CHAPTER 05

BASICS OF THE SHELL ENVIRONMENT

⚠ WARNING

This chapter is part of the sample edition of **Self-Deployment for Software Developers**.

Get the complete edition with all chapters, examples, and updates at selfdeployment.io↗.

In the previous chapter, we covered the basics of the Linux operating system. In this chapter, we will go over the basics of the shell environment including the prompt, standard streams, exit codes, variables, and built-in commands.

Imagine controlling your computer's every move with simple text commands, bypassing the slower, buggier, **and sometimes nonexistent graphical user interfaces**. This is what the shell, also known as the command line interpreter, offers: a direct and efficient way to interact with the machine.

Being able to use the shell effectively is **vital for any developer or system administrator**. You may not like it, but you cannot avoid it. Even if you prefer using a graphical user interface, knowing how to use the shell environment will provide a deeper understanding of how the tools work and enable you to troubleshoot problems more efficiently.

In today's world, with powerful graphical user interfaces and bright, high-resolution screens, why would anyone prefer using the command line over a GUI? This preference can be attributed to several compelling reasons:

- **It is faster.** The shell is designed to be as efficient as possible, allowing you to perform complex operations with fewer keystrokes. When you use a GUI, you have to navigate through menus and click on buttons, which can be time-consuming and error-prone. Ultimately, for many applications, a GUI is simply a layer on top of the command line.
- **It is powerful.** By nature, the shell supports more advanced features than graphical user interfaces, such as piping, redirection, and scripting. On top of that, the basic set of commands supplied by the

developer demonstrates the intended use. This way, we can use it efficiently.

- **It is more flexible and manageable.** The shell allows you to automate repetitive tasks and customize your workflow to suit your needs. Also, you can easily share the scripts you have created with others or use scripts written by others. Would you like to collect error logs from a program and send them to you via email? You can do it with just a few lines of a shell script.

Last but not least, the shell environment is everywhere. It is available on every operating system and serves as the common language that every program understands. Therefore, learning how to use the shell or at least understanding its basics is an excellent idea.

Whenever we open a terminal, we initiate a shell environment. When a shell is initiated, it executes (loads) the initialization files, sets up the environment variables, and waits for the user to enter commands. It indicates the waiting state of the shell by showing a prompt and a blinking cursor.

The prompt is the text that appears before the cursor, determined by the `PS1` variable. By default, the typical prompt consists of the following information:

- The username that is currently logged in is indicated with the `\u` character.
- The short hostname of the machine, indicated with the `\h` character, is the part of the hostname before the first dot. If you are constantly connecting to different hosts, you might want to use the longer (full)

version of it using the `\H` character. For example, if the full hostname is `node0.chlabs.io`, the short hostname will be `node0`.

- The current working directory (the directory the user is currently in) is indicated with the `\w` character.
- The `$` sign for a regular user and `#` for a superuser.

For instance, `root@node0:~#` prompt means the current user is `root`, it is connected to the host with the shortname of `node0`, currently in its home directory represented by the tilde character `~` (usually `/root` for the root user), and has superuser privileges. As mentioned before, some users and shells override the default prompt to save horizontal space and replace the whole prompt or the user indicator characters with `>` character. Though good for demonstration, using a short prompt will ultimately lead to running commands as an unwanted user on an unintended host in the long run. Because of that, if you need to save on horizontal space, just add a newline character `\n` to the prompt (inside the `PS1` variable). In this way, you will have the best of both worlds.

In this book, we will not use the entire prompt for simplicity and to save horizontal space. Instead, we will use the characters (`$` for regular users and `#` for superusers) to represent it.

As a different common symbol in the shell, a `backslash`, the `\` character, can be used to **split a command into multiple lines**. This is particularly useful because command readability matters as much as code readability, and we want to read it from top to bottom as much as possible, rather than from left to right. We will use this character extensively to split the commands into multiple lines. This way, we can explain the commands line by line or argument by argument. As a side note, please make sure to

not have any spaces after the backslash character **at the end of a line** because it will be considered a part of the command and break the command's syntax.

