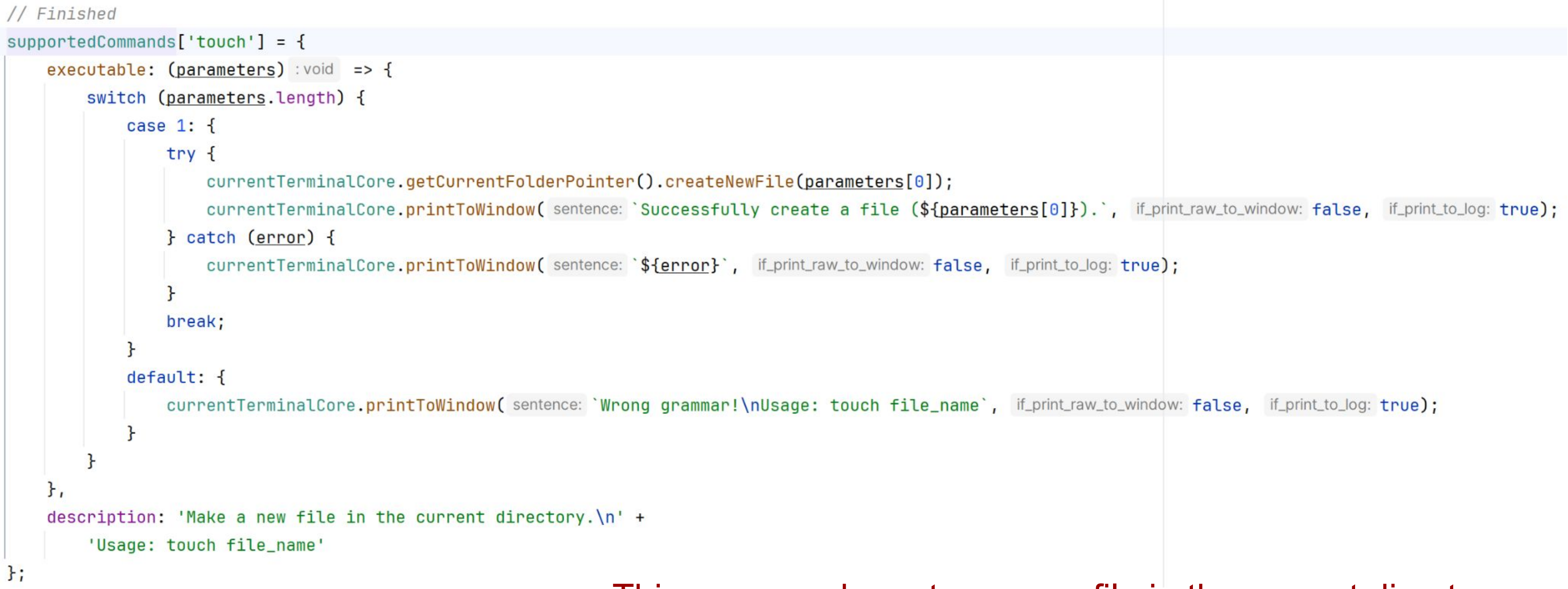


```
movePath(type, oldPath, newPath) : void  {  Show usages  Vicayo Zhang
    /*  When moving a single file, if the destination is...*/
    switch (type) {
        case 'file':...
        case 'directory':...
        default: {
            throw new Error(`Path type is illegal`);
        }
    }
}

copyPath(type, oldPath, newPath) : void  {  Show usages  Vicayo Zhang
    /*  When copying a single file, if the destination i...*/
    switch (type) {
        case 'file':...
        case 'directory':...
        default: {
            throw new Error(`Path type is illegal`);
        }
    }
}

deletePath(type, path) : void  {  Show usages  Vicayo Zhang +1
    switch (type) {
        case 'file':...
        case 'directory':...
        default: {
            throw new Error(`Path type is illegal`);
        }
    }
}
```

There are three new methods added:

- movePath() is the actual data logic part of the terminal command “mv”, which can move an existing file or folder to another location. When the destination location is already existing, a file moving will be prevented while a folder moving will combine the folders.
- copyPath() is the actual data logic part of the terminal command “cp”, which can copy an existing file or folder to another location. When the destination location is already existing, a file copying will be prevented while a folder copying will combine the folders.
- deletePath() is the actual data logic part of the terminal coimmand “rm”, which can delete an existing file or folder.

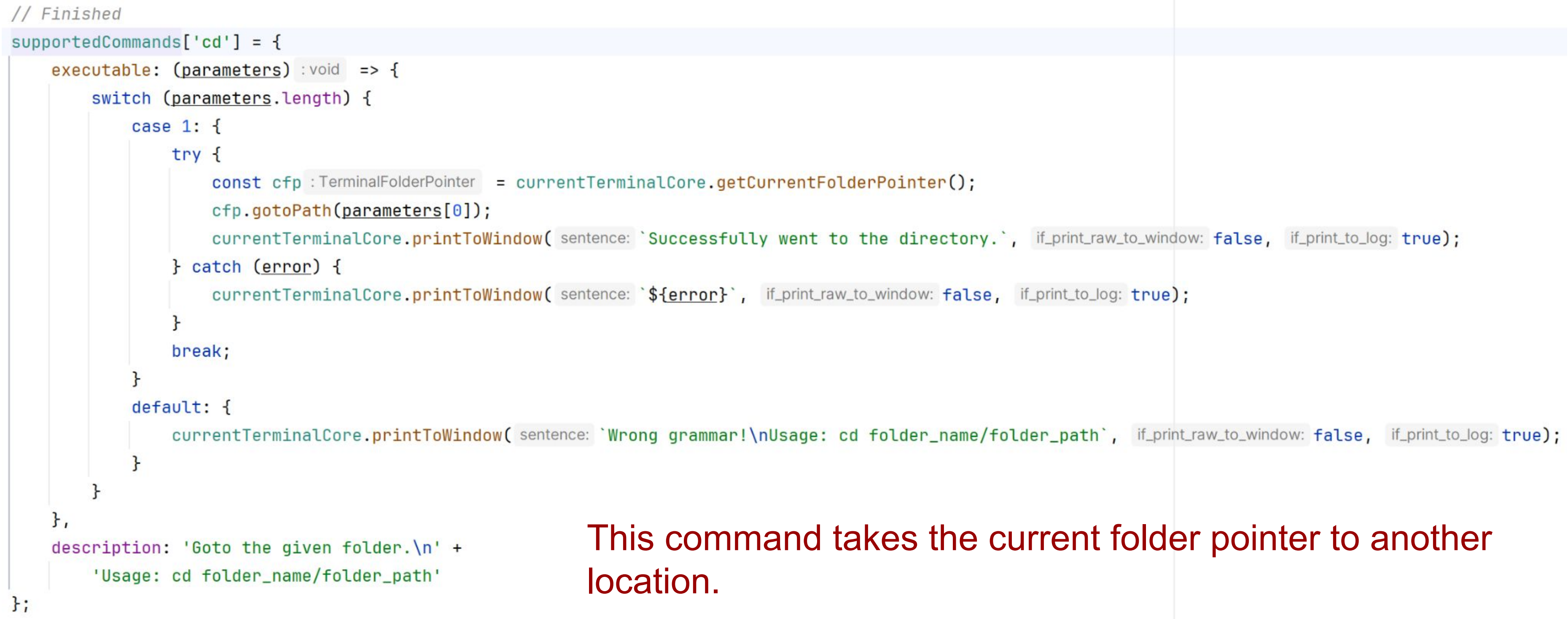
Vicayo Zhang - Screenshot 2 (Code & UI Explanation)



```
// Finished
supportedCommands['touch'] = {
    executable: (parameters) :void  => {
        switch (parameters.length) {
            case 1: {
                try {
                    currentTerminalCore.getCurrentFolderPointer().createNewFile(parameters[0]);
                    currentTerminalCore.printToWindow( sentence: `Successfully create a file (${parameters[0]}).`,  if_print_raw_to_window: false,  if_print_to_log: true);
                } catch (error) {
                    currentTerminalCore.printToWindow( sentence: `${error}`,  if_print_raw_to_window: false,  if_print_to_log: true);
                }
                break;
            }
            default: {
                currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: touch file_name`,  if_print_raw_to_window: false,  if_print_to_log: true);
            }
        }
    },
    description: 'Make a new file in the current directory.\n' +
        'Usage: touch file_name'
};
```

This command creates a new file in the current directory.

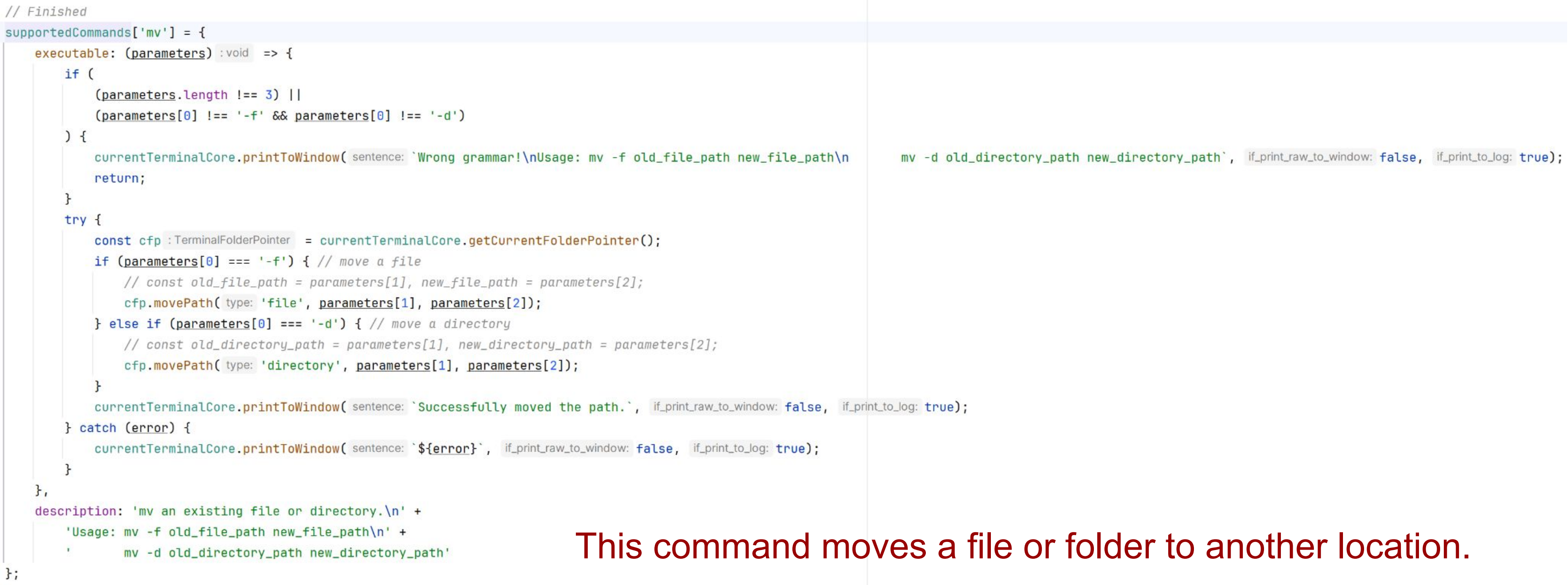
Vicayo Zhang - Screenshot 3 (Code & UI Explanation)



```
// Finished
supportedCommands['cd'] = {
    executable: (parameters) : void  => {
        switch (parameters.length) {
            case 1: {
                try {
                    const cfp : TerminalFolderPointer  = currentTerminalCore.getCurrentFolderPointer();
                    cfp.gotoPath(parameters[0]);
                    currentTerminalCore.printToWindow( sentence: `Successfully went to the directory.`,  if_print_raw_to_window: false,  if_print_to_log: true);
                } catch (error) {
                    currentTerminalCore.printToWindow( sentence: `${error}`,  if_print_raw_to_window: false,  if_print_to_log: true);
                }
                break;
            }
            default: {
                currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: cd folder_name/folder_path`,  if_print_raw_to_window: false,  if_print_to_log: true);
            }
        }
    },
    description: 'Goto the given folder.\n' +
        'Usage: cd folder_name/folder_path'
};
```

This command takes the current folder pointer to another location.

