

---

## Final Report for wSSH

---

**Group #8:**

Cyle Dawson

David Johnston

James McGinnis

Jonathon Saunders

Kerrick Staley

## Document Revisions

Version	Date	Author	Change
1	2012-12-2	Kerrick Staley	Fill in boilerplate
2	2012-12-3	McLoven	rough drafted my bits
3	2012-12-4	David Johnston	Drafted Sections 1.1 - 1.3.
4	2012-12-4	Kerrick Staley	Completed Section 7
5	2012-12-05	David Johnston	Drafted Section 1.4.
6	2012-12-07	Jonathon Saunders	Finished Section 3 and started section 4
7	12/7/2012	McLoven	touched up my parts, not sure what “other sections is”
8			
9			
10			

# Table of Contents

---

## **1 Introduction**

- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, Acronyms, Abbreviations
- 1.4 Design Goals (based on Deliverables, Functional, Non-Functional, User-Interface Requirements)

## **2 References**

## **3 User Interface Description**

## **4 Decomposition Description**

- 4.1 Module Decomposition
- 4.2 Concurrent Process
- 4.3 Data Decomposition
- 4.4 OverALL System Specification

## **5 Dependency Description**

- 5.1 Intermodule Dependencies
- 5.2 InterProcess Dependencies
- 5.3 Data Dependencies

## **6 Interface Description**

- 6.1 Module Interface
- 6.2 Process Interface

## **7 Design Rationale**

- 7.1 Handling directory changes: GUI to Terminal
- 7.2 Handling directory changes: Terminal to GUI
- 7.3 Performing GUI operations on the backend

## **8 Tracability**

## **9 Language/Framework/Tools Used**

## **10 Individual Responsibilities**

- 10.1 Cyle Dawson
- 10.2 David Johnston
- 10.3 James McGinnis
- 10.4 Jonathon Saunders

10.5 Kerrick Staley

## Section 1: Introduction

### 1.1: Purpose

This document is meant to introduce the reader to both the wSSH project and the components of the web application itself. In particular, it should:

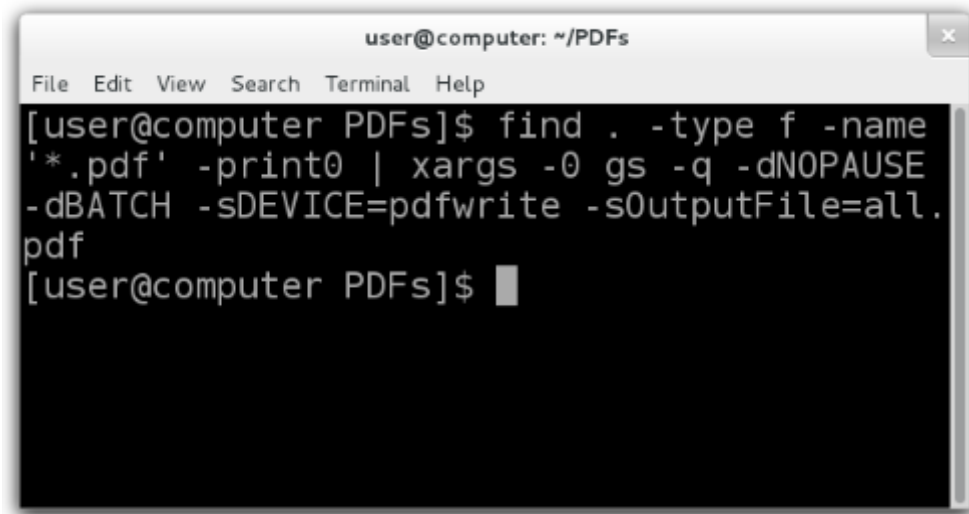
- Describe and illustrate the general problem which motivated the project as well as the much more focused scope of our solution. (See Section 1.2)
- Introduce the primary components of the system. (See Section 1.3)
- Describe both the broader goals of the project and the features provided by the current implementation. (See Section 1.4)
- Show the functionality provided to the user via the current user interface. (See Section 3)
- Describe in detail the project's various components (see Sections 4-8). In particular we define:
  - how functionality has been divided between separate modules,
  - the interfaces and interactions between modules,
  - and some of their more interesting inner-workings.
- Describe the roles and responsibilities which each group member has performed (see Section 10) as well as some of the design and implementation difficulties which we have collectively discovered along the way (see Section 7).
- List the frameworks upon which the project is built (see Section 9) and direct the reader to documentation describing these frameworks (see Section 2).