See the example below about how adding a backslash improves the readability of a command.

```
$ docker login \
  -u nologin \
  --password-stdin \
  rg.fr-par.scw.cloud/chlabs-io
```

05.01. STANDARD STREAMS. STDIN, STDOUT, STDERR

Every shell has three standard streams that are used for communication with the system: standard input (stdin), standard output (stdout), and standard error (stderr). These streams are used to send and receive data between the shell and the running programs. By default, all of these streams are attached to the terminal emulator.

- **Standard input (stdin)** is the stream that is used to send data into a command or program. The terminal emulator connects stdin to the keyboard by default, allowing us to type commands and provide input to the running shell.
- **Standard output (stdout)** receives output from a command or program. This is the type of output most programs produce, including regular log statements. The terminal emulator writes stdout to the

screen, which lets us view the output. Before the video terminals, in the 1960s and 1970s, the teletype terminals printed the stdout to a paper roll.

- **Standard error (stderr)** is where error and most warning messages are sent, such as Node.js' `console.error(...)` and `console.warn(...)` methods or Python's `print` function with `file=sys.stderr` parameter[1]. Similar to stdout, stderr will also be written to the screen, displaying any errors and warnings that occur during execution but often in a different color.

  The standard error stream is separated from standard output for an important reason. This way, users can redirect `stderr` to a file or another program to handle errors separately and avoid missing them in the output.

In some environments, such as a CI/CD pipeline or a cron job, users typically *do not have access to the standard input (stdin) stream* due to the nature of the task. Such environments are referred to as **non-interactive**. The programs intended to be executed in them are designed not to require any user input. Take a package manager, for example, like `apt`. Users can pass the `-y` flag to the `$ apt` command to prevent it from asking for confirmation for anything. Sometimes, shell scripts are also designed to check whether they are running in an interactive shell and behave accordingly. [2]

> **⚠ WARNING**
>
> In Unix-like operating systems, most resources are treated as files, allowing us to read from and write to them when applicable. This includes regular files, named pipes, domain or network sockets, character devices (`/dev/null`, `/dev/tty`, `/dev/random`), and special files (`/proc`).
>
> Since they are treated as files, all these resources have their file descriptors, such as the relative or absolute path for a regular file. For the rest of this chapter, we will mostly use the word *"file"* to refer to **any of these resources.**

In the shell environment, the standard streams are also treated as files and have their file descriptors. These descriptors are `0` for stdin, `1` for stdout, and `2` for stderr. Users can combine and/or redirect these streams, then write them to files or pipe them to other programs using operators.

However, before jumping into the examples, we need to learn three basic commands: `$ cat`, `$ ls`, and `$ grep`. The `$ cat` command is used to output the content of a file (or multiple files) to stdout. The `$ ls` command will list the files and directories in the given path and if no path is given, it will list the files and directories in the current working directory. Lastly, the `$ grep` command will write the lines that match the given regex pattern to the stdout. All of these commands will write errors and warnings to the stderr, for instance, if the file or directory does not exist for the `$ cat` and `$ ls` commands.

Here is the list of the most used operators and their usage examples:

- `>` (redirect) operator redirects the stdout from the left side command to the file at the right. For example, we can use `$ cat file.txt > output.txt` to write the contents of the `file.txt` file to the `output.txt` file. If the `output.txt` file already exists, it will be overwritten. If any error happens during this process, the error will be printed to the terminal because the stderr is not redirected.

- `>>` (append) operator appends the stdout of the left side command to the file specified at the right. For example, we can use `$ cat file.txt >> output.txt` to add the contents of the `file.txt` file to the `output.txt` file. Like the redirect operator, if the `output.txt` file does not exist, it will be created. However, unlike the redirect operator, this operator will not overwrite the file but will simply append to the end of it and *that's why it is my go-to operator* for writing outputs to a file to **prevent accidental overwriting**.