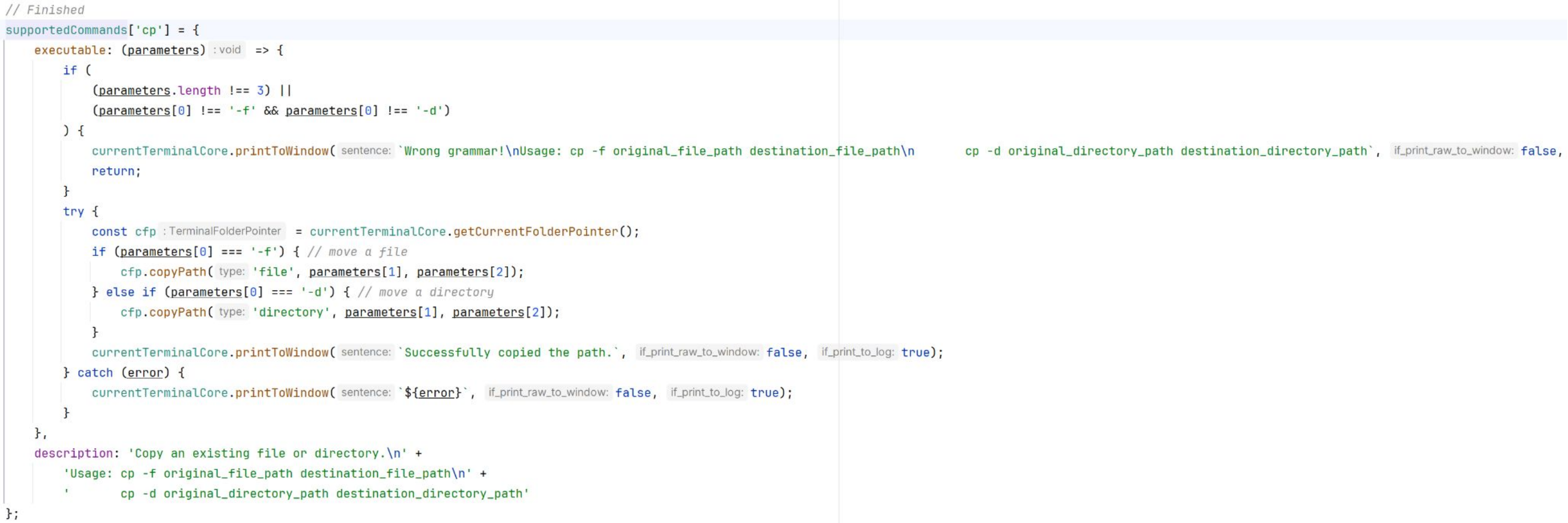
Vicayo Zhang - Screenshot 4 (Code & UI Explanation)



```
// Finished
supportedCommands['mv'] = {
    executable: (parameters) :void  => {
        if (
            (parameters.length !== 3) ||
            (parameters[0] !== '-f' && parameters[0] !== '-d')
        ) {
            currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: mv -f old_file_path new_file_path\n       mv -d old_directory_path new_directory_path`,  if_print_raw_to_window: false,  if_print_to_log: true);
            return;
        }
        try {
            const cfp :TerminalFolderPointer  = currentTerminalCore.getCurrentFolderPointer();
            if (parameters[0] === '-f') { // move a file
                // const old_file_path = parameters[1], new_file_path = parameters[2];
                cfp.movePath( type: 'file', parameters[1], parameters[2]);
            } else if (parameters[0] === '-d') { // move a directory
                // const old_directory_path = parameters[1], new_directory_path = parameters[2];
                cfp.movePath( type: 'directory', parameters[1], parameters[2]);
            }
            currentTerminalCore.printToWindow( sentence: `Successfully moved the path.`,  if_print_raw_to_window: false,  if_print_to_log: true);
        } catch (error) {
            currentTerminalCore.printToWindow( sentence: `${error}`,  if_print_raw_to_window: false,  if_print_to_log: true);
        }
    },
    description: 'mv an existing file or directory.\n' +
        'Usage: mv -f old_file_path new_file_path\n' +
        '       mv -d old_directory_path new_directory_path'
};
```

This command moves a file or folder to another location.

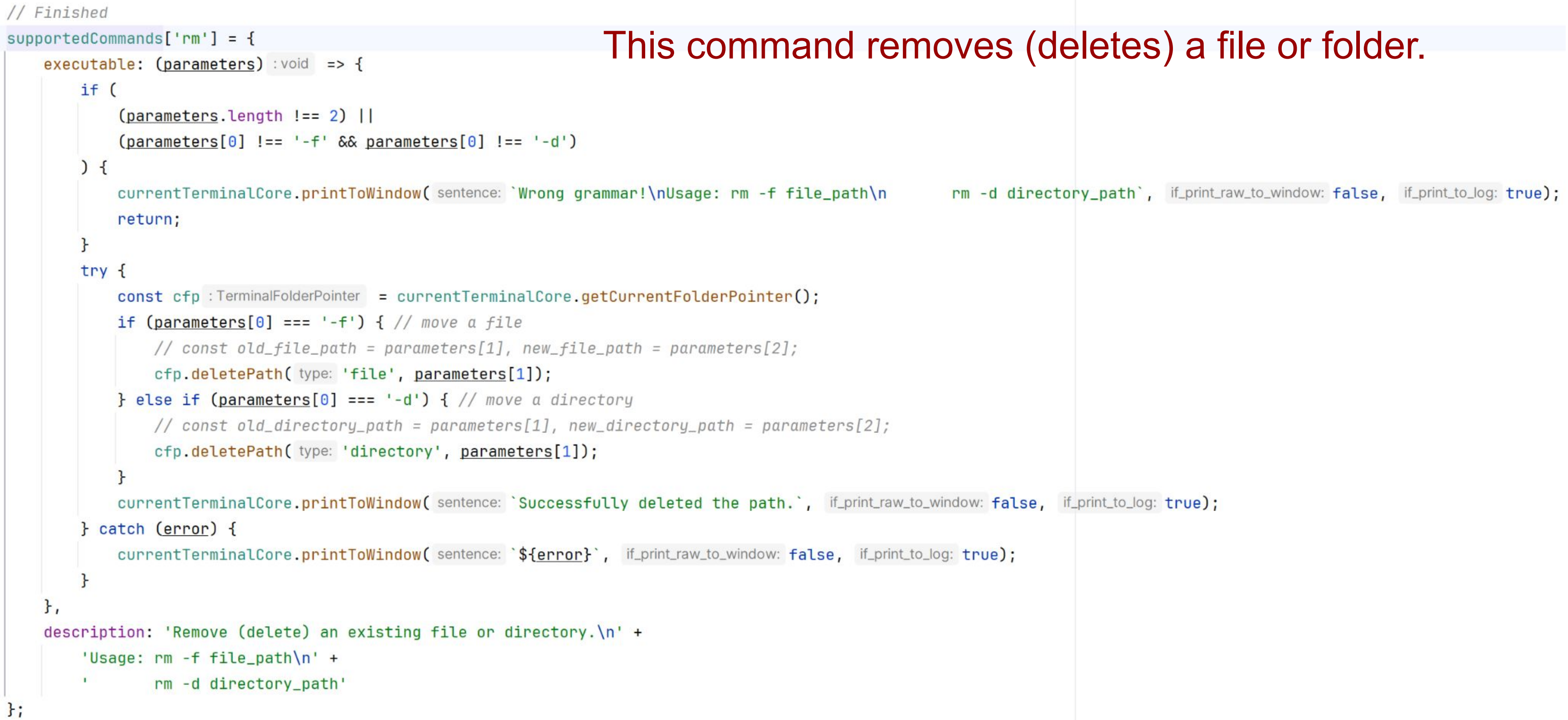
Vicayo Zhang - Screenshot 5 (Code & UI Explanation)



```
// Finished
supportedCommands['cp'] = {
    executable: (parameters) :void  => {
        if (
            (parameters.length !== 3) ||
            (parameters[0] !== '-f' && parameters[0] !== '-d')
        ) {
            currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: cp -f original_file_path destination_file_path\n       cp -d original_directory_path destination_directory_path`, if_print_raw_to_window: false,
            return;
        }
        try {
            const cfp :TerminalFolderPointer  = currentTerminalCore.getCurrentFolderPointer();
            if (parameters[0] === '-f') { // move a file
                cfp.copyPath( type: 'file', parameters[1], parameters[2]);
            } else if (parameters[0] === '-d') { // move a directory
                cfp.copyPath( type: 'directory', parameters[1], parameters[2]);
            }
            currentTerminalCore.printToWindow( sentence: `Successfully copied the path.`, if_print_raw_to_window: false, if_print_to_log: true);
        } catch (error) {
            currentTerminalCore.printToWindow( sentence: `${error}`, if_print_raw_to_window: false, if_print_to_log: true);
        }
    },
    description: 'Copy an existing file or directory.\n' +
        'Usage: cp -f original_file_path destination_file_path\n' +
        '       cp -d original_directory_path destination_directory_path'
};
```

This command copies a file or folder to another location.