This document thus represents a single resource for detailing both the intentions of wSSH's design and the realities of its implementation.

## 1.2: Scope

### Motivating Problem

Many technical users of modern computer systems are often required to use both graphical user interfaces (GUI) and command line interfaces (CLIs) interfaces in concert. Most obviously, this is because many critical programs have been designed to be accessed via *either* GUI or CLI, but it is also because some software tasks just seem more suited to one interface more than the other.



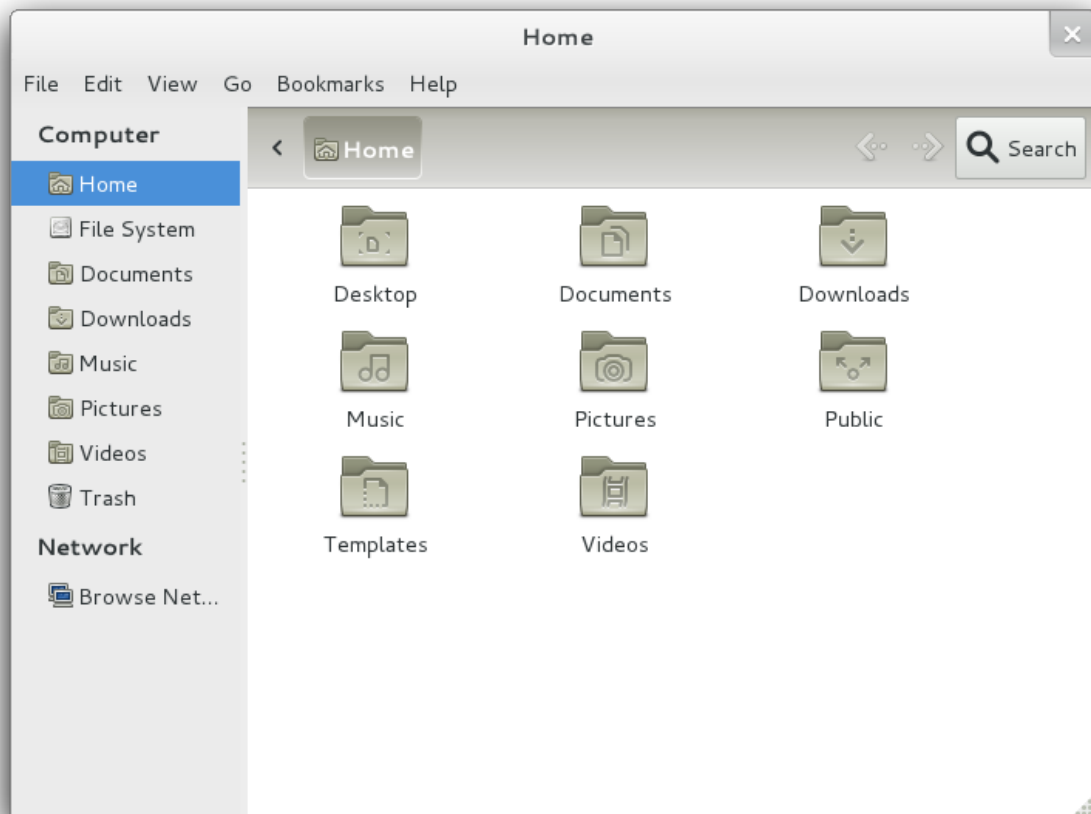
```
user@computer: ~/PDFs
File Edit View Search Terminal Help
[user@computer PDFs]$ find . -type f -name
'*.pdf' -print0 | xargs -0 gs -q -dNOPAUSE
-dBATCH -sDEVICE=pdfwrite -sOutputFile=all.
pdf
[user@computer PDFs]$
```

To illustrate this point, first consider the expressivity of the command invoked above. With one line of code it combines two separate programs to recursively search the current working directory for all pdf files, then concatenate their contents into a single file named `all.pdf`. There is no doubt that a program with a graphical user interface could be made to perform this task. However, it would not likely have the flexibility of `find` and `xargs`.

This use of a UNIX FIFO pipe is an example of an especially useful and simple programming paradigm available to CLIs which does not have any direct GUI analog (though it could be argued that contextual dragging-and-dropping can be made to emulate some of this behaviour. However, its utility is somewhat diminished by the obscurity of its syntax. As with many command line

programs, it may be difficult for a user without a great deal of background knowledge to fully interpret, let alone construct, such a command.

On the other hand, consider the simplicity and facility of a modern GUI-based file-browser, such as the Nautilus File Browser shown below.



