Hako User Manual

Authors: Niall Ryan (21454746), Cathal O'Grady (21442084)

Supervisor: Prof. Stephen Blott

2025-05-02

Contents

1 Hako User Manual

1.1 Hako

Hako is a platform agnostic Unix-like integrated development platform for the purposes of educating the youth about programming in a more “systems-oriented” way.

1.2 Target Readers

This manual is intended for Hako’s two target audiences:

- Students
- Mentors

1.3 Content

If you want to build from source, and host locally, this manual will describe:

- How to build hako from source
- How to run it locally

This manual will also describe the features of Hako:

- Applications
- Core-Utils
- System API

1.4 Building

1.4.1 Dependencies

Build instructions applicable Unix/Posix/Linux environments

- Justfile
- ldoc (lua-ldoc)
- Meson + Ninja (fortnite)
- Gcc
- Emscripten
- Node (npm)

- gpg
- cp

1.4.2 Build instructions

```
just
```

You can do a clean build with:

```
just clean && just
```

You can also build the site as a static bundle with:

```
just site
```

1.5 Running

1.5.1 Development Server (Easiest)

If you wish to simply run Hako for personal use, you can run it using a development server.

A development server is a simple server on your own machine for personal use, and allows you to use the application easily - often used for the actual development of Hako.

Once you build Hako (see build instructions above), a development server can be ran via:

```
just site-run-dev
```

or more succinctly

```
just srd
```

It will then be available on `127.0.0.1:5173` (`localhost:5173`) in your browser.

1.5.2 Self-Hosting

Hako is mostly simple to self-host as it is **completely client-side**, which means it all runs on your device. There are no servers or any other compute required to run the application.

If you are trying to deploy the website yourself, you might find the `Containerfile` useful. This can be used to build a container image, using a tool like docker. Building the image can be done as follows, from the root of the repository (using docker):

```
docker build -f Containerfile -t hakob .
```

Then you can run the container like so:

```
docker run -it -p 8000:80 --rm --name hakob localhost/hakob
```

This will host the website on port 8000.

If you want to use your own solution and webserver, do note that the following headers need to be set:

```
Cross-Origin-Embedder-Policy require-corp
Cross-Origin-Opener-Policy same-origin
```

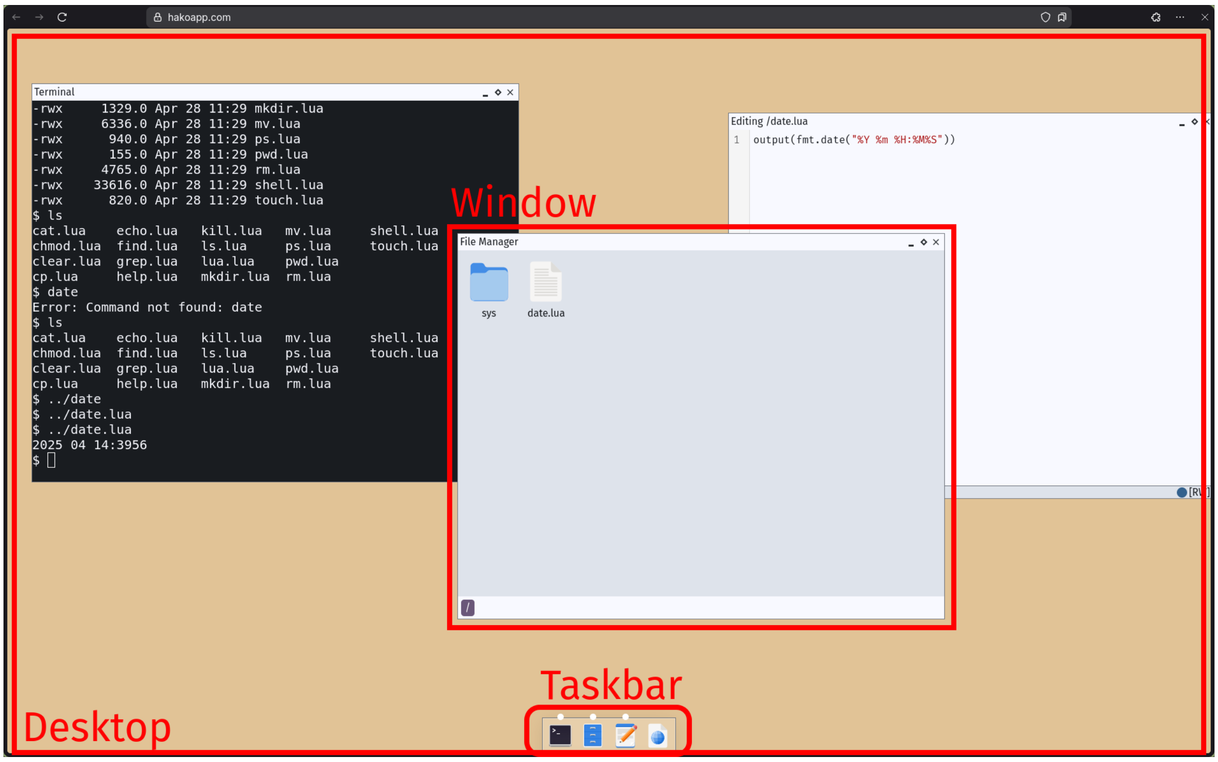
1.6 Features

Hako has a whole bunch of features to assist you in learning (or teaching) systems programming!

A high-level category of these features are:

- **The Desktop** – The graphical user interface for the system
- **Applications** – The applications available for you to use
- **Core-Utils** – Useful tools and utilities for you to use in the terminal
- **System APIs** – Interfaces to manipulate and use your system with

1.6.1 Desktop



```
-rwx     1329.0 Apr 28 11:29 mkdir.lua
-rwx     6336.0 Apr 28 11:29 mv.lua
-rwx      940.0 Apr 28 11:29 ps.lua
-rwx      155.0 Apr 28 11:29 pwd.lua
-rwx     4765.0 Apr 28 11:29 rm.lua
-rwx    33616.0 Apr 28 11:29 shell.lua
-rwx      820.0 Apr 28 11:29 touch.lua
$ ls
cat.lua    echo.lua   kill.lua   mv.lua     shell.lua
chmod.lua  find.lua   ls.lua     ps.lua     touch.lua
clear.lua  grep.lua   lua.lua    pwd.lua
cp.lua     help.lua   mkdir.lua  rm.lua
$ date
Error: Command not found: date
$ ls
cat.lua    echo.lua   kill.lua   mv.lua     shell.lua
chmod.lua  find.lua   ls.lua     ps.lua     touch.lua
clear.lua  grep.lua   lua.lua    pwd.lua
cp.lua     help.lua   mkdir.lua  rm.lua
$ ../date
$ ../date.lua
$ ../date.lua
2025 04 14:3956
$ 
```

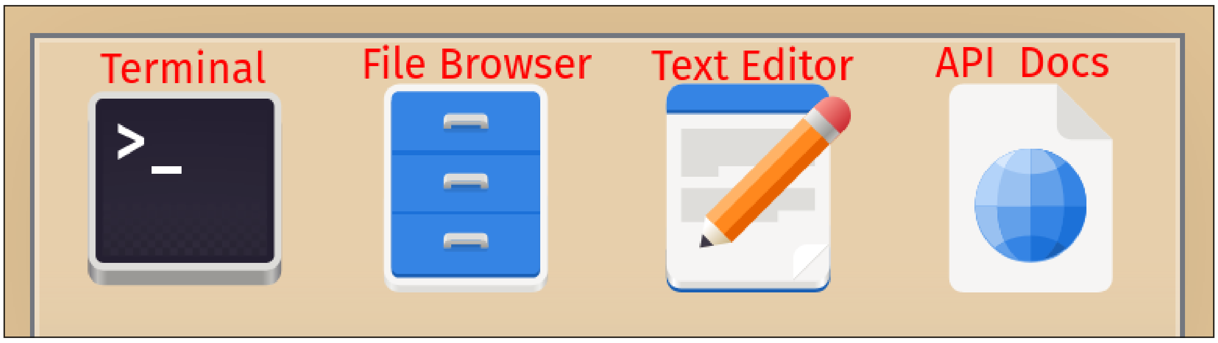
```
output(fmt.date("%Y %m %H:%M%S"))
```

The desktop should be a familiar interface to those who have used a typical operating system before (such as Windows, MacOS, ChromeOS).

