Lu Sun, and many more.

A Notebook on Linux Operating System

To my family, friends and communities members who have been dedicating to the presentation of this notebook, and to all students, researchers and faculty members who might find this notebook helpful.

Contents

Foreword

If software and e-books can be made completely open-source, why not a notebook?

This brings me back to the summer of 2009 when I started my third year as a high school student in Harbin No. 3 High School. In the end of August when the results of Gaokao (National College Entrance Examination of China, annually held in July) are released, people from photocopy shops would start selling notebooks photocopies that they claim to be from the top scorers of the exam. Much curious as I was about what these notebooks look like, never have I expected myself to actually learn anything from them, mainly for the following three reasons.

First of all, some (in fact many) of these notebooks were more difficult to understand than the textbooks. I guess we cannot blame the top scorers for being so smart that they sometimes make things extremely brief or overwhelmingly complicated.

Secondly, why would I want to adapt to notebooks of others when I had my own notebooks which in my opinion should be just as good as theirs.

And lastly, as a student in the top-tier high school myself, I knew that the top scorers of the coming year would probably be a schoolmate or a classmate. Why would I want to pay that much money to a complete stranger in a photocopy shop for my friend's notebook, rather than requesting a copy from him or her directly?

However, my mind changed after becoming an undergraduate student in 2010. There were so many modules and materials to learn for a college student, and as an unfortunate result, students were often distracted from digging deeply into a module (For those who were still able to do so, you have my highest respect). The situation became worse when I started pursuing my Ph.D. in 2014. As I had to focus on specific research areas entirely, I could hardly split much time on other irrelevant but still important and interesting contents.

This motivated me to start reading and taking notebooks for selected books and articles, just to force myself to spent time learning new subjects out of my comfort zone. I used to take hand-written notebooks. My very first notebook was on *Numerical Analysis*, an entrance level module for engineering background graduate students. Till today I still have on my hand dozens of these notebooks. Eventually, one day it suddenly came to me: why not digitalize them, and make them accessible online and open-source, and let everyone read and edit it?

—

As most of the open-source software, this notebook (and it applies to the other notebooks in this series as well) does not come with any "warranty" of any kind, meaning that there is no guarantee for the statement and knowledge in this notebook to be absolutely correct as it is not peer reviewed. **Do NOT cite this notebook in your academic research paper or book!** Of course, if you find anything helpful with your research, please trace back to the origin of the citation and double confirm it yourself, then on top of that determine whether or not to use it in your research.

This notebook is suitable as:

- a quick reference guide;
- a brief introduction for beginners of the module;
- a "cheat sheet" for students to prepare for the exam (Don't bring it to the exam unless it is allowed by your lecturer!) or for lecturers to prepare the teaching materials.

This notebook is NOT suitable as:

- a direct research reference;
- a replacement to the textbook;

because as explained the notebook is NOT peer reviewed and it is meant to be simple and easy to read. It is not necessary brief, but all the tedious explanation and derivation, if any, shall be "fold into appendix" and a reader can easily skip those things without any interruption to the reading experience.

—

Although this notebook is open-source, the reference materials of this notebook, including textbooks, journal papers, conference proceedings, etc., may not be open-source. Very likely many of these reference materials are licensed or copyrighted. Please legitimately access these materials and properly use them.

Some of the figures in this notebook is drawn using Excalidraw, a very interesting tool for machine to emulate hand-writing. The Excalidraw project can be found in GitHub, *excalidraw/excalidraw*.

Preface

Some references of this notebook are the Linux Bible (10th edition) that I borrowed from National Library Singapore, and also many Bilibili and YouTube videos, which I will cite as I go through the notebook.

List of Figures

List of Tables

-

Part I

Basic Linux

1

Brief Introduction to Linux

CONTENTS

This chapter gives a brief introduction to Linux, including its key features, advantages and disadvantages over other operating systems (OS).

1.1 Linux as an Operating System

Linux is an OS. An OS is essentially a special piece of software running on a machine (desktop, laptop, server, mobile devices, edge device, etc.) that manages hardware resources and supports application software in the system. An OS shall be able to:

- Detect and prepare hardware
- Manage process
- Manage memory
- Manage storage and files
- Provide user and application interface, and associated authentication methods
- Provide software development kits (SDK) for developing applications

Linux has been overwhelmingly successful and adopted in many areas.

For example, Android operating system for mobile phones is developed using Linux. Google Chrome is also backed by Linux. Many websites such as Facebook are also running on Linux servers.

Some of the most favorable features of Linux, especially to large-size enterprises, are as follows.

- Clustering. It is possible to group multiple Linux machines and let them work as a whole. The group of machines appears to be a single powerful machine to the upper layer.
- Visualization. It is possible to share a server among multiple users and applications in a logically separated manner, so that each of the users thinks that he is working on a dedicated machine.
- Cloud computing. Cloud computing is an advanced usage of Linux clustering and virtualization features. Linux servers can be configured flexibly to support cloud computing functions. It is convenient to manage and audit the users and the resources they deploy.
- Real-time computing and edge computing. Embedded Linux can be implemented on micro-controllers or micro-computers for real-time edge control.

This list can go on and on.

Linux differs from Microsoft Windows and MacOS in many ways, though they are all very successful OSs. Among the three OSs, Linux is the only one that is completely open-source, in the sense that its source code can be viewed and customized by the users per requested.

1.2 A Brief History of Linux

The initial motivation of Linux is to create a UNIX-like operating system that can be freely distributed in the community.

Many modern OSs including MacOS and Linux are inspired by UNIX. UNIX operating system was created by AT&T in 1969 as a software development environment that it used internally. In 1973, UNIX was rewritten in C language, thus gaining useful features such as portability. Today, C is still the primary language used to create UNIX and Linux kernels.

AT&T, who originally owned UNIX, tried to make money from it. Back then AT&T was restricted from selling computers per required by the government. Therefore, AT&T decided to license UNIX source code to universities for a nominal fee. Researchers from universities started learning and improving UNIX, which speeded up the development of UNIX. In 1976, UNIX V6 became the first UNIX that was widely spread. UNIX V6 was developed at

UC Berkeley and was named the Berkeley Software Distribution (BSD) of UNIX.

From then on, UNIX moved towards two separated directions. While BSD remained "open", AT&T started steering UNIX towards commercialization. By 1984 AT&T was pretty ready to start selling commercialized UNIX, namely "AT&T: UNIX System Laboratories (USL)". USL did not sell very well. As AT&T was not allowed to sell PCs, the only thing it could do was to license the source code to other PC manufactures. For this reason the price for the source code had to be set high as it was targeting PC manufactures, not end users. This effectively prevented an end user from procuring UNIX source code from AT&T directly. The PC manufactures were more profitable than AT&T just by selling UNIX-based PCs and workstations to the end users. Overall, although the community acknowledged that UNIX was useful, UNIX source code was extremely costly and was not popular among the end users.

In 1984, Richard Stallman started the GNU project as part of the Free Software Foundation. It is recursively named by phrase "GNU is Not UNIX", intended to become a recording of the entire UNIX that could be open and freely distributed. The community started to "recreate" UNIX based on the defined interface protocols published by AT&T.

Linus Trovalds started creating his version of UNIX, i.e. Linux, in 1991. He managed to publish the first version of the Linux kernel on August 25, 1991, which only worked on a 386 processor. Later in October, Linux 0.0.2 was released with many parts of the code rewritten in C language, making it more suitable for cross-platform usage. This Linux kernel was the last and the most important piece of code to complete a UNIX-like system under GNU General Public License (GPL). It is so important that people call this operating system "Linux OS" instead of "GNU OS", although GNU is the host of the project and Linux kernel is just a part (the most important part) of it.

1.3 Linux Distributions

As casual Linux users, people do not want to understand and compile the Linux source code to use Linux. In response to this need, different Linux distributions have emerged. They share the same Linux OS kernel but differ from each other in many ways such as software management tools and user interfaces.

Today, there are hundreds of Linux distributions in the community. The most famous two categories of distributions are as follows. The major difference is the way they manage software applications.

- Red-Hat-Based Distributions
  - Red Hat Enterprise Linux (RHEL)

  - Fedora
  - CentOS
- Debian-Based Distributions
  - Debian
  - Ubuntu
  - Linux Mint
  - Elementary OS
  - Raspberry Pi OS

Notice that although the source code of all the distributions above is publicly available as required by GPL license (GPL requires that any modified versions of a GPL-licensed product shall also be made open-source with a GPL license, as long as the modifications spread in the community), some of the distributions may come with a "subscription fee". The subscription fee is not for the OS source code, but for the technical support, paid maintenance, and other add-on services that the developers of the distributions provide to the end users.

1.3.1 Red-Hat-Based Distributions

Red Hat created the Red Hat Package Manager (RPM) to manage software applications. The RPM packaging contains not only the software files but also its metadata, including version tracking, the creator, the configuration files, etc. In the OS, a local RPM database is used to track all software on the machine. Yellow Dog Updater Modified (YUM) is an open-source Linux package management application that uses RPM plus additional features for enhanced user experience. YUM is very popular among Red-Hat-based distributions.

Red Hat Enterprise Linux (RHEL) is a commercial, stable and well-supported OS that can host mission-critical applications for big business and governments. To use RHEL, customers pay for subscriptions which allow them to deploy any version of RHEL as desired. Different tiers of supports are available depending on the subscriptions. Many add-on features are available for the customers such as the cloud computing integration.

CentOS is a "recreation" version of RHEL using freely available RHEL source code. In this sense, CentOS experience should be very similar with RHEL and it is free of charge, but the users will not enjoy the professional technical support from RHEL engineers. Recently, Red Hat took over the development of CentOS project.

Fedora is a free, cutting-edge Linux distribution sponsored by Red Hat. It is less stable than RHEL, and plays as the "testbed" for Red Hat to interact with the community. From this perspective, Fedora is very similar to RHEL, just with more dynamics and uncertainties. Some functions, especially server related functions, will be tested on Fedora before implemented on RHEL.

1.3.2 Debian Based Distributions

Different from Red-Hat-based distributions that use RPM, Debian and Debian-based distributions use Advanced Packaging Tool (APT) to manage software applications. ATP simplifies the process of managing software by automating the retrieval, configuration and installation of software packages. Among all the Debian-based distributions, Ubuntu is the most successful and popular one. Ubuntu has a variety of graphical tools and focuses on full-featured desktop system while still offering popular server packages. It has a very active community to support its development.

Ubuntu has larger software pool than Fedora. Ubuntu and its associated software usually have a longer "lifespan" than Fedora because Ubuntu servers as a stable platform while Fedora is more of a "testbed". Ubuntu is more for casual users and beginners, while Fedora more for advanced users or developers, especially developers for RHEL.

1.4 Linux Graphical Desktop

Graphical user interface is not necessary to run Linux OS. Yet, many Linux distributions support graphical desktops to convene the end users. When installing these distributions, the user can choose whether or not to install a graphical desktop environment along with the OS. The most popular desktop environment is GNOME. There are other choices such as KDE, LXDE and Xfce desktops. GNOME and KDE are more for regular PCs while LXDE and Xfce, being light in size, more for low-power-demanding systems.

Figures 1.1, 1.2, 1.3 and 1.4 give the flavors of each desktop environment mentioned above. From the figures we can see that GNOME adopts a more Linux/MacOS style desktop environment, while KDE "Windows 7" style. LXDE and Xfce are more simple in graphics presentations and they are more for embedded systems.

It is possible to install multiple desktop environments on one machine, in which case the user can choose which desktop environment to use each time the computer is started.

1.5 Linux Installation

Linux can be installed both on a fixed hard drive or on a mobile storage such as a thumb drive. The installation of different distributions may differ. Thanks to the graphical installation tools for the popular distributions, the installations can be done fairly easily.

FIGURE 1.1
GNOME desktop environment.

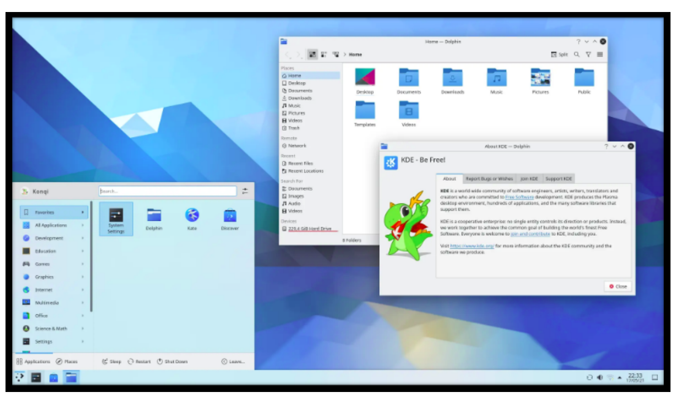


FIGURE 1.2
KDE desktop environment.

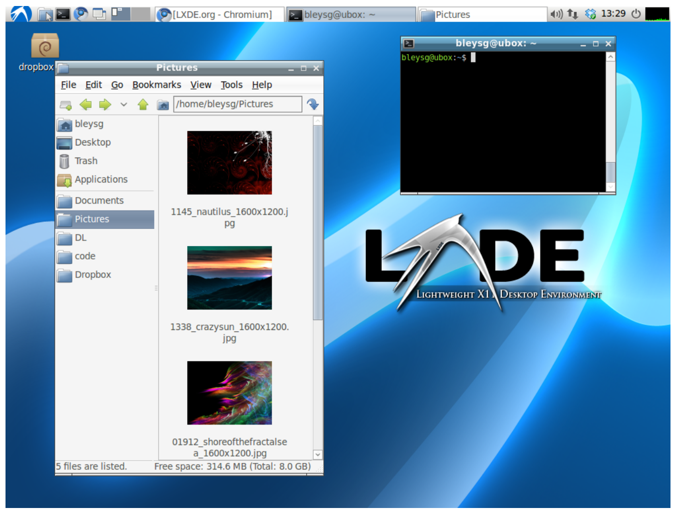


FIGURE 1.3
LXDE desktop environment.

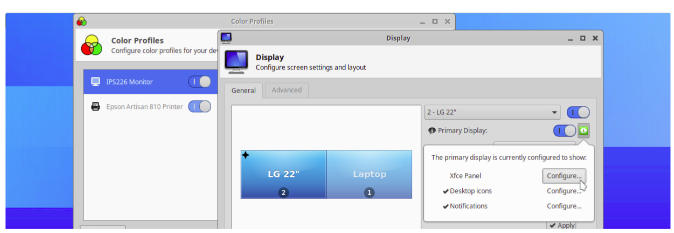


FIGURE 1.4
Xfce desktop environment.

Instructions of installing Ubuntu is given by *https://ubuntu.com*. Instructions of installing Fedora is given by *https://getfedora.org*. For the use of RHEL, consult with Red Hat at *https://www.redhat.com*. Red Hat provides different types of RHEL licenses for different using purpose, including developer license, which is cheaper than a standard enterprise-level license and serves well for learning purpose.

2

Shell

CONTENTS

Linux command line interface (CLI), usually known as the "shell", is the most available, flexible and powerful tool for the users and programs to interact with the OS and perform certain actions.

Notice that the use of the shell is not compulsory for casual users when the graphical desktop is present. Still, it is strongly recommended that the users shall understand at least the basics of shell because it is more flexible and powerful than the graphical desktop and hence can become handy time to time.

Linux shell will be used rapidly in the remaining of this notebook.

2.1 Shell as a Command Line Interface

Linux's default CLI, usually known as the "shell", was invented before the graphical tools, and it has been more powerful and flexible than the graphical tools from the first day. On those machines where no graphical desktops are installed, the use of shell is critical.

2.1.1 Shell Types

There are different types of shells. The most commonly used shell is the "bash shell" which stands for "Bourne Again Shell", derived from the "Bourne Shell" used in UNIX. An example `calculate_fib.sh` written in bash shell script is given below, where Fibonacci series is calculated "1, 1, 2, 3, 4, 8, 13, 21, 34, 55".

```
#!/usr/bin/bash
n=10
function fib
{
  x=1; y=1
  i=2
  echo "$x"
  echo "$y"
  while [ $i -lt $n ]
  do
      i=‘expr $i + 1 ‘
      z=‘expr $x + $y ‘
      echo "$z"
      x=$y
      y=$z
  done
}
r=‘fib $n‘
echo "$r"
```

Some other shells such as "C Shell" and "Korn Shell" are also popular among certain users or certain Linux distributions. For example, C Shell supports C-like shell programming, which is sometimes more convenient than the bash shell. In case where the Linux distribution does not have these shells pre-installed, the user can install and use these shells just like installing any other software.

In this notebook, bash shell is used unless otherwise mentioned.