Vicayo Zhang - Screenshot 6 (Code & UI Explanation)

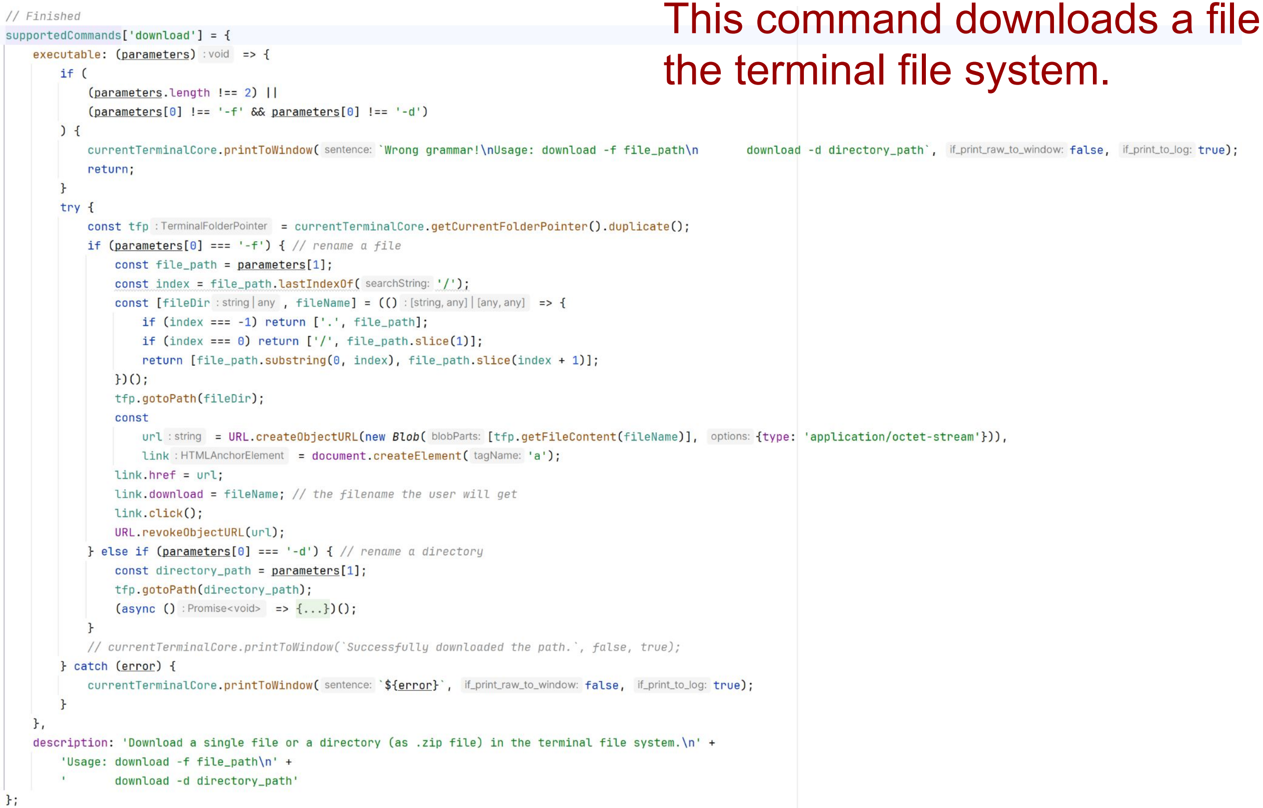


```
// Finished
supportedCommands['rm'] = {
    executable: (parameters) :void  => {
        if (
            (parameters.length !== 2) ||
            (parameters[0] !== '-f' && parameters[0] !== '-d')
        ) {
            currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: rm -f file_path\n       rm -d directory_path`,  if_print_raw_to_window: false,  if_print_to_log: true);
            return;
        }
        try {
            const cfp :TerminalFolderPointer  = currentTerminalCore.getCurrentFolderPointer();
            if (parameters[0] === '-f') { // move a file
                // const old_file_path = parameters[1], new_file_path = parameters[2];
                cfp.deletePath( type: 'file', parameters[1]);
            } else if (parameters[0] === '-d') { // move a directory
                // const old_directory_path = parameters[1], new_directory_path = parameters[2];
                cfp.deletePath( type: 'directory', parameters[1]);
            }
            currentTerminalCore.printToWindow( sentence: `Successfully deleted the path.`,  if_print_raw_to_window: false,  if_print_to_log: true);
        } catch (error) {
            currentTerminalCore.printToWindow( sentence: `${error}`,  if_print_raw_to_window: false,  if_print_to_log: true);
        }
    },
    description: 'Remove (delete) an existing file or directory.\n' +
        'Usage: rm -f file_path\n' +
        '       rm -d directory_path'
};
```

This command removes (deletes) a file or folder.

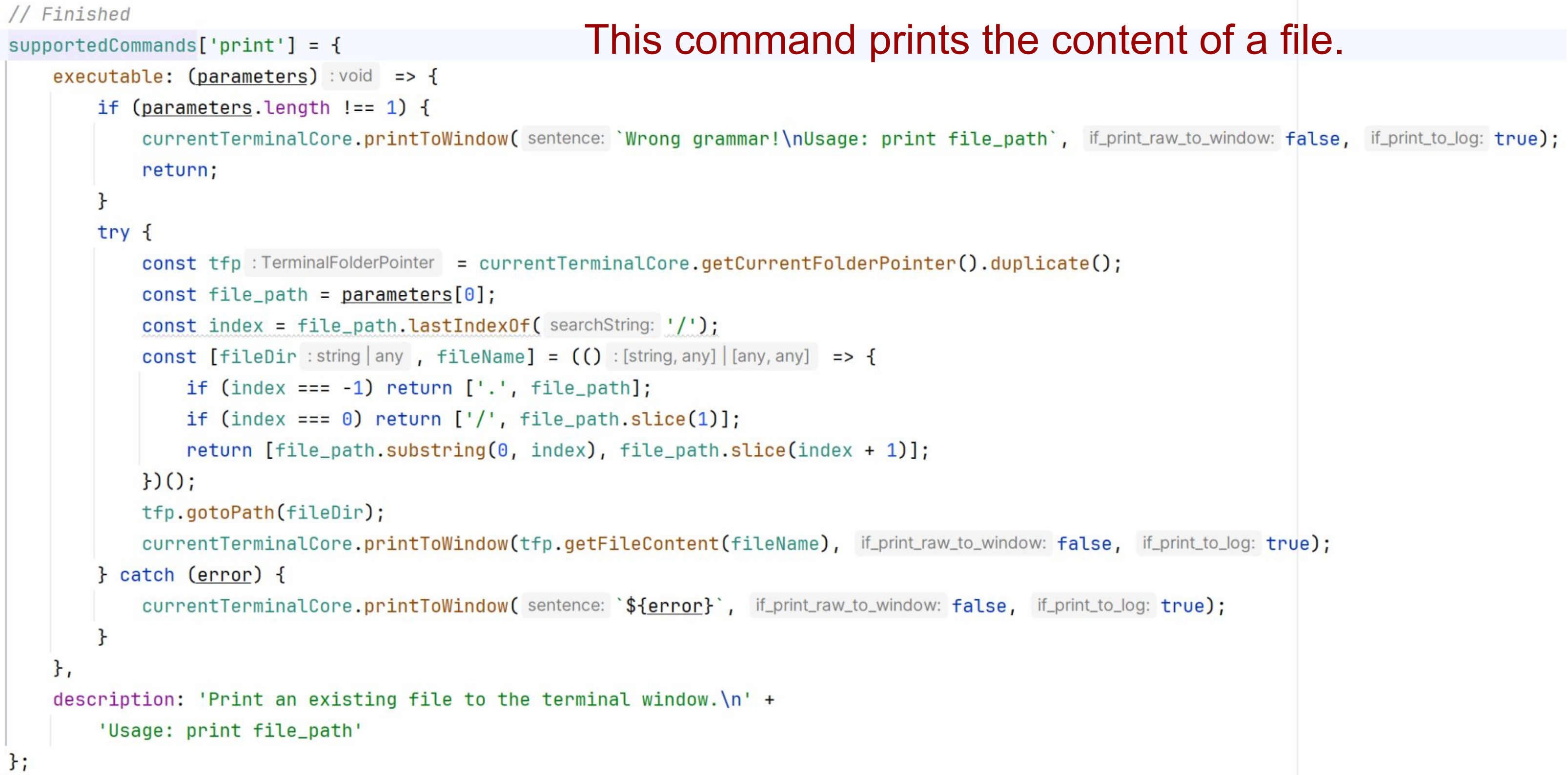
Vicayo Zhang - Screenshot 7 (Code & UI Explanation)

This command downloads a file or folder (as .zip file) from the terminal file system.



```
// Finished
supportedCommands['download'] = {
    executable: (parameters) :void  => {
        if (
            (parameters.length !== 2) ||
            (parameters[0] !== '-f' && parameters[0] !== '-d')
        ) {
            currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: download -f file_path\n       download -d directory_path`,  if_print_raw_to_window: false,  if_print_to_log: true);
            return;
        }
        try {
            const tfp :TerminalFolderPointer  = currentTerminalCore.getCurrentFolderPointer().duplicate();
            if (parameters[0] === '-f') { // rename a file
                const file_path = parameters[1];
                const index = file_path.lastIndexOf( searchString: '/');
                const [fileDir :string | any , fileName] = (() :[string, any] | [any, any]  => {
                    if (index === -1) return ['.', file_path];
                    if (index === 0) return ['/', file_path.slice(1)];
                    return [file_path.substring(0, index), file_path.slice(index + 1)];
                })();
                tfp.gotoPath(fileDir);
                const
                    url :string  = URL.createObjectURL(new Blob( blobParts: [tfp.getFileContent(fileName)],  options: {type: 'application/octet-stream'})),
                    link :HTMLAnchorElement  = document.createElement( tagName: 'a');
                link.href = url;
                link.download = fileName; // the filename the user will get
                link.click();
                URL.revokeObjectURL(url);
            } else if (parameters[0] === '-d') { // rename a directory
                const directory_path = parameters[1];
                tfp.gotoPath(directory_path);
                (async () :Promise<void>  => {...})();
            }
            // currentTerminalCore.printToWindow(`Successfully downloaded the path.`, false, true);
        } catch (error) {
            currentTerminalCore.printToWindow( sentence: `${error}`,  if_print_raw_to_window: false,  if_print_to_log: true);
        }
    },
    description: 'Download a single file or a directory (as .zip file) in the terminal file system.\n' +
        'Usage: download -f file_path\n' +
        '       download -d directory_path'
};
```