It is a “floating window manager”, which means windows can:

- Be dragged and placed anywhere on the screen
- Overlap eachother (windows can be on top of others)
- Be manually resized and positioned
- No strict rules are automatically enforced

1.6.1.1 Taskbar



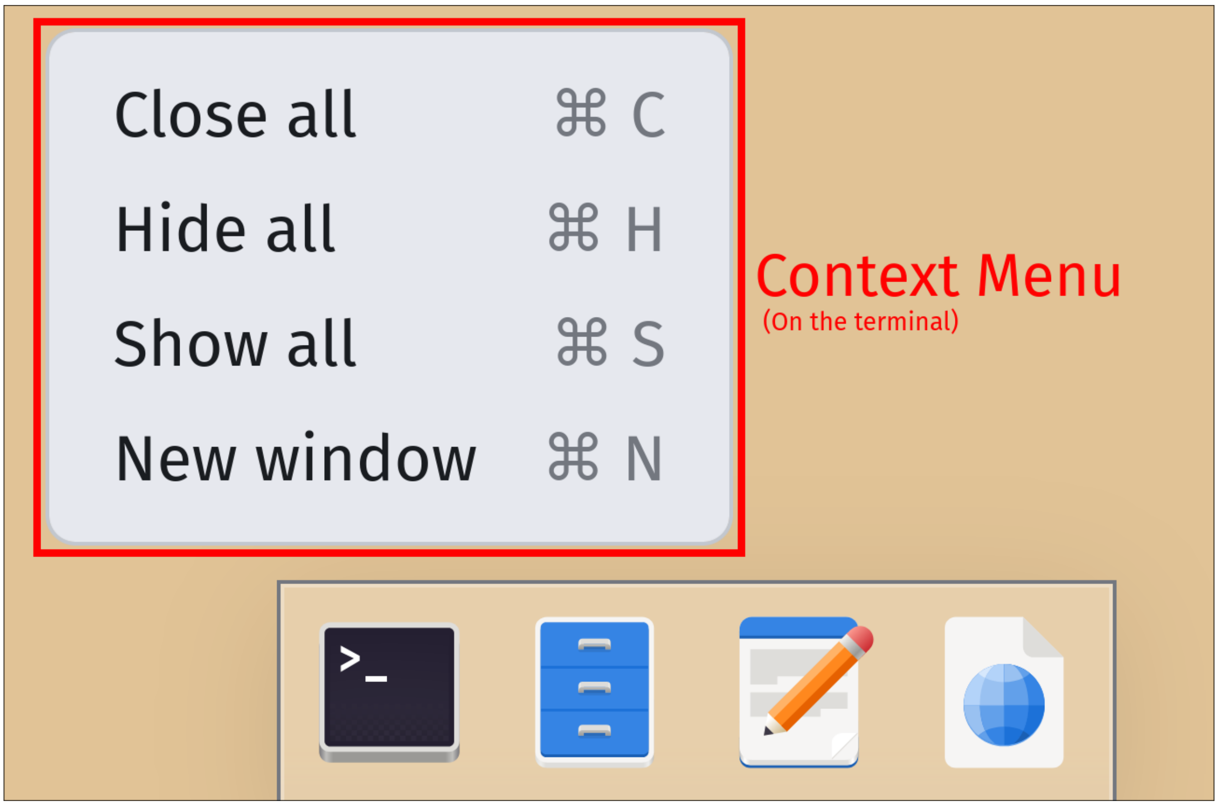
You can “left click” on an icon to open one instance of that application type.

Furthermore, if an instance of this application type already exists when you “left click” them, it will cycle bringing the instances to the forefront, starting with the closest going to the furthest.

Hako comes with four built-in graphical user interface (GUI) applications:

- **Terminal** (first icon)
- **File** (second icon)
- **Text Editor** (third icon)
- **System API Documentation** (fourth icon)

The functionality of these can be read about below in the “Applications” section.



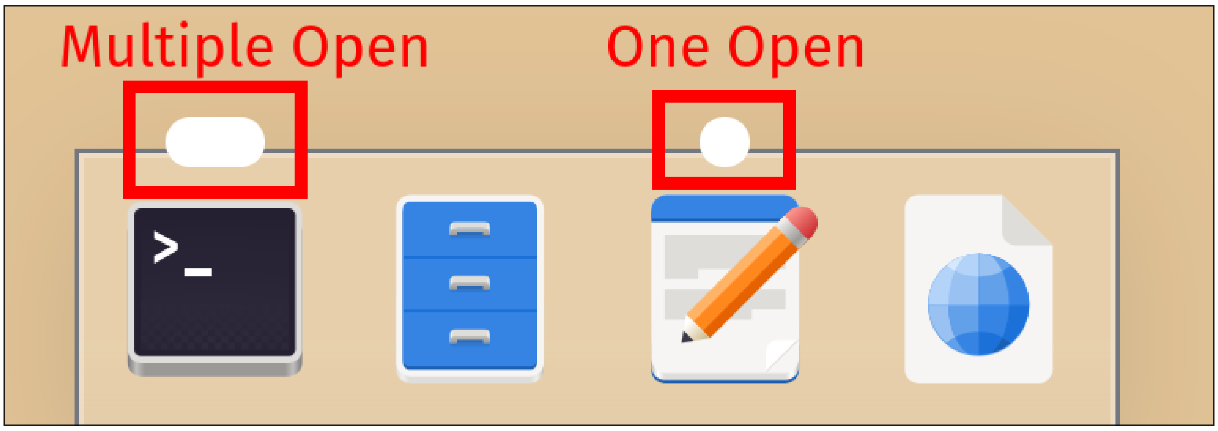
You can "right click" on any application icon to open the context menu.

The context menu allows you to perform actions on a single application type, these actions include:

- **Close all** – Closes all instances of that application type
- **Hide all** – Hides all instances of that application type
- **Show all** – Shows all instances of that application type
- **New window** – Creates a new instance of that application type

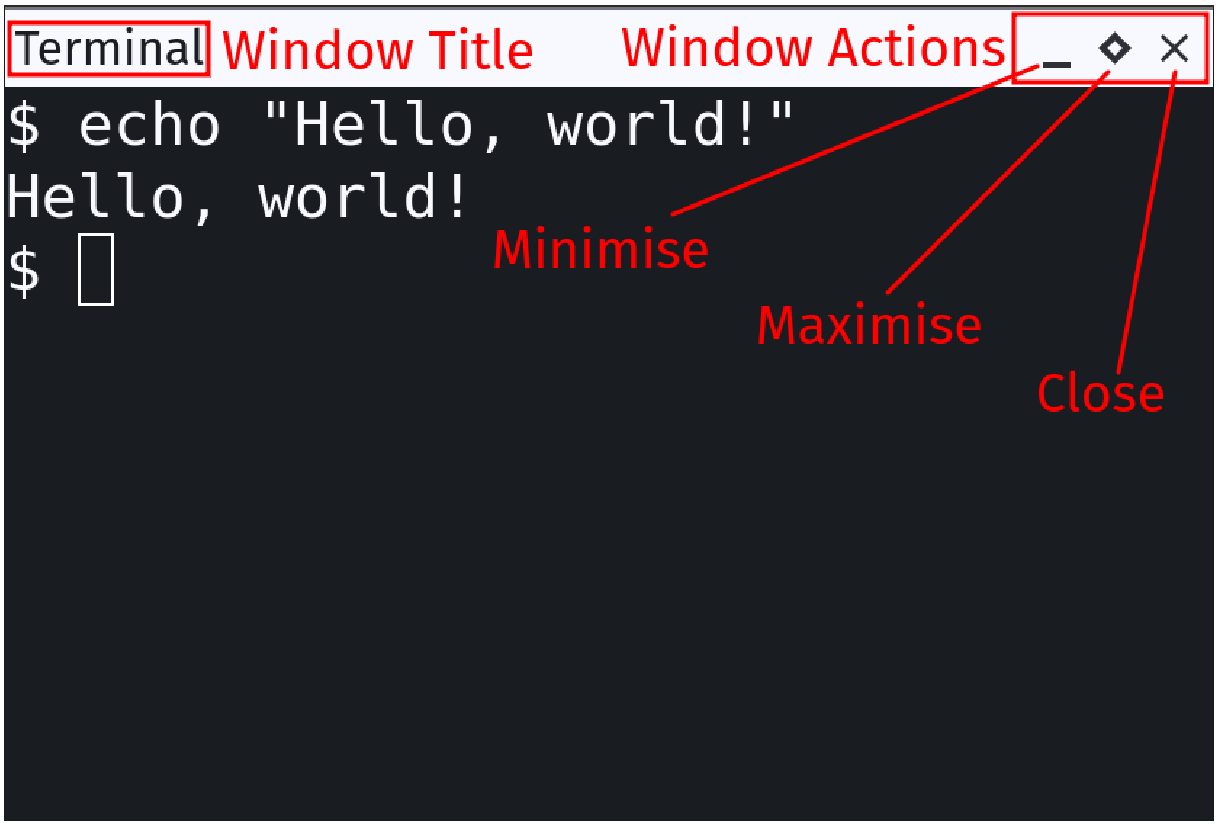
You can also hit a keybind whilst the context menu is open to perform the associated action, as denoted above.