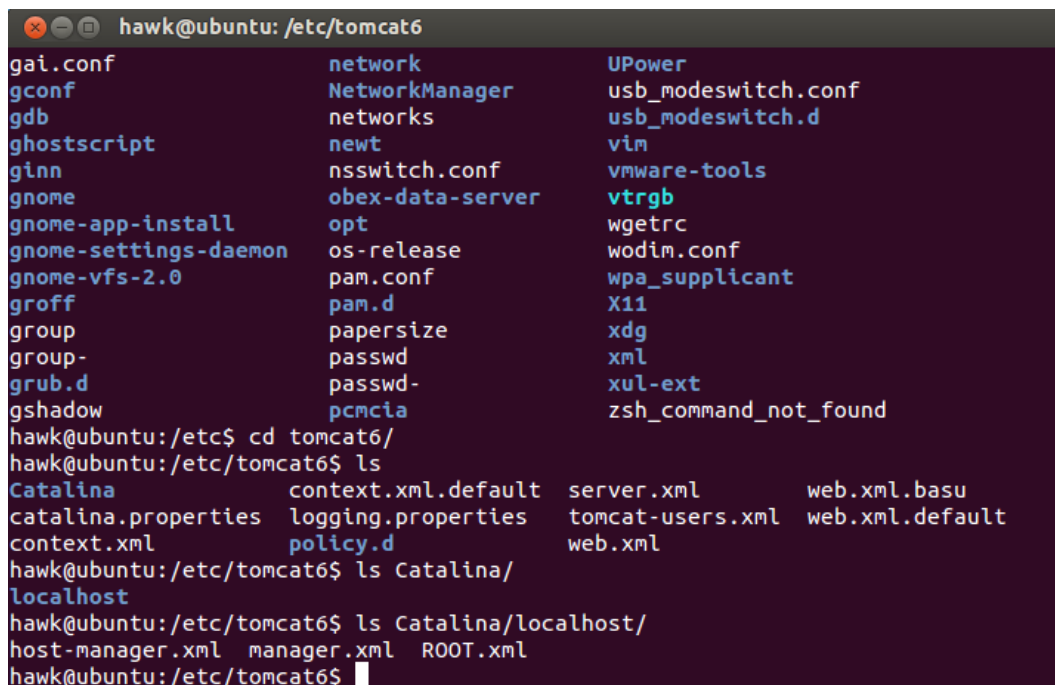
Source: [https://upload.wikimedia.org/wikipedia/commons/d/d2/Nautilus\\_screenshot.png](https://upload.wikimedia.org/wikipedia/commons/d/d2/Nautilus_screenshot.png)

Through such an interface it is simple and expedient to perform some of the most common file-system manipulation tasks via some combinations icons, buttons, and menus.

- Changing the current working directory.
- Listing a directory's contents.
- Opening a file.
- Moving, renaming and copying files.

Through a CLI, one can construct sophisticated file manipulations, which could not reasonably be performed via a GUI interface. However, the most basic (and most common) of actions can seem relatively cumbersome compared to their GUI analogs.

For instance, the output of an `ls` command provides very few visual cues to help the user interpret and navigate a directory's contents.

A terminal window titled 'hawk@ubuntu: /etc/tomcat6' showing a directory listing. The first 'ls' command lists files in the current directory, including configuration files like 'gai.conf', 'gconf', 'gdb', 'ghostscript', 'ginn', 'gnome', 'gnome-app-install', 'gnome-settings-daemon', 'gnome-vfs-2.0', 'groff', 'group', 'group-', 'grub.d', 'gshadow', and system directories like 'network', 'NetworkManager', 'networks', 'newt', 'nsswitch.conf', 'obex-data-server', 'opt', 'os-release', 'pam.conf', 'pam.d', 'papersize', 'passwd', 'passwd-', 'pcmcia', 'UPower', 'usb\_modeswitch.conf', 'usb\_modeswitch.d', 'vim', 'vmware-tools', 'vtrgb', 'wgetrc', 'wodim.conf', 'wpa\_supplicant', 'X11', 'xdg', 'xml', 'xul-ext', and 'zsh\_command\_not\_found'. The second 'ls' command is run after 'cd tomcat6/', listing 'Catalina', 'context.xml.default', 'server.xml', and 'web.xml.basu'. The third 'ls' command is run inside the 'Catalina' directory, listing 'localhost', 'context.xml', 'logging.properties', 'tomcat-users.xml', and 'web.xml.default'. The fourth 'ls' command is run inside the 'localhost' directory, listing 'host-manager.xml', 'manager.xml', and 'ROOT.xml'.