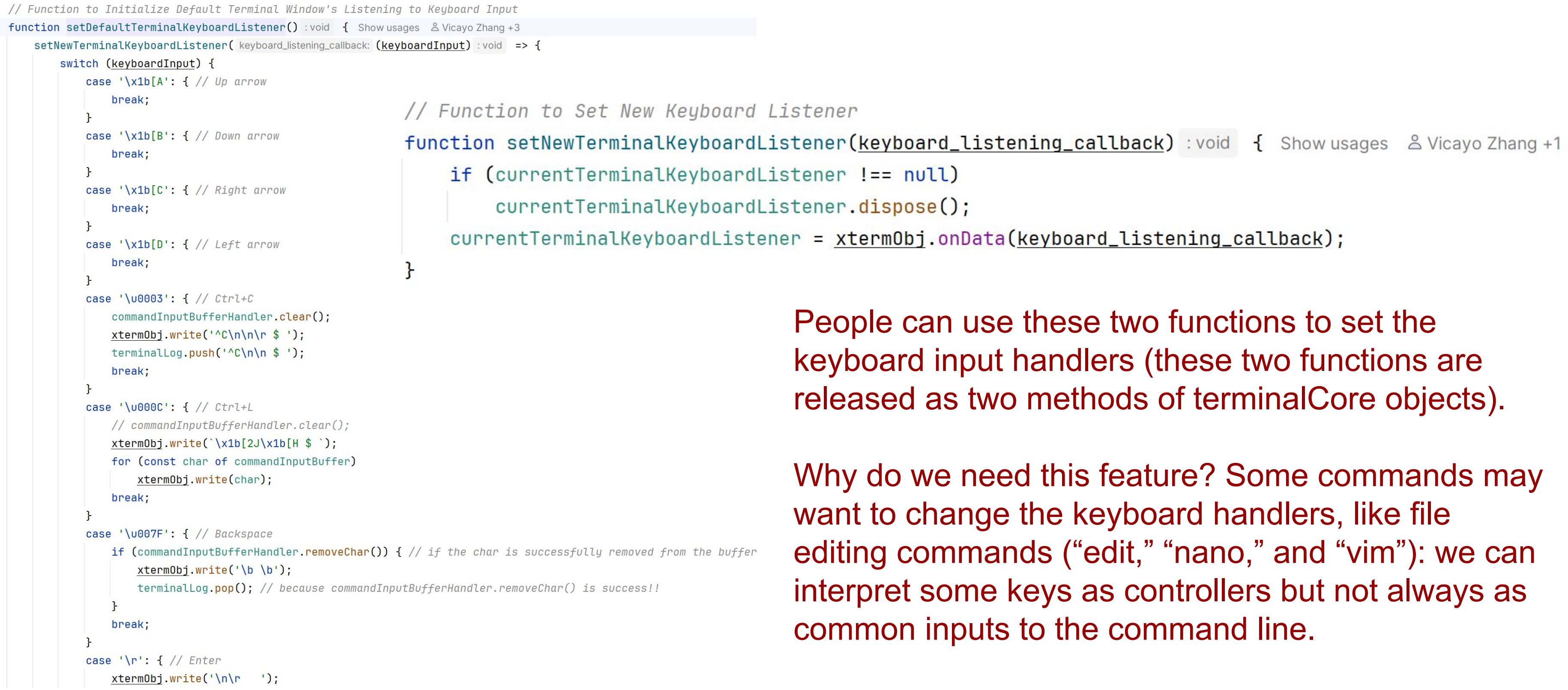
Vicayo Zhang - Screenshot 8 (Code & UI Explanation)



This command prints the content of a file.

```
// Finished
supportedCommands['print'] = {
    executable: (parameters) : void  => {
        if (parameters.length !== 1) {
            currentTerminalCore.printToWindow( sentence: `Wrong grammar!\nUsage: print file_path`,  if_print_raw_to_window: false,  if_print_to_log: true);
            return;
        }
        try {
            const tfp : TerminalFolderPointer  = currentTerminalCore.getCurrentFolderPointer().duplicate();
            const file_path = parameters[0];
            const index = file_path.lastIndexOf( searchString: '/');
            const [fileDir : string | any , fileName] = (() : [string, any] | [any, any]  => {
                if (index === -1) return ['.', file_path];
                if (index === 0) return ['/', file_path.slice(1)];
                return [file_path.substring(0, index), file_path.slice(index + 1)];
            })();
            tfp.gotoPath(fileDir);
            currentTerminalCore.printToWindow(tfp.getFileContent(fileName),  if_print_raw_to_window: false,  if_print_to_log: true);
        } catch (error) {
            currentTerminalCore.printToWindow( sentence: `${error}`,  if_print_raw_to_window: false,  if_print_to_log: true);
        }
    },
    description: 'Print an existing file to the terminal window.\n' +
        'Usage: print file_path'
};
```

Vicayo Zhang - Screenshot 9 (Code & UI Explanation)



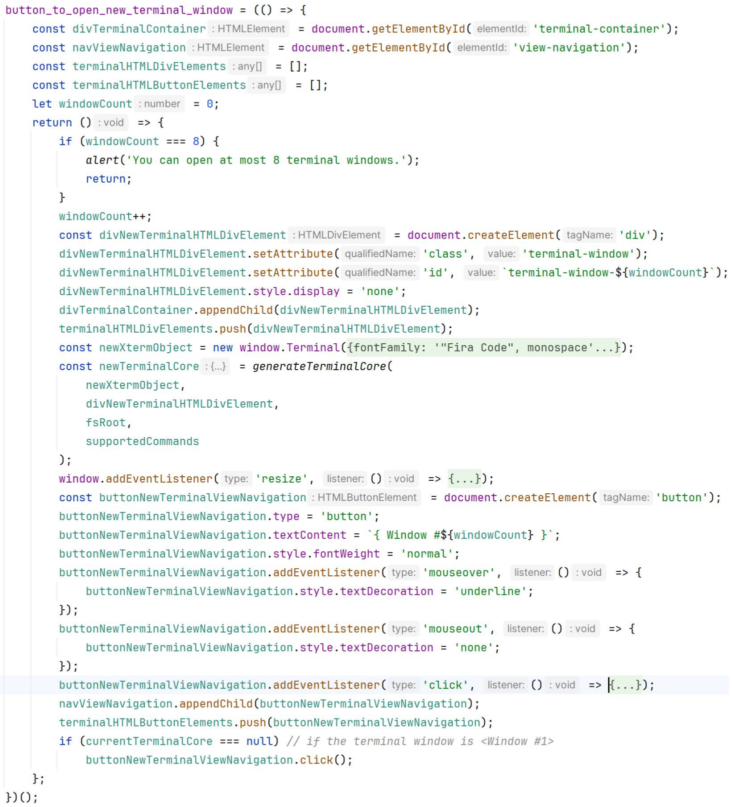
```
// Function to Initialize Default Terminal Window's Listening to Keyboard Input
function setDefaultTerminalKeyboardListener() : void  {  Show usages  Vicayo Zhang +3
    setNewTerminalKeyboardListener( keyboard_listening_callback: (keyboardInput) : void  => {
        switch (keyboardInput) {
            case '\x1b[A': { // Up arrow
                break;
            }
            case '\x1b[B': { // Down arrow
                break;
            }
            case '\x1b[C': { // Right arrow
                break;
            }
            case '\x1b[D': { // Left arrow
                break;
            }
            case '\u0003': { // Ctrl+C
                commandInputBufferHandler.clear();
                xtermObj.write('^C\n\n\r $ ');
                terminalLog.push('^C\n\n $ ');
                break;
            }
            case '\u000C': { // Ctrl+L
                // commandInputBufferHandler.clear();
                xtermObj.write(`\x1b[2J\x1b[H $ `);
                for (const char of commandInputBuffer)
                    xtermObj.write(char);
                break;
            }
            case '\u007F': { // Backspace
                if (commandInputBufferHandler.removeChar()) { // if the char is successfully removed from the buffer
                    xtermObj.write('\b \b');
                    terminalLog.pop(); // because commandInputBufferHandler.removeChar() is success!!
                }
                break;
            }
            case '\r': { // Enter
                xtermObj.write('\n\r   ');
```

```
// Function to Set New Keyboard Listener
function setNewTerminalKeyboardListener(keyboard_listening_callback) : void  {  Show usages  Vicayo Zhang +1
    if (currentTerminalKeyboardListener !== null)
        currentTerminalKeyboardListener.dispose();
    currentTerminalKeyboardListener = xtermObj.onData(keyboard_listening_callback);
}
```

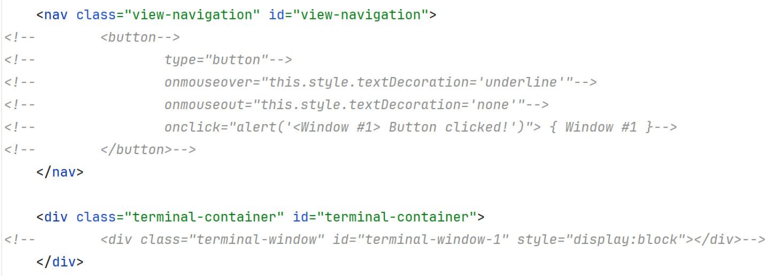
People can use these two functions to set the keyboard input handlers (these two functions are released as two methods of terminalCore objects).

Why do we need this feature? Some commands may want to change the keyboard handlers, like file editing commands (“edit,” “nano,” and “vim”): we can interpret some keys as controllers but not always as common inputs to the command line.

Vicayo Zhang - Screenshot 10 (Code & UI Explanation)



```
button_to_open_new_terminal_window = (() => {
    const divTerminalContainer : HTMLElement  = document.getElementById( elementId: 'terminal-container');
    const navViewNavigation : HTMLElement  = document.getElementById( elementId: 'view-navigation');
    const terminalHTMLDivElements : any[]  = [];
    const terminalHTMLButtonElements : any[]  = [];
    let windowCount : number  = 0;
    return () : void  => {
        if (windowCount === 8) {
            alert('You can open at most 8 terminal windows.');
            return;
        }
        windowCount++;
        const divNewTerminalHTMLDivElement : HTMLDivElement  = document.createElement( tagName: 'div');
        divNewTerminalHTMLDivElement.setAttribute( qualifiedName: 'class',  value: 'terminal-window');
        divNewTerminalHTMLDivElement.setAttribute( qualifiedName: 'id',  value: `terminal-window-${windowCount}`);
        divNewTerminalHTMLDivElement.style.display = 'none';
        divTerminalContainer.appendChild(divNewTerminalHTMLDivElement);
        terminalHTMLDivElements.push(divNewTerminalHTMLDivElement);
        const newXtermObject = new window.Terminal({fontFamily: '"Fira Code", monospace'...});
        const newTerminalCore : {...}  = generateTerminalCore(
            newXtermObject,
            divNewTerminalHTMLDivElement,
            fsRoot,
            supportedCommands
        );
        window.addEventListener( type: 'resize',  listener: () : void  => {...});
        const buttonNewTerminalViewNavigation : HTMLButtonElement  = document.createElement( tagName: 'button');
        buttonNewTerminalViewNavigation.type = 'button';
        buttonNewTerminalViewNavigation.textContent = `{ Window #${windowCount} }`;
        buttonNewTerminalViewNavigation.style.fontWeight = 'normal';
        buttonNewTerminalViewNavigation.addEventListener( type: 'mouseover',  listener: () : void  => {
            buttonNewTerminalViewNavigation.style.textDecoration = 'underline';
        });
        buttonNewTerminalViewNavigation.addEventListener( type: 'mouseout',  listener: () : void  => {
            buttonNewTerminalViewNavigation.style.textDecoration = 'none';
        });
        buttonNewTerminalViewNavigation.addEventListener( type: 'click',  listener: () : void  => {...});
        navViewNavigation.appendChild(buttonNewTerminalViewNavigation);
        terminalHTMLButtonElements.push(buttonNewTerminalViewNavigation);
        if (currentTerminalCore === null) // if the terminal window is <Window #1>
            buttonNewTerminalViewNavigation.click();
    };
})();
```