2.1.2 Prompt

After opening the shell or terminal, a string (usually containing username, hostname, current working directory, etc.) followed by either a `$` or `#` should appear. Following the string, the user can key in the shell commands. The string may look like the following:

```
<username>@<hostname>:~$
```

The above string is called a *prompt*, indicating the start of a user command. By default, for regular user, the ending of the prompt is `$` while for the root user, the ending is `#`. The prompt can be customized by changing the environment variable `PS1`. See Sections 2.1.3, 2.2.4 for details about environment variable and shell configuration, respectively.

For the term “root user”, we are referring to a special user with username and user ID (UID) “root” and 0 respectively. This UID gives him the administration privilege over the machine, such as adding/removing users, changing ownership of files, etc. To avoid fatal damage to the entire system by human error, root user shall not be used unless absolutely necessary. For this reason, the root user’s authentication is often deactivated by default (by setting its login password to invalid).

Notice that the root user is different from a “sudoer”, later of which is basically a regular user equipped with *sudo privilege*. A sudoer can temporarily switch to root user by using sudo su as follows.

```
<username>@<hostname>:~$ sudo su
[sudo] password for <username>:
root@<hostname>:/home/<username>#
```

More about sudo privilege, sudo and su commands are introduced later.

Key in a command and press Enter to execute the command. A Linux shell command looks like the following in general

```
$ <command> [<option>] [<input>]
```

2.1.3 Shell Environment Variables

To execute a command by its name, the OS needs to know where the command is located at. Commonly used commands shall be included in the PATH environment of the shell, so that they can be executed anytime from any working directory. The PATH environment is a series of directories (locations) in the system, and it is initialized automatically when the shell is started. Check the PATH environment by

```
$ echo $PATH
<directory-1>:<directory-2>:<directory-3>: ...
```

where echo displays a line of text, and $PATH is a built-in environment variable that records the PATH environment of the current bash. Some default directories in the PATH environment are introduced as follows.

Most Linux built-in commands for regular users are stored under /bin, /usr/bin, and for administrators, in /sbin, /usr/sbin. Commands for a specific user is often stored under /home/<username>/bin. To determine the location of a particular command, use type if the command can be found in the PATH environment. Alternatively, use locate, mlocate to search all the accessible files in the system to find a command. An example is given below.

```
$ type <command>
<command-location>
```

In addition to PATH, there are other shell environment variables for the user to monitor and control the OS. Table 2.1 summarizes commonly used shell environment variables. Command echo $<variable-name> can be used

TABLE 2.1
Commonly used shell environment variables.

Variable	Description
BASH	Full pathname of the bash command.
BASH_VERSION	Current version of the bash command.
EUID	Effective user ID number of the current user, which is assigned when the shell starts, based on the user's entry in /etc/passwd.
HISTFILE	Location of the history file.
HISTFILESIZE	Maximum number of history entries.
HISTCMD	The number index of the current command.
HOME	Home directory of the current user.
PATH	Path to available commands.
PWD	Current directory.
OLDPWD	Previous directory.
SECONDS	Number of seconds since the shell starts.
RANDOM	Generating a random number between 0 and 99999.

to check the values of these variables. Use command `env` to check a list of environment variables in the shell.

The environmental variables can be created or updated as follows.

```
<variable-name> = <variable-value> ; export <variable-name>
```

For example,

```
PATH = $PATH:/getstuff/bin ; export PATH
```

adds a new directory `/getstuff/bin` to the `PATH` environmental variable. This allows temporary change to the PATH environment variable. Notice that each time the shell is restarted, the environmental variables are also reset.

To permanently change the value of an environmental variable, consider adding the above command in the bash start scripts such as `~/.bashrc`, which is the user script (initialization script) of the user's shell.

2.2 Basic Commands

Some basic and useful commands are briefly introduced in this section by categories.

2.2.1 Displaying User/Machine Information

Administrative users may need to frequently check the basic system information, such as hardware configuration, OS version, username, hostname, disk usage, running process, system clock, etc. Some useful commands are summarized below.

The following commands show basic information of a user.

```
$ whoami
<username>
$ grep <username> /etc/passwd
<username>:x:<uid>:<gid>:<gecos>:<home-directory>:<shell>
```

In the above, `whoami` is used to display the current login user's username. Command `grep` is used to search a content, in this case the user name, in a selected file, in this case `/etc/passwd` where the user information is stored. This should return the username, the password (for encrypted password, an "x" is returned), UID, group id (GID), user id info (GECOS), home directory and default shell location of the user. Another command `id` also returns the UID and GID information of the current user.

The following commands show the date and hostname of the machine.

```
$ date
<date, time and timezone>
$ hostname
<hostname>
```

The following command `lshw` lists down hardware information in details. Sudo privilege is recommended when using this command, to give more detailed and accurate information of the system. Sometimes the displayed information can be too detailed. Use `-short` argument with the command to shorten the output.

```
$ sudo lshw
```

2.2.2 Performing Files Operations

The most important commands for navigating in the file system is to display the current working directory and list down files and subdirectories in the current working directory as follows.

```
$ pwd
<working-directory-path>
$ ls
<file-list>
```

The aforementioned `ls` command can be used flexibly. Commonly seen arguments that come with `ls` are `-l` (implement long listing with more details of each item), `-a` (include hidden item in the list) and `-t` (list by time).

More file operations related commands are introduced in Chapter 4.

2.2.3 Set Alias and Shortcuts

Command `alias` is used to create short-cut keys for commands and associated options, which makes it more convenient for the system operators to work on the shell. Some alias has already been created automatically when the shell is started. Use `alias` to check the existing alias in the shell. An example is given below.

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
```

A temporary alias can be added to the shell by using

```
$ alias <shortcut command>='<original command and options>'
```

for example

```
$ alias pwd='pwd; ls -CF'
```

To permanently add alias to the shell, the alias needs to be added to the bash start scripts such as `~/.bashrc`.. See Section 2.2.4 for details.

2.2.4 Other Commands

Many commands can be used flexibly and it is impossible to illustrate all their details. Consider use the following two methods to check the detailed manual about a command.

```
$ man <command>
$ <command> --help
```

Use `history` to check history commands. Use `!<history command index>` to repeat a history command, or use `!!` to repeat the latest previous command. It is possible to disable history recording function for privacy purpose.

Shell configuration files are loaded each time a new shell starts. User-defined permanent configurations (such as useful alias) can be put into these files so that the configurations can be implemented automatically. Some useful files are summarized in Table 2.2.

2.3 A Taste of Bash Shell Script Programming

A truly power feature of the shell is its ability to redirect inputs/outputs of commands, thus to chain the commands together. Meta-characters pipe (`|`),

TABLE 2.2
Shell configuration files.

File pathname	Description
`/etc/profile`	The environment information for every user, which executes upon any user logs in. Root privilege is required to edit this file.
`/etc/bashrc`	Bash configuration for every user, which executes upon any user starts a shell. Root privilege is required to edit this file.
`~/.bash_profile`	The environment information for current user, which executes upon the user logs in.
`~/.bashrc`	Bash configuration for current user, which executes upon the user starts a shell.
`~/.bash_logout`	Bash log out configuration for current user, which executes upon the user logs out or exit the last bash shell.

ampersand (`&`), semicolon (`;`), dollar (`$`), parenthesis (`()`), square bracket (`[]`), less than sign (`<`), greater than sign (`>`), double greater than sign (`>>`), error greater than sign (`2>`) and a few more more, are used for this feature. Details are given below.

The pipe (`|`) connects the output of the first command to the input of the second command. The following example searches keyword "function" in `calculate_fib.sh` which was given previously.

```
$ cat calculate_fib.sh | grep function
function fib
```

where `cat` concatenates files and print on the standard output, and `grep` prints lines that match patterns in each file.

The semicolon (`;`) allows inputting multiple commands in the same line in the script. The commands are then executed one after another from left to right.

The ampersand (`&`) can be put in the end of a line so that the command on that line will run in the background. The commands or process running in the background does not occupy the shell standard display, and the users can continue working on other commands in parallel. This is particularly useful when a task is going to take a long time to be executed. To manage the tasks running in the background, check more details in Chapter 6.

Use the dollar sign `$` (not the prompt) to indicate a command expansion. The command in `$(<command>)` will be executed as a whole, then treated as a single input. The content in `()` is sometimes called sub-shell. For example, to display the function defined in `calculate_fib.sh` previously,

```
$ echo Display functions: $(cat calculate_fib.sh | grep function)
Display functions: function fib
```

Use `$[<arithmetic expression>]` for simple calculations, such as

```
$ echo 1+1=$[1+1]
1+1=2
```

Another example to count the number of files/folders in the current directory is

```
$ echo There are $(ls -a | wc -w) files in this directory.
There are 69 files in this directory.
```

where `wc` counts the number of lines, words or bytes in a file.

The dollar sign `$` is also used to expand the value of a variable, either environmental variable or self-defined variable, as explained previously in 2.1.3.

The less than sign `<` and greater than sign `>` are used for input/output direction of a file. They are useful when a command needs to pull input and/or push output to a file instead of the standard input and output. An example using command `sort` together with input direction `<` is given as follows. Considering sorting characters "a", "c", "b", "g", "e", "f", "d" using `sort` command. The letters are input from the console as follows. Use `ctrl+D` to quit the input, and the output after sorting will be displayed in the console as follows.

```
$ sort
a
c
b
g
e
f
d
a
b
c
d
e
f
g
```

For demonstration purpose, create a file `before_sort` in the current working directory. Inside `before_sort` are letters "a", "c", "b", "g", "e", "f", "d", each occupying a separate row. There are several ways to create the file, which will be explained later. For now, just assume that the file already exists. Use `cat` to quickly check its content as follows.

```
$ cat before_sort
a
c
b
g
e
```

```
f
d
```

Use `sort` to sort `before_sort` as follows. In this case, the input to `sort` becomes a file, rather than the standard input from the keyboard. Notice that in this example, `sort before_sort` also works, as `sort` will by default take its first argument as the location of the file to be sorted.

```
$ sort < before_sort
a
b
c
d
e
f
g
```

Use `>` to redirect the output of a command to a file as given in the following example.

```
$ sort < before_sort > after_sort
$ cat after_sort
a
b
c
d
e
f
g
```

where `sort` does not output the result to the console, but instead saves the result in a file named `after_sort`. The double greater sign `>>` works similarly with `>` except that `>>` will append the output to an existing file, while `>` overwrites the existing file.

With the above been said, it is possible to use the following to create the `before_sort` file that has been used in the example.

```
$ echo -e "a\nc\nb\ng\ne\nf\nd\n" > before_sort
```

The error greater sign `2>`, `2>>` works similarly with `>`, `>>` except that instead of redirecting standard output messages, it redirects the error messages.

3

Text File Editing

CONTENTS

Many text editors are supported in Linux, to name a few, *Vim*, *Emacs*, *gedit*, *Visual Studio Code*. These text editors come with different features. Some of the text editors may even play the role of an integrated development environment (IDE).

Among the vast number of choices, *Vim* is probably the most popular one. It works perfectly in a shell environment without relying on graphical desktop, thus is adopted by many Linux distributions as the built-in text editor. *Vim* is introduced in this chapter, followed by a brief review of some other commonly used text editors.

3.1 General Introduction to Vim

Vim is a free and open-source software initially developed by Bram Moolennar, and has become the default text editor of many Unix/Linux based operating systems.

Some people claim *Vim* to be the most powerful text file editor as well as integrated development environment for programming on a Linux machine (and potentially on all computers and servers). The main reasons are as follows.

- *Vim* is usually built-in to Linux during the operating system installation, making it the most available and cost-effective text editor.
- *Vim* can work on machines where graphical desktop is not supported.
- *Vim* is light in size and is suitable to run even on an embedded system.
- *Vim* operations are done mostly via mode switch and shortcut keys; as a result, **the brain does not need to halt and wait for the hand to grab and move the mouse**, thus speeding up the text editing.
- *Vim* is highly flexible and can be customized according to the user's habit (for example, through `~/.vimrc`), and it allows the users to define shortcut keys.
- *Vim* can automate repetitive operations by defining macros.
- *Vim* can be integrated with third-party tools which boost its features to a higher level.

Vim can become very powerful and convenient for the user if he is very used to it. However, *Vim* is not as intuitive as other text editors such as *gedit*, and there might be a learning curve for the beginners.

3.2 Vim Modes

Unlike other text editors, *Vim* defines different "modes" during the operation, each mode has some unique features. For example, in the *insert* mode, *Vim* maps keyboard inputs with the text file just like an conventional text editor. In the *normal* mode (this is the default mode when opening *Vim*), *Vim* uses useful and customizable shortcut keys to quickly navigate the document and perform operations such as cut, copy, paste, replace, search, and macro functions. In the *virtual* mode, *Vim* allows the user to select partial of the document for further editing. In the *cmdline* mode, *Vim* takes order from command lines and interact with Linux to perform tasks such as save and quit.

The following Table 3.1 summarizes the commonly used modes in Vim.

As a start, the following basic commands can be used to quickly create, edit and save a text file using vim. In home directory, start a shell and key in

```
$ vim testvim
```

TABLE 3.1
Commonly used modes in *Vim*.

Mode	Description
Normal	Default mode. It is used to navigate the cursor in the text, search and replace text pieces, and run basic text operations such as undo, redo, cut (delete), copy and paste.
Insert	It is used to insert keyboard inputs into the text, just like commonly used text editors today.
Visual	It is similar to normal mode but areas of text can be highlighted. Normal mode commands can be used on the highlighted text.
Cmdline	It takes in a single line command input and perform actions accordingly, such as save and quit.

to create a file named "testvim" and open the file using *Vim*. Notice that in some Linux versions, *vi* might be aliased to *vim* by default.

In the opened file, use `Esc` and `i/a` to switch between normal mode and insert mode. In the normal mode, use `h`, `j`, `k`, `l` to navigate the position of the cursor. Finally, in the normal mode, use `:w` to save the file, and `:q` to quit *vim*, or use `:wq` to save and quit *Vim*.

The above basic commands and their relationships are summarized in Fig. 3.1. A flowchart to create/open, edit, save, and quit a text file using the aforementioned commands are given in Fig. 3.2.

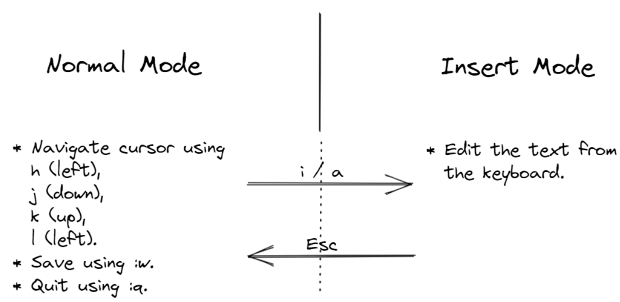


FIGURE 3.1
Mode switching between normal mode and insert mode, and basic functions associated with the modes.

There are other shortcuts to switch from normal mode to insert mode. Some of them are summarized in Table 3.2.

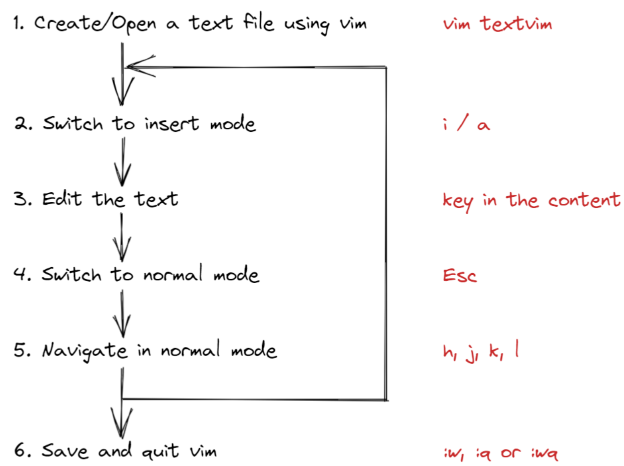


FIGURE 3.2
A flowchart for simple creating, editing and saving of a text file using *Vim*.

TABLE 3.2
Commonly used shortcuts to switch from normal mode to insert mode.

Operator	Description
`i`	Insert before the character at the cursor.
`I`	Insert at the beginning of the row at the cursor.
`a`	Insert after the character at the cursor.
`A`	Insert at the end of the row at the cursor.
`o`	Create a new row below the cursor and switch to insert mode.
`O`	Create a new row above the cursor and switch to insert mode.

3.3 Vim Profile Configuration

With the basic operations introduced in Section 3.2, we are able to create and edit a text file as we want to, just like using any other text editor. Though at this point the advantages of using *Vim* over other text editors are not obvious yet, the *Vim* editor is at least useable.

Before introducing more advanced features of *Vim* for a better user experience, we can now customize the user profile to suit our individual habit. Notice that the customization is completely optional and personal. This section only introduces the ideas and basic methods of such customization, such as re-mapping keys and creating user-defined shortcuts. Everything introduced here are merely examples and it is completely up to the user how to design and implement his own profile.

In Linux, navigate to home directory. Create the following path and file `~/.vim/vimrc` or `~/.vimrc`. Open the *vimrc* file as a blank file using *Vim*. The individual user profile can be customized here.

3.3.1 Mapping Shortcuts

It is desirable to re-map some keys to speed up the text editing. For example, by mapping `jj` to `Esc` in insert mode, one can switch from insert mode to normal mode more quickly (notice that consequent "jj" is rarely used in English). Another example coud be mapping `j`, `k`, `i` to `h`, `j`, `k` respectively in normal and visual modes, making the navigation more intuitive. In that case, an alternative key needs to be mapped to `i`.

The mapping of keys and keys combinations can be done as follows in *vimrc*.

