



Architecture & Deployment

2025-2026 v0.1.0 on branch main Rev: 03b1cdace14bb0b0720e24be862097b3792214ea









Hello Shell

In this exercise, you'll set up a mini treasure hunt for adventurers navigating your server via the Command Line. You'll automate this hunt using Shell Scripting. This activity aims to help you become more familiar with the command line's fundamental tools and the automation of tasks through basic scripting.



Execute these tasks solely using Terminal or Git Bash. Utilizing GUI tools would defeat the purpose.

Table of contents

-  Legend
-  Creating directories and files
-  Adding clues
-  Test the treasure hunt
-  Automate the treasure hunt using shell scripting
-  Make auto_hunt executable
-  Running auto_hunt from any directory.
-  What just happened?



Legend

Parts of this exercise are annotated with the following icons:



A task you MUST perform to complete the exercise

- ? An optional step that you may perform to make sure that everything is working correctly, or to set up additional tools that are not required but can help you
- 🏁 The end of the exercise
- 🏠 The architecture of the software you ran or deployed during this exercise.
- 🔥 Troubleshooting tips: how to fix common problems you might encounter

! Creating directories and files

We're already familiar with the `pwd`, `cd`, `ls`, `mkdir`, `touch`, `echo`, and `cat` commands. Using the relevant commands, perform the tasks below:

- Starting from your home (`~`) directory, create a new directory named `treasure_hunt`.
- Within the `treasure_hunt` directory, craft three additional directories: `cave`, `forest`, and `lake`.
- Create the following files: `cave/echo.txt`, `lake/fish.txt`, and `forest/tree.txt`.



In Unix-like operating systems, the tilde (`~`) symbol is a shorthand representation for a user's home directory. It's a convenient way to refer to this directory without needing to know or type the full path.

For instance, if a user's home directory is `/home/username`, typing `cd ~` in the terminal would navigate them directly to that location. The tilde is recognized and expanded to the full path by the shell, making it an efficient shortcut. Additionally, the tilde can be combined with other directory or file names, such as `~/Documents`, to quickly reference subdirectories or files within the home directory.

The adoption of the tilde as a shortcut has become a deeply ingrained convention in the command-line world, providing users with a quick and consistent way to access their personal files and settings.

Challenge

As an added challenge: find a way to perform this step of the exercise using no more than two commands.

! Adding clues

Using your preferred method, update the files we just created with the specified content:

- `cave/echo.txt`: To uncover the next clue, explore where the water flows.
- `lake/fish.txt`: Venture deep into the woods to discover the last hint.
- `forest/tree.txt`: `curl parrot.live`



The `curl` command is a versatile tool used primarily for transferring data using various protocols, most commonly HTTP and HTTPS. For beginners diving into the world of command-line operations, think of `curl` as a way to communicate with websites and servers directly from the terminal without the need for a web browser.

Whether you're trying to fetch the contents of a web page, download a file, or interact with APIs, `curl` is your go-to utility. Its name stands for **C**lient **U**RL, underscoring its capability to work with URLs to retrieve or send data.

Beginners often start with basic `curl` commands, like `curl https://example.com`, which fetches and displays the content of the specified web page in the terminal. As users become more accustomed to it, they'll find that `curl` offers a wide range of options and parameters to

customize requests, making it an indispensable tool for many developers and system administrators.

? Test the treasure hunt

Ensure your setup is in order:

- Transition to the `treasure_hunt` directory.
- Examine the contents of `echo.txt` within the cave directory.
- Guided by the clue, proceed to the subsequent directory.
- Unravel the next hint and move forward accordingly.
- Input the concluding hint into the terminal.
- Relish your discovered treasure.

! Automate the treasure hunt using shell scripting

Follow these steps to script and automate your entire treasure hunt:

- In the `treasure_hunt` directory, create a file named `auto_hunt`.
- Launch your go-to command-line text editor to edit the `auto_hunt` file.
- Add the following line at the top of the file: `#!/bin/bash`
- Systematically script the commands to journey through the treasure hunt. Introduce a theatrical pause of 2 seconds between commands using the `sleep` command for heightened suspense.



For a touch of efficiency, consider crafting a function that merges the file reading and short delay. This promotes reusability throughout your script.



The `sleep` command is a simple yet useful utility in Unix-like operating systems that pauses the execution of a program or script for a specified duration. For beginners getting acquainted with scripting or command-line tasks, think of sleep as a way to introduce deliberate delays. By inputting sleep followed by a number, the system will pause for that many seconds. For instance, `sleep 5` will introduce a pause of five seconds.

- To execute the command found within `forest/tree.txt`, incorporate: `sh forest/tree.txt`.
- Preserve your hard work by saving the script and gracefully exiting your text editor.
- Jumpstart your treasure hunt automation with the command:

```
$> sh auto_hunt
```



Solution

```
#!/bin/bash
cd ~/treasure_hunt

read_clue() {
    cat $1
    sleep 2
}

read_clue cave/echo.txt

read_clue lake/fish.txt

read_clue forest/tree.txt
```

```
sh forest/tree.txt
```

! Make `auto_hunt` executable

Currently, to execute the shell script, you must use the `sh` command followed by the script's precise filepath. Assuming you are in the home directory, try running:

```
$> ./auto_hunt  
permission denied: ./auto_hunt
```

The error message `permission denied: ./auto_hunt` that you see indicates that the shell has been denied the permission to execute the file named `auto_hunt`. In Unix-like operating systems, files have certain permissions associated with them, determining who can read, write, or execute them. When you try to run `./auto_hunt` without the necessary execute permission, the system prevents it from being executed, leading to this error.