In the most common case, a GUI file browser is a superior interface for listing a directory's contents because it provides file and folder icons to help the user more easily perceive a directory's contents and traverse a file-system.

## GUI + CLI Integration

Given that both interface paradigms have clear context-specific benefits and drawbacks, a technical user may wish to be able to use some combination of graphical and command line interactions so that he/she may decide what tool best fits a given task. Within the domain of file-system shells,



we are not aware of any tools which truly combine both paradigms together. Currently, a user can interact with GUI and CLI shells side-by-side, but the two programs will be decoupled.

It was with this context that we decided that the central goal of this project was to create a shell which would enable the user to dynamically decide which UI paradigm would best fit any given task. In particular, we focused our attention on creating a system which would synchronize the current working directory of a terminal interface and a graphical file-browser. We believe that reflecting directory changes between both user interfaces is the crucial first step in uniting the two UI paradigms to create a single coherent interface which can facilitate a single coherent workflow for the user.

Because SSH is one of the most featureful and capable terminal emulation environments available and because each of us use SSH very often, we decided to build our program around a remote SSH connection. Inspired by the remote connectivity afforded to us by SSH, we decided that a web client should be our target environment. Thus we had found our project, wSSH, a modern web app which would combine and synchronize a graphical file-browser interface with a remote SSH connection.

### 1.3: Definitions, Acronyms, Abbreviations

Web Client (or just Client)	<p>This is the browser within which the front-end to the web-application is run.</p> <p>It provides the portal for all user interaction with the remote system.</p>
Web Server (or just Server)	<p>This is the system which formats and relays communication between the client and the remote system to which the user is interacting.</p> <p>Its roles include: Servicing the web client (over both HTTP and WebSockets), and making requests of the web server (over both SSH and SFTP).</p>
SSH Server	<p>This is the remote system to which the user is connecting</p> <p>Note that the web server and SSH server are not necessarily the same host.</p>
Graphical User Interface (or just GUI)	<p>Within the web client, this is the collection of UI components by which the user can perceive and manipulate the remote system via icons and menus.</p> <p>Collectively, they emulate many aspects of a file browser within a traditional desktop environment.</p>
Command-Line Interface (or just CLI)	<p>Within the web client, this is the UI component by which the user can invoke commands on the remote system and receive their textual output.</p> <p>This is meant to closely reflect the interactions which the user would have if they were using a traditional SSH client.</p>

## 1.4: Design Goals

### Overview of Project Goals

Our primary objectives have been to accurately emulate terminal behavior and to accurately depict the contents of the CLI's current working directory as icons within the GUI. It was also important to us that we could provide both of these features without customizing or installing software onto the SSH server. We designed the web server to relay and format requests to and from another remote host which is running a vanilla SSH/SFTP server.

As a secondary objective, we also wished to provide the user with the ability to perform a variety of common and convenient file manipulation tasks via traditional GUI elements such as mouse clicks, icons, buttons, menus, dragging-and-dropping. These GUI interactions should seem simple and intuitive from the user's experience with traditional desktop file-browsers.

The project's members have throughout this project's design and implementation phases consistently pursued simplicity in application's architectural design.

If we were to continue development there are a number features which we would wish to add to our implementation. The most important would be to address an array of security concerns. In particular, the websocket connection should be encrypted via SSL/TLS.

### **Enumeration of Functional and Nonfunctional Deliverables (by Priority):**

1. Accuracy terminal emulation.
2. Synchronized depiction of the contents of the CLIs current working directory in the GUI.
3. Can connect to most SSH servers without installing additional software or making significant configuration changes.
4. Simplicity of user experience using familiar GUI interactions:
  - a. Changing the current working directory in both the CLI and GUI via:
    - i. Double clicking a directory icon.
    - ii. Clicking a button to move to any ancestor of the current working directory.
  - b. Multi-icon selection with contextual drag-and-drop:
    - i. Dragging selected files into another directory.
    - ii. Removing files and folders by dragging them to the trash.
  - c. Context dependent popup menus to provide appropriate actions.
5. Simplicity of architectural design