  For these two operators, the `>` and `>>` operators are respectively equal to the `1>` and `1>>` since they redirect the stdout. If we want to redirect the stderr to elsewhere, we can use the `2>` or `2>>` syntax. Additionally, we can use the `2>&1` or its shorthand, `&>`, in `bash` to redirect the stderr to the stdout.

- `|` (pipe) operator pipes the stdout of the left side command to the stdin of the right side command. For instance, the `$ cat file.txt | grep "hello"` command will print the lines that contain the word "hello" in the `file.txt` file to the stdout. *This is probably the most used operator in the shell environment.*

- `<` (input redirect) operator redirects the contents of the file on the right side as the stdin to the command on the left. As a simple example, we can use `$ cat < file.txt` to print the contents of the `file.txt` file to the stdout. Even though we can have the same effect by using other commands and operators, this operator is more readable and concise.

- `&` (background execution) operator runs the command in the background like in the example of `$ npm run dev &`. In this way, we can continue running other commands in parallel (in the foreground or background) and attach to the background process when we need to. However, since stdout and stderr of all processes will still be connected to the terminal, this operator is primarily used when we need to run multiple related, long-running commands together, such as a database and a web server, and access logs and warnings from a single location as they are generated.

- `&&` (and) operator executes the right side command only if the command on the left is successful. For instance, we can use `# apt update && apt upgrade` to update the package list and then upgrade the packages only if the package list update was successful. This operator is mainly used in long chains of commands in CI/CD pipelines.

- `||` (or) operator executes the right side command only if the left side command fails. For example, we can use `$ ls /backups || echo "Directory does not exist"` to list the backup files in the `/backups` directory and print "Directory does not exist" if the directory does not exist.

The first four operators are called the *redirection operators* and are used to redirect the standard streams. By default, these work with standard output rather than the standard error, which helps to enforce a separation of concerns. In this way, we can read the standard error, which contains error and warning messages, without affecting the normal flow of the program.

However, even if it's not common, we can always redirect the standard error to the standard output using the `2>&1` redirect syntax on a command-by-command basis or for the whole chain of commands.

05.02. EXIT CODES

Every command or program executed and **completed** in the shell environment returns an 8-bit unsigned integer as the exit code whether it was successful or failed[3]. It is between `0` and 255, and you may see the exit code referred to as the status, exit status, or return code in some resources. This exit code indicates whether the command or program was completed successfully and, if not, the type of error that occurred. They are essential for understanding the outcome of programs and are used by the shell environment to determine the flow of execution.

Typically, exit code `0` means the command was executed successfully and any other exit code indicates an error. While some programs use exit code `1` as a general error and print the exact failure reason to stderr, some others use specific non-zero codes to indicate the particular failure reason.

Even if a program exited with a `0` indicating a successful execution, it can still print errors or warnings to stderr. For example, a database backup program ran successfully, but it can also print a warning to stderr to indicate the backup was not encrypted or the backup space is running out.

A shell environment has many special parameters, such as `$?`. We can check the exit code of the last command by using the `$?` variable with the command `$ echo $?`.

```
$ ls -alh /nonexistent
ls: cannot access '/nonexistent': No such file or directory
$ echo $?
2
```

When commands are executed in a shell script or a terminal session, the consecutive commands will continue to run (the session will remain active) even if the previous command fails. To change this behavior, we can use the `$ set -e` setting before running the commands to let the shell know that it should exit immediately if any command fails. Likewise, the exit code of a pipeline, such as `$ cmd0 | cmd1 | cmd2`, is determined by the last command, and all the commands will run even if the previous commands failed. To change this behavior and cause a pipeline to fail early, we can use the `$ set -o pipefail` setting.