```
inoremap jj <Esc>
noremap j h
noremap k j
noremap i k
noremap h i
```

where `inoremap` is used to map keys (combinations) in insert mode, and `noremap` in normal and visual modes.

The upper case letter `S` and lower case letter `s` in normal mode are originally used to delete and substitute texts, and they are rarely used due to the more powerful shortcut `c` which does similar tasks. We can re-map `S` to saving, and disable `s`. Similarly, upper case letter `Q` is mapped to quitting *Vim*.

```
noremap s <nop>
map S :w<CR>
map Q :q<CR>
```

where `<nop>` stands for "no operation" and `CR` stands for the "enter" key on the keyboard. The keyword `map` differs from `noremap` in the sense that `map` is for recursive mapping.

3.3.2 Syntax and Color Scheme

By default *Vim* displays white colored contents on a black background. Use the following command in *vimrc* to enable syntax highlighting or change color schemes. Use `:colorscheme` in cmdline mode in *Vim* to check for available color schemes.

```
syntax on
colorscheme default
```

The following setups in *vimrc* displays the row index and cursor line (a underline at cursor position) of the text, which can become handy during the programming. Furthermore, it sets auto-wrap of text when a single row is longer than the displaying screen.

```
set number
set cursorline
set wrap
```

The following command opens a "menu" when using cmdline mode, making it easier to key in commands.

```
set wildmenu
```

3.3.3 Other Useful Setups

Use `scrolloff` to make sure that when scrolling in *Vim*, there are always margins lines in the top and bottom of the screen, so that the cursor is always close to the centre of the screen.

```
set scrolloff=3
```

Enable spell check in *Vim* as follows.

```
map sc :set spell!<CR>
```

where `sc` can be used to quickly turn on and off the spell check function. In addition, when the cursor is put on the wrongly spelled word, use `z=` to open a list of possible corrections.

3.3.4 Plug Tools

In the Linux community, many plug tools have been created to add useful features for *Vim*. As a demonstration, in this section *vim-plug*, a light-size vim plugin management tool created on GitHub, is used to install selected *Vim* plugins. Details about *vim-plug* can be found at GitHub under *junegunn/vim-plug*.

Following the instructions given by GitHub under *junegunn/vim-plug*, to use *vim-plug* on Linux, the very first step is to use *cURL*, a command-line tool for transferring data specified with URL syntax, to download *vim-plug*. To confirm that *cURL* is already installed (this is often the case), use the following command in Linux shell

```
$ apt-cache policy curl
```

and if *cURL* is installed, the shell is expected to return somthing like

```
curl:
  Installed: 7.68.0-1ubuntu2.7
  Candidate: 7.68.0-1ubuntu2.7
  Version table:
 *** 7.68.0-1ubuntu2.7 500
        500 http://cn.archive.ubuntu.com/ubuntu focal-updates/main amd64
             Packages
        500 http://security.ubuntu.com/ubuntu focal-security/main amd64
            Packages
        100 /var/lib/dpkg/status
     7.68.0-1ubuntu2 500
        500 http://cn.archive.ubuntu.com/ubuntu focal/main amd64
            Packages
```

If cURL is not pre-installed, use `sudo apt install curl` to install curel.

With *cURL* installed, use the following in the shell to install *vim-plug*

```
$ curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
    https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

In the very beginning of *vimrc*, add the following to specify the plugins to be installed. As an example, *vim-airline/vim-airline* and *joshdick/onedark.vim* are to be installed, the first of which adds a status line at the bottom of the *Vim* window, and the second adds a popular color scheme “onedark”.

```
call plug#begin()
Plug 'vim-airline/vim-airline'
Plug 'joshdick/onedark.vim'
call plug#end()
```

Finally, reload *vimrc*, then run `:PlugInstall` in cmdline mode to install the plugins. Use `colorscheme onedark` instead of `colorscheme default` in *vimrc* to enjoy the onedark color scheme.

Notice that instead of setting up configurations permanently in *vimrc*, the user can also apply a setup in cmdline mode for temporary use in an open session. A full list of *vimrc* configurations used in this chapter can be found in the appendix.

3.4 Basic Operations in *Vim*

In normal mode, the most frequently used operation is probably `u`, which stands for undo. Other commonly used operations, such as delete, cut, copy, paste, replace and search, are mostly done in normal mode through shortcut keys. For example, `dd` deletes (cuts) the entire row at the cursor and `p` pastes the content in the clipboard to the cursor position. For beginners, remembering shortcut keys can be difficult. Therefore, it is suggested looking for the consistent patterns of the different commands, rather than brute-force remembering the operations.

Many *Vim* shortcut keys in normal mode has the following structure, namely an operator command followed by a motion command without a space in between.

```
<operator><motion>
```

The operator command tells *Vim* what to do (say, copy), and the motion command tells the applicable range of the operation (say, a row, or a word, or a character). Some operator commands may work alone without motion commands.

3.4.1 Cut, Change, Copy and Paste

The following lines taken from Wikipedia under "William Shakespeare" is used as an example to demonstrate delete/cut, change, copy and paste functions. In the text file, each sentence takes a separate row as shown in Figure 3.3.

> William Shakespeare (bapt. 26 April 1564 – 23 April 1616) was an English playwright, poet and actor, widely regarded as the greatest writer in the English language and the world's greatest dramatist.
>
> He is often called England's national poet and the "Bard of Avon" (or simply "the Bard").

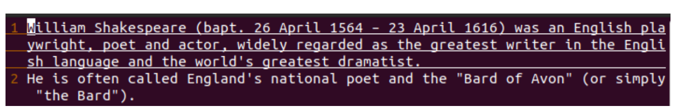


FIGURE 3.3
A piece of text of "William Shakespeare", for demonstration.

Use either `x` or `X` to delete the character at the cursor or previous to the cursor, respectively. To delete multiple characters, one way is to press `x` or `X` multiple times (or hold the keys). Alternatively, it is possible for *Vim* to automatically repeat the procedure. For example, `20x` tells *Vim* to perform `x` for 20 times. The same applies for other operators or motions commands. For example, `10l` executes `l` for 10 times, moving the cursor to the right for 10 characters.

Operator `d`, similar with `dd`, deletes the contents of the text, but it requires a motion command and can be used more flexibly. The motion shall tell *Vim* the applicable range to delete/cut.

For example, `dl` deletes one character to the right, i.e. deletes the character at the cursor just like `x`. Likewise, `dh` deletes one character to the left just like `X`. Similarly, `d20l` deletes 20 characters to the right, where "20l" as a whole plays as the motion of "20 characters to the right". A combination by using things like `5d4l` also works and leads to the same result as $20 = 5 \times 4$.

Command `d` can be used even more flexibly. For example, by using word-related motions, `d` can delete/cut by words instead of by characters. Move the cursor to the beginning of a word, (for example, "S" in "Shakespeare"), use `dw` to delete the word. The word motion `w` is similar with `l`, except that `l` directs to the next character, while `w` directs to the beginning of next word. Similarly, `b` directs to the beginning of the current/previous word. Thus, `db` can be used to delete word to the left. Examples `d10b`, `10db`, `d20w`, `5d4w` can be used to delete multiple words at a time. Motions `w` and `b` can also be used to navigate in the text just like `l` and `h`.

When in the middle of a word, `dw` will delete the characters from the cursor to the beginning character of the next word. For example, if the cursor is currently at "k" in "Shakespeare", `dw` will delete "kespeare " (notice that the space between "Shakespeare" and "(bapt." will also be deleted). To delete from the beginning of the word instead, you can use `b` first to navigate back to the beginning of the word, then apply `dw`. Alternatively, use "inner-word" motion `iw` to indicate that the whole word of the cursor shall be deleted.

In addition to character-related motions `h`, `l` and word-related motions `b`, `w`, there are similar motions for sentence `(` (previous), `)` (next) and paragraph `{` (previous), `}` (next). There are also inner-sentence motion `is`, inner-paragraph motion `ip`, inner-quotation motion `i'`, `i"`, ``i` `` and inner-block motion `i(`, `i<`, `i{`, and many more. For example, when the cursor is at "A" of "26 April 1564", `di(` will delete everything inside "()", i.e. deleting "bapt. 26 April 1564 - 23 April 1616".

The operators and motions introduced so far are summarized in Tables 3.3 and 3.4. Notice that motions `aw`, `as`, `ap` are also given in the table. They are similar with their corresponding `iw`, `is`, `ip` except that when deleting, the consequent blank space (for word and sentence) or blank row (for paragraph) will also be deleted.

To change contents, use operator `c`, which works the same way as `d` but it automatically switch to insert mode after removing the content. To copy a piece of text to clipboard, use `y` (stands for "yank") followed by its associated motion to indicate the range of text. The motions also follow Table 3.4. To paste the text in the clipboard to the cursor, use `p`. No motion is required.

TABLE 3.3
Commonly used operators related to delete/cut, change, copy and paste.

Operator	Description
x	Delete/Cut the character at cursor.
X	Delete/Cut the character before cursor.
dd	Delete/Cut the entire row.
d	Delete/Cut selected text according to the motion command.
cc	Change the entire row.
c	Change selected text according to the motion command.
yy	Copy the entire row.
y	Copy selected text according to the motion command.
p	Paste clipboard to the cursor.

TABLE 3.4
Commonly used motions.

Motion	Description
h, l	One character to the left or right.
j, k	One row to the up or down.
b	First character of the current word, or fist character of the previous word.
e	Last character of the current word.
w	First character of the next word.
(,)	One sentence to the previous or next.
{, }	One paragraph to the previous or next.
iw, is, ip	Inner-word, inner-sentence, inner-paragraph.
aw, as, ap	A word, a sentence, a paragraph (including the end blank).
i', i", i`	Inner-quotation for different types of quotations.
i(, i<, i[, }	inner-block for different types of brackets.
0	Beginning of the row.
$	Ending of the row.
gg	Beginning of the text.
G	Beginning of the last row of the text.

In addition to Table 3.4, another commonly used type of motion is to "find by character". For example, consider the following row of text. The cursor is currently at letter "A".

```
ABCDEFG;HIJKLMN;OPQ;RST;UVW;XYZ
```

In normal mode, using `f` followed by a character will navigate the cursor to the nearest corresponding character that appears in the text. For example, `fG` will move the cursor to letter "G". Similarly, `f;` will move the cursor to the ";" between "G" and "H". Key in `f;` again and the cursor will move to ";" between "N" and "O". From here key in `2f:` and the cursor will go to ";" between "T" and "U", as it is equivalent to executing `f;` twice. If `df;` is used when the cursor is at letter "A", "ABCDEFG;" will be deleted.

3.4.2 Search in the Text

In normal mode, use `/<content>` to search a keyword or a phrase. The following setup in *vimrc* shall give a better searching experience.

When searching in the text for a particular word or phrase (searching in the text will be will be covered in a later of the chapter), to make the searching result highlighted, add the following line to the user profile *vimrc*.

```
set hlsearch
exec "nohlsearch"
set incsearch
noremap <Space> :nohlsearch<CR>
set ignorecase
noremap = nzz
noremap - Nzz
```

where `hlsearch` enables highlighting all matching results in the text, and `incsearch` enables highlighting texts along with typing the keyword. *Vim* remembers the keyword from the previous search and may automatically highlight them in the text on a new session, which can be confusing sometimes. The command `exec "nohlsearch"` (`exec` command in the user profile makes *Vim* execute that command when starting a new session) that comes after `set hlsearch` forces *Vim* to clear its searching memory on a new session. Finally, to quit searching, use `:nohlsearch` in cmdline mode and the highlights shall be gone. For convenience, consider mapping it with a customized shortcut key as well, for example to `Space`. As a bonus, set `ignorecase` to ignore case-sensitive during the searching.

Keys `n` and `N` is used to navigate through the searching result, and `zz` is used to pin cursor position in the central of the screen. They are mapped to `-` and `=` in *vimrc*. Notice that they can also be used as motion together with delete/cut, change and copy as given in Table 3.3. Many users in the community have posted their recommended *Vim* user profile configurations, which could be good references when setting up the configurations of your own.

With the above setup, searching for “he” using `/he` leads to the following result given in Fig. 3.4. From Fig. 3.4, it can be seen that all appearances of

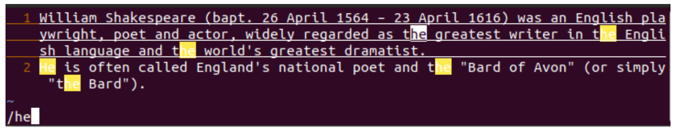


FIGURE 3.4
Search “he” in the piece of text of “William Shakespeare”.

“he” (case insensitive) is highlighted, and the cursor is automatically moved to its first appearance, i.e. “he” in “and the world’s greatest dramatist”. Click `Enter` to confirm the searching content and enabling free move of the cursor.

3.4.3 Other Tips

Use `Ctrl+o`, `Ctrl+i` to “undo” and “redo” cursor positions, respectively. They only move the cursor position and don’t change the actual texts.

To save as a file, use `:w <new path>` in cmdline mode.

In cmdline mode, use `!` to interact with the bash shell. For example, consider a case where a read-only file needs to be edited and saved by a sudoer who forgot to start *Vim* using `sudo`. The common `:w` will be rejected. In this case, use `:w !sudo tee %` to perform the save, where `tee` is a Linux command that takes standard input and writes to a file, and `%` stands for the current file. In another example where an existing file’s content is to be insert into the current text, navigate the cursor to the place to insert the text, and use `:r !cat <filename>`.

To convert and save the current text into an *html* file, use `:%TOhtml`. From there, the file can be further converted into a PDF file.

3.5 Visual Modes

The use of a mouse makes selecting a block of text very intuitive. In most text editors, the selected text will be highlighted, as if the cursor expands from one character to the entire block of text. Sequentially, operations such as delete and copy can be performed on the selected text.

The three visual modes of *Vim*, namely “visual”, “visual-line” and “visual-block”, provide similar experience where the user can select and highlight a block of text.

Use `v` to enter the visual mode, then navigate the cursor to select a block of text. This allows the user to select text between any two characters. An example is given by Fig. 3.5. Alternatively, use `V` to enter the visual-line mode where multiple lines can be easily selected, and use `<ctrl>+v` to enter visual-block mode to select a rectangular block of text.

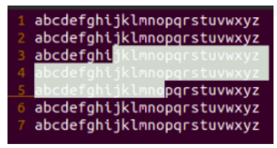


FIGURE 3.5
An example of visual mode where a block of text is selected.

In any of the above visual mode, use `:normal + <operation>` to execute operation(s) form the normal mode for each line. This allows convenient editing of multiple lines of text all together. For example, use `V` to select a few lines of contents, followed by `:normal 0iprefix-` (or `:normal 0hprefix-`, if the insert key `i` has been mapped by `h`), and "`prefix-`" will be added to the beginning of each of the selected lines.

In visual-block mode, after selecting a block of content, use `I` to enter insert mode and insert content in the first row of the selected block. When existing the insert mode using `<Esc>`, the changes will apply to all selected rows.

3.6 Vim Macros

Vim macros can become handy for frequent and repetitive works. Use `q` in normal mode to start and end a macro recording. The syntax follows `q<macro-name><operations><q>` where `<macro-name>` is a single character that labels the macro.

Consider the following example where there is a text file as follows

```
apple.jpg
pear.jpg
orange.jpg
banana.jpg
peach.jpg
```

and we would like to edit it to get the following revised text

```
image: 'apple.jpg'
ttl: 5
image: 'pear.jpg'
ttl: 5
image: 'orange.jpg'
```

```
ttl: 5
image: 'banana.jpg'
ttl: 5
image: 'peach.jpg'
ttl: 5
```

In this example, repetitive work is involved and it is time consuming to do it manually if there are thousands of items in the file, and it is better to record a macro to automate the procedure. Navigate the cursor to the first row of the text, and type the following sequence of characters.

```
q<macro name>0iimage: '<Esc>A'<Enter>ttl: 5<Esc>jq
```

where `<macro name>` can be any character, for example `s`. The string in the middle `0iimage: '<Esc>A'<Enter>ttl: 5<Esc>j` is the necessary procedure to perform the revision for one row. If everything is done correctly, the following text should be obtained

```
image: 'apple.jpg'
ttl: 5
pear.jpg
orange.jpg
banana.jpg
peach.jpg
```

and the cursor should be somewhere at row `pear.jpg`.