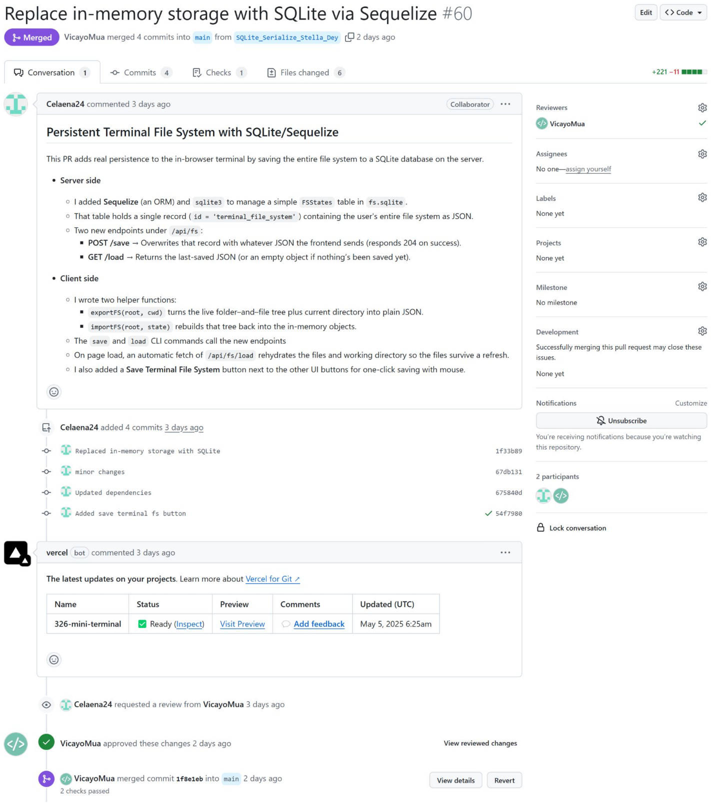
```
    <nav class="view-navigation" id="view-navigation">
<!--        <button-->
<!--                type="button"-->
<!--                onmouseover="this.style.textDecoration='underline'"-->
<!--                onmouseout="this.style.textDecoration='none'"-->
<!--                onclick="alert('<Window #1> Button clicked!')"> { Window #1 }-->
<!--        </button>-->
    </nav>

    <div class="terminal-container" id="terminal-container">
<!--        <div class="terminal-window" id="terminal-window-1" style="display:block"></div>-->
    </div>
```

This function “opens” new terminal window tabs and sets up new terminal cores and navigation buttons.

In each terminal window, the terminal core uses the same file system but can take different keyboard input handlers.

When the navigation button is clicked, the view will be switched to the corresponding terminal tab, and the font of the navigation button will be bold.

Vicayo Zhang - Screenshot 11 (Code & UI Explanation)



Replace in-memory storage with SQLite via Sequelize #60

Merged VicayoMua merged 4 commits into `main` from `SQLite_Serialize_Stella_Dey` 2 days ago

Celaena24 commented 3 days ago

Persistent Terminal File System with SQLite/Sequelize

This PR adds real persistence to the in-browser terminal by saving the entire file system to a SQLite database on the server.

- **Server side**
  - I added **Sequelize** (an ORM) and `sqlite3` to manage a simple `FSStates` table in `fs.sqlite`.
  - That table holds a single record (`id = 'terminal_file_system'`) containing the user's entire file system as JSON.
  - Two new endpoints under `/api/fs`:
    - **POST /save** → Overwrites that record with whatever JSON the frontend sends (responds 204 on success).
    - **GET /load** → Returns the last-saved JSON (or an empty object if nothing's been saved yet).
- **Client side**
  - I wrote two helper functions:
    - `exportFS(root, cwd)` turns the live folder–and–file tree plus current directory into plain JSON.
    - `importFS(root, state)` rebuilds that tree back into the in-memory objects.
  - The `save` and `load` CLI commands call the new endpoints
  - On page load, an automatic fetch of `/api/fs/load` rehydrates the files and working directory so the files survive a refresh.
  - I also added a **Save Terminal File System** button next to the other UI buttons for one-click saving with mouse.

Celaena24 added 4 commits 3 days ago

```
Replaced in-memory storage with SQLite      1f33b89
minor changes                               67db131
Updated dependencies                        675840d
Added save terminal fs button             ✓ 54f7980
```

vercel bot commented 3 days ago

The latest updates on your projects. Learn more about Vercel for Git ↗

Name	Status	Preview	Comments	Updated (UTC)
326-mini-terminal	✅ Ready (Inspect)	Visit Preview	💬 Add feedback	May 5, 2025 6:25am

Celaena24 requested a review from VicayoMua 3 days ago

VicayoMua approved these changes 2 days ago

VicayoMua merged commit `1f8e1eb` into `main` 2 days ago
2 checks passed

Reviewers
VicayoMua ✓

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
You're receiving notifications because you're watching this repository.

2 participants

As a project manager who designs the core services of the project, I reviewed all the pull request since the last milestone, solve all the conflicts, and merge all the relevant code branches.

Vicayo Zhang - Challenges and Insights

Reflections on Challenges I Faced:

- At first, it’s very difficult to implement the file system commands, such as cd, mv, rm, and cp, mainly because I need to access the real directory tree in the command closure.
- After the terminal folder pointer class has more powerful methods, such as gotoPath(), movePath(), copyPath(), and deletePath(), the problem is solved.

Insights and Takeaways from Collaborative Teamwork:

- GitHub Issues and Pull Requests can help divide the team workload and organize the team work.
- Regular communication can help solve integration issues quickly.
- Following consistent coding standards (naming, commenting) makes merging different parts easier.

Vicayo Zhang - Future Improvements

- Add the command “edit” to edit files in the terminal file system
- Add the command “webass” to support WebAssembly files.
- Revise the draft code for the commands, such as wget and ping…

Aryan Ghosh - Assigned Work Summary

Tasks Completed

- Designed and implemented full-stack integration for edit command
- Built modal-based text editor for modifying virtual file system contents
- Implemented async saveFSState() logic to persist edits to SQLite via backend API
- Refactored front-end button handlers to support JavaScript module scope (type="module")
- Debugged and resolved import/module issues in modularized JS setup
- Diagnosed and fixed broken UI behavior due to scope mismatch between inline handlers and ES module functions
- Validated working terminal UI with functional edit, save, and print capabilities

PR Closed: https://github.com/VicayoMua/326_Mini_Terminal/pull/75

Files Contributed To

- src/terminal_setup_core_and_commands.js
  - Added edit command with frontend modal integration
  - Hooked command into terminal input parser
  - Patched button_to_* logic for modular scope
- src/editor_utils.js (new file created)
  - Added showEditor() modal with save callback
  - Implemented saveFSState() via fetch('/api/fs/save')
- Index.html
  - Switched <script> to type="module" to enable ES imports

Aryan Ghosh - Feature Demonstration

Feature Implemented: edit Command

The edit command allows users to open and modify any text-based file directly in the terminal UI via a modal text editor. Changes made in the editor are:

- Instantly reflected in the in-memory virtual file system
- Persisted to the SQLite backend using the /api/fs/save endpoint

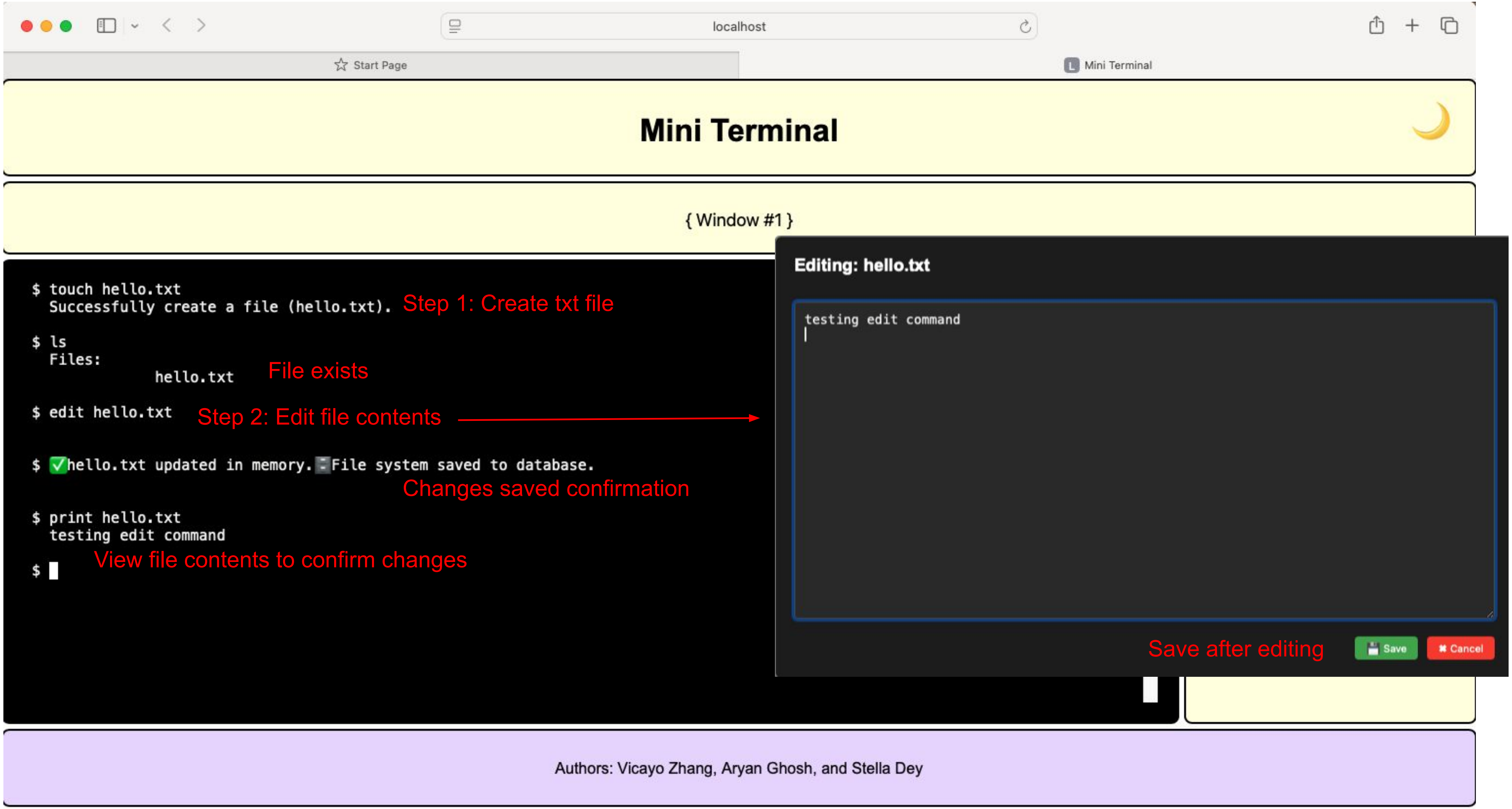
This enables seamless editing and saving of terminal-based files without needing external tools.