- **C** – Close all
- **H** – Hide all
- **S** – Show all
- **N** – New window



The taskbar also denotes if none, one or multiple instances of an application type are open. See image above.

1.6.1.2 Windows

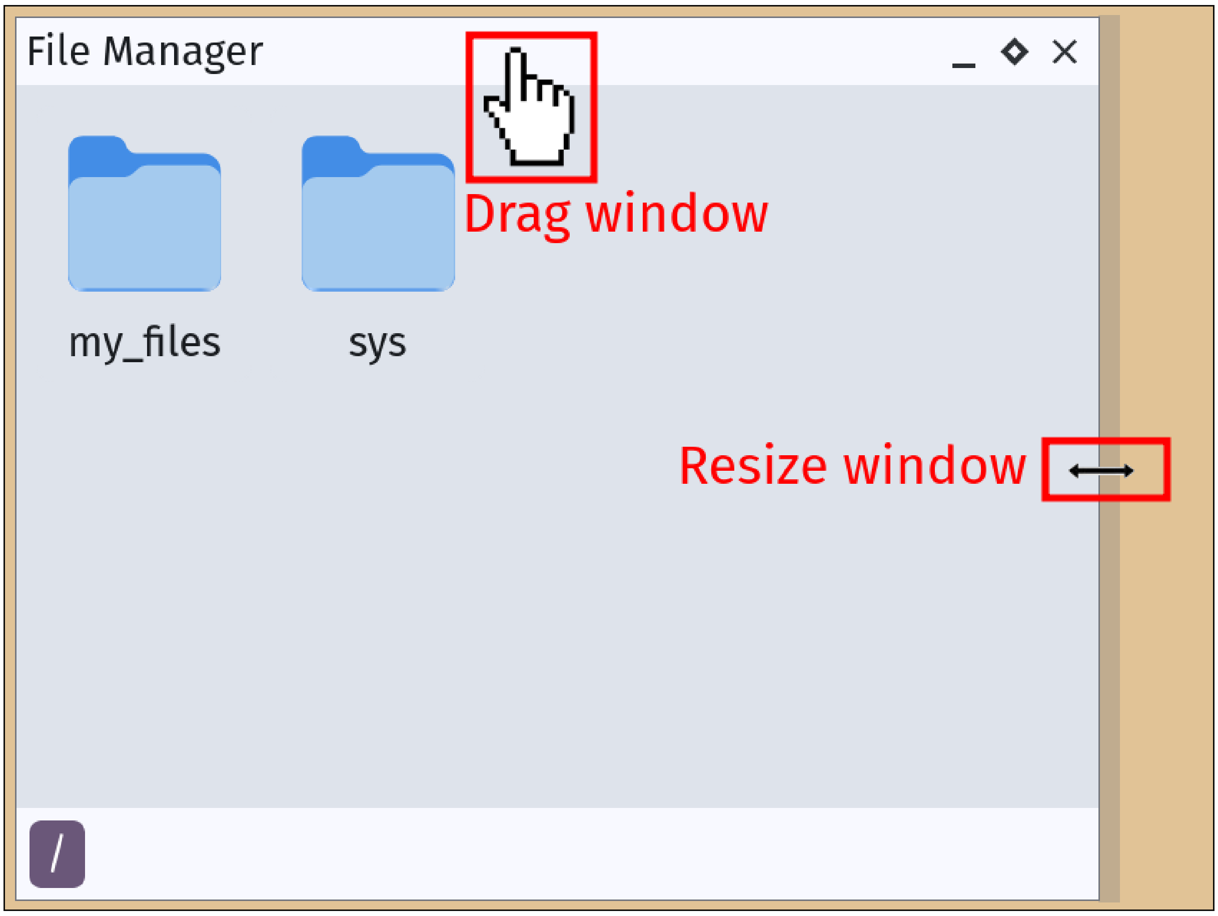


```
$ echo "Hello, world!"
Hello, world!
$ 
```

Windows contain the application's graphical interface, information about the application, and methods to manipulate the application instance.

The window title in the top-left corner displays the window's application's name.

The window actions in the top-right corner allow you to minimise, maximise and close the window.

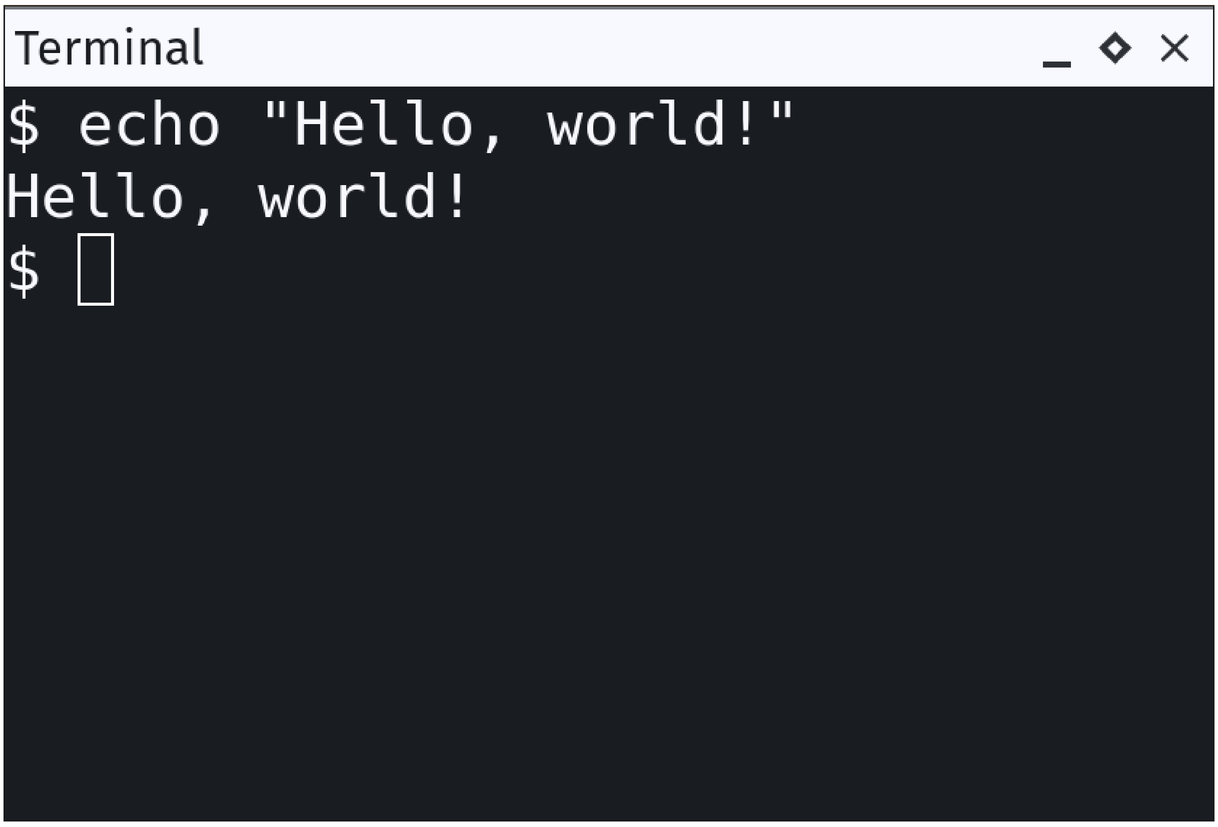


Windows can be dragged around the desktop by holding “left click” on the window’s title bar.

Windows can be resized by hovering on any of the window’s four borders, and holding “left click” whilst dragging to increase or decrease in the direction you wish.

1.6.2 Applications

1.6.2.1 Terminal



```
$ echo "Hello, world!"
Hello, world!
$ 
```

The terminal is an application that provides a text-based interface that lets you interact directly with the Hako operating system or other programs.

Through a terminal you can:

- **Run commands** to control the system (like managing files, processes, windows, etc)
- **Automate tasks** by writing and running Lua scripts
- **Interact with programs** that don't have a graphical user interface (GUI)

The default program that the terminal runs is the **shell**, which you can read more about below in the "core-utils" section.