05.03. VARIABLES AND ENVIRONMENT VARIABLES

When we open the terminal or create a new tab, we are actually starting a new shell session. A shell session is a long-running interactive shell process (like a script) with its own variables. The shell variables are local to the shell session; they can be created, updated, and deleted, and will not be visible to the child processes of the current process. However, when we mark a shell variable with the `export` keyword, it becomes a special type of shell variable called an environment variable, making it inherited by child processes spawned from that shell session.

> **⚠ WARNING**
>
> Like other shell variables, environment variables are copied to the child processes at the time of the process creation. Therefore, updating a variable in the shell will not be carried over (propagated) to the child processes.

Both types of variables store simple key-value information, which programs can use to determine their behavior, such as the shell's current working directory or a database connection string. These variables are case-sensitive and can be accessed using the `$` character followed by the variable name. By default, the shell variables are string pairs and they will be passed to the child processes as strings (both keys and values), however, they can be used as integers, arrays, and other data types within the shell itself and in shell scripts.

In a shell session, we can create a new variable either by assigning a value to it, such as `$ MY_VARIABLE="Hello, World!"`, or we can use the

`$ export` command to create a new variable and declare it as an environment variable at the same time, such as `$ export MY_VARIABLE="Hello, World!"`. Both of these assignments will update an existing variable if it is already defined. Keep in mind that the `$` character at the beginning has nothing to do with the variable; it simply indicates that this expression belongs to a shell, and we shouldn't use spaces around the equal sign.

In the example below, we create a new environment variable, print it using the `$ echo` command, and then clear it using the `$ unset` command. At the end, we list all the environment variables using the `$ env` command to see which variables would be passed to the child processes if we ran a new command.

```
$ export MY_VARIABLE="Hello, World!"
$ echo $MY_VARIABLE
Hello, World!
$ unset MY_VARIABLE
$ env
COMMAND_MODE=unix2003
HOME=/Users/chris
LOGNAME=chris
MallocNanoZone=0
ORIGINAL_XDG_CURRENT_DESKTOP=undefined
...
```

To handle or use **space or special characters** such as `#`, `;`, `&`, `*`, `?`, etc. in variable values, command arguments, or other places where special characters might be interpreted by the shell, we should use the single `'` or double quotes `"` to enclose the value. The difference between the two

is that single quotes do not expand variables, whereas double quotes do. For instance:

```
$ MY_PLANET="World"
$ echo 'Hello, $MY_PLANET!'
Hello, $MY_PLANET!
$ echo "Hello, $MY_PLANET!"
Hello, World!
```

Besides single and double quotes, the backticks (`` ` ``), were used to execute a command and get its output. However, nowadays their use is discouraged, and the `$()` operator is preferred.

```
$ echo date
date
$ echo `date`
Thu Jul 24 15:35:35 -07 2025
$ echo $(date)
Thu Jul 24 15:35:52 -07 2025
$ echo 'The current date is: $(date)'
The current date is: $(date)
$ echo "The current date is: $(date)"
The current date is: Thu Jul 24 15:36:44 -07 2025
```

When a shell session starts, it executes (loads) the initialization scripts, which contain the initial variables and configurations. The order in which these scripts are run depends on the invocation and startup mode, such

as login or non-login, and whether the session is interactive or non-interactive.

Although different shells, such as `zsh`, provide more advanced elements for each case, `bash` offers a simple and easy-to-understand approach with two files. First, if it's a login shell, meaning the user is logging in to the system with this shell session, such as an `SSH` connection or changing the user with the `$ sudo -i` command, the shell will execute the `.bash_profile` file. If it's a non-login shell, such as opening a terminal emulator, which creates a new shell session, this `.bash_profile` file will not be executed, so it is not loaded.

If a terminal or terminal emulator (TTY) is connected to the shell session, it is considered an interactive shell so that it will run the `.bashrc` file regardless of the startup mode. This is why we typically store and override environment variables and other configurations in this file, ensuring they are loaded every time a new interactive session is opened.