Branch Name: ping-Aryan

- **Frontend command logic** (terminal_setup_core_and_commands.js)
- **Modal UI logic** (editor_utils.js)
- **Backend persistence** (server.js, /api/fs/save)

Layer	Status	Details
Frontend UI	Complete	Modal-based editor with editable <textarea>, Save button, and real-time feedback
Command Parsing	Complete	edit <file_path> registered and fully integrated into supportedCommands
File Editing Logic	Complete	Handles both read and update operations in the virtual FS
Backend Integration	Complete	Uses fetch() to call /api/fs/save with serialized FS state
Persistence to DB	Complete	SQLite backend via server.js and Sequelize ORM

Aryan Ghosh - Feature Demonstration (UI Screenshot)



Mini Terminal

{ Window #1 }

```
$ touch hello.txt
  Successfully create a file (hello.txt).

$ ls
  Files:
              hello.txt

$ edit hello.txt


$ ✅hello.txt updated in memory.💾File system saved to database.


$ print hello.txt
  testing edit command

$ █
```

Step 1: Create txt file

File exists

Step 2: Edit file contents

Changes saved confirmation

View file contents to confirm changes

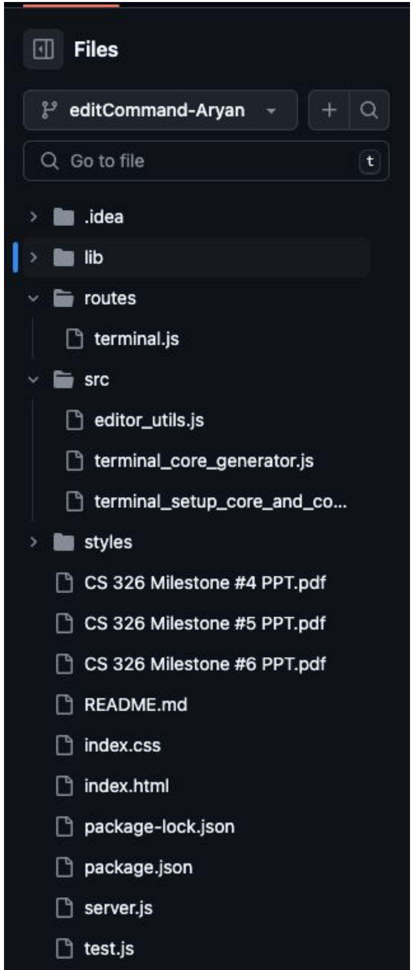
Editing: hello.txt

```
testing edit command
|
```

Save after editing

Authors: Vicayo Zhang, Aryan Ghosh, and Stella Dey

Aryan Ghosh - Code Structure & Organization



```
.idea
lib
routes
  terminal.js
src
  editor_utils.js
  terminal_core_generator.js
  terminal_setup_core_and_co...
styles
CS 326 Milestone #4 PPT.pdf
CS 326 Milestone #5 PPT.pdf
CS 326 Milestone #6 PPT.pdf
README.md
index.css
index.html
package-lock.json
package.json
server.js
test.js
```

Front-End & Back-End Separation

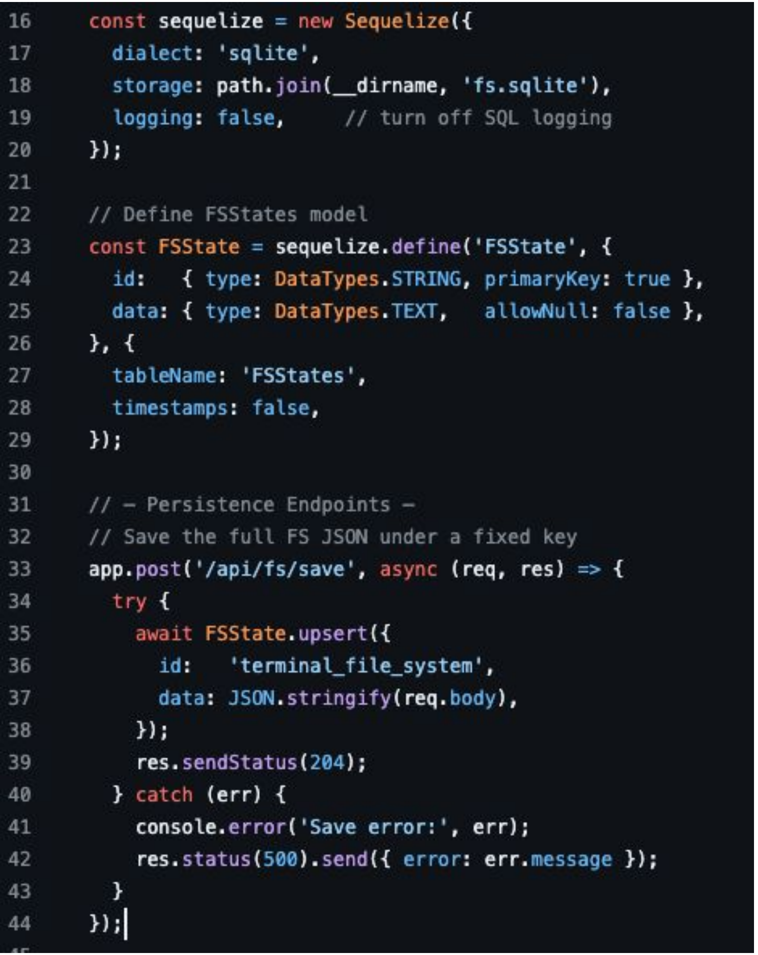
- **Front-end files are located in the src/, styles/, and root-level index.html:**
  - src/terminal_setup_core_and_commands.js: main command interface logic, terminal behavior
  - src/editor_utils.js: modal-based file editing and save logic
  - index.html, index.css: static UI and styling

- **Back-End Logic is encapsulated in:**
  - server.js: Express server with RESTful endpoints (/api/fs/save, /api/fs/load)
  - routes/terminal.js: backend route for shell command execution (ping integration)

Component Labeling & Placement

- All terminal UI logic is grouped under src/ for clarity and modularity
- editor_utils.js is introduced as a focused utility module for editing functionality
- routes/ houses API route logic (e.g., terminal.js handles server-side command execution securely)
- Naming conventions (e.g., terminal_core_generator.js, editor_utils.js) clearly describe the file's role

Aryan Ghosh - Back-End Implementation



```
const sequelize = new Sequelize({
  dialect: 'sqlite',
  storage: path.join(__dirname, 'fs.sqlite'),
  logging: false,     // turn off SQL logging
});

// Define FSStates model
const FSState = sequelize.define('FSState', {
  id:   { type: DataTypes.STRING, primaryKey: true },
  data: { type: DataTypes.TEXT,   allowNull: false },
}, {
  tableName: 'FSStates',
  timestamps: false,
});

// — Persistence Endpoints —
// Save the full FS JSON under a fixed key
app.post('/api/fs/save', async (req, res) => {
  try {
    await FSState.upsert({
      id:   'terminal_file_system',
      data: JSON.stringify(req.body),
    });
    res.sendStatus(204);
  } catch (err) {
    console.error('Save error:', err);
    res.status(500).send({ error: err.message });
  }
});
```

- **Key File:** server.js
- **What It Does:**
  - Sets up Sequelize to use a local SQLite database (fs.sqlite)
  - Defines a model (FSState) to store the entire virtual file system as a single JSON object
  - Provides two API endpoints:
    - POST /api/fs/save to save the updated file system after editing
    - GET /api/fs/load to load the file system on app startup
- **Code Snippet Shown in Slide:**
  - Sequelize initialization
  - FSState model definition
  - /api/fs/save route to persist the FS after editing a file
- **Integration with Front-End:**
  - The edit command triggers saveFSState() in editor_utils.js
  - That function sends a POST request to /api/fs/save with the full file system tree
  - Backend receives and stores the updated state in SQLite
  - Data is restored on reload via /api/fs/load
- **How SQLite & Sequelize Are Used:**
  - Sequelize is configured with SQLite as the storage engine
  - FSState is the model that maps to the FSStates table in fs.sqlite
  - File system data is stored as a JSON string in the data field
  - Sequelize handles all DB operations with upsert(), findByPk(), and sync()
- **Visible Impact on UI:**
  - After saving a file via the editor modal, the terminal displays:
    - ✅ File updated in memory
    - 🗄️ File system saved to database
- **Challenges & Solutions:**
  - *Challenge:* Needed a way to persist deeply nested virtual FS
    Solution: Used Sequelize with a single JSON blob in SQLite
  - *Challenge:* Express server didn't accept incoming JSON
    Solution: Added express.json() middleware
  - *Challenge:* Confirming FS saved properly
    Solution: Console logs + DB inspection + visible confirmation in terminal

Aryan Ghosh - Challenges and Insights

Technical Challenges Faced

- Encountered a SyntaxError due to import statements not working outside of module scope
  → **Resolved by adding** type="module" to the script tag in index.html
- Inline onclick button handlers stopped working after switching to ES modules
  → **Solution:** Reattached button event listeners programmatically using querySelector(...).onclick
- Edits made in the file editor weren't persisting between sessions
  → **Implemented backend integration** using fetch('/api/fs/save') and a Sequelize-powered SQLite database
- Needed to persist a nested in-memory file system object
  → **Solved by serializing** the entire FS object as a single JSON blob and storing it using Sequelize

Lessons & Insights

- Learned how to build full-stack features that connect terminal commands, modal UI, and backend storage
- Gained experience in structuring ES module–based JavaScript for better code organization and scalability
- Realized the importance of modularizing responsibilities (e.g., editor logic in editor_utils.js)
- Understood how to use Sequelize models effectively for non-relational JSON storage in SQLite

Team Collaboration Takeaways

- Practiced clear **task ownership** (e.g. owning the edit command from frontend to backend)
- Regularly merged changes with the main branch to avoid stale code and conflicts
- Collaborated through pull requests and GitHub issue tracking, ensuring transparency and accountability
- Learned to debug across multiple layers (UI → logic → API → DB), improving full-stack problem-solving skills

Aryan Ghosh - Future Improvements

1. **Error Logging System for Debugging**

   Add a centralized client-side error logger that logs failed commands, API calls, or file system actions to a downloadable log file

   Github issue: https://github.com/VicayoMua/326_Mini_Terminal/issues/76

2. **Modularization and Cleanup of Terminal Command Logic**

   Move each terminal command (edit, touch, print, etc.) to separate modules/files for better maintainability and scalability

   Github issue: https://github.com/VicayoMua/326_Mini_Terminal/issues/77

Areas of Technical Debt / Optimization

- The save/load process does not validate schema — JSON corruption could break the app
- No validation exists for file size or content when editing or uploading files

Stella Dey - Assigned Work Summary

Assigned Issues:

- #69: Sqlite & Serialize

https://github.com/VicayoMua/326_Mini_Terminal/issues/69

Completed tasks:

Integrated Sequelize and SQLite on the server:

- Installed sequelize and sqlite3
- Configured a Sequelize instance pointing at fs.sqlite
- Defined an FSState model (id PK, data TEXT) and synced it on startup

Added two new REST endpoints under /api/fs:

- POST /save – receives the serialized filesystem JSON and upserts it into FSStates
- GET /load – retrieves and parses that JSON (returns an empty object if none exists)

Built frontend serialization helpers:

- serializeFolder – deep‑copies the in‑memory folder tree into plain objects
- exportFS(root, cwd) – packages the folder tree plus current directory into JSON
- importFS(root, state) – rebuilds the live folder/file objects from saved JSON

Enhanced the UI:

- Added a “Save Terminal File System” button in index.html
- Wired its onclick to invoke the same save command, giving one‑click persistence

Closed PRs:

- Persistent Terminal file system with SQLite/Serialize

  https://github.com/VicayoMua/326_Mini_Terminal/pull/60

Commits Authored:

- 1f33b89: Replaced in-memory storage with SQLite

https://github.com/VicayoMua/326_Mini_Terminal/pull/60/commits/1f33b8955673083231324cddde753f3feaf7e7b0

- 67db131: Minor changes

https://github.com/VicayoMua/326_Mini_Terminal/pull/60/commits/67db131026b081187cd4d46723f2ecfbe2276c7d

- 675840d: Updated dependencies

https://github.com/VicayoMua/326_Mini_Terminal/pull/60/commits/675840d5b9ba1d628415fa97454802fe2c3c08f5

- 54f7980: Added save terminal fs button

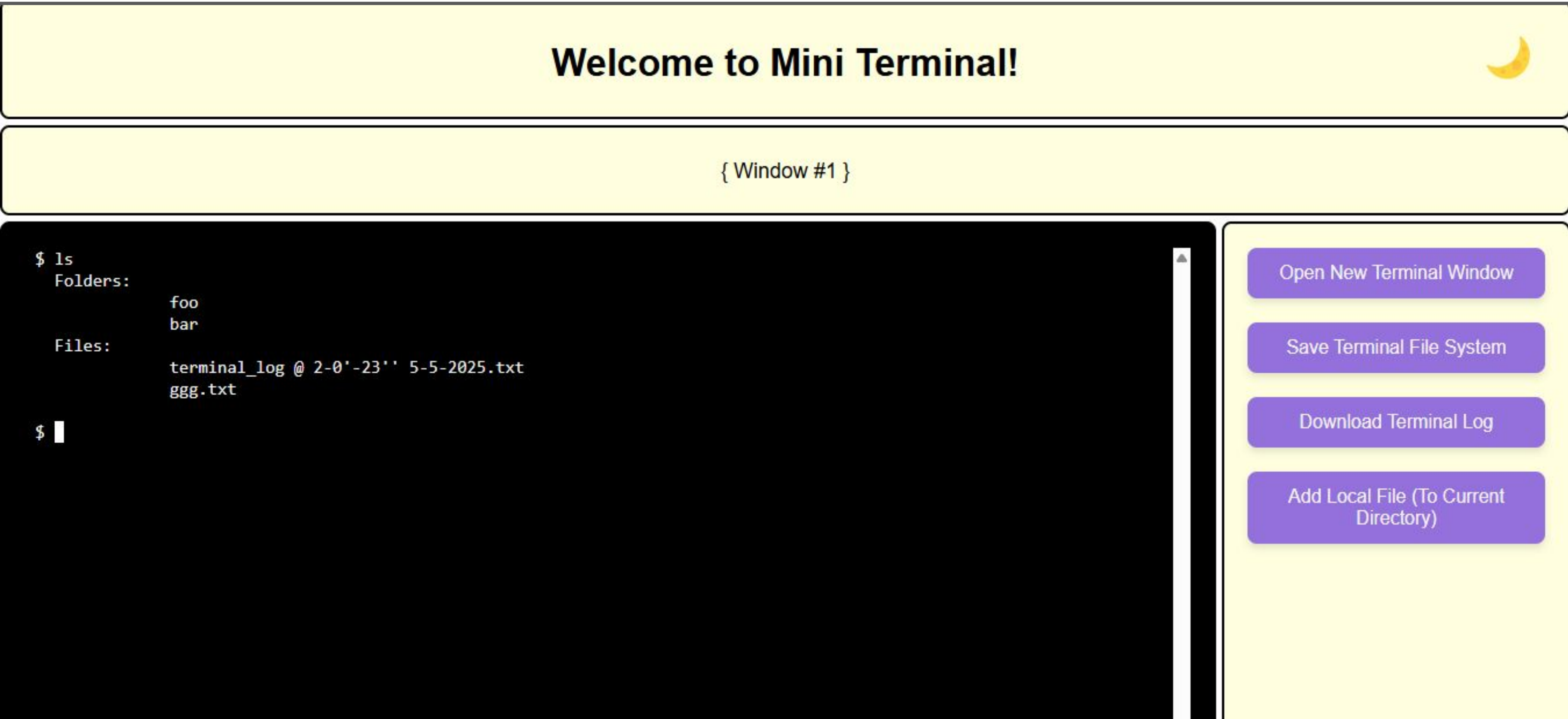
https://github.com/VicayoMua/326_Mini_Terminal/pull/60/commits/54f79808f706eddec4e0cf75fdc05caf408b7deb

Feature Demonstration

- **Feature implemented:** Save & reload the in-browser terminal’s file system via a SQLite database using Sequelize.
- **Completion Progress:**
  - Front-end (4/4):
    - exportFS & importFS helpers
    - save & load CLI commands
    - Automatic load on page startup
    - “Save Terminal File System” UI button
  - Back-end (3/3):
    - FSState Sequelize model
    - POST /api/fs/save endpoint
    - GET /api/fs/load endpoint
- **Git Branch:** SQLite_Serlialize_Stella_Dey

  https://github.com/VicayoMua/326_Mini_Terminal/tree/SQLite_Serialize_Stella_Dey

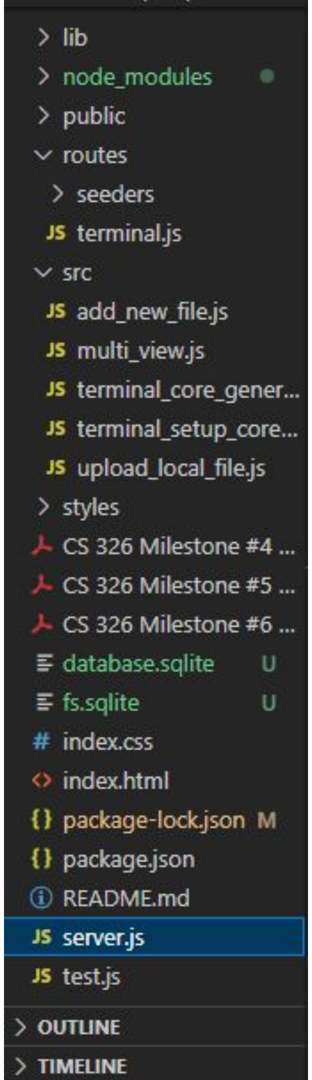
Stella Dey - Screenshot



Welcome to Mini Terminal!
{ Window #1 }
Open New Terminal Window
Save Terminal File System
Download Terminal Log
Add Local File (To Current Directory)

```
$ ls
  Folders:
              foo
              bar
  Files:
              terminal_log @ 2-0'-23'' 5-5-2025.txt
              ggg.txt

$ █
```

Code Structure Screenshot



```
> lib
> node_modules
> public
v routes
  > seeders
  terminal.js
v src
  add_new_file.js
  multi_view.js
  terminal_core_gener...
  terminal_setup_core...
  upload_local_file.js
> styles
CS 326 Milestone #4 ...
CS 326 Milestone #5 ...
CS 326 Milestone #6 ...
database.sqlite        U
fs.sqlite              U
index.css
index.html
package-lock.json      M
package.json
README.md
server.js
test.js
> OUTLINE
> TIMELINE
```

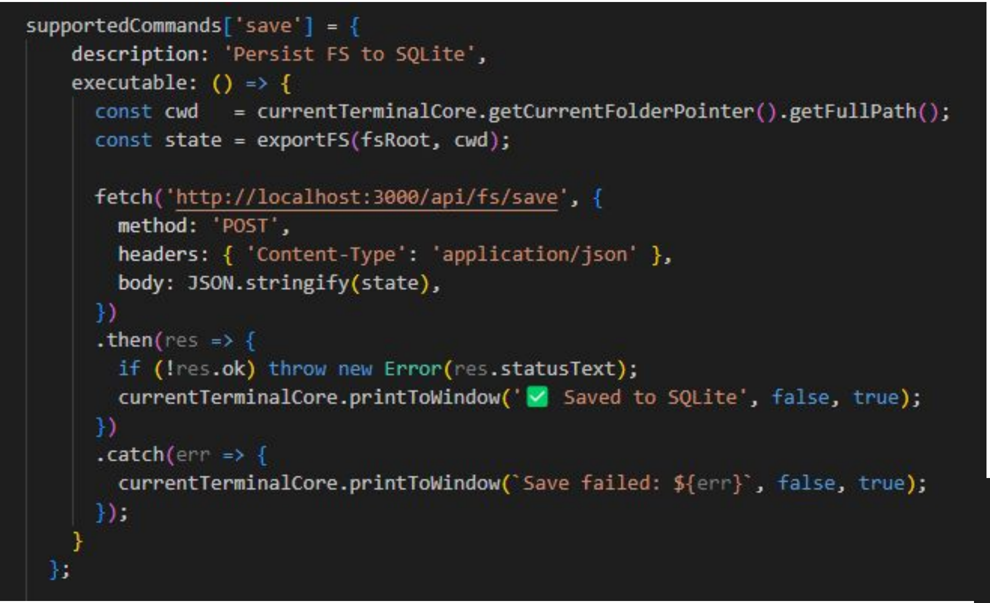
Stella Dey - Code Structure and Organization

Code Structure & Organization

- **Directory Layout**
  - **routes/**
    - terminal.js – all Express API endpoints (/ping, /proxy, /fs/save, /fs/load)
  - **server.js** – bootstraps Express, CORS, Sequelize/SQLite and mounts routes/ under /api
  - **Front-end files** (served statically)
    - index.html, index.css – UI skeleton & styling
    - js/
      - terminal_core_generator.js – initializes the xterm window & keyboard listeners
      - terminal_setup_core_and_commands.js – defines exportFS/importFS, CLI commands, and links UI buttons
      - other helpers (add_new_file.js, multi_view.js, upload_local_file.js, etc.)
- **Front-End vs. Back-End Separation**
  - **Front-End** lives entirely in the static src/ files and HTML/CSS; it handles command parsing, terminal rendering, and fetch calls.
  - **Back-End** is pure Node.js in server.js and routes/; it exposes REST endpoints and manages persistence via Sequelize/SQLite.
- **Key Components & Their Locations**
  - **Serialization Helpers** (exportFS, importFS): in terminal_setup_core_and_commands.js
  - **Terminal Core** (generateTerminalCore): in terminal_core_generator.js
  - **API Routes** (/api/fs/save, /api/fs/load): in routes/terminal.js
  - **Sequelize Model** (FSState): defined and synced in server.js

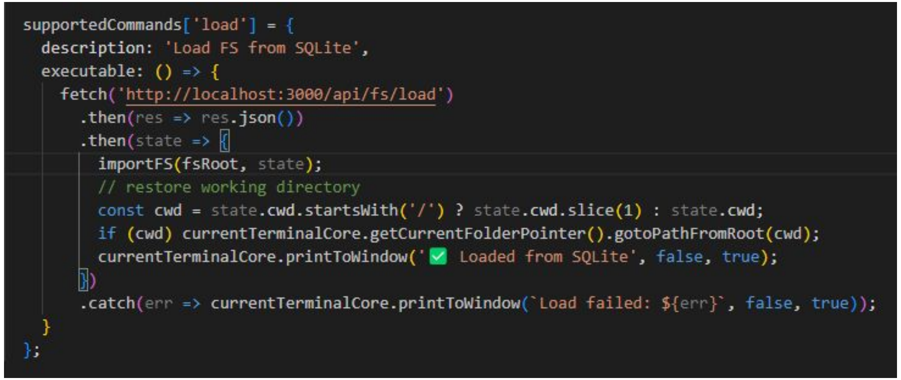
This organization cleanly separates UI logic from data persistence and makes it easy to locate each feature in the codebase.