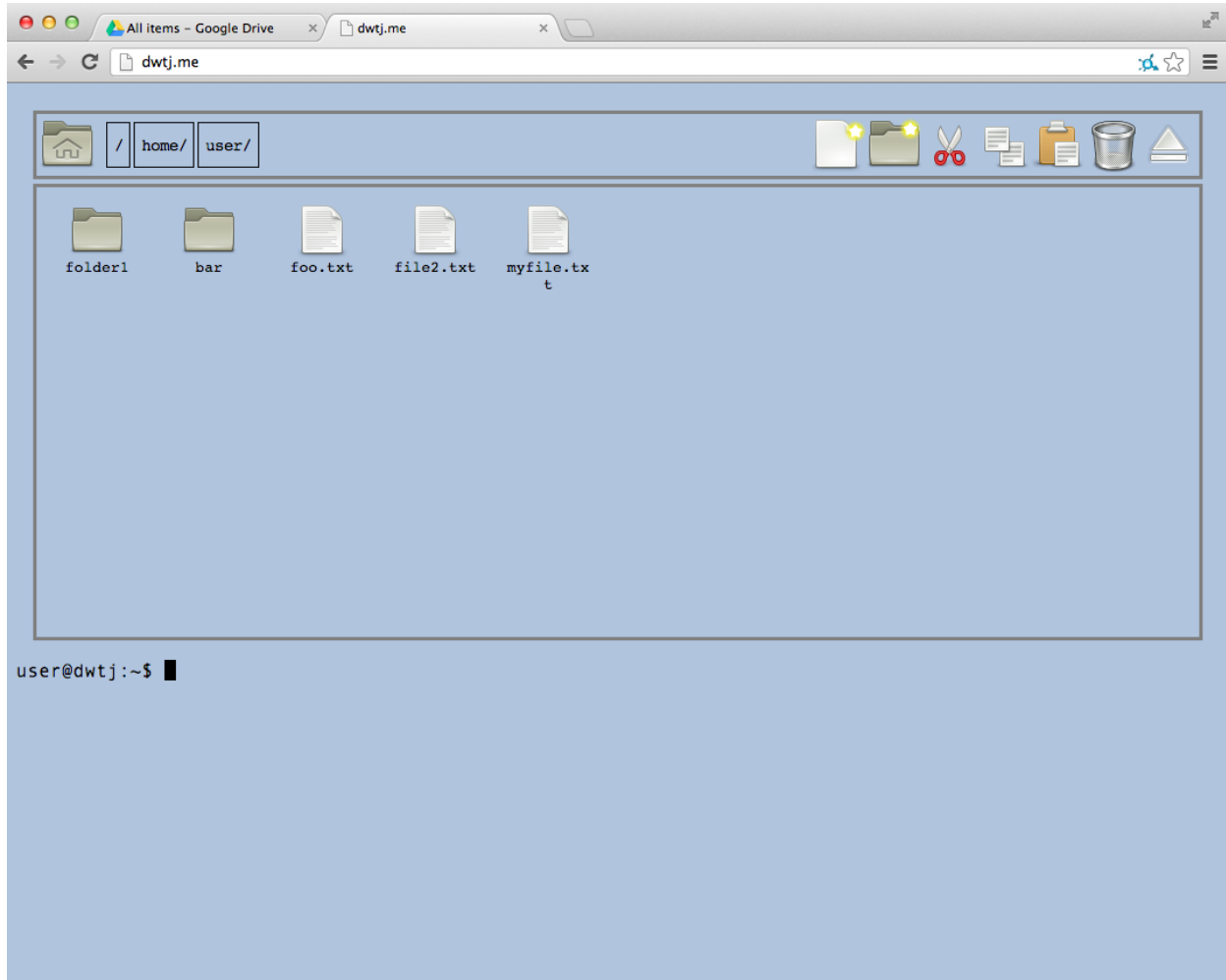
### **Enumeration of Deliverables Which Are Not Yet Implemented (by Priority):**

- Security
  - SSL/TLS of the websocket connection.
  - More security considerations.
- UI Features
  - Let user collapse either GUI or CLI
  - File uploads and downloads
  - Keyboard Shortcuts
- Performance/Architecture
  - Intelligently buffer communications.
  - Push changes to the current working directory rather than polling for them.