The terminal also has additional features, such as:

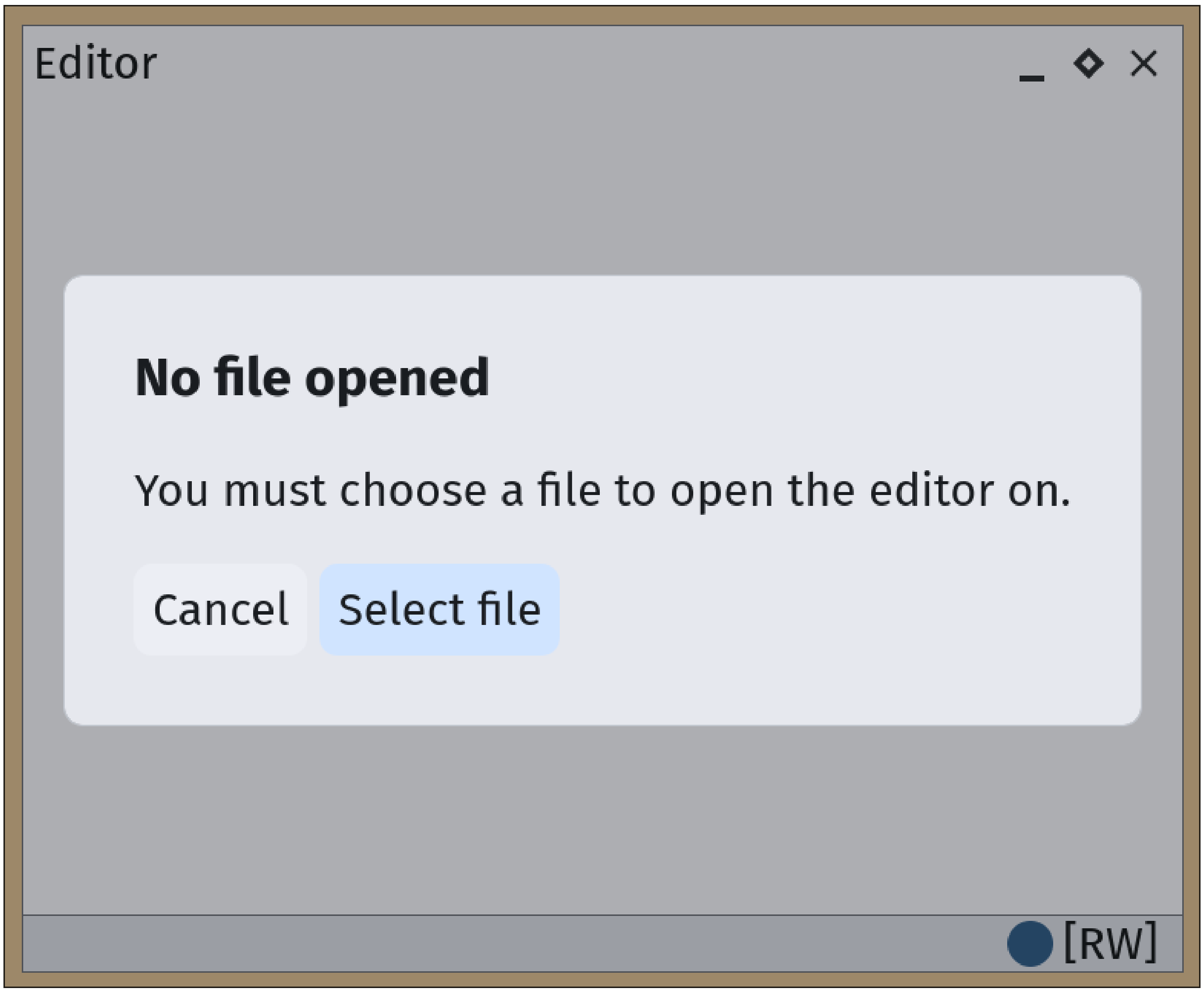
- **Line manipulation** – You can edit a line like you typically would, going back and forth with arrows keys
- **Command history** – Go to previous commands ran with arrows keys
- **Command search** – Enter "Ctrl + R" to search your history for previous commands

If the shell process attached to a terminal dies, the terminal instance will close.

1.6.2.2 Editor The text editor is a lightweight and fast application that lets you to edit and view plain text easily.

It's primary features are:

- Allow file editing
- Syntax highlighting for Lua files
- Auto-save to avoid loss of data

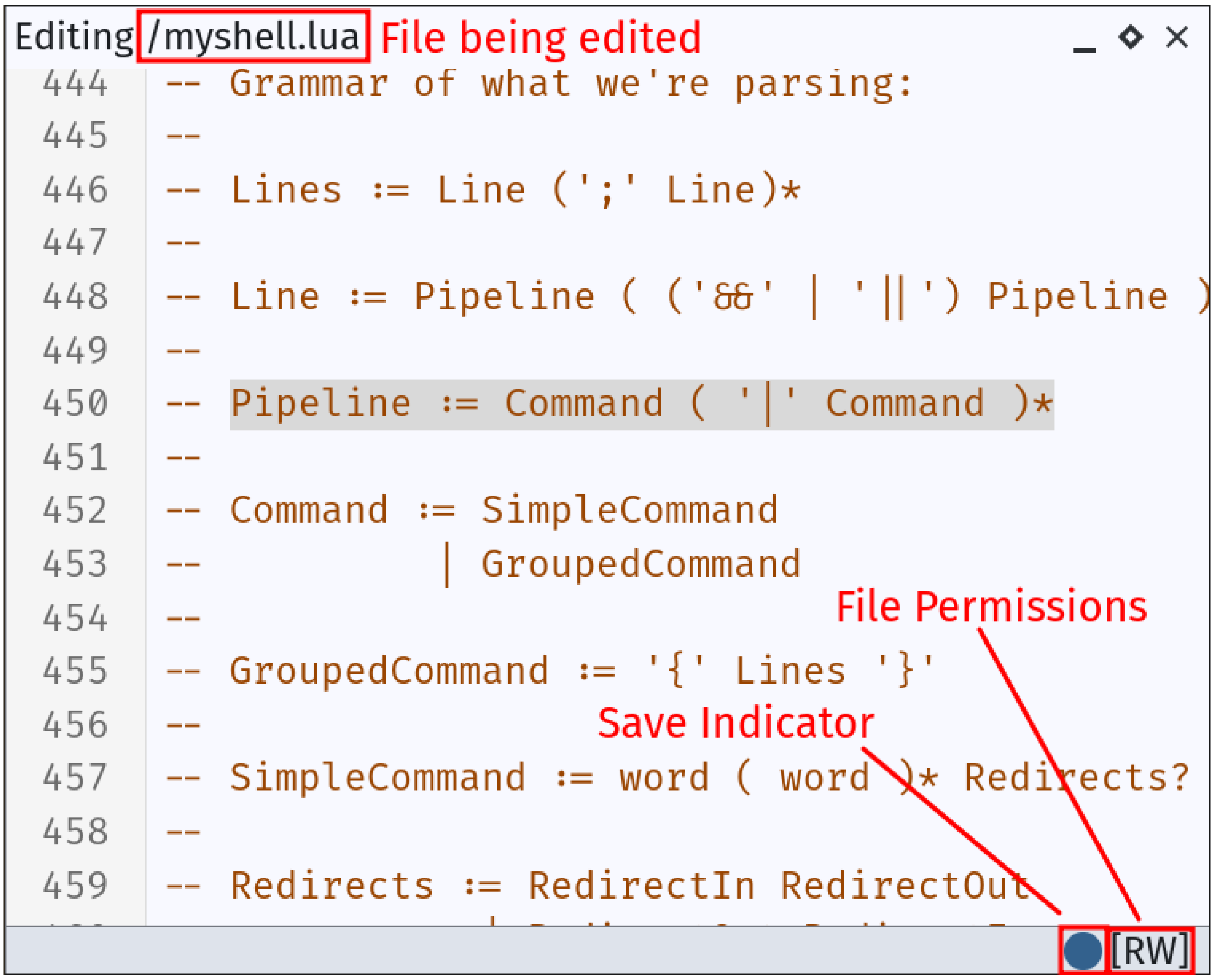


No file opened

You must choose a file to open the editor on.

When you open the text editor from the task bar, it will request a file to open.

Upon clicking "Select file", it will open the file browser, allowing you to select or create a file.



```
-- Grammar of what we're parsing:
--
-- Lines := Line (';' Line)*
--
-- Line := Pipeline ( ('&&' | '||') Pipeline )
--
-- Pipeline := Command ( '|' Command )*
--
-- Command := SimpleCommand
--          | GroupedCommand
--
-- GroupedCommand := '{' Lines '}'
--
-- SimpleCommand := word ( word )* Redirects?
--
-- Redirects := RedirectIn RedirectOut
```

The editor window's name reflects the file being edited.