To repeat the recorded procedures, use `@<macro-name>`. In this example, just key in `@s` in the normal mode, and the text shall become

```
image: 'apple.jpg'
ttl: 5
image: 'pear.jpg'
ttl: 5
orange.jpg
banana.jpg
peach.jpg
```

Repetitively using `@s` proceeds with the revision. In the case the procedure needs to be repeated for many times, use `<number>@<macro-name>`, in this example `3@s`, to complete the remaining task.

3.7 File Explorer and Screen Splitting

Many IDEs come with project folder navigation and screen splitting features. In these IDEs, there is often a built-in file explorer, from where the user can navigate in the file system, select a file to edit, and the IDE will split the window for the selected file. *Vim* has similar features that supports file explorer and screen splitting, either via built-in functions or third-party support tools.

In *Vim*, use `:Explore`, `:Sexplore` or `:Vexplore` (or `:Ex`, `:Sex`, `:Vex` for short) in cmdline mode to open a file explorer, from where the user can navigate the cursor to select a file. *Vim* will then open the file in a split window that allows the user to further editing the file.

There are many third-party plugin tools that enable convenient file explorer functions. For example, *github.com/scrooloose/nerdtree*. More details can be found in the associated repository on *GitHub*.

Use `:split` and `:vsplit` for horizontal and vertical screen splitting, respectively. A second split window would show up with the same text file opened. For simplicity, these commands can be mapped in *vimrc* as follows.

```
noremap sj :set nosplitright<CR>:vsplit<CR>
noremap sl :set splitright<CR>:vsplit<CR>
noremap si :set nosplitbelow<CR>:split<CR>
noremap sk :set splitbelow<CR>:split<CR>
```

where `splitright` and `splitbelow` is used to setup the default cursor position after splitting the screen.

In a split window, open a new file using `:e <path>`. To navigate the cursor across different split windows, use `Ctrl+w` followed by `h`, `j`, `k` and `l`. For simplicity, they can be mapped as follows.

```
noremap <C-j> <C-w>h
noremap <C-l> <C-w>l
noremap <C-i> <C-w>k
noremap <C-k> <C-w>j
```

where `<C->` stands for `Ctrl+`.

Resize the selected split window using `:res+<number>`, `:res-<numer>`, `:vertical resize+<number>`, `:vertical resize-<number>`. For simplicity, map these commands as follows.

```
noremap J :vertical resize-2<CR>
noremap L :vertical resize+2<CR>
noremap I :res+2<CR>
noremap K :res-2<CR>
```

3.8 Other Text Editors

Apart from *Vim*, many other text editors are also widely used in Linux, each with different features. For demonstration purpose, *Vim* and other text editors are used to open a bash shell script that calculates the first 10 elements of Fibonacci series.

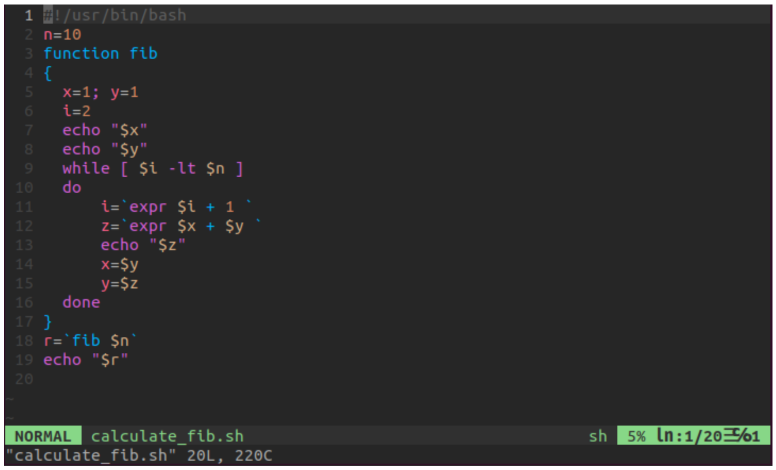


FIGURE 3.6
Vim (with user's profile customization as introduced in this chapter).

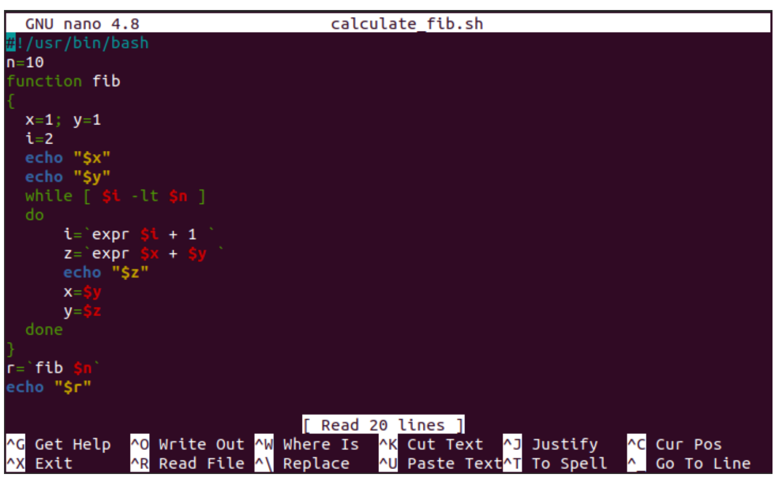


FIGURE 3.7
Nano.

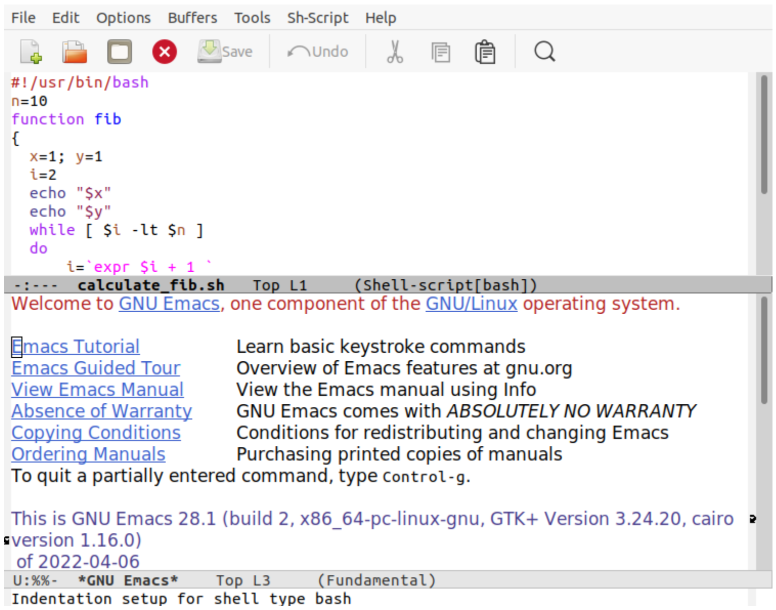


FIGURE 3.8
Emacs.

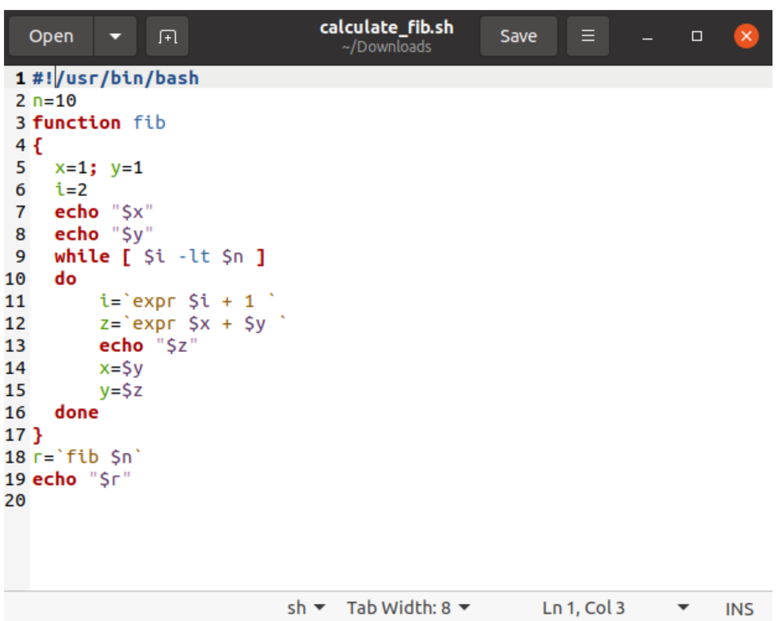


FIGURE 3.9
Gedit.

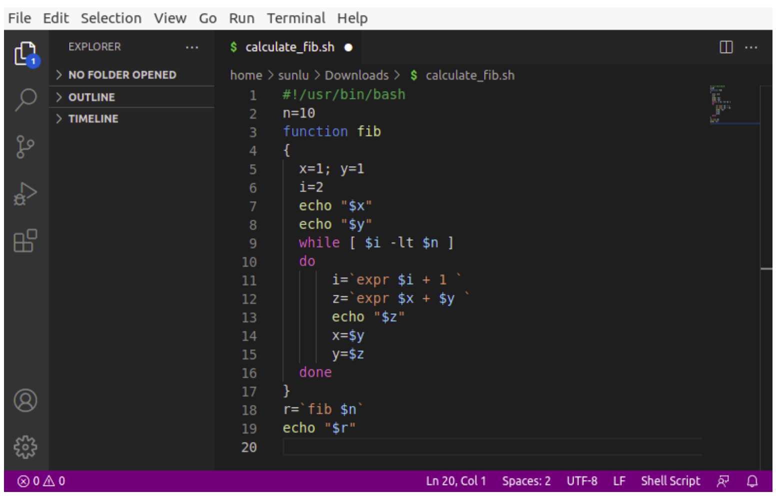


FIGURE 3.10
Visual Studio Code.

4

File Management

CONTENTS

File management is a big portion of OS functionality. In Linux, each device (such as a printer) is treated and managed as a file, and Linux uses a tree hierarchy to manage devices and files. This chapter introduces the filesystem hierarchy and commonly used file management commands.

4.1 Filesystem Hierarchy Standard

The root directory is denoted by a single forward slash "`/`". All sub directories or files can be located by its full path, which looks like the following

```
/<directory>/<subdirectory>/.../<directory-name>
/<directory>/<subdirectory>/.../<file-name>
```

where the first / in each row represents the root directory, and sequential / represents entering a subdirectory.

Upon Linux installation, a file hierarchy is by default created under the root directory. A user can create new files under this hierarchy framework, but should not change the framework itself. The hierarchy is given in Fig. 4.1. Notice that different Linux distributions may differ slightly on how the architecture looks like. The "/" in the figure, as introduced, stands for the root directory, and "`root`" in the figure is a subdirectory under / whose directory name is "root" and it is used store root user related documents. They are two different directories.

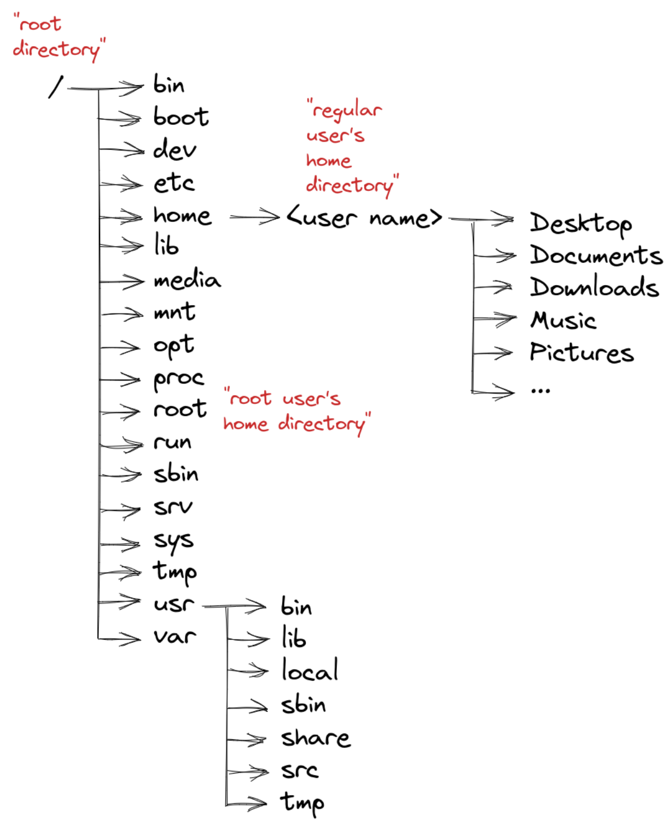


FIGURE 4.1
An example of Linux file system hierarchy.

A (regular) user's home directory is often located at `/home/<user name>`. When logging in as a regular user, his home directory is shorten by the tilde ~ for convenience. Hence, for example `ls ~` will list down the files and directories under his home directory.

As can be seen from Fig. 4.1, the hierarchy contains quite a few predetermined subdirectories, each has some unique purpose. For convenience of illustration, these subdirectories are categorized by functionalities and depressibilities given in Fig. 4.2. Notice that this categorization is rough and may not reflect the truth for all applications. For example, the commonly used command `ls` may appear under `/bin` or `/usr/bin` depending on the Linux distributions.

	"Used by the OS" Administration (System) Level	"Used by all users" All-Users Level	"Used by a specific user" Individual User Level
Executable	/bin /sbin	/usr/bin /usr/sbin	/home/<user name>
Library	/lib	/usr/lib	/home/<user name>
Data / Program Storage	/opt /var	/usr/local /usr/share /usr/src	/home/<user name>
Device	/dev /media /mnt		
Configuration	/etc		/home/<user name>
System	/boot /proc /sys		
Other	/tmp /usr /root	/usr/tmp	

FIGURE 4.2
A rough categorization of commonly used directories in Linux file hierarchy standard.

A brief introduction to the directories are summarized in Table 4.1.

Linux file hierarchy standard differs from MS-DOS and Windows in several ways. Firstly, Linux stores all files (regardless of their physical location) under root directory, while Windows uses drive letters such as `C:\`, `D:\` to distinguish different hard drives. Secondly, Linux uses slash (`/`) to separate directory names, e.g. `/home/username` while Windows uses back slash (`\`), e.g. `C:\Users\username`. Lastly, Linux uses "magic numbers" to tell file types and permissions and ownership to tell whether a file is executable, while Windows (almost always) uses suffixes to tell file types and distinguish executables. Distinguishing file types using magic numbers can be more reliable than using suffixes, though a bit less intuitive.

Magic numbers of a file refer to the first few bytes of a file that are unique to a particular file type, for example, PNG file is hex `89 50 4e 47`. Linux compare the magic numbers of a file with an internal database to decide the file types and features.

TABLE 4.1
Introduction to commonly used directories in Linux file hierarchy standard.

Directory	Description
`/bin`, `/sbin`	Executables used by the OS, the administrator, and the regular users.
`/lib`	Libraries to support `/bin` and `/sbin`.
`/usr/bin`, `/usr/sbin`	Executables used by the administrator and the regular users.
`/usr/lib`	Libraries to support `/usr/bin` and `/usr/sbin`.
`/opt`	Application software installed by OS and administrator for all users.
`/var`	Directories of data used by applications.
`/usr/local`	Application software installed by administrator for all users.
`/usr/share`	Architecture-independent sharable text files for applications.
`/usr/src`	Source files or packages managed by software manager.
`/dev`	Files representation of devices, such as CPU, RAM, hard disks.
`/media`	System mounts of removable media.
`/mnt`	Manual mounts of devices.
`/etc`	Configuration files for OS, users, and applications.
`/boot`	Linux bootable kernel and initial setups.
`/proc`	System resources information.
`/sys`	Linux kernel information, including a mirror of the kernel data structure.
`/tmp`, `usr/tmp`	Temporary files.
`/root`	Root user's home directory.
`/home/<user name>`	A regular user's home directory, containing executables, configurations and files specifically belong to this user.

TABLE 4.2
Commonly used commands to navigate in the Linux file system.

Command	Description
`pwd`	Print working directory.
`ls`	List the subdirectories and files (and their detail information) in a given directory.
`touch`	Create an empty file.
`mkdir`	Create an empty subdirectory.
`mv`	Move (cut-and-paste) a directory or a file; change name of a directory or a file.
`cp`	Copy-and-paste a directory or a file.
`rm`, `rmdir`	Remove a directory or a file (not to Trash, but just gone).
`chmod`	Change permission.
`chown`	Change ownership.