## Section 2: References

N/A

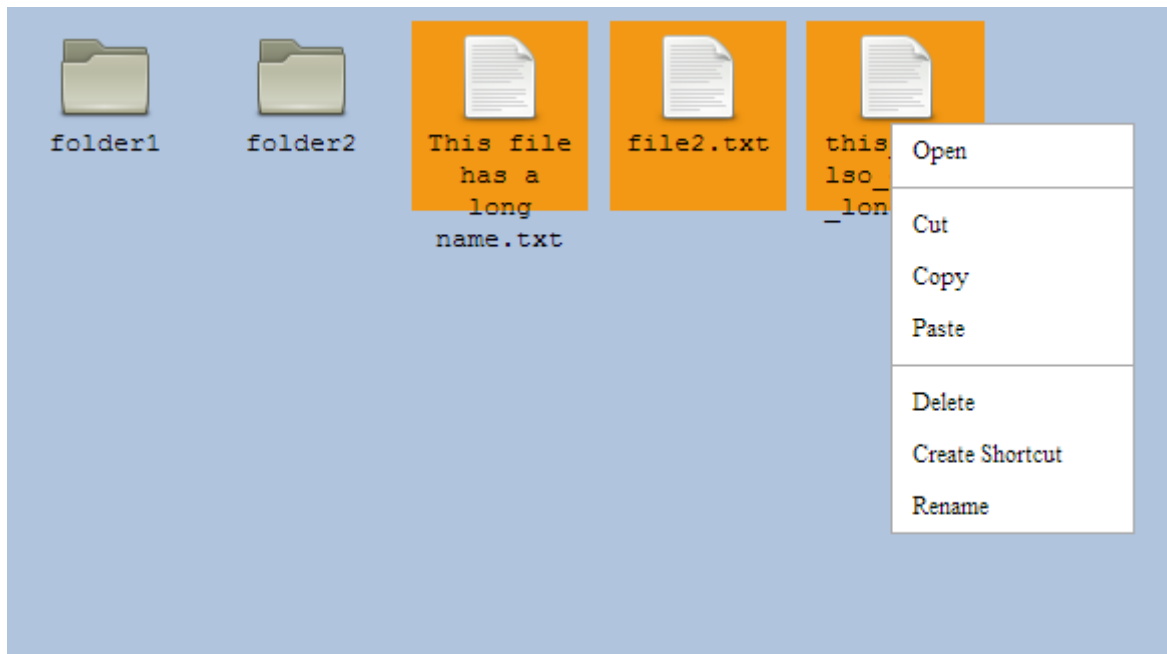
## Section 3: User Interface Description



Our user interface is split into three main parts the header, the file system, and command line interface.

- **Header:** The header has several components. First is the home icon that will take the user to their home. The next part is the path bar that gives the path from root to the current working directory so that the user can quickly move to any of those folders. This GUI convention is often known as breadcrumbs. Finally there are action buttons with provide the user with various functionality such as making a new file, making a new folder, cut, copy, paste, deleting files and logging out.

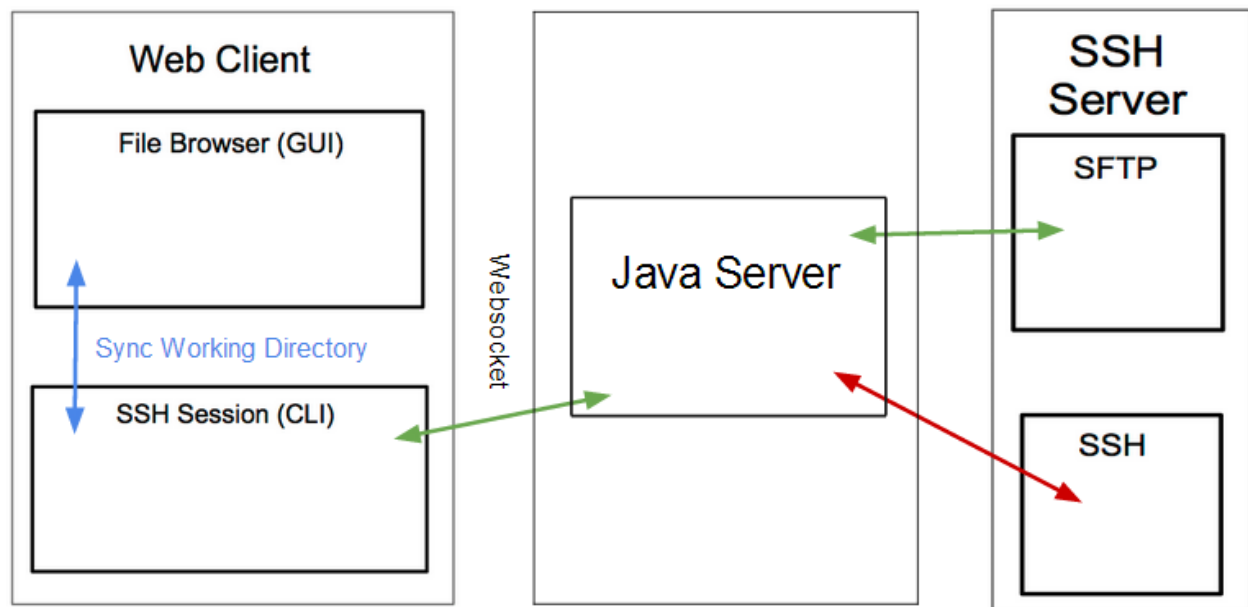
- **Filesystem:** The filesystem is where all the files from the current working directory are loaded and has various functionality for those files. For all the icons in the file system, you can select several and perform functions on them. All the selected files can also be dragged and dropped to move them in the file system. If you right click on any of the files or folders there is also a menu that opens to perform various actions.
- **CLI:** The command line interface is provides all the functionality that a regular shell would.