There are two indicators in the bottom right:

- **Save indicator** – The circle. Blue indicates saved, Red indicates not-saved.
- **File Permissions** – Whether the file is Read Only [RO] or Read Write [RW]

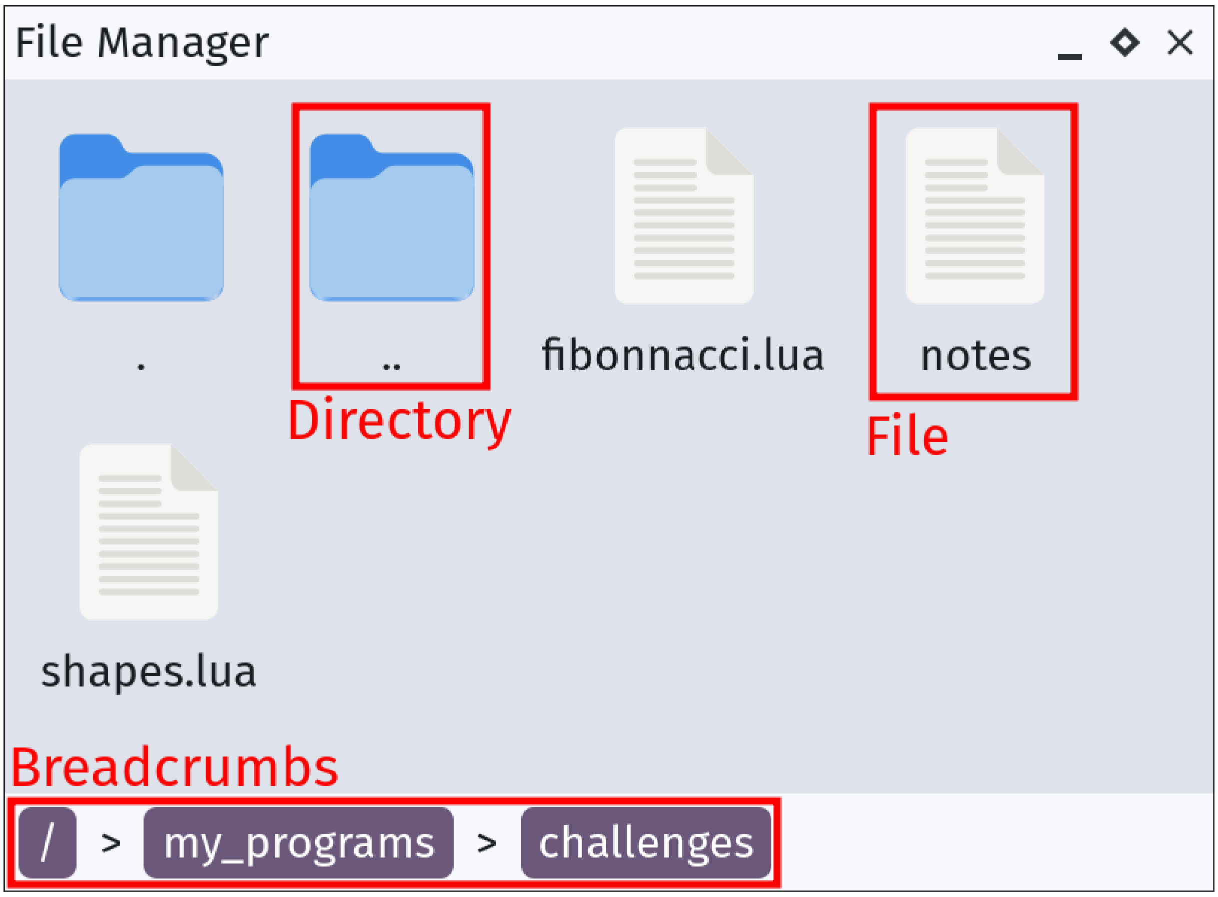
You can also alternatively force a save with the 'Ctrl + S'.

1.6.2.3 File Browser The file browser is an application that provides a graphical interface for you to view, organise and manage the files and folders on Hako.

The primary features of the file browser are:

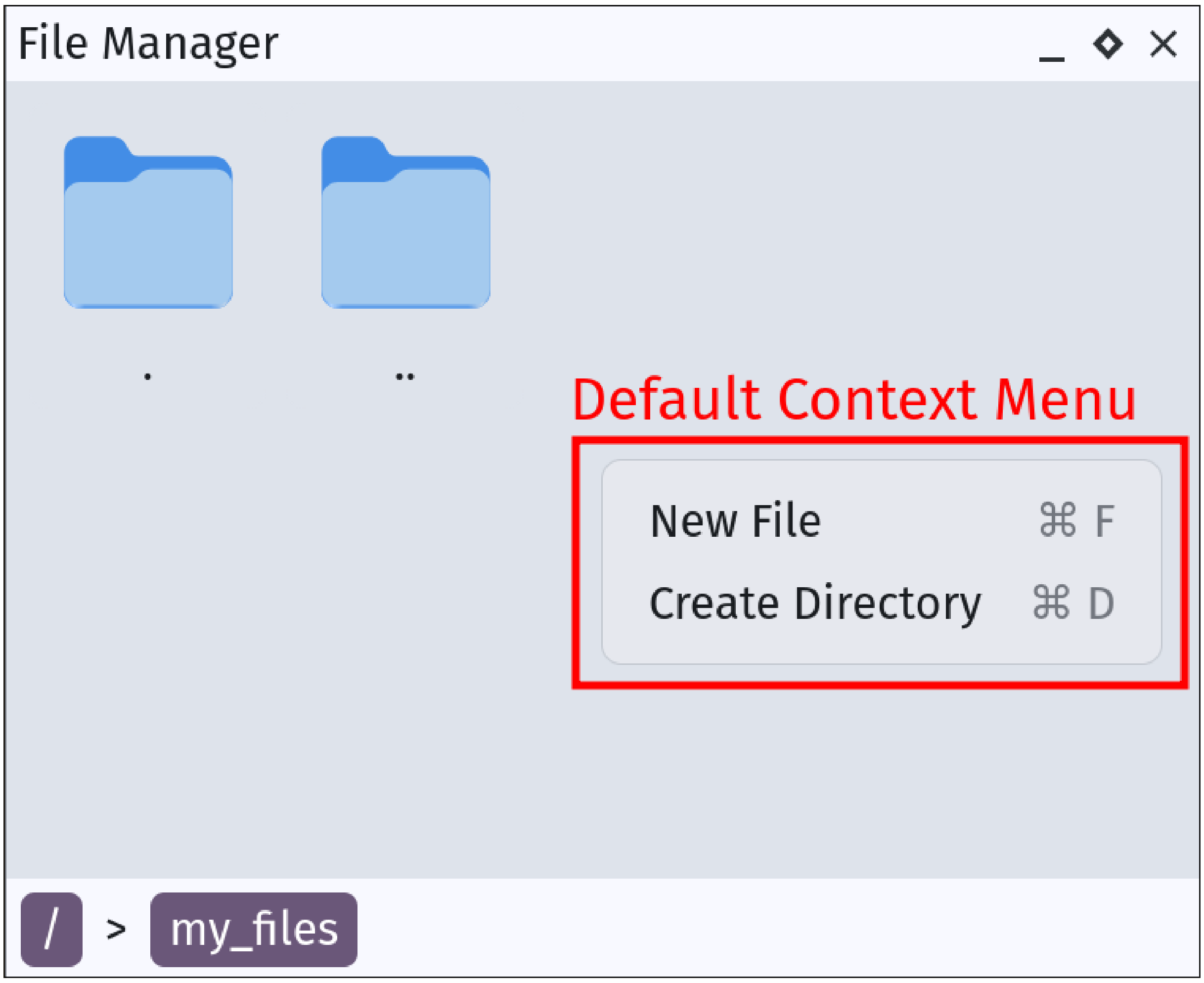
- Viewing your filesystem
- Traversing your filesystem
- Creating files or directories

- Renaming files or directories
- Deleting files or directories
- Drag and drop for moving files or directories



Directories have a "folder" icon. Files have a "file" icon.

The bottom-left contains "breadcrumbs", which shows your relative path from the root (NOTE: you can click them too).