Stella Dey - Frontend Implementation (Screenshots)



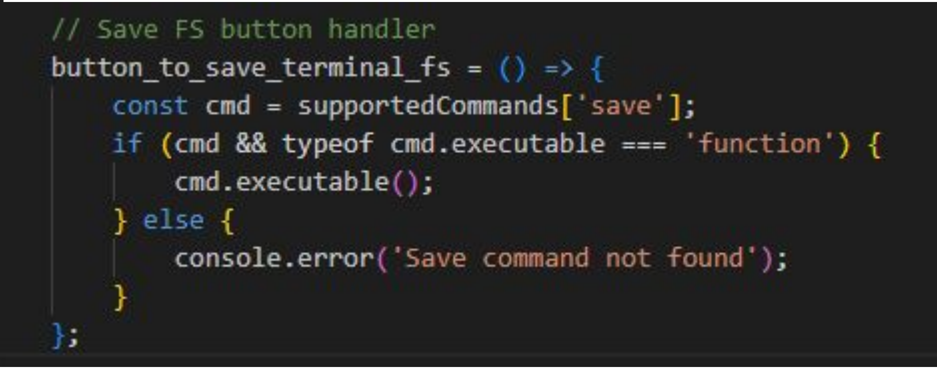
```
supportedCommands['save'] = {
    description: 'Persist FS to SQLite',
    executable: () => {
      const cwd   = currentTerminalCore.getCurrentFolderPointer().getFullPath();
      const state = exportFS(fsRoot, cwd);

      fetch('http://localhost:3000/api/fs/save', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(state),
      })
      .then(res => {
        if (!res.ok) throw new Error(res.statusText);
        currentTerminalCore.printToWindow('✅ Saved to SQLite', false, true);
      })
      .catch(err => {
        currentTerminalCore.printToWindow(`Save failed: ${err}`, false, true);
      });
    }
  };
```

Save command



```
supportedCommands['load'] = {
  description: 'Load FS from SQLite',
  executable: () => {
    fetch('http://localhost:3000/api/fs/load')
      .then(res => res.json())
      .then(state => {
        importFS(fsRoot, state);
        // restore working directory
        const cwd = state.cwd.startsWith('/') ? state.cwd.slice(1) : state.cwd;
        if (cwd) currentTerminalCore.getCurrentFolderPointer().gotoPathFromRoot(cwd);
        currentTerminalCore.printToWindow('✅ Loaded from SQLite', false, true);
      })
      .catch(err => currentTerminalCore.printToWindow(`Load failed: ${err}`, false, true));
  }
};
```

load command



```
// Save FS button handler
button_to_save_terminal_fs = () => {
    const cmd = supportedCommands['save'];
    if (cmd && typeof cmd.executable === 'function') {
        cmd.executable();
    } else {
        console.error('Save command not found');
    }
};
```

Save button

Stella Dey - Frontend Implementation

UI Integration

- Added a **Save Terminal File System** button in index.html
- Button’s onclick="button_to_save_terminal_fs()" invokes the same supportedCommands['save'] logic

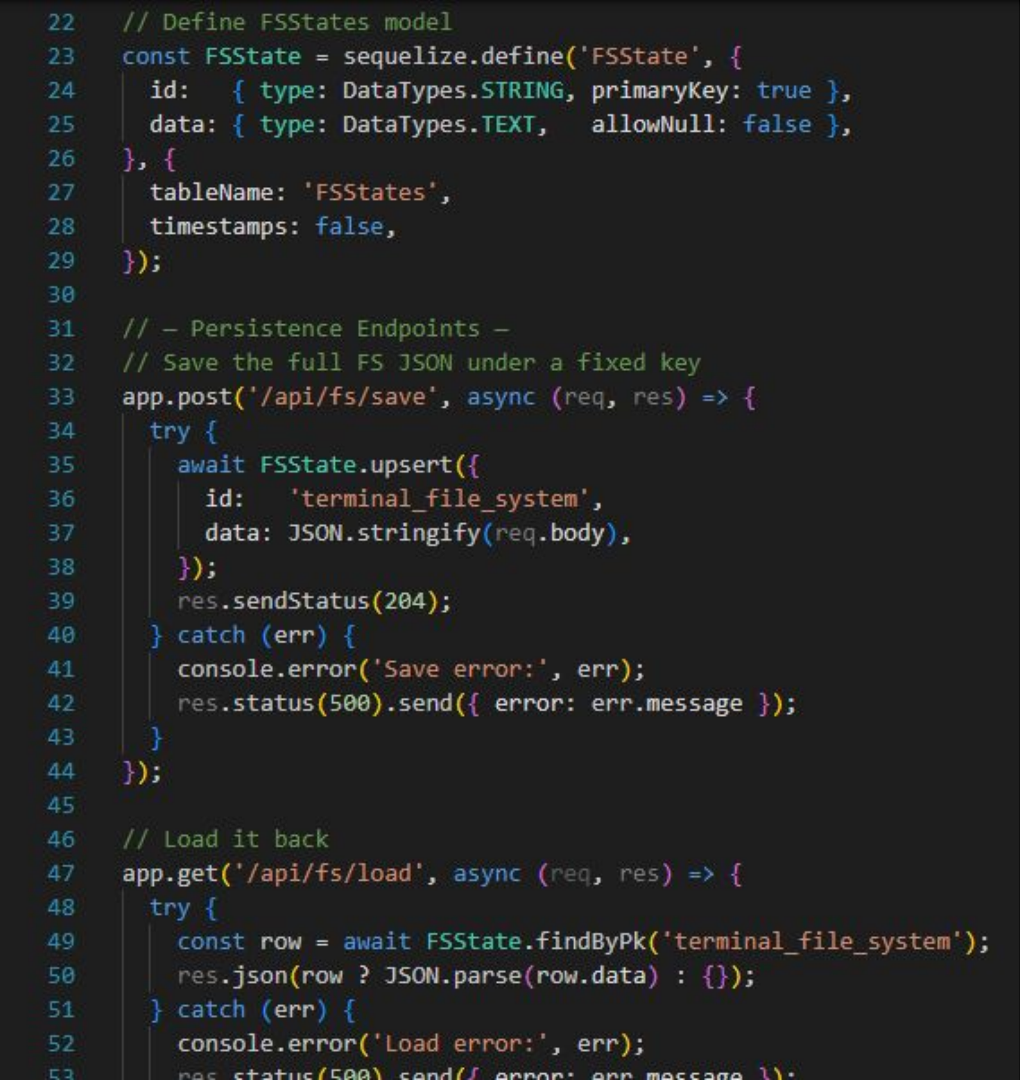
How It Fits Together

1. **exportFS() / importFS()** convert the in-memory folder/file graph to/from JSON
2. CLI commands and UI buttons call fetch to hit our backend REST endpoints
3. Front-end remains purely responsible for state serialization and user interaction

Challenges & Solutions

- **Recursive Serialization:** Turning nested FolderObjects into plain JSON → solved with serializeFolder() helper
- **Startup Hydration:** Needing to rebuild state before user input → solved by an async IIFE that calls /api/fs/load on page load and runs importFS

Stella Dey - Backend Implementation (Screenshot)



```
// Define FSStates model
const FSState = sequelize.define('FSState', {
  id:   { type: DataTypes.STRING, primaryKey: true },
  data: { type: DataTypes.TEXT,   allowNull: false },
}, {
  tableName: 'FSStates',
  timestamps: false,
});

// — Persistence Endpoints —
// Save the full FS JSON under a fixed key
app.post('/api/fs/save', async (req, res) => {
  try {
    await FSState.upsert({
      id:   'terminal_file_system',
      data: JSON.stringify(req.body),
    });
    res.sendStatus(204);
  } catch (err) {
    console.error('Save error:', err);
    res.status(500).send({ error: err.message });
  }
});

// Load it back
app.get('/api/fs/load', async (req, res) => {
  try {
    const row = await FSState.findByPk('terminal_file_system');
    res.json(row ? JSON.parse(row.data) : {});
  } catch (err) {
    console.error('Load error:', err);
    res.status(500).send({ error: err.message });
```

Stella Dey - Backend Implementation

Architecture & Organization

- **Models**
  - FSState in server.js (could be moved to models/)
- **Routes & Controllers**
  - All API logic in routes/terminal.js (mounted at /api)
- **Middleware & Initialization**
  - app.use(cors()), app.use(express.json())
  - ;(async ()=>{ await sequelize.sync(); app.listen(...) })()

Integration & Impact

- Express app listens on port 3000
- Front-end fetch calls (/api/fs/save, /api/fs/load) reach these handlers
- The database file fs.sqlite stores the single JSON blob

Challenges & Solutions

- **Upsert semantics**: used Model.upsert() to overwrite or insert in one call
- **Nested JSON**: stored entire state as a TEXT field rather than multiple tables
- **Startup sync**: wrapped sequelize.sync() in an async IIFE to ensure table exists before requests

CRUD Queries with Sequelize::

- **Create/Update:** covered by `FSState.upsert()`
- **Read:** covered by `FSState.findByPk()`

Stella Dey - Challenges and Insights

Challenges:

- Bridging front-end and back-end was trickier than I thought. I ran into some CORS errors, mixed-up URLs, and “405 Method Not Allowed” when our dev server didn’t match the API.
- ESM vs. CommonJS headaches popped up when I tried to `require('node-fetch')`. Realizing Node 20 already has a global `fetch` saved me a lot of time and showed that sometimes fewer dependencies are better.

Insights:

Keep things modular by separating serialization helpers, CLI commands, UI button wiring, and API routes into distinct files and functions. This way each piece stayed focused and was easy to swap out or refactor later.

Collaborative Takeaways

Splitting UI vs. persistence workstreams, doing daily sync-ups, and keeping branch scopes narrow all helped us catch bugs before they snowballed into merge conflicts.

Stella Dey - Future Improvements

Auto-Save on Idle
Continuously persist changes after a period of inactivity, so users never lose work.

Issue: https://github.com/VicayoMua/326_Mini_Terminal/issues/72

Enhanced UI: Tree View & Context Menus
Provide a clickable folder-tree sidebar and right-click menus for file operations.

Issue: https://github.com/VicayoMua/326_Mini_Terminal/issues/73