4.2 Commonly Used File Management Commands

Some of the most widely used file management commands are summarized in Table 4.2. Notice that `chmod` and `chown` are administration related commands that change the accessibility of a directory or a file, and will be introduced in a later sections together with the Linux permission system. The rest commands are categorized and introduced in the following subsections.

4.2.1 Print Working Directory

As introduced earlier in Chapter 2 Table 2.1, `HOME`, `PWD` and `OLDPWD` are three default environmental variables used to store the home directory, the current directory and the previous directory of the shell, respectively. Therefore, to print the current working directory in the console, use command `echo $PWD`. Alternatively, just use `pwd` which has the safe effect, as follows.

```
$ pwd
```

4.2.2 List Information about the Files

As one of the most frequently used commands, `ls` lists down information about the files in the current directory (or any other directory if specified), and by default sort the entries alphabetically. Features can be specified for the items to be listed, to make the result more selective. The user is able to decide what information to be displayed, such as file name, file access control list, etc., as follows.

```
$ ls [<option>] [<path>]
```

An example is given in Fig. 4.3 below. By default, command `ls` alone shows only the name of files and subdirectories (excluding hidden files and subdirectories). With the additional arguments (option, as given in the syntax above), more information can be displayed. For example in Fig. 4.3, the `-l` argument displays the information in long listing mode, which includes the owner and access control list information. More details about files and directories access control list are given in later part of this section.

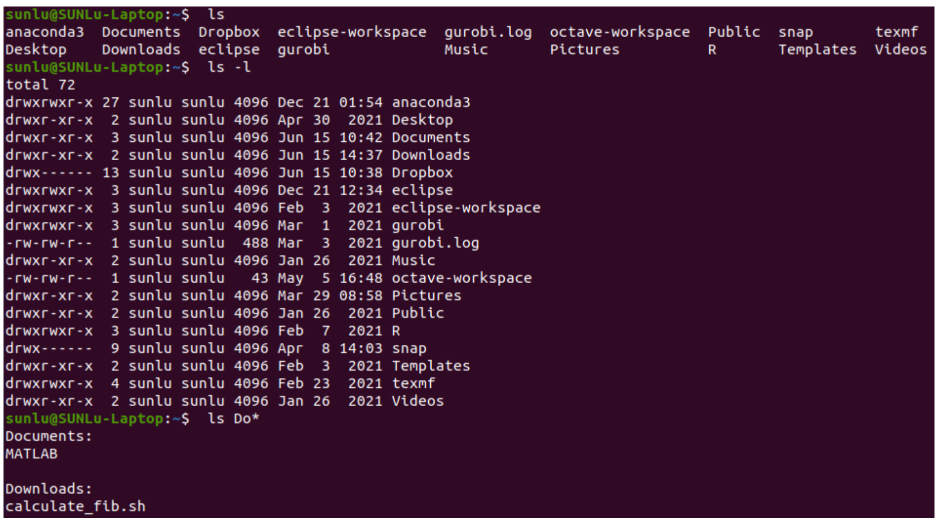


FIGURE 4.3
List down information of files and subdirectories in the current working directory.

More information can be found by reading the `ls` command manual, which is accessible via `ls --help`. Some commonly used `ls` arguments are summarized in Table 4.3. It is also possible to use a combination of arguments in a single line of command. For example, `ls -al` aggregates the effects of using `ls -a` and `ls -l`.

Notice that some Linux distributions may come by default an alias about `ls`, which usually helps to displays the information in a clearer manner. For example, when `ls='ls --color-auto'` is used, the displayed content will be colored based on the type of the files and subdirectories.

4.2.3 Create an Empty or a Simple Text File

To create an empty file in the current working directory, simply use `touch` followed by the path of the file (including file name) as follows.

```
$ touch [<option>] <path>
```

TABLE 4.3
Commonly used arguments and their effects for *ls* command.

Directory	Description
`-a`, `--all`	Include hidden files and subdirectories in the display, including current directory “`.`” and parent directory “`..`” in the list.
`-A`, `--almost-all`	Include hidden files and subdirectories in the display, excluding “`.`” and “`..`”.
`-C`, `--color[=WHEN]`	Colorize the output.
`-l`	Use a long listing format.
`-s`, `--size`	Print the allocated size of each file, in blocks.
`-S`	Sort the displayed content.
`-t`	Sort by modification time.

For example,

```
$ touch ~/test
```

will create an empty file “*test*” under the user’s home directory. If only the name of the file is given, it will by default create the file under the current working directory. Notice that if a file or subdirectory name starts with “`.`”, it will be treated as a hidden file or subdirectory automatically.

Multiple empty files will be created if multiple paths are given in the command separated by spaces.

To create a simple text file, such as a text file with a single line of contents inside, consider using `echo` command with `>` as follows. It is more convenient than using *Vim* for the same task, although also possible.

```
$ echo '<content>' > <path>
```

For example,

```
$ echo '<html><body><h1>Hello world!</h1></body></html>' > ~/test.html
```

creates a simple static HTML web page that says “Hello world!” in the home directory.

4.2.4 Create an Empty Directory

Similar with `touch`, use `mkdir` followed by the path of the directory (including directory name) to create a directory as follows.

```
$ mkdir [OPTION] <path>
```

TABLE 4.4
Commonly used arguments and their effects for *mv* and *cp* command.

Directory	Description
`-b`	Make a backup before overwrite.
`-u`	Overwrite only when source target item is newer than the target path item.
`-i`	Prompt before overwrite.
`-f`	Do not prompt before overwrite.

4.2.5 Move, Copy-and-Paste, and Remove Files and Directories

To move a file or a directory from an existing PATH to a new PATH, simply use `mv` command as follows.

```
$ mv [<option>] <source> <target>
```

Different from the conventional cut-and-paste, while moving the item, it is possible to also rename the item simultaneously. For example,

```
$ mv ~/dog.png ~/Pictures/puppy.png
```

will not only move the file `dog.png` in the home directory to the subdirectory `Pictures`, but also chance the file name to `puppy.png`. For this reason, `mv` can also be used to rename an item rather than moving the item, just by "move" it to the same directory but with a differen name.

Some commonly used arguments of `mv` is summarized in Table 4.4, many of which concerns about the case where there is already an existing item with the identical name in the target path.

The copy-and-paste command `cp` works similar with the move command `mv`, except that it will not remove the item from the source path. Similar syntax applies to `cp` as follows, and arguments in Table 4.4 also apply to `cp`.

```
$ cp [<option>] <source> <target>
```

To permanently delete an item, use `rm` command as follows.

```
$ rm [<option>] <path>
```

For safety, usualy when using `rm`, the OS will keep prompting messages asking user to confirm whether to permenantly delete an item or not. In some OS setups, it is by default forbidden to delete a directory, unless all files and subdirectories in that file have been priorily removed. The following arguments in Table 4.5 can be used to change the setup.

It is possible though, that removed items using `rm` be recovered by expertise. For greater assurance that the deleted contents are truly unrecoverable, consider using `shred` which can physically overwrite the portion of hardware drive where the item is located. More details of `shred` can be found by using

```
$ shred --help
```

TABLE 4.5
Commonly used arguments and their effects for *rm* command.

Directory	Description
`-f`	Ignore nonexistent files and arguments and do not prompt.
`-r`	Remove directories and their contents recursively.
`-i`	Prompt before every removal.
`-d`	Remove empty directories.

TABLE 4.6
Commonly used wildcard characters.

Directory	Description
`*`	Matches any number of characters.
`?`	Matches one character.
`[...]`	Matches characters given in the square bracket, which can include a hyphen-separated range of characters.

4.2.6 Use of Wildcard Characters

When performing moving, copying, removing or otherwise acting on files, wildcard characters can be used to make the work more efficient sometimes. For example, `ls a*` will list all items in the current directory that starts with letter "a". Commonly used metacharacters are summarized in Table 4.6.

4.3 Access Control List

Each file or directory in the Linux OS is assigned with an owner and a permission list known as the Access Control List (ACL). The permission list prevents unauthorized persons to access the item. The permission list of a file can be checked by using `ls -l`. An example is given in Fig. 4.3.

The first column of the output in Fig. 4.3 gives the type and permission of the item. The leading `d` and `-` indicate subdirectory and regular file respectively. Other commonly seen indicators are `l` for a symbolic link, `b` for a block device, `c` for a character device, `s` for a socket and `p` for a named pipe.

Following by the item type indicator is the 9-bit permission that may look like `rwxrwxrwx`. The characters `r`, `w` and `x` stand for three types of permissions "read", "write" and "execute" respectively. An explanation to these permissions is summarized in Table 4.7 and more details can be found in the `ls` command manual available using `ls --help`. The 9-bit permission of an item indicates the permissions of 3 types of users to the item, the first 3 bits the file owner, the middle 3 bits the file group, and the last 3 bits other users. If

TABLE 4.7
Three types of permissions.

Directory	Description
`r`	View what is in the file or directory.
`w`	Change file contents; rename file; delete file. Add or remove files or subdirectories in a directory.
`x`	Run a file as a program. Change to the directory as the current directory; search through the directory; access metadata (file size, etc.) of files in the directory.

any bit in the 9-bit permission is overwritten by a dash `-`, it means that the associated permission for the associated users is banned.

Commands `chown` and `chmod` can be used to change the ownership and 9-bit permission of an item respectively. Details are given in following subsections.

4.3.1 Change Ownership and Group of a File or Directory

Administrative privilege is required to run `chown` command to change the ownership and group of a file or a directory as follows.

```
# chown [<option>] <new_owner>[:<new_group>] <path>
```

For example, in Fig. 4.4,

```
$ sudo chown root:root calculate_fib.sh
```

is used to change the ownership and group of file `calculate_fib.sh` from `sunlu` to `root`. Use `ls` with longlist to check the ownership of a file. Notice that elevated privilage is required to change its ownership, otherwise the request will be rejected as shown in Fig. 4.4.

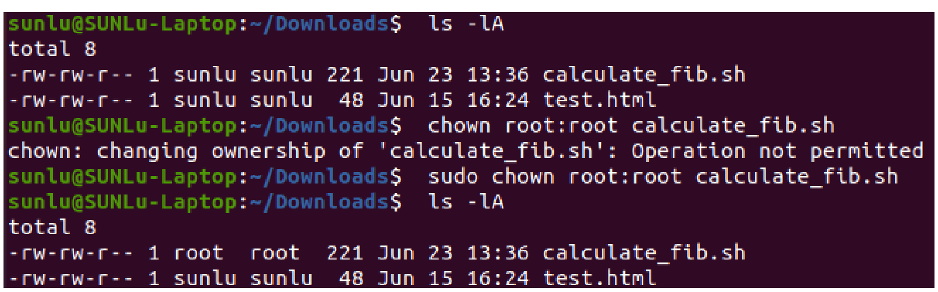


FIGURE 4.4
Change ownership and group of a file.

4.3.2 Change Permissions of a File or Directory

Both the owner of a file or directory and the users with administrative privilege can change the 9-bit permission of the file using `chmod` as follows. The 9-bit permission, in this context, is called the mode of the file.

```
$ chmod [<option>] <new_mode> <path>
```

For example, in Fig. 4.5, `g-w` is used to subtract "writing" permission from "group", and `go+w` is used to add "writing" permission to 'group" and "other", respectively. Here, `u`, `g` and `o` represents "user" (owner), "group" and "other", and `r`, `w` and `x`, "read", "write" and "execute", respectively. Alternatively, 3-

```
sunlu@SUNLu-Laptop:~/Downloads$  ls -l
total 8
-rw-rw-r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu  48 Jun 15 16:24 test.html
sunlu@SUNLu-Laptop:~/Downloads$  chmod g-w calculate_fib.sh
sunlu@SUNLu-Laptop:~/Downloads$  ls -l
total 8
-rw-r--r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu  48 Jun 15 16:24 test.html
sunlu@SUNLu-Laptop:~/Downloads$  chmod go+w calculate_fib.sh
sunlu@SUNLu-Laptop:~/Downloads$  ls -l
total 8
-rw-rw-rw- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu  48 Jun 15 16:24 test.html
sunlu@SUNLu-Laptop:~/Downloads$  chmod 664 calculate_fib.sh
sunlu@SUNLu-Laptop:~/Downloads$  ls -l
total 8
-rw-rw-r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu  48 Jun 15 16:24 test.html
```

FIGURE 4.5
Change 9-bit permission (mode) of a file.

digit numbers such as `664` as shown in Fig. 4.5 can also represent a permission. The first digit is associated with the permission given to the "user", where in this example `6` represents `rw-`. Notice that each permission is assigned a number: `r` is 4, `w` is 2, and `x` is 1. From the sum of the numbers, the OS understands which permission(s) are assigned. For example, $6 = 4 + 2$, hense `r` and `w` permissions. Likewise as another two quick examples, a `7` would mean `rwx` and `5`, `r-x`, respectively. The second and third digits are associated with the permission given to "group" and "other" respectively. Hense, `664` would assign `rw-rw-r--` to the file.

4.4 Search through the System

There are roughly 3 types of searching commands that a user would use frequently:

- Look for the location of a command using its name
- Look for the location of a file using its name (and other metadata such as size, permission, etc.)
- Look for the location of a file using a portion its content

There can be multiple ways to reach each of the above goals. Details are as follows.

4.4.1 Looking for a Command

Use `type` to look for a command as follows.

```
$ type <command>
```

For example

```
$ type cd
cd is a shell builtin
$ type python
python is /usr/bin/python
$ type ls
ls is aliased to ‘ls --color=auto’
```

4.4.2 Looking for a File by Metadata

Many Linux distributions come with built-in command `locate` that can be used to quickly locate a file by (a fraction of) its path as follows. Notice that as long as a file or directory's full path contains the searched content, there is a chance that it will appear in the result. As a result, if the name of a directory is used for searching, all the items in that directory will likely to appear in the result (as their full paths contain the name of the directory).

```
$ locate <file-name>
```

The mechanism behind `locate` is that behind the users' eyes, the OS runs `updatedb` in the background usually once a day to update an internal database that gathers the names of files, and `locate` searches the database for a file. Notice that `locate` may fail to find recently added files if it has not been added to the database by `updatedb`. Besides, not all files are covered by `updatedb` by default, and a configuration file at `/etc/updatedb.conf` determines which files to be covered by `updatedb`. It is also worth mentioning that it will take some time to run `updatedb` for the first time, as it has a lot of things to add to the database during its initial run.

One may get confused by commands `locate` and `mlocate`. The concepts of the commands are very similar, and when both commands are available on a machine, `mlocate` functions by default even the user types `locate`.

An example of using `locate` is given in Fig. 4.6. It can be seen from this example that the searching is done globally and does not rely on the current working directory.

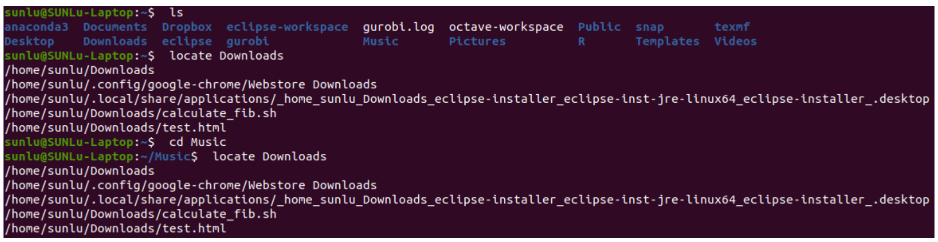


FIGURE 4.6
Search for files and directories using *locate*.

It is worth mentioning that for safety and privacy reasons, `locate` only shows the items that the user would be able to detect manually using `cd` and `ls` in the first place. Therefore, a regular user cannot locate any file under `/root` our other users' home directory using this method.

A more common and widely accepted way of looking for a file by its variety of attributes is using `find` as follows.

```
$ find [<options>] [<path>] <expression>
```

where `options` can be used to specify whether to follow a symbolic link in the result, and expression the filtering method, such as `-f` for file, `-size` for searching by size. It is possible to search by name, size, extension, type, size, and other metadata.

4.5 File Archive

The `tar`, `gzip` and `zip` commands can all be used in files archive, but with different features, as given in Table 4.8.

Only `tar` command is introduced here, as it is the most commonly used and can satisfy most use cases. A common way of using `tar` is as follows. Use

```
$ tar -cvzf <archive-file> <file1> <file2> <file3> ...
```

to archive and zip files, and

TABLE 4.8
Commonly used file archive tools.