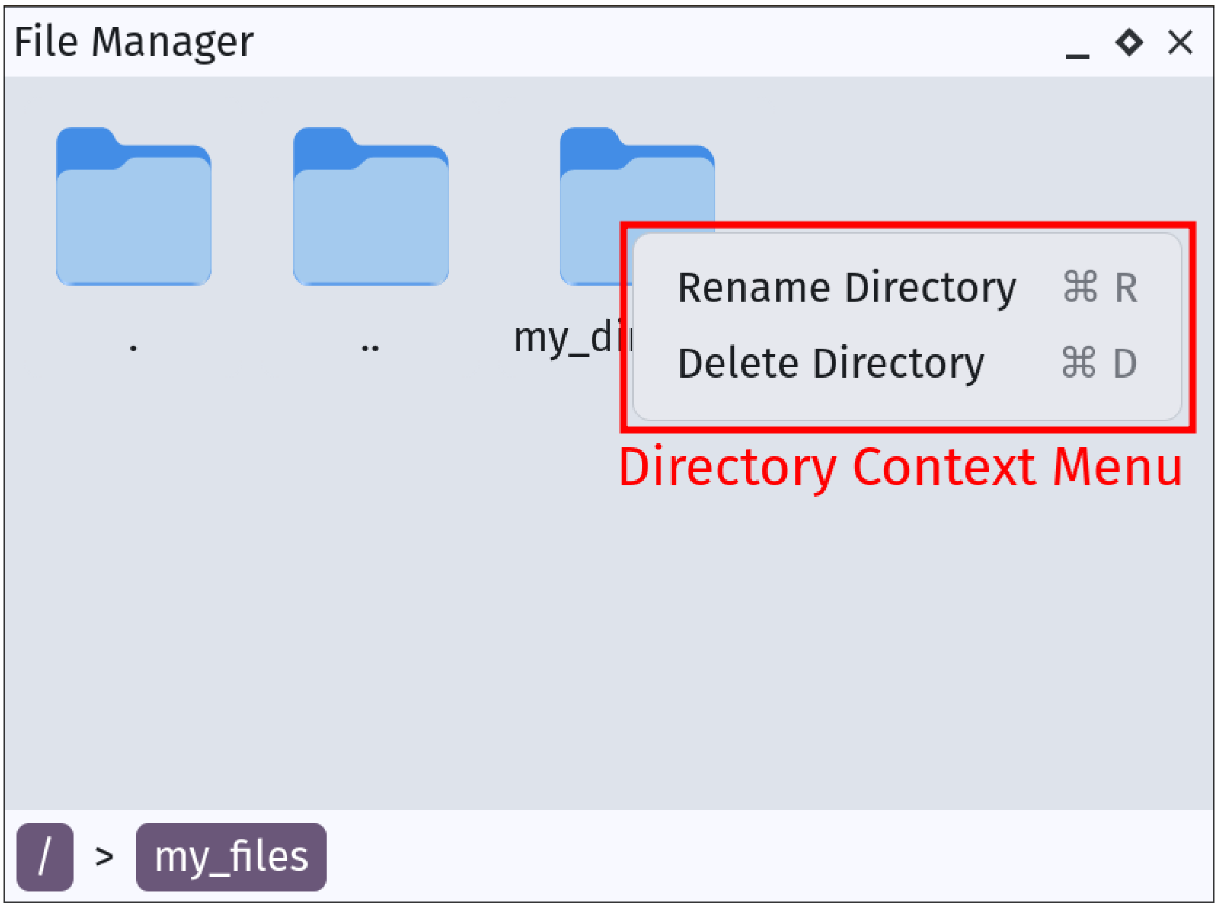


You can “right-click” on the background of the file browser to open the general context menu.

This provides you with two options:

- **New file** – Creates a new file, prompts you for the name
- **Create directory** – Creates a new directory, prompts you for the name

You can also use the keybinds denoted in the context menu instead of directly clicking the options.

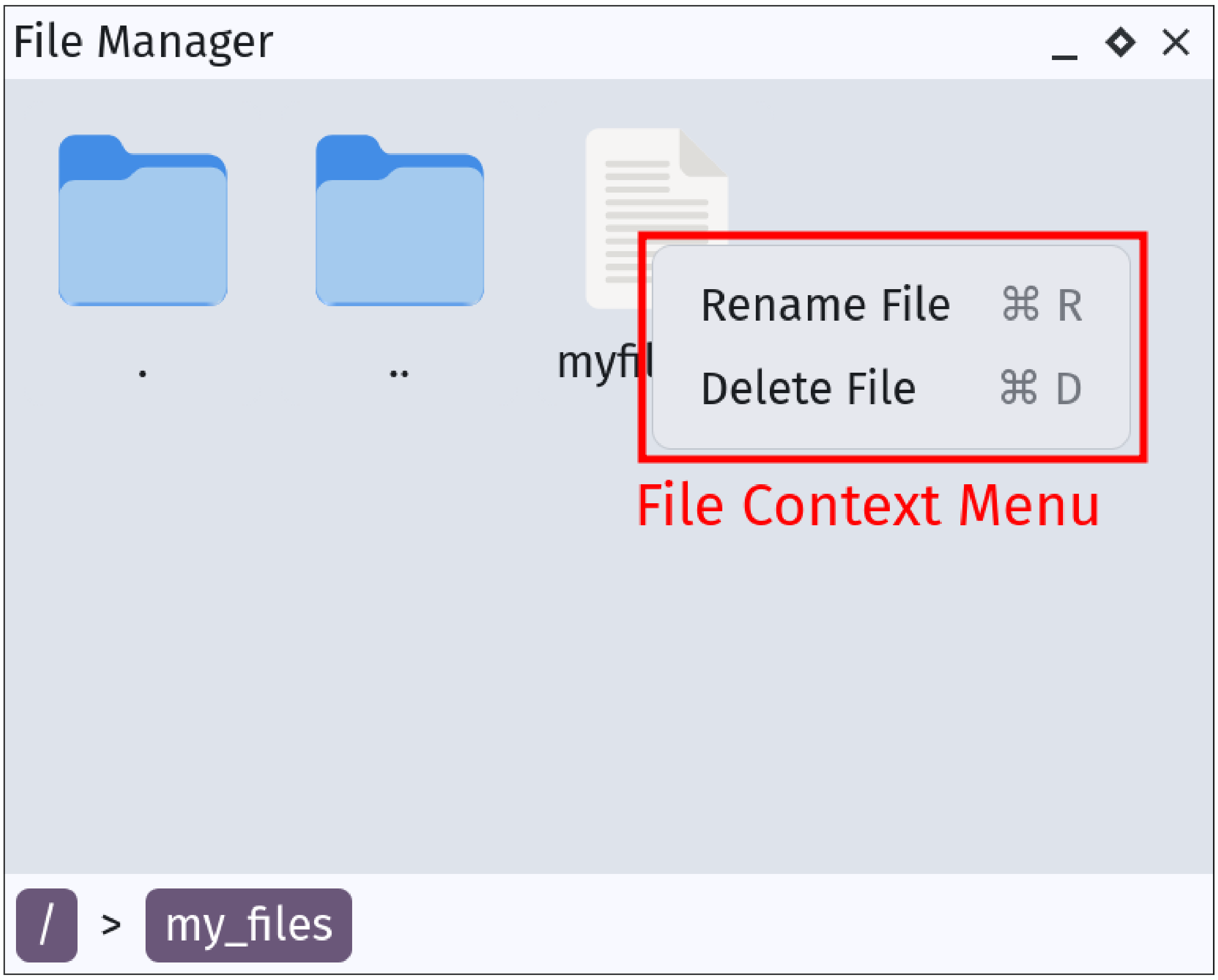


You can “right-click” on a directory to open the directory context menu.

This provides you with two options:

- **Rename directory** – Rename the directory
- **Delete directory** – Delete the directory

You can also use the keybinds denoted in the context menu instead of directly clicking the options.



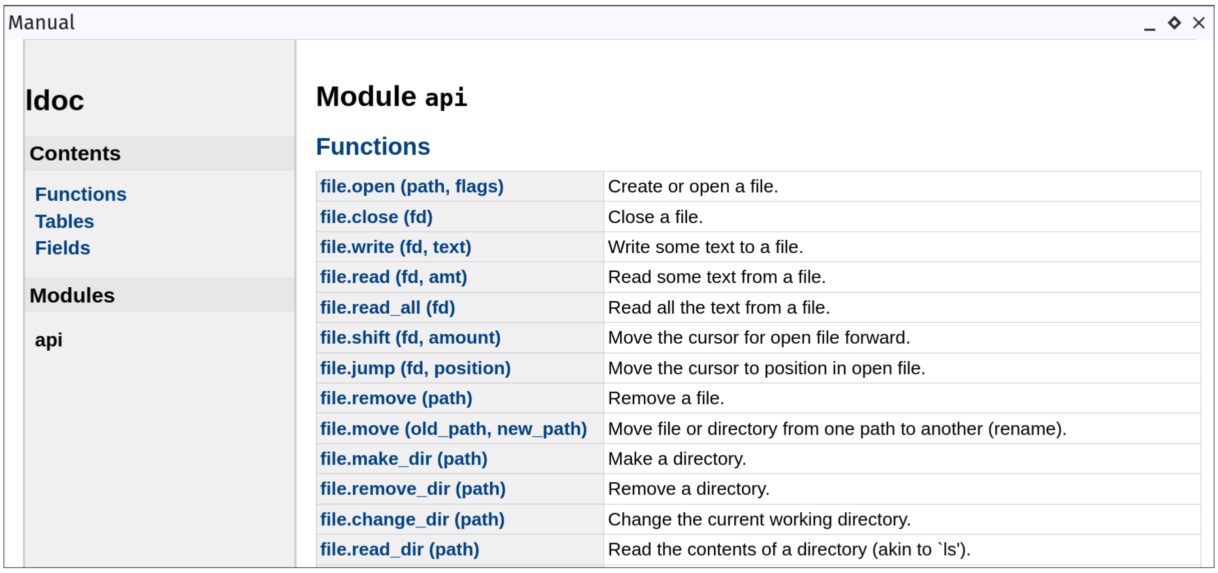
You can “right-click” on a file to open the file context menu.