## Section 4: Decomposition Description

### Module Decomposition

We had three basic modules each with two parts. The GUI interface in the web browser, our web server that is redirecting the traffic, and the SSH server that the user is actually trying to connect to.



GUI File System - The front end display of the current working directory.

CLI - The front end command line interface to the SSH server.

Java Server - The web server running Java did all the work to send the traffic to the ssh server, and handle sending the commands to the SFTP server.

SFTP Server - The SFTP server you are connecting to.

SSH Server - The SSH Server you are connecting to.

### Concurrent Process

### Data Decomposition

### Overall System Specification



## Section 5: Dependency Description

### ◦ **Intermodule Dependencies**

For the front end, the GUI and shell depend on one another only to keep track of directory changes. On the server-side, it follows a layered design. Each WebSocket connection talks with its own copy of a connection module. This connection module talks with both the SSH connection module and the SFTP connection module. Those modules represent the actual connections to the SSH server, and thus talk with the SSH server directly.

### ◦ **Interprocess Dependencies**

Although the server-side is highly threaded, the threads remain more or less independent. The WebSocket server itself is the only thing that talks with each WebSocket connection interdependently. Beyond that, each WebSocket connection just communicates with the modules it calls on to do its work.

### ◦ **Data Dependencies**

Our application acts pretty much like an in-between for the user and the desired SSH server. It does not keep track of a significant amount of data of its own, besides what is required to allow proper communication between the user and the SSH server.

## Section 6: Interface Description

### Module Interface

ServerMain: onOpen, onMessage, onClose, onError

An implementation of java-websocket's WebSocketServer. Methods are called by the library.

Connection:

SSHConnection:

SFTPConnection:

### Process Interface

There wasn't any process communication done outside of the java-websocket library, and the library handled creating the threads for us. So, we didn't create any interface on that level.

## Section 7: Design Rationale

### Handling directory changes: GUI to Terminal

#### ■ Description:

If a directory change is performed in the GUI, the terminal needs its state updated to match that of the GUI.

#### ■ Factors affecting Issue

It is impossible to change the current working directory of a running Bash process from another process in a programmatic way: the only way to change the CWD is to use “cd” (or a similar command such as pushd) inside the Bash session. However, this cd command needs to be sent in a way that does not interfere with the use of the terminal. In particular, if the Bash shell is not currently in the foreground (e.g. if Vim is running), the cd command will not have the desired effect, and may in fact have very adverse effects (imagine sending a sequence of random keystrokes to Vim).

#### ■ Alternatives and their pros and cons

- Determine if another process is in the foreground, and if so, prevent directory changes in the GUI

Pros: Directory changes as we implement them only affect the Bash process anyway, so if another process is running in the foreground, this child process will ignore the new directory, which could violate the user’s expectations.

Cons: Preventing directory changes in certain contexts creates an inconsistent experience and could frustrate the user. Also, detecting whether a non-Bash process is in the foreground is non-trivial, and implementing this feature would divert resources away from more important aspects of the project.

- Suspend the foregrounded process, if any, and then run the cd command; resume any suspended process afterwards using the fg command.

Pros: Easier to implement, and creates a more consistent user experience.

Cons: Still some implementation complexity, because one needs to detect whether a process was actually suspended, and only run fg if a process was suspended.

### ■ Resolution of Issue

We went with the latter option. We also decided to actually display the `cd` command in the terminal (rather than implicitly sending it and then intercepting its output) so that the user would be aware of what was going on.

## Handling directory changes: Terminal to GUI

### Description

When the directory is changed in the terminal, the GUI needs to be refreshed (the breadcrumbs widget needs to display the new path and the file pane needs to display the new directory's contents).

### Factors affecting Issue

There are two ways to determine the CWD of a running Bash process: either issue a `pwd` (or similar) command inside the shell, or inspect the `/proc/<pid>/cwd` symlink.