Directory	Description
`tar`	Save many files together into a single tape or disk archive, and can restore individual files from the archive. By default, it does not compress the files. However, `-z` option can be used in combination of the command to add compression feature.
`gzip`	Compress or restore files.
`zip`	Compress multiple files one-by-one and integrate them together into a single file.

```
$ tar -xvzf <archive-file>
```

to restore files from the archive file. The commonly used archive file name, in this scenario, is `<filename>.tgz`.

The detailed explanation to all available options for `tar` can be found using `tar --help`. The most commonly used options are `-c`, `-x`, `-z`, `-f` and `-v`, standing for creating compress tape, extracting (restoring) file, adding compressing feature, using file archive, and listing processed files in the console, respectively.

4.6 *Ranger* for File Management

Ranger is a *Vim* based terminal file explorer tool that is highly flexible and customizable. Just like *Vim*, Ranger does not rely on a mouse or a graphical interface, and can manipulate file system in an intuitive way.

To install Ranger, use

```
$ sudo apt install ranger
```

and to run ranger in the current working directory, simply type `ranger`.

An example to demonstrate ranger interface is given in Fig. XXX.

5

Software Management

CONTENTS

...

5.1 Linux Kernel and Repository Management

...

5.2 Linux Package Management Utilities

...

5.3 Software Management

...

5.3.1 Searching for Software

...

5.3.2 Installation of Software

...

5.3.3 Upgrading Software

...

5.3.4 Uninstallation of Software

...

6

Process Management

CONTENTS

A process refers to an instance of a computer program that is running in the system. Managing processes is one of the essential tasks of an OS. In a Windows system, the user can use the task manager, which is a graphical tool to check and manage all the running processes. In a Linux system, the user can manage process in the prompt console using bash commands.

6.1 General Introduction to Process

A process is the fundamental unit for the OS to manage the resources used by a running program.

6.1.1 Process

To improve the efficiency of the CPU, the OS allows multiple processes (also called tasks, jobs) to share the computational capability and memory of the system, each thinking that it is exclusively using the all machine resources, as shown in Fig. 6.1.

The status of a process is stored in its *Process Control Block (PCB).* The PCB is a special data structure used to describe the dynamic of a process. The OS manages the PCBs and control the processes accordingly. Some of the attributes of a PCB are summarized in Table 6.1.

A process shall at least have the following states. The fundamental states of a process and their transferring are introduced in Fig 6.2, where "blocked" indicates that the process is waiting for other inputs to carry out the remaining part of the program, and "suspended" indicates that the process is hold for

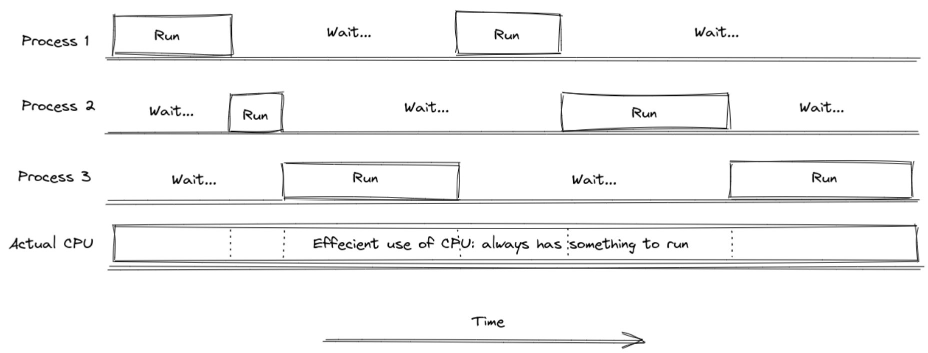


FIGURE 6.1
A demonstration of running multiple processes on a single-core CPU.

TABLE 6.1
Some attributes of a PCB.

Name	Description
Identifier	The unique ID of the process.
State	The state of the process, for example, running, suspended, terminated.
Priority	Priority level in comparison with other processes.
Program Counter	A pointer to the next line of program to be executed.
Memory regions	A pointer to the RAM where the code and data of the process is stored.
Accounting Information	Time limits, clock time used, etc.

some reasons. When a process is offloaded from the CPU, its context is moved from the CPU registers to the PCB of the progress.

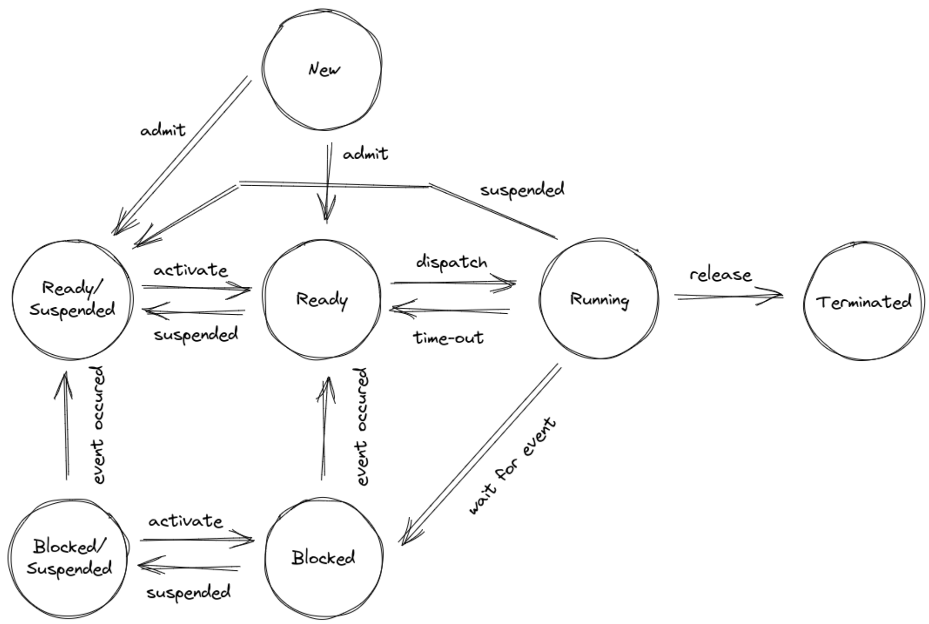


FIGURE 6.2
Fundamental states of a process and their transferring.

There are different types of processes. For example, based on the source of the processes, there are OS triggered processes and user triggered processes, the first of which usually has a higher priority. Based on the running environment, there are front-ground processes and background processes. Based on the resources used, there are CPU processes and I/O processes.

A process usually has a relatively isolated environment, and it does not share memory storage with other processes. Special inter process communication mechanism, which is often referred as "pipe", is required for processes to talk to each other. Inter process communication requires OS level controls.

6.1.2 Thread

A "thread" is like a work dispatch inside a process. There can be multiple threads in a process, as shown in Fig 6.3. Each thread has its own CPU register values and stack, but they share the same program, memory and file storage addresses.

Threads differ from processes in the following aspects:

- A thread is lighter than a process, occupying less resources to create.

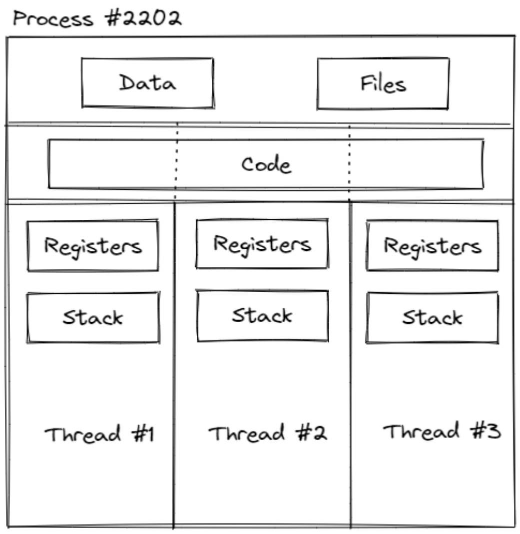


FIGURE 6.3
A demonstration of multiple threads in a process.

- Sharing memories and resources among threads in a process is easier than sharing among processes, because they naturally share address space.
- It is easier to enable parallel computation for the threads in the process when it is running on a multi-core CPU.

Notice that for many OS, including Linux, the kernel can provide thread level services.

6.2 Process Management in Linux

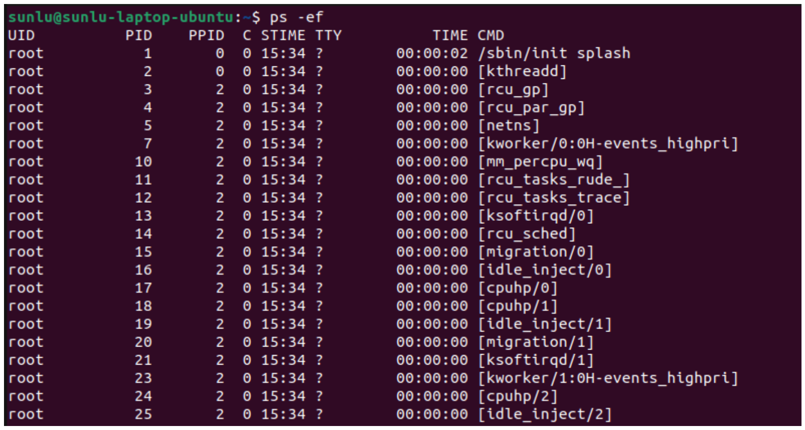
Two commands, `ps` and `top`, are widely used in monitoring the running process in the OS. They can be used stand-alone, without additional arguments like follows.

```
$ ps
```

or

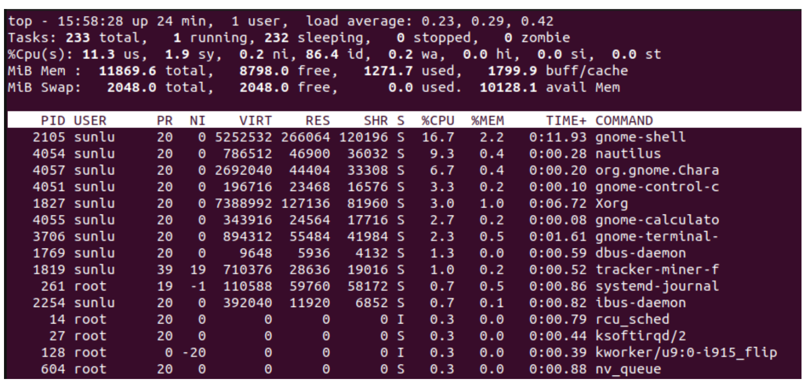
```
$ top
```

The major difference between these two commands is that `ps` provides a screenshot (in a text format) of a list of processes including their names, process IDs (PID) and owners, etc., running at the instance, while `top` provides a frequently-refreshing display of the running processes as well as their associated resources usage. Figs 6.4 and 6.5 give a quick demo of how the execution of the two commands look like.



```
sunlu@sunlu-laptop-ubuntu:~$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 15:34 ?        00:00:02 /sbin/init splash
root           2       0  0 15:34 ?        00:00:00 [kthreadd]
root           3       2  0 15:34 ?        00:00:00 [rcu_gp]
root           4       2  0 15:34 ?        00:00:00 [rcu_par_gp]
root           5       2  0 15:34 ?        00:00:00 [netns]
root           7       2  0 15:34 ?        00:00:00 [kworker/0:0H-events_highpri]
root          10       2  0 15:34 ?        00:00:00 [mm_percpu_wq]
root          11       2  0 15:34 ?        00:00:00 [rcu_tasks_rude_]
root          12       2  0 15:34 ?        00:00:00 [rcu_tasks_trace]
root          13       2  0 15:34 ?        00:00:00 [ksoftirqd/0]
root          14       2  0 15:34 ?        00:00:00 [rcu_sched]
root          15       2  0 15:34 ?        00:00:00 [migration/0]
root          16       2  0 15:34 ?        00:00:00 [idle_inject/0]
root          17       2  0 15:34 ?        00:00:00 [cpuhp/0]
root          18       2  0 15:34 ?        00:00:00 [cpuhp/1]
root          19       2  0 15:34 ?        00:00:00 [idle_inject/1]
root          20       2  0 15:34 ?        00:00:00 [migration/1]
root          21       2  0 15:34 ?        00:00:00 [ksoftirqd/1]
root          23       2  0 15:34 ?        00:00:00 [kworker/1:0H-events_highpri]
root          24       2  0 15:34 ?        00:00:00 [cpuhp/2]
root          25       2  0 15:34 ?        00:00:00 [idle_inject/2]
```

FIGURE 6.4
Execution of `ps` command.



```
top - 15:58:28 up 24 min,  1 user,  load average: 0.23, 0.29, 0.42
Tasks: 233 total,   1 running, 232 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.3 us,  1.9 sy,  0.2 ni, 86.4 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11869.6 total,   8798.0 free,   1271.7 used,   1799.9 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.  10128.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2105 sunlu     20   0 5252532 266064 120196 S  16.7   2.2   0:11.93 gnome-shell
   4054 sunlu     20   0  786512  46900  36032 S   9.3   0.4   0:00.28 nautilus
   4057 sunlu     20   0 2692040  44404  33308 S   6.7   0.4   0:00.20 org.gnome.Chara
   4051 sunlu     20   0  196716  23468  16576 S   3.3   0.2   0:00.10 gnome-control-c
   1827 sunlu     20   0 7388992 127136  81960 S   3.0   1.0   0:06.72 Xorg
   4055 sunlu     20   0  343916  24564  17716 S   2.7   0.2   0:00.08 gnome-calculato
   3706 sunlu     20   0  894312  55484  41984 S   2.3   0.5   0:01.61 gnome-terminal-
   1769 sunlu     20   0    9648   5936   4132 S   1.3   0.0   0:00.59 dbus-daemon
   1819 sunlu     39  19  710376  28636  19016 S   1.0   0.2   0:00.52 tracker-miner-f
    261 root      19  -1  110588  59760  58172 S   0.7   0.5   0:00.86 systemd-journal
   2254 sunlu     20   0  392040  11920   6852 S   0.7   0.1   0:00.82 ibus-daemon
     14 root      20   0       0      0      0 I   0.3   0.0   0:00.79 rcu_sched
     27 root      20   0       0      0      0 S   0.3   0.0   0:00.44 ksoftirqd/2
    128 root       0 -20       0      0      0 I   0.3   0.0   0:00.39 kworker/u9:0-i915_flip
    604 root      20   0       0      0      0 S   0.3   0.0   0:00.88 nv_queue
```

FIGURE 6.5
Execution of `top` command.

Notice that `top` is a running application where the user can keep interacting with to filter for particular processes, while `ps` is more of a one-time command and its output can be saved into a text file for further processing.

To kill a process, use the `kill` command as follows.

```
$ kill <option> <process ID>
```

The `kill` command offers different options to kill a process. Use `kill -l` to list down the options as given below (and many more).

```
1) SIGHUP   2) SIGINT   3) SIGQUIT  4) SIGILL
5) SIGTRAP  6) SIGABRT  7) SIGBUS   8) SIGFPE
9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT
17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
...
```

Commonly used `kill` options are `kill -9` and `kill -15` followed by the PID. Notice that in Linux the process is arranged in a tree structure; killing a parent process will automatically terminate its children processes, and killing a child process may result in its parent process to restart a new child process.

7

Programming on Linux

CONTENTS

“nobreak

7.1 IDE

“nobreak

7.2 Python Programming

“nobreak

7.3 MATLAB/Octave Programming

Part II

Advanced Linux

8

Administration

CONTENTS

...

8.1 Introduction to Linux Administration

...

8.2 Root Account Management

Managing and protecting the *root* user account is a key portion in Linux administration. It is worth mentioning that the root user is different from a "*sudoer*", i.e., a user that can use `sudo` command, although they both have elevated privileges when comes to system administration. A more detailed explanation is given below.

The root user refers to the user that is created by the system upon first installation of the OS. Its username is by default "root", with a UID of 0, and a GID of 0. Its home directory is by default `/root` instead of `/home/<user name>`. The default prompt for the root user is often `#` instead of `$` for other users. The root user has administrative privileges and can do almost anything without being denied or questioned by the system. As a commonly used safety practice, the password for the root user is usually disabled, thus nobody can login to the system as the root user using username and password authentication.

Notice that in theory the root user does not necessarily need to use the username "root", although it is a default convention. The administrative comes with the UID of 0, not the username. Thus, it is possible to assign the administrative account a different username. It is also possible to create multiple accounts with administrative privileges by assigning UID of 0 to multiple accounts, although it is not recommended to share the same UID among different users.

Although the administrative privilege of root user does not come with its username, in some systems, the username "root" is given elevated privilege anyway. More details about this is introduced later.

Check the root user information as follows.

```
$ cat /etc/passwd | grep ^root
root:x:0:0:root:/root:/bin/bash
```

where the third and fourth column of above give the UID and GID of the root user, respectively.