Both of these files are typically located in the user's home directory. However, this path can be overridden in the `/etc/passwd` file, which also contains the other information about the user, such as the user's name, ID, and the default shell program's path.

In addition to the shell's initialization scripts, most package managers (like `apt`, `brew`, etc.) also have their initialization scripts, which load or modify variables they need when a session starts. For example, for the shell to be able to run programs installed via the package manager without specifying the full path, the package manager will add a new directory to the `PATH` environment variable. Then, after every installation, the package manager will **add symbolic links** of these programs to this

directory. Additionally, some programs may need to modify variables or execute commands to set up the environment for the program. Because of this, they might want you to restart the shell session. If this is necessary, they typically warn you about it at the end of the installation.

Here are some of the most used environment variables:

- `HOME` variable stores the user's home directory. This is where the `$ cd` command will move you when you run it without any path argument.
- `PWD` is the present working directory. This is the directory you are currently in, which will be used by tools like `Node.js` or `Python` to determine the root directory of the project. When you use the `$ cd` command, it will be updated to the new directory.
- `SHELL` is the path to the current shell program being used. Some shell scripts may need to know the current shell in which they are running specific scripts so that they can behave differently.
- `LANG` variable stores the system's language and locale settings, such as `en_US.UTF-8`. It is used by the programs to determine the language of the output. With this variable, multi-language programs can print the output in your preferred language or encode the output in your preferred encoding.
- `TMPDIR` variable points to the directory where temporary files are stored. If it is empty or not set, the programs will use the `/tmp` directory. Typically, the contents of the `/tmp` directory will regularly be cleaned up by the system.

Before continuing to the next part, I have three tips for you about the shell and the environment variables:

1. Since different shells use different initialization scripts, add an `$ echo "loading from <filename>"` command to the initialization script to understand which file is being loaded at the initialization. Although this might slow down the shell startup time, it is a good way to keep track of the initialization files, especially when using multiple shells or hosts.

2. You can use shell environment variables to your advantage. To illustrate, change the `EDITOR` to your favorite editor so that when you need to edit a file from the command line, it opens your favorite editor. Or, before running a program, you might want to set `TMPDIR` to a different directory to store the temporary files for debugging purposes.

3. For security concerns, you may not want certain environment variables to be visible to all programs. You can use the `$ env | grep -i <term>` command to list the environment variables and filter them in both the keys and the values case-insensitively.

05.04. BUILT-IN SHELL COMMANDS

Shells have built-in commands that are executed by the shell itself without invoking an external program. These commands perform basic operations, such as changing the current working directory, creating aliases, and printing messages to stdout. They are essential for navigating the system and performing basic operations without depending on external programs.

Here are some of the most commonly used built-in shell commands:

- `$ cd` is probably the most used shell command of all time and changes the current working directory. We can use `$ cd <path>` to move to a different directory or use `$ cd ..` to move to the parent directory. Because it is a built-in command, some shell executables come with their unique options. For example, `$ cd -` will switch between the current directory and the previous directory.

  The `PWD` environment variable will be updated to the new directory whenever we use the `$ cd` command. If we execute the `$ cd` command without any arguments, it will take us to our home directory.

- `$ pwd` prints the current working directory to stdout. It is practical when we have a minimal prompt and want to know which directory we are in. We could also echo the `PWD` environment variable for the same result.

- `$ echo <...>` will print its arguments to the stdout and will add a newline character at the end.

- `$ alias` makes a shortcut for a command or set of commands. We can use `$ alias <alias_name>='<command>'` to create shortcuts for long or complex commands, making them easier to remember and use. If we run `$ alias` without any parameters, it will list all the aliases of the current session, which is the same behavior many shell tools do when called without parxameters.

  This is one of the most loved built-in commands by developers and system administrators. For example, most Kubernetes developers alias the `$ kubectl` to `$ k` for easier access. Moreover, many developers carry their alias list to every machine either by changing the shell's initialization files or by opening custom SSH sessions.