This provides you with two options:

- **Rename file** – Rename the file
- **Delete file** – Delete the file

You can also use the keybinds denoted in the context menu instead of directly clicking the options.

1.6.2.4 Manual



Module `api`

Functions

file.open (path, flags)	Create or open a file.
file.close (fd)	Close a file.
file.write (fd, text)	Write some text to a file.
file.read (fd, amt)	Read some text from a file.
file.read_all (fd)	Read all the text from a file.
file.shift (fd, amount)	Move the cursor for open file forward.
file.jump (fd, position)	Move the cursor to position in open file.
file.remove (path)	Remove a file.
file.move (old_path, new_path)	Move file or directory from one path to another (rename).
file.make_dir (path)	Make a directory.
file.remove_dir (path)	Remove a directory.
file.change_dir (path)	Change the current working directory.
file.read_dir (path)	Read the contents of a directory (akin to `ls').

Hako has multiple system APIs, described below in the "System APIs" section.

The manual is a full document describing all the different ways to interact with Hako.

The alternative is the `help` command that you can enter into your terminal, which describes Hako's system APIs at a higher level with examples

1.6.3 Core-Utils

Hako comes with a powerful set of core-utils to make interacting with your system easy and productive.

Core-utils are a set of essential command-line programs that are fundamental for interacting with your operating system.

The core-utils are in the `/bin` directory, and by default, any files in this directory are accessible anywhere in your system from your shell.

For example, the `ls.lua` core-util can be accessed no matter where you are in your filesystem with the `ls` or `ls.lua` command – and you can place any tool you create yourself in here to access them from anywhere in your system too!

1.6.3.1 ls

List the files in a directory.

If there is no directory listed, it defaults to listing the current directory.

Help Message:

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default)
Sorts entries alphabetically by default.

Options:
 -1, --one          list one file per line
 -a, --all          do not ignore directories starting with .
 -A, --almost-all   do not list implied . and ..
 -R, --recurse      follow sub-directories and list their contents too
 -l, --long         long format
 -i, --inode        list inode numbers
 -s, --block        lists allocated blocks
 -p,                append / to directory names
 -c, --ctime        list ctime
 -u, --atime        list atime
 -h, --human        list human readable sizes (1K, 243M, 2G)
 -S, --size         sort by size
 -X, --ext          sort by extension
 -t, --smtime       sort by mtime
 -m, --sctime       sort by ctime
 -e, --satime       sort by atime
 -r, --reverse      reverse sort order
```

Code:

The code is available at `/bin/ls.lua`.

1.6.3.2 cat

Con**cat**enates and prints files.

Typically used to print files out at the command line to see their contents.

Help Message:

```
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s) to standard output.
With no FILE, or when FILE is -, read standard input.

Options:
  -n, --number      number all output lines
  -E, --show-ends   display $ at the end of each line
  -T, --show-tabs   display TAB characters as ^I
  -h, --help        display this help and exit
```

Code:

The code is available at `/bin/cat.lua`.

1.6.3.3 cp

Copies files or directories.

Help Message:

```
Usage: cp [OPTION]... SOURCE... DEST

Copy SOURCE to DEST, or multiple SOURCE(s) into directory DEST.

Options:
  -f, --force         overwrite without prompting
  -n, --no-clobber    do not overwrite existing files
  -i, --interactive   prompt before overwrite
  -r, --recursive     copy directories recursively
  -h, --help          display this help and exit
```

Code:

The code is available at `/bin/cp.lua`.

1.6.3.4 mv

Moves / renames files or directories.

Help Message:

```
Usage: mv [-finT] SOURCE DEST
or: mv [-fin] SOURCE... DIRECTORY

Rename SOURCE to DEST, or move SOURCEs to DIRECTORY

Options:
 -f  Don't prompt before overwriting
 -i  Interactive, prompt before overwrite
 -n  Don't overwrite an existing file
 -T  Refuse to move if DEST is a directory
 -h  Show this help message
```

Code:

The code is available at `/bin/mv.lua`.

1.6.3.5 rm

Removes files or directories.

Will not remove a directory unless `-r` is specified.

Help Message:

```
Usage: rm [OPTION]... FILE...
Remove (unlink) each FILE.
By default, it does not remove directories. Use -r to do so.

Options:
  -f, --force          ignore nonexistent files; never prompt
  -i, --interactive    prompt before every removal
  -r, --recursive      remove directories and their contents recursively
  -h, --help           display this help and exit
```

Code:

The code is available at `/bin/rm.lua`.

1.6.3.6 mkdir

Makes a directory.

Help Message:

```
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

Options:
  -h, --help     displays this help and then exits
```

Code:

The code is available at `/bin/mkdir.lua`.

1.6.3.7 rmdir

Removes a directory (only if it's empty).

Help Message:

```
Usage: rmdir DIRECTORY...
Remove DIRECTORY if it is empty
```

Code:

The code is available at `/bin/rmdir.lua`.

1.6.3.8 touch

Create empty files.

Useful for creating files before opening or manipulating.

Example: `touch script.lua`.

Help Message:

```
Usage: touch FILE...

Create or access files.
```

Code:

The code is available at `/bin/touch.lua`.

1.6.3.9 find

Search for files.

Also allows much more complex behaviour, such as executing a command on each file via the `-exec` flag.

Help Message:

```
Usage: find [OPTION|PATH]...

Search for files and perform actions on them.

Options:
    -name PATTERN Match file name (without directory name) to PATTERN
   -iname PATTERN Case insensitive -name
    -path PATTERN Match path to PATTERN
   -ipath PATTERN Case insensitive -path
   -atime DAYS    Match access time greater or equal than DAYS
   -mtime DAYS    Match modified time greater or equal than DAYS
   -ctime DAYS    Match creation time greater or equal than DAYS
    -type X       File type is X (one of: f,d)
 -pattern PATTERN Match pattern (lua pattern match string)
-maxdepth N       Descend at most N levels
-mindepth N       Don't act on first N levels
   -empty         Match empty file/directory
   -print         Print file name (default)
       -h         Print this help message.
```

Code:

The code is available at /bin/find.lua

1.6.3.10 grep

Search for patterns in text.

Help Message:

```
Search for PATTERN in each FILE.
Example: grep -r ipairs sys

Options:
 -l         Show only names of files that match
 -L         Show only names of files that don't match
 -o         Show only the matching part of line
 -i         Ignore case
 -n         Add 'line_no:' prefix
 -r         Recurse directories
 -h         Do not add 'filename:' prefix
 -H         Add 'filename:' prefix
 -v         Select non-matching lines
 -q         Quiet. Return 0 if PATTERN is found, 1 otherwise
 -s         Suppress open and read errors
 -e PATTERN Pattern to match
```

Code:

The code is available at /bin/grep.lua.

1.6.3.11 chmod

Change file and directory permissions.

Help Message:

```
Usage: chmod [OPTION]... [r|w|x] FILE...
Change the permissions of files.

Options:
 -R  Recurse on directories
 -f  Hide errors
 -h  Print this help message
```

Code:

The code is available at /bin/chmod.lua.

1.6.3.12 Shell

All operating systems (Windows, MacOS, Linux), including Hako, have shells to control them with.

The shell gives you a text interface, letting you type commands instead of clicking on graphical elements, to control your operating system.

Change Directory:

To change the directory of your shell, you can use the `cd` (Change Directory) command.

Example:

- `cd /sys` will bring you into the `sys` directory.

Pipelines:

You can pipe the output of one command into the input of another using the | operator.

Example:

- `ls -l | grep hello.txt` causes grep to search the ls comman's output for `hello.txt`

Short Circuit Evaluation:

You can join pipelines together with logical operators '&&' (AND) and '||' (OR).

- && – Executes the next pipeline IFF the previous one was succesful (the exit code was 0)
- || – Executes the next pipeline IFF the previous one failed (the exit code was not 0)

Examples:

- `cat hello.txt || echo "hello world"> hello.txt` – if `cat` fails (hello.txt doesn't exist), we create it by echoing into `hello.txt`
- `ls bad_dir && rm -rf bad_dir` – if `ls` succeeds (bad_dir exists), then we delete it.

I/O Redirection:

You can redirect the input or output to your commands.

- < – redirects stdin
- > – redirects stdout

Examples:

- `echo hello world > hello.txt` – writes to (or creates) hello.txt, with content "hello world"

- `cat - < hello.txt` – cat uses stdin, and stdin is replaced with `hello.txt`, so we print the contents of `hello.txt` (equivelant to `cat hello.txt`)

Grouping:

You can group your commands with operators `{`, `}`. This does not spin up a sub-shell.

Multiple Lines:

You can have multiple lines by using the `;` operator.

Example:

- `ls -l ; touch foobar.txt` – runs the two commands seperately

Subshell:

Can use `--subshell` to execute commands in a sub-shell (seperate shell), everything after `--subshell` will be the input to the subshell.

Code:

The code is available at `/bin/shell.lua`.

1.6.3.13 ps

List running processes.

Help Message:

```
Usage: ps [OPTION]
List current processes.