For comparison, a regular user would have far different UID and GID as shown below.

```
$ cat /etc/passwd | grep ^sunlu
sunlu:x:1000:1000:Sun Lu,,,:/home/sunlu:/bin/bash
```

A sudoer refers to the regular users who can temporarily elevate its privileges and execute administration commands using `sudo <privileged-command>`. A sudoer can also switch to root user prompt using `sudo su` (and quit root user prompt using `exit`). Their sudo privilege comes from the fact that they are included in the "sudo group".

Use `groups` to check existing defined groups in the system, and `groups <user name>` the groups a user is engaged. An example is given below.

```
$ groups
sunlu adm cdrom sudo dip plugdev lpadmin lxd sambashare docker
$ groups sunlu
sunlu : sunlu adm cdrom sudo dip plugdev lpadmin lxd sambashare docker
```

The elevated privilege of the sudoer group is defined in `/etc/sudoers`. A section of the file is given below, where `%` appeared in front of `admin` and `sudo` is used to indicate group name.

```
# User privilege specification
root   ALL=(ALL:ALL) ALL

# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL

# Allow members of group sudo to execute any command
%sudo  ALL=(ALL:ALL) ALL
```

It is possible to give the root account a password, thus enabling root authentication. This approach is sometimes used for troubleshooting and recovering purposes, but it is not a good practice in routine operations.

8.3 User Management

xxx

9

Storage Management

CONTENTS

Upon installation of the system, the OS shall scan the machine and automatically mount (the majority part of) the hard disks under the root directory `/`. All software, data files, devices, etc., shall be registered somewhere in the filesystem hierarchy in Fig. 4.1 introduced in Section 4.1.

Although it is often not necessary for the user to manage the storage and the filesystem by himself, Linux provides flexible tools to monitor and manage the storage of the machine, including manipulating partition table, format partition, and managing its mounting point.

9.1 Monitor Storage Status

A secondary storage, such as a hard disk, needs to be "registered" in the OS before it can be displayed correctly to the user. The registration procedure includes creating disk partitions, and specify mount point and filesystem type for each partition, which looks like Fig.

Use `df` to monitor the status of the mounted storages, including the filesystem name, size, and used percentage. Command `df -h` gives a nice snapshot of the storage usage in a human-readable format. An example is given below.

```
$ df -h
Filesystem     Size  Used Avail Use% Mounted on
tmpfs          1.2G  2.1M  1.2G   1% /run
/dev/sda3      916G   51G  818G   6% /
tmpfs          5.8G     0  5.8G   0% /dev/shm
tmpfs          5.0M  4.0K  5.0M   1% /run/lock
/dev/sda2      512M  5.3M  507M   2% /boot/efi
```

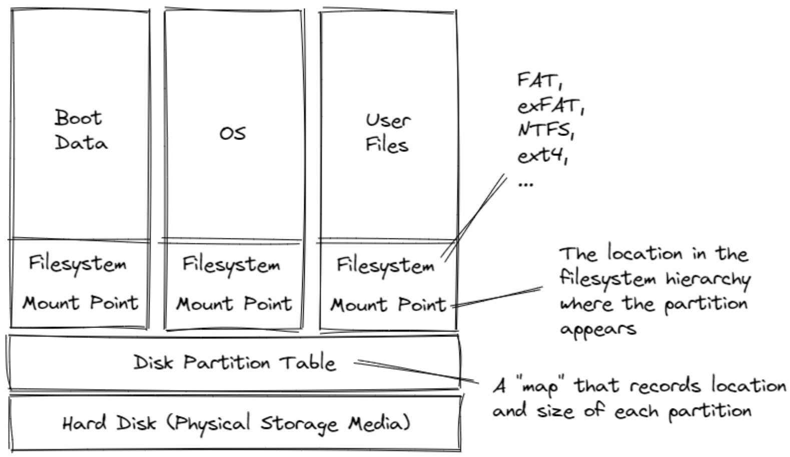


FIGURE 9.1
A demonstration of how a hard disk is “registered” in the OS.

```
tmpfs          1.2G  120K  1.2G   1% /run/user/1000
```

from where it can be seen that the most of the storage of the system, in this case $818G$, is mounted on the root directory.

Use `lsblk` to list information about block devices in the OS. Block devices refer to nonvolatile mass storage devices whose information can be accessed in any order, such as hard disks and CD-ROMs. An example is given below.

```
$ lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
loop0   7:0    0     4K  1 loop /snap/bare/5
loop1   7:1    0    62M  1 loop /snap/core20/1593
loop2   7:2    0    62M  1 loop /snap/core20/1611
loop3   7:3    0 163.3M  1 loop /snap/firefox/1670
loop4   7:4    0   177M  1 loop /snap/firefox/1749
loop5   7:5    0 400.8M  1 loop /snap/gnome-3-38-2004/112
loop6   7:6    0 248.8M  1 loop /snap/gnome-3-38-2004/99
loop7   7:7    0  81.3M  1 loop /snap/gtk-common-themes/1534
loop8   7:8    0  91.7M  1 loop /snap/gtk-common-themes/1535
loop9   7:9    0  45.9M  1 loop /snap/snap-store/575
loop10  7:10   0    47M  1 loop /snap/snapd/16010
loop11  7:11   0  45.9M  1 loop /snap/snap-store/582
loop12  7:12   0    47M  1 loop /snap/snapd/16292
loop13  7:13   0   284K  1 loop /snap/snapd-desktop-integration/10
loop14  7:14   0   284K  1 loop /snap/snapd-desktop-integration/14
sda     8:0    0 931.5G  0 disk
|-sda1  8:1    0     1M  0 part
|-sda2  8:2    0   513M  0 part /boot/efi
|-sda3  8:3    0   931G  0 part /
```

```
sr0    11:0   1  1024M 0 rom
```

which gives the size, the type and the mount point of all the storages.

Use `blkid` to print block device attributes. An example is given below.

```
$ blkid
/dev/sda3: UUID="d0b15b7c-71f2-41f4-b67a-e7c69446feab" BLOCK_SIZE
    ="4096" TYPE="ext4" PARTUUID="4b570507-6aa0-46c7-ac1b-06cb4c8bdb61
    "
```

9.2 Disk Partition Table Manipulation

Due to the development of computer science and OS, manual disk partition is less necessary than before for a casual user. Nevertheless, Linux provides necessary tools for disk partition table manipulation and they are introduced in this section.

9.2.1 Disk Partition

Disk partitioning refers to the action of creating one or more regions on secondary storage (e.g., disk) so that they can be managed separately, as if they are different "virtual disks". An example of disk partitioning on a Windows PC would be to partition a $1TB$ hard drive into $512GB$ of `C:\` drive and $512GB$ of `D:\` drive. Partitioned regions are logically separated. The partitions locations and sizes are stored in the disk partition table.

Some of the reasons for using disk partition include:

- Due to capability limitation, OS cannot handle a very large disk storage as a whole (this is less the case nowadays).
- Different types of files, for example system data and user data, can be stored separately for easy management.
- Different filesystems can be used on different partitions.
- Different partitions can be configured differently with unique settings.
- Sometimes partitioning can speed up hard disk accessing.

9.2.2 Disk Partition Table Manipulation

From the output of `lsblk` command earlier, it is clear that the system's hard disk name is "`sda`". To get a bit more details on this disk and its current partitioning status, use

```
$ sudo fdisk -l | grep sda
Disk /dev/sda: 931.51 GiB, 1000204886016 bytes, 1953525168 sectors
/dev/sda1    2048      4095      2048   1M BIOS boot
/dev/sda2    4096   1054719   1050624 513M EFI System
/dev/sda3 1054720 1953523711 1952468992 931G Linux filesystem
```

From the above result, it can be seen that the disk registered under `/dev/` (this is where the devices are represented by default) has been partitioned into 3 partitions, and the user space that can be further modified is `/dev/sda3` which is $931GB$.

There are variety of tools that can be used for disk partitioning. A common way of doing that is to use

```
$ sudo fdisk /dev/sda3
```

to enter the fdisk utility, and follow the wizard.

Other tools such as `cfdisk` and `parted` can also be used similarly for disk partition.

9.3 Mount, Unmount and Format a Partition

The hard disk can be formatted by partition, from where a new filesystem can be created. To format a partition, double check using `lsblk` to make sure that it is not mounted in the system. Use `sudo umount <partition name>` to unmount a partition.

Use `sudo mkfs` to format and create a new formatted filesystem, then use `mount` to mount it back to the OS. The mount of a partition needs to be recorded into `/etc/fstab` so that the OS would remember it after a reboot.

10

Shell Advanced

CONTENTS

On top of Chapter 2, more advanced shell commands and script programming skills are introduced in this chapter.

10.1 Service Control

There are many services running in the background of the OS, some of which started by the OS while the other by the user. For example, *Apache service* might be used when the system is hosting a webpage. Other commonly used services include keyboard related services, bluetooth services, etc.

To quickly have a glance of the running services, use

```
$ systemctl --type=service
```

These services can be managed using service managing utilities such as `systemctl` and `service`. Some commonly used terminologies are concluded in Table 10.1 with explanations about their differences.

In short, `systemd` is the back-end service of Linux that manages the services. Both `systemctl` and `service` are tools to interact with `systemd` (and other back-end services) to manage the services. Generally speaking, `systemctl` is more straightforward, powerful and more complicated to use, while `service` is usually simpler and user-friendly.

Use the following commands to check the status of a service, and start, stop or reboot the service.

```
$ sudo systemctl status <service name>
$ sudo systemctl start <service name>
$ sudo systemctl stop <service name>
$ sudo systemctl restart <service name>
```

Use the following commands to enable and disable a service. An enabled

TABLE 10.1
Commonly seen terminologies regarding service control.

Term / Tool name	Description
`systemd`	The `systemd`, i.e., *system daemon*, is a suite of basic building blocks for a Linux system that provides a system and service manager that runs as PID 1 and starts the rest of the system.
`systemctl`	The `systemctl` command interacts with the `systemd` service manager to manage the services. Contrary to `service` command, it manages the services by interacting with the `Systemd` process instead of running the init script.
`service`	The `service` command runs a pre-defined wrapper script that allows system administrators to start, stop, and check the status of services. It is a wrapper for `/etc/init.d` scripts, Upstart's `initctl` command, and also `systemctl`.

service automatically starts during the system boot, and a disabled service does not.

```
$ sudo systemctl enable <service name>
$ sudo systemctl disable <service name>
```

Use the following command to mask and unmask a service. A masked service cannot be started even using `systemctl start`.

```
      $ sudo systemctl mask <service name>
      $ sudo systemctl unmask <service name>
```

The `service` command can be used in a similar manner as follows.

```
$ sudo service <service name> status
$ sudo service <service name> start
$ sudo service <service name> stop
$ sudo service <service name> restart
```

10.2 Advanced Shell Programming

...

11

Git

CONTENTS

Git is a distributed version-control system for tracking changes during the software development, and it can be used on cross platforms including Windows, Unix/Linux and MacOS. This chapter introduces the basic use of *Git* on a local Linux machine and with a remote repository host server such as *GitHub*. Continuous integration and continuous development (CI/CD), an important concept in nowadays software development and operation, is also briefly covered. For CI/CD, *GitHub* Action is introduced.

11.1 Introduction

Git, created by Linus Trovalds in 2005, is a distributed version-control system for tracking changes in source code and files. It is helpful with maintaining data integrity during the collaborative development of a software in distributed non-linear work flows. *Git* is free and open-source software under GNU general public license.

With *Git*, all computers participating in the software development store a copy of the full-fledged repository locally with complete history, and it can synchronize with a centralized remote server. *Git* uses "master" and "slave" branches to manage the concurrent development of different features of the project, where the master branch is the stable and shared repository among everyone, and the slave branches are copies of the master branch where individual features can be developed. For a slave branch, once its developed feature is approved, it can be merged back to the master branch. A demonstration is given in Fig. 11.1.

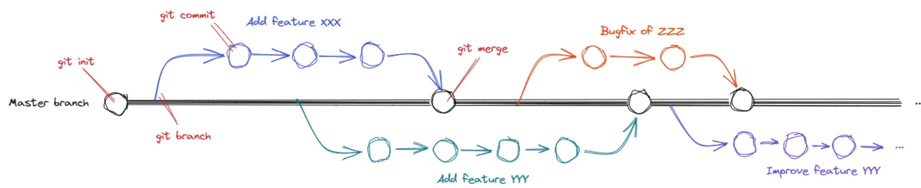


FIGURE 11.1
Git for software development management.

CLI is often used to manage a *Git* repository. For example, `git init` starts a new repository with an empty master branch, and `git branch <branch-name>` creates a new slave branch from the master branch, etc. Some of these commands are shown in Fig. 11.1 and more to be introduced later. Notice that a graphical user interface is also available to interact with `git`. However, in the scope of this notebook, command line is mostly used.

11.2 Setup

To use *Git*, follow the instructions to install and configure the software.

11.2.1 Installation

Git and its relevant documents can be obtained from its official website *https://git-scm.com/*. In many Linux distributions, *Git* is pre-installed. If *Git* is missing in the machine, follow the instructions on the official website to install it on the computer. Notice that the installation procedure may differ with different OS.

11.2.2 Configuration

Upon successful installation, it is recommended to use `git config` for some basic configurations. Notice that there are two types of configurations, namely the global configurations (apply to the machine and the user), and the repository configurations (apply to a particular *Git* repository). By default, the global level configurations are stored under `~/.gitconfig` and the repository level configurations `./.git/config` in the repository. A good practice to edit these configurations is to use *Git* commands rather than editing the files directly.

To add user name and email to the global configuration, use

```
$ git config --global user.name '<user name>'
$ git config --global user.email '<user email>'
```

To retrieve the global configuration, use

```
$ git config --global -l
```

To revoke a global configuration, use

```
$ git config --global --unset <configuration>
```

For example,

```
$ git config --global --unset user.name
```

removes the user name.

More details about `git config` can be found at *https://git-scm.com/docs/git-config*. Usually, the user name and email configurations are mandatory, as they are very useful information in developing collaborative projects.

11.3 Local Repository Management

In practice, repositories are managed locally and they may or may not have their remote associations known as upstreams. *Git* is used to manage local repositories and synchronize them with their associated upstreams. *Git* local repository management is introduced in this section. *Git* is able track versions and control branches.

11.3.1 Initialization of a Repository

Navigate to the project directory. Use the following command to create a new *Git* repository for the project.

```
$ git init
```

From this point forward, *Git* monitors everything that happens inside this directory and its subdirectories and tries to track any change to the files, unless otherwise configured specifically. Many CLI commands, for example `git status` to check the status of the repository, become available. More details are introduced later.

TABLE 11.1
Different file status in a *Git* managed project.

Status	Description
Untracked	Newly added or renamed items in the project directory.
Modified (Unstaged)	Modified items from the last version that has not been registered in the staged area.
Modified (Staged)	Modified items from the last version that has been registered in the staged area. Notice that an untracked item can be staged directly, skipping "modified (unstaged)" step. Use `git add` to stage items.
Unmodified	Unmodified items from the last version. Use `git commit` to update a new version, which would change all the staged items into unmodified items.

11.3.2 Version Tracking

For simplicity, assume that there is only one branch in the repository, namely the master branch. Notice that when there are multiple branches, the version-tracking works the same for each and every branch in a separate and independent manner.

The mechanism behind version tracking is briefly introduced as follows. This helps the user to better understand how *Git* works and how the CLI commands are designed.

The project directory is split into two parts, outside `./.git/` the workspace, and inside `./.git/` the *Git* repository. The workspace has the up-to-date project contents and it is directly managed by the user, while *Git* repository is managed by *Git*. The user should not manage the repository directly unless using the *Git* interface.

Inside the *Git* repository are metadata of the workspace files such as which files have been changed since the last deployment, etc. It also stores a full back up of every historical versions of the project (with powerful compressing mechanisms, of course). It is worth mentioning that instead of recording the changes of a file from version to version, *Git* records the snapshot of the file in every version, unless it is left untouched between two consecutive versions.

Figure 11.2 gives a demonstration of how *Git* manages the project directory. *Git* classifies files in the workspace into different status, as shown in Fig. 11.2. A brief explanation of each types is given in Table 11.1. Detail explanations of "stage" and "commit" are given later.