You might think `$ ls` or `$ mkdir` commands should be on this list because of the `$ cd`. However, the `$ cd` command changes the state of the shell environment, whereas the `$ ls` and `$ mkdir` commands do not, so they belong to the userland utilities rather than the shell. You can access the complete list of the built-in shell commands from the manual page↗ or using the `$ help` command in `bash`. Additionally, we can use the `$ type <command_name>`, another built-in command, to see which path is associated with the command or if it is an alias, which program it aliases.

05.05. SHELL SCRIPTING

Shell scripting is the process of creating scripts that contain commands for the shell to execute. They are used to automate repetitive tasks, perform complex operations, and customize the behavior of the shell environment. They are essential for system administrators and developers, who use them to perform various tasks, such as managing files, creating backups, installing software, and monitoring system performance.

There are four main ways to run shell scripts within the shell environment:

1. The direct call to the shell script file using either a relative path such as `$ ./script.sh` or an absolute path. With this approach, the `script.sh` file must be executable by the user so that the script can be executed with the correct interpreter, as specified by the shebang line. For example, if the script starts with `#!/bin/bash`, it will be

executed with the `bash` interpreter; if it begins with `#!/usr/bin/python`, it will be executed with the `python` interpreter.

2. Running the script with the `$ bash <script_path>` approach will run the script with the `bash` interpreter regardless of the shebang. It has the same effect as the first approach for the `bash` scripts, but it's more explicit and easier to understand. In both of these methods, a new child process will be created as if it were a new non-interactive shell session.

3. Running the script with the `$ source <script_path>` approach will run the script within the current shell, meaning it will be executed as if it were a part of the current shell session, and there will not be a new child process. In this method, the script can access the shell variables and functions, in addition to the environment variables that are accessible by all the child processes. Moreover, the script will be able to override the environment variables. **This is how the shell's initialization scripts are loaded.**

4. The `$ exec <script_path>` approach will run the script by replacing the shell process. This will give the new script the ability to run commands with `exec` and delegate the control to the new commands by replacing the current process ID (PID). As a result, the old shell process will be terminated. This method is primarily used in container initialization scripts to ensure that signals are properly sent to the intended processes.

Although shell scripts are still usable and remain a foundational tool in DevOps, they are no longer as popular as they were a decade ago. Being older than most programming languages, including C++, shell scripts are

not designed to handle the complex operations we use them for today. Because of this, most teams are not developing shell scripts anymore and are replacing the current ones with newer tools such as Ansible Playbooks↗.

In this book, we will avoid using shell scripts when covering tasks that can be automated, like CI/CD pipelines or backups. Instead, we will use new and more powerful tools, such as zx↗ or bun↗. These tools are designed to replace shell scripts for most use cases and offer more robust features than the shell environment, including better error handling, the ability to run multiple commands in parallel effortlessly, and access to modern third-party libraries.

In this chapter, we covered the basics of the shell environment of the `bash`. In the next chapter, we will cover the basic Linux commands and how to use them to perform various tasks.

FOOTNOTES

1. e.g., `print("Error: User not found...", file=sys.stderr)`. ↵
2. Even if you try to load the `.bashrc` file in a non-interactive environment, the script will check if the shell is running interactively in the shell's logic, usually in the first lines of the configuration file. If the environment is non-interactive, it returns early without executing the rest of the file.

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

Since the rest of the file contains many useful configurations, such as aliases, functions, `PATH` additions, and other environment variables, if a shell is running in a non-interactive environment, the user will not be able to use many commands as they would in an interactive one. The easiest way to solve this problem is to call the commands using their full path or to create another script and set it as the `BASH_ENV` environment variable, so it is loaded in non-interactive environments. ↵

3. Even if a program exits with more than 8 bits of information, the shell will only use the least significant 8 bits (% 256) as the exit code. ↵