### Alternatives and their pros and cons

- Upon login, alter `PROMPT_COMMAND` (a command that Bash runs before every prompt is displayed; typically used to set the title of an xterm) so that it echos a special escape sequence followed by the current working directory followed by another escape sequence. Whenever the escape sequence is encountered in the terminal's output, intercept the characters up to and including the next escape sequence, and query the backend for the contents of that directory.
  - Pros: Very simple to implement.
  - Cons: David didn't like it, and for some reason I didn't think it had much merit either, even though was probably the best approach in retrospect. It also had the requirement that we send a request from the client to the web server querying the

directory's contents every time a new prompt was displayed.

- Have the web server periodically poll the /proc/<pid>/cwd symlink, and push an update to the client every time the symlink points to a new location or the contents of the linked directory change.
  - Pros: Doesn't mess with the user's shell. Doesn't result in any traffic between the webserver and the client when there are no directory changes.
  - Cons: Creates a lot of overhead between the web server and the SSH server and leads to a delay between actions in the terminal and the appearance of their result in the GUI.

## Resolution of Issue

We chose the latter option.

## Performing GUI operations on the backend

### Description

When the user performs an operation in the GUI (e.g. deleting a file), that operation needs to be executed on the backend.

### Factors affecting Issue

Bash has a rather inconsistent escaping mechanism, making it difficult to properly escape filenames.

SSH servers typically expose an SFTP (SSH File Transfer Protocol) interface.

### Alternatives and their pros and cons

- Insert the commands directly into the terminal, either executing them immediately or waiting for the user to press [Enter]

- Pros: Simple implementation.
- Cons: Can interfere with the user's shell use. Need to escape filenames, which can lead to complexity and edge-cases. If the user has set unusual aliases, it could also interfere with command execution.
- Open a second SSH session and use it exclusively for performing file operations.
  - Pros: Simple implementation..
  - Cons: Still requires escaping filenames, and might be slightly more complex than the above method.
- Use SFTP to perform the file operations
  - Pros: SFTP provides a very regular interface; no need to worry about escaping or other edge-cases.
  - Cons: Increases implementation complexity.

## Resolution of Issue

We chose the last option.

## Section 8: Tracability

No	Use Case/ Non-functional Description	Subsystem/Module/classes that handles it
1	Get a shell, Use the shell	shell in a box front end communicates with java server class OneShot method named StartShell() creating a stream to the ssh server when provided proper credentials
2	Get the Graphical User Interface	jquery sends json object to java server which uses Class, OneShot method, openSFTP to relay pwd contents to gui
3	Update the Graphical User Interface	Java backend sends json update object via open AJAX call to frontend to alert the JQuery of changes to pwd
4	Delete a file, Make a file, Move a file, Copy a file	Json object is sent to the Java server which will use class OneShot method openSFTP to utilize a sftp connection to perform the action.

## Section 9: Language/Framework/Tools Used

1. Apache HTTP Server
2. java-websocket
3. JQuery
4. ShellInABox
  - a. ShellInABox is an open source tool implementing a browser-based terminal emulator that communicates via AJAX to a backend, allowing arbitrary commands (e.g. bash) to be accessed from a browser. We integrated the VT100 emulator provided by this project into our client, obviating the need to implement a terminal emulator ourselves.
5. JSch
  - a. JSch is a java library which allows the use of SSH and SFTP protocols by a java application and much more.
  - b. In this project it is used to create a input/output stream to an ssh server and to make a connection to a SFTP for the purpose of manipulating files directly.
  - c. The library is hosted and documented here <http://www.jcraft.com/jsch/>
6. java-websocket
  - a. This library was used to handle the websocket handshake and spawning a connection object through with the rest of the server could communicate with the client. <https://github.com/TooTallNate/Java-WebSocket>



## Section 10: Individual Responsibilities

- **Cyle Dawson**

Implemented the websocket interface provided by java-websocket and developed the communication layer between the websocket server and the SSH/SFTP connections to the SSH server.

- **David Johnston**

Implemented many features of the GUI interface, including support for dynamic file-system display, icon event handling, and use of the Selectable, Draggable, and Droppable JQuery UI APIs.

- **James McGinnis**

Implemented the server's interface to SSH and SFTP.

- **Jonathon Saunders**

Worked on the right click menu system for the GUI and scrollbar.

- **Kerrick Staley**

Implemented the terminal portion of the client and the "Connect" dialog; documented protocol for client<->server communication; contributed to overall architecture design.