Options:
  -h        display this help and exit
  --help    display this help and exit
```

Code:

The code is available at `/bin/shell.lua`.

1.6.3.14 kill

Kills a running process.

Help Message:

```
Usage: kill PID...
```

Code:

The code is available at `/bin/kill.lua`.

1.6.3.15 lua

Runs Lua code in the shell

Example:

- `lua 'output("Hello, Hako!")'`

Code:

The code is available at `/bin/lua.lua`.

1.6.3.16 clear

Clears the screen.

You can also type `Ctrl+L` to clear the terminal too!

Code:

The code is available at `/bin/clear.lua`.

1.6.3.17 pwd

Prints the current directory.

Code:

The code is available at `/bin/pwd.lua`.

1.6.3.18 echo

Displays a line of text.

Example:

- `echo hello Hako "this is a test!"` outputs `hello Hako this is a test!`.

Code:

The code is available at `/bin/echo.lua`.

1.6.4 System APIs

Hako exposes many APIs for you to manipulate the operating system with, it's how you can ask Hako to "do something".

1.6.4.1 Desktop Environment API (window)

Hako's desktop environment API (Also known as window api) lets you create, hide/show, move, resize and focus GUI windows of predefined types.

Window Types (Global constants in Lua):

```
  TERMINAL = 0.0,
  FILE_MANAGER = 1.0,
  EDITOR = 2.0,
  MANUAL = 3.0,
```

API Methods:

```
size = window.area()
list = window.list()
size = window.dimensions(id)
pos  = window.position(id)
id   = window.open(window_type)
window.hide(id)
window.show(id)
window.focus(id)
window.move(id, x, y)
window.resize(id, w, h)
window.close(id)
```

Examples:

```
-- Open a terminal window
local id = window.open(TERMINAL)

-- Create a window and keep moving it
local id = window.open(TERMINAL)
for true do
```

```lua
    local position = window.position(id)
    window.move(id, position.x + 10, position.y + 10)
end
```

For more information on the API methods, see the API Manual in Hako.

1.6.4.2 Process API (process)

Hako's process API allows you to launch new Lua scripts as subprocesses, send and receive their I/O, wait for them to exit, or prematurely terminate them.

Creation (`create`) does not start execution until `start(pid)`!

Process I/O Labels (Global Constants in Lua):

```lua
  STDIN = 0.0,
  STDOUT = 1.0,
```

API Methods:

```lua
pid, err  = process.create(path, opts)
err       = process.start(pid)
err       = process.kill(pid)
err       = process.wait(pid)
list      = process.list()
err       = process.output(text, opts)
err       = process.close_output()
str, err  = process.input()
line, err = process.input_line()
str, err  = process.input_all()
err       = process.close_input()
ok, err   = process.isatty(I_O_LABEL)
err       = process.pipe(in_pid, out_pid)
pid, err  = process.get_pid()
process.exit(code)
```

> Furthermore, `process.output` has additional alias to `output`, as it's a common function call, so you can just call `output` in your code.

Options for the `create` method and their default values:

```lua
opts = {
    argv = {}, -- A list of the arguments to pass to the process
    pipe_in = false, -- Tells process to take input from pipe instead
       of terminal
    pipe_out = false, -- Tells process to output to pipe instead of
       terminal
    redirect_in = "", -- Path to redirect input from the filesystem
    redirect_out = "", -- Path to redirect output to in the filesystem
}
```

Options for the `output` method and their default values:

```
opts = {
    newline = true -- Appends a newline character after the text
}
```

Examples:

```
-- run 'ls' in a process
local pid, err = process.create("/bin/ls.lua", { argv = {"-l", "."}})
if err then
    output("Failed to create process: " .. errors.as_string(err))
end
process.start(pid)
process.wait(pid)

-- get the current process's pid
local pid, err = process.get_pid()
if not err then output("My pid: " .. tostring(pid)) end
```

For more information on the API methods, see the API Manual in Hako.

1.6.4.3 Filesystem API (file)

The filesystem API provdes low-level primitives for creating, reading, writing, moving and deleting files and directories.

API Methods:

```
fd, err = file.open(path, flags)
err     = file.close(fd)
n, err  = file.write(fd, text)
n, err  = file.read(fd, amt)
s, err  = file.read_all(fd)
err     = file.shift(fd, amount)
err     = file.jump(fd, position)
err     = file.remove(path)
err     = file.move(old_path, new_path)
err     = file.make_dir(path)
err     = file.remove_dir(path)
entries, err = file.read_dir(path)
info, err    = file.stat(path)
info, err    = file.fdstat(fd)
cwd, err     = file.cwd()
```

```lua
err      = file.change_dir(path)
err      = file.permit(fd, flags)
```

Open flags for opening a file are any combination of:

- `"r"` = read – Open with read access
- `"w"` = write – Open with write access
- `"c"` = create – Create if it doesn't exist (fails if it does exist)

Permission flags for changing the permissions of a file/directory are any combination of:

- `"r"` = read – Can be opened for reading
- `"w"` = write – Can be opened for writing
- `"x"` = execute – Can be executed

`stat` returns a data structure of the filesystem node stat'd, containing:

```lua
info = {
    ctime = { -- When the node was created
        nsec = 0.0, -- nano seconds
        sec = 0.0 -- seconds
    },
    mtime = { -- The last time the node was changed
        nsec = 0.0,
        sec = 0.0
    },
    atime = { -- The last time the node was accessed
        nsec = 0.0,
        sec = 0.0
    },
    blocks = 0.0, -- how many blocks of storage the node uses
    blocksize = 4096.0, -- the size of a block on disk
    ino = 0.0, -- the inode of the node (its unique identifier in the
       filesystem)
    perm = "rwx", -- The permissions of the node
    size = 0, -- The size of the node, in bytes
    type = DIRECTORY | FILE -- What type of node it is

}
```

> `DIRECTORY` and `FILE` are global constants in Lua.

Examples:

```lua
-- Creating and writing "Hello" into new file `notes.txt`
local fd, err = file.open("notes.txt", "wc")
if err then output(errors.as_string(err)) end
file.write(fd, "Hello")
file.close(fd)
```

```lua

-- List the current directory's contents
local entries, err = file.read_dir(".")
for _, entry in ipairs(entries) do
    output(entry)
end
```

For more information on the API methods, see the API Manual in Hako.

1.6.4.4 Terminal API (terminal)

The `terminal` API offers simple control over your text console:

- Clearing the screen
- Prompting the user and receiving input
- Getting the width of the terminal
- Getting the height of the terminal

API Methods:

```lua
err  = terminal.clear()
text = terminal.prompt(prompt_text)
w    = terminal.width()
h    = terminal.height()
```

Examples:

```lua
-- Clearing the terminal
terminal.clear()

-- Prompting the user for their name
local name = terminal.prompt("What is your name? ")
output("Your name: " .. name)
```

For more information on the API methods, see the API Manual in Hako.

1.6.4.5 Format API (fmt)

The `fmt` API allows you to easily format time. It wraps Lua's existing `os.date` and `os.time` methods.

- `fmt.date` – returns a formatted string (or table) for a given epoch seconds or the current time
- `fmt.time` – returns the current timestamp or converts a date-table into seconds since epoch

API Methods:

```
str = fmt.date(format, time)
t   = fmt.time(date_table)
```

Examples:

```
output("Now: " .. fmt.date("Y%-%m-%d %H:%M:%S"))
```

For more information on the API methods, see the API Manual in Hako.

1.6.4.6 Error API (errors)

The `errors` API converts numeric error codes into human-readable strings and provides a convenient `ok()` helper to abort with context when a call fails.

API Methods:

```
-- Aborting with `errors.ok`
local fd, err = file.open("foo.txt", "r")
errors.ok(err, "opening foo.txt") -- exits if `err` is not nil

-- Converting an error code to a string
local fd, err = file.open("foo.txt", "r")
if err then output("Error: " .. errors.as_string(err)) end
```

For more information on the API methods, see the API Manual in Hako.