To address this, let's grant the `auto_hunt` script execute permissions:

```
$> chmod +x auto_hunt
```

More information

At this juncture in the course, delving into the intricacies of Unix permissions isn't required. We'll embark on a deeper exploration of this topic as the semester progresses.

Running the script now works:

```
$> ./auto_hunt
```

To find the next clue, search where the water flows

...

! Running `auto_hunt` from any directory

Wouldn't it be convenient to execute this script without specifying its full path, much like the other commands we've utilized so far? Give this a shot by trying:

```
$> auto_hunt
```

```
command not found: auto_hunt
```

The error message `command not found: auto_hunt` essentially means that the shell couldn't find a command or program named `auto_hunt` in the places it usually looks for such commands. When you type a command in the terminal, the shell searches for that command in a list of directories specified in a variable called `PATH`. If the command or program isn't located in any of these directories, you'll get the "command not found" error.

Let's probe where our shell currently scouts for executable programs:

```
$> echo $PATH
```

```
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
```

The above command displays the contents of the `PATH` environment variable. The response enumerates the directories where the shell scans for executable files. As evident, the `~/treasure_hunt` directory is conspicuously absent from this compilation. Consequently, our shell remains oblivious to any executable linked with the `auto_hunt` command.

To add `auto_hunt` to your `PATH`, you need to update the `PATH` environment variable to include the directory containing the `auto_hunt` script. Here's how you can do it:

You can temporarily add the directory to your `PATH` for the current session with:

```
$> export PATH=$PATH:~/treasure_hunt
```

If you now run the following, you will see that your script is executed:

```
$> auto_hunt
To find the next clue, search where the water flows
...
```

Restart your shell and attempt executing `auto_hunt` once more. Oops! It seems our `PATH` has reverted to its original configuration.

To permanently change the `PATH`, you'll need to add the export line to your shell's initialization file. The specific file depends on the shell you're using:

- For `bash` (Git Bash), it's typically `~/.bashrc` or `~/.bash_profile`.
- For `zsh` (MacOS), it's `~/.zshrc`.

With your preferred command-line text editor, append the following line to the end of your shell's initialization file:

```
export PATH=$PATH:~/treasure_hunt
```

Let's break this line down.

- `export`: This command tells the shell to make a variable available for other processes or commands that come after. When you export a variable, it's like

announcing to programs and scripts you might run next, “Hey, you can use this!”

- **PATH**: This is one of the most critical environment variables in Unix-like operating systems. It tells the shell where to look for executable files in response to commands entered by the user. Its value is a list of directories separated by colons (:).
- **\$PATH**: Here, the \$ is used to retrieve the current value of the **PATH** variable. So, **\$PATH** represents whatever directories are currently in your **PATH**.
- **:**: In the context of the **PATH** variable, the colon (:) is used as a delimiter to separate different directory paths.
- **~/treasure_hunt**: This is a directory named treasure_hunt located within the user’s home directory.

In this command, we merge several elements together. Firstly, **\$PATH** retrieves the present **PATH** value. Then, **~/treasure_hunt** gets tacked onto that value. Essentially, this operation adds the **treasure_hunt** directory in the user’s home to the roster of directories the shell peruses when seeking executables. Put plainly, after initiating this command, the shell will extend its search to the **~/treasure_hunt** directory whenever a command is run, supplementing the directories already listed in your **PATH**.

To incorporate the modifications made to the startup file without restarting your terminal, simply “source” the file:

For Bash:

```
$> source ~/.bash_profile
```

For zsh:

```
$> source ~/.zshrc
```

Now, the **auto_hunt** command should be accessible from any location in the terminal.

```
$> auto_hunt
```

To find the next clue, search where the water flows

```
...
```

```
        .ccccccc.
      ,,,,;cooolccoo;,,,
    .d0x;..;lllll;..;x0d.
  .cd0;' ,lo0XXXXXkll;' ;odc.
,ol; ;c, ':oko:cccccc,...ckl.
; c.;kXo.....;c::' .....oc
,dc..oXX0kk0o.':lll;..cxxc.,ld,
kNo.'oXXXXXXo',:lll;..oXX0o;c0d.
K0c;o0XXXXXXo.':lol;..dXXXXl';xc
0l,:k0XXXXXX0c.,clc':.0XXXXx,.oc
K0c;d0XXXXXXXl..'';'..lXXXXXo..oc
dNo..oXXXXXXX0x:..' 'lx0XXXXXk,..; ..
cNo..lXXXXXXXXX0olkXXXXXXXXXXkl,..;:';
.,;'. ,dkkkkk0XXXXXXXXXXXXXXXXX0xxl;,,,;l:.
; c.;:' ''':do0XXXXXXXXXXXXXXXXXXXX0do;' ;clc.
; c.l0dood:' ''oXXXXXXXXXXXXXXXXXXXXXk,..;ol.
';.:xxxxxocccoxxxxxxxxxxxxxxxxxxxxxxl::'.';,.
';.....;l'
```

What just happened?

In this exercise, we navigated through basic Unix commands. We started by creating directories and files, and then crafted a “treasure hunt”, where we used these commands to create and modify files.

We further explored the curl command and its capabilities in interacting with the Internet directly from the command line. Transitioning to shell scripting, we automated the treasure hunt sequence with the `auto_hunt` script.

A challenge arose when trying to run the script from any directory, which led us to tinker with file permissions and the `PATH` environment variable. By modifying permissions and adjusting the `PATH`, we ensured our script was easily accessible from any location in the terminal.

[↑ Back to top](#)