Use `git status` to check the file status in the project. An example is given below.

```
$ git status
On branch master
```

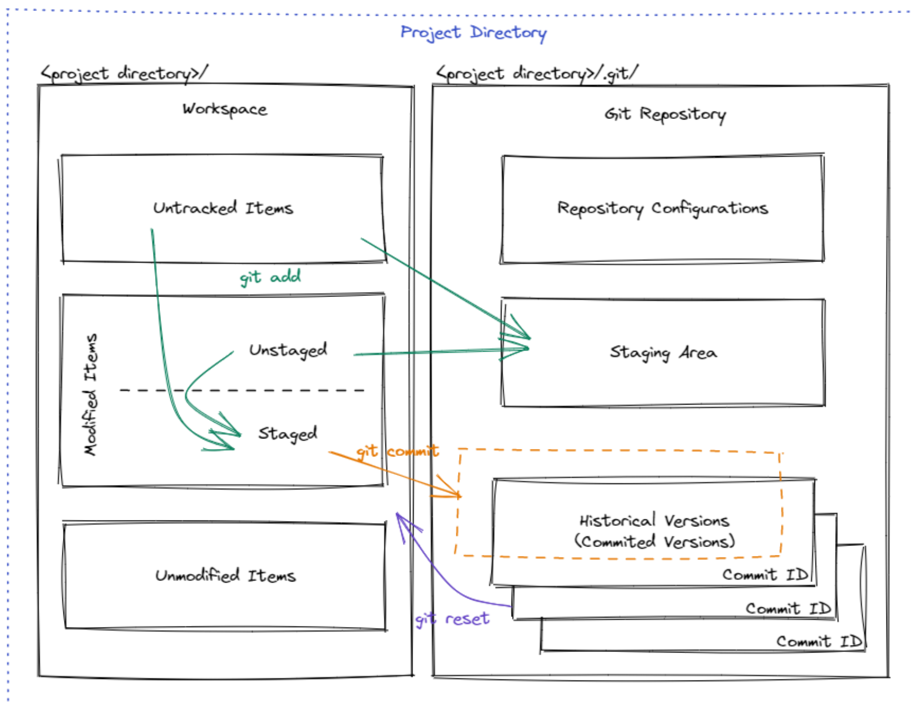


FIGURE 11.2
The project directory managed by *Git*.

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   chapters/ch-software-management-advanced/ch.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   appendix/ap.tex
        modified:   main.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        chapters/ch-software-management-advanced/figures/
```

where `ch.tex` is a modified (staged) item; `ap.tex` and `main.pdf` modified (unstaged) items, and `figures/` an untracked item.

It takes a two-step procedure to back up the project in the *Git* repository. In the first step, the user flags the changed items (either newly added, renamed, or modified) to be backed up in the next version. In the second step, the user actually backs up the items. The first and second steps are called "stage" and "commit" respectively. Notice that it is possible to run a single line of command to execute both steps, but logically it still takes two steps.

Git tracks the name and content of the items that the user has staged in the "staging area" as shown in Fig. 11.2. Think of staging items as taking a snapshot of the items. However, the snapshot at this stage is temporary and has not been backed up in the repository yet. The actual backup happens later when the staged items are committed. To stage an item, use

```
$ git add <item name>
```

which registers the item in the staging area, thus also changes its status from untracked or modified (unstaged) to modified (staged). If an item is modified after it has been staged, *Git* will distinguish the "staged portion" and "unstaged portion" of that item. If using `git status` to check its status, the item will be listed as both staged and unstaged. Unstaged items, either untracked or modified, will remain its status after the commit. Sometimes for convenience, `git add -A` can be used to add all untracked or modified items to the staging area.

Use `git commit` to commit the staged items and back them up in the repository as follows.

```
$ git commit [<item name>]
```

The above command commits the project and creates a version in the *Git* repository. It is possible to specify items, in which case *Git* only commits the specified items and leave the rest items as they are. A commit ID is automatically assigned to the commit. Notice that the user will be asked to provide a "comment message" with the commit, which should be used to briefly explain what has been changed in this commit.

A flag `-a` with `git commit` stages all changes made to the project, then implements the commit command. A flag `-m` simplifies the message recording process and allows the user to key in the message directly after the command. An example is given below.

```
$ git status
On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:  A Notebook on Linux/chapters/ch-software-management-
           advanced/ch.tex
       modified:  A Notebook on Linux/main.pdf

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -am "add introduction to git command"
[master e2e977e] add introduction to git command
 2 files changed, 5 insertions(+), 4 deletions(-)

$ git status
On branch master

nothing to commit, working tree clean
```

To check the commit logs, i.e., all historical commits including their associated timestamps, authors, commit IDs and comment messages, use `git log` as shown in the example below. Notice that the commit logs can be very long. Only a few commits are given for illustration purpose.

```
$ git log
commit e3475e673d8c2a087de6b4423188c51e80af3e5d (HEAD -> master)
Author: sunlu <sunlu.electric@gmail.com>
Date:   Wed Aug 31 15:16:52 2022 +0800

    git

commit 3ab6d1473a7e48d4d890509ddb5a87274c023e6c
Author: sunlu <sunlu.electric@gmail.com>
Date:   Tue Aug 30 15:50:06 2022 +0800

    k8s

commit f6a1e3d779305e966602ecd7589c05f56dd2ad0f
Author: sunlu <sunlu.electric@gmail.com>
Date:   Mon Aug 29 16:31:18 2022 +0800

    more on docker and kubernetes
```

```
commit e2de5b28db7982f57d0ad51361d5161b884f2efe
Author: sunlu <sunlu.electric@gmail.com>
Date:   Sun Aug 28 21:38:48 2022 +0800

    add docker sections
```

where notice that `HEAD` is a reference that points to the latest commit in a branch. Filters can be added as optional arguments for the `git log` command. More details are given at *https://git-scm.com/docs/git-log*.

And finally, to restore to a previous commit version, use `git reset` or `git revert`. Notice that `git reset` and `git revert` can both be used to restore the workspace to a previous stage, but they differ significantly. In general, `git reset` "erases" the past commits, and it is often used in the case where the changes and commits to be reset have not been uploaded to the upstream or shared distributively. Command `git revert` on the other hand create a new commit that undoes all the changes, and it is often used when the changes and commits to revert are already distributed.

The command `git reset` is often used as follows.

```
$ git reset <option> <commit ID>
```

where `<option>` is often `--hard`, `--mixed` or `--soft`, and `<commit ID>` can be the ID of any commit in the git log, or for shorcut `HEAD` the latest commit, `HEAD^` or `HEAD~1` the second latest commit, `HEAD~2` the third latest commit, and so on.

The options `--hard`, `--mixed` and `--soft` work differently. All these options move `HEAD` to the specified commit, and remove all commits afterwards from the repository. The `--hard` option reverts the workspace back to when the specified commit happened (meaning that there would be no way to undo the reset command). Both `--mixed` and `--soft` do not change the workspace. The `--mixed` option leaves all the changes from the specified commit to today as unstaged, while `--soft` leaves them as staged. If no `--hard`, `--mixed` or `--soft` is given, `--mixed` will be used as the default option.

Notice that `git reset` does not help if the unwilling changes and commits to be reverted have already been pushed and synchronized to a remote server or to other collaborators. This is because `git reset` erases the history, and as a result *git* would think that the reset repository is "left behind" when it synchronizes with other repositories. Consequently, it will simply synchronizes the changes and commits back.

In such cases, consider using `git revert` instead. Syntax wise they work similarly. Instead of erase the history, `git revert` creates a new commit whose content is copied and pasted from a past commit.

11.3.3 Branch Management

"Branch" is one of the core features of *Git*, and it plays an important role in collaborative development of a project. There are two types of branches, namely the local branch and the remote branch. The remote branch is often the upstream mirror of the local branch used for sharing and synchronizing the project development with others. The details will be introduced in Section 11.4. Only local branches are considered in this section.

To list down all the local branches, use

```
$ git branch
```

and the current working branch will be highlighted. The current working branch is also referred as the "head branch" or the "active branch".

To create a new branch from the current branch, and to delete a branch, use

```
$ git branch <new branch name> [<commit ID>]
$ git branch -d <branch name>
```

respectively. The optional `<commit ID>` when creating a new branch allows the user to create a branch on top of a specified historical commit instead of the latest commit.

To rename a branch, use

```
$ git branch -m [<old branch name>] <new branch name>
```

where if no `<old branch name>` is specified, the current working branch will be renamed.

To switch to a different working branch, use `git checkout` or `git switch` as follows.

```
$ git checkout <branch name>
$ git switch <branch name>
```

Notice that when there are uncommitted changes in the current branch, *Git* may forbid the user to switch to another branch (the rules are too complicated to be explained here). Therefore, it is recommended to commit the changes before switching to a different branch. When switching to another branch, the workspace will change accordingly to the target branch.

To merge a branch back to the current branch, use

```
$ git merge <branch name>
```

and fill in the comments accordingly. For example, checkout to master branch, and use `git merge <slave branch name>` to merge the features in the slave branch to the master branch. This often happens when the slave branch has finished developing and testing a new feature and is prepared to add the feature to the stable master branch. By default, *Git* automatically deletes the merged branch. This behavior can be changed in the configuration.

Pull Request

In a collaborative project, the master branch is managed very carefully because everything on that branch will affect the project deeply. There is often a group of senior developers who manage the master branch. Any code to be added to the master branch must be checked by one or a few of them.

When a slave branch wants the features developed on that branch to be merged to the master branch, the owner of the slave branch needs to raise a "pull request". As its name suggests, it requests the master branch to pull the updates from the slave branch. The master branch managers will then view and check the slave branch, making sure that there is no bugs, low-quality codes or conflicts, and then approve the merging.

An alternative way of integrating two branches is `git rebase`, which "rewires" the two branches into a linear single branch. Use `git rebase` as follows.

```
$ git rebase <branch name>
```

A demonstrative Fig. 11.3 explains the differences between `git merge` and `git rebase`. It is clear from Fig. 11.3 that by using `git rebase`, all feature commits are integrated into the master repositories to form a single repository tracking line, which differs largely from `git merge`.

An intuitive way to understand `git rebase` is given as follows. Imagine two branches A and B that have diverged from the same commit O. On branch A, execute `git rebase B`. Here is a step-by-step breakdown of what happens:

1. Identify common ancestor of the current branch A and the rebase branch B, which is commit O in this example.

2. Back up commits happened on branch A since commit O. Assume they are commits A_1, A_2, ..., A_n.

3. Reset branch A to Branch B and A now starts from the tip of B.

4. Reapply commits A_1, A_2, ..., A_n on the reset A in a sequential manner, potentially leading to conflicts that need to be resolved manually.

After the rebase operation, the history of branch A appears as if it was developed sequentially from the end of branch B, effectively integrating the changes of B into A in a linear history.

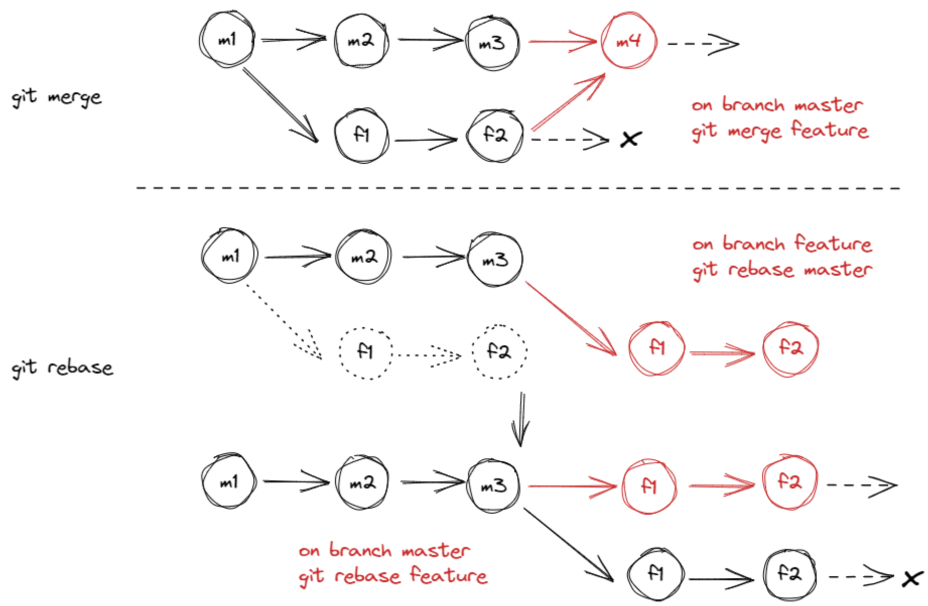


FIGURE 11.3
Two approaches of integrating branches, `git merge` VS `git rebase`.

11.4 Remote Repository Management

In collaborative projects, the repository and branches are stored and shared in one or more cloud servers. Each developer can interact with the repository on the cloud by pulling changes or pushing updates. The developer does not directly make changes to the repository or branches on the cloud. Instead, he clone the repository to his local machine and manipulate branches locally. The local repository and branches are then synchronized with the cloud repository and branches using *Git* commands. This is known as remote repository management and it is introduced in this section.

Depending on the cloud servers that host the remote repository, different services might be available. In the scope of this notebook, *GitHub* is considered as the remote repository host. *GitHub* is an internet hosting service for software development and sharing using *Git*. A user can create, manage, share and co-develop remote repositories on *GitHub*. Notice that *GitHub* is not the only place to host remote repositories. Some alternatives include *GitLab* and *Gitee*. Using *Git*, it is even possible to build a repository hosting server on premises using a regular computer. In this notebook, only *GitHub* is studied.

An *https* URL is associated with each remote repository on *GitHub*, for example *https://github.com/torvalds/linux.git* for Linux kernel. This URL can be used to download the remote repository to a local machine, or to link (synchronize) a local repository with that remote repository.

Consider the case where there is already a remote repository, either under the user's *GitHub* account or from the community. To download a clone of an existing repository to the local machine, use

```
$ git clone <repository URL> [<local directory>]
```

and to maintain the local repository up-to-date with the remote repository, regularly use

```
$ git remote update
$ git pull
```

to synchronize the local repository with the remote one.

Should I use `git pull`?

One may argue whether it is a good idea to use `git pull`. Although convenient, `git pull` is an integration of two commands, `git fetch` and `git merge` (or `git rebase`) executed sequentially. It may be "safer" to manually run the two commands separately.

The local commits can also be pushed to the remote repository using

```
$ git push
```

if the remote repository is under the user's account, or if the owner of the remote repository gives the user the permission. Notice that when pushing updates to the remote repository, *GitHub* may require login credentials, for example username and password pair or SSH key, for permission checks.

Consider another case where there is already a well developed local repository. An empty remote repository has just been created on *GitHub* which is to be setup to mirror the local repository. To link the remote and local repositories, navigate to the local repository and use

```
$ git remote add <remote repository name> <repository URL>
```

which will register the remote repository URL in the local configuration file. A commonly used remote repository name is "origin". The next step is to map the local current working branch with a branch in the remote repository, which would be used as the default source/target branch when running `git pull` and `git push`. This can be done as follows.

```
$ git branch --set-upstream-to <remote repository name> <remote
    repository branch>
```

Notice that this step is not required if the project is cloned from a remote repository using `git clone` introduced earlier, as it will automatically setup the upstream using the source of the clone.

From there on, use `git push` and `git pull` normally.

By assigning upstream branch to a local branch and `git push` the local branch to the upstream, *Git* will update or create the remote branch accordingly. A remote branch can be deleted with a `-d` flag while using `git push`.

11.5 *GitHub Actions*

GitHub Actions is essentially a tool that allows the user to define a pipeline of actions when something changed in the *GitHub* remote repository, for example a pull request is raised, a new commit is pushed, etc. *Actions* is useful as part of CI/CD. More details are given in Appendix B.

Part III

Widely Used Services

12

Relational Database

CONTENTS

Database and database manage systems (DBMS) are introduced in this chapter. Both relational and non-relational databases are covered. Relational database is introduced in this chapter, while non-relational database in the consequent chapter. Tools and languages to manipulate databases, such as structured query language (SQL) for relational databases, are introduced.

12.1 Brief Introduction to Relational Database

Database, in a broad view, refers to an organized collection of data of any format. In this sense, any file format that hosts information in a meaningful and explainable way, such as *CSV*, *XML* and *JSON*, is a database. These file formats often work fine when the data is stored in a centralized manner and its size small.

As the data size grows, the robustness and efficiency of storing and retrieving data become challenging, and different database models have been proposed to tackle it. Dedicated software, namely DBMS, is developed to manage and maintain the database and provide an interface for the users to create, retrieve, update and delete data. Different database models require different database engines and DBMS, and different DBMS may provide different interfaces for the users.

There are many database models available in the market. The most widely seen database models can be divided into two categories, namely the relational databases (RDB) and the non-relational databases.

Relational database was proposed in 1970s by IBM. Some important features of a relational database include the following.

- Structure the data as "relations", which is a collection of tables, each consisting of a set of rows (also known as tuple/record) and columns (also known as attribute/field).
- For each table, a primary key, either being a column or a combination of several columns, is defined that can uniquely distinguish a row from others.
- Provide relational operations that manipulate the data in the tables, for example, joining tables together and aligning them using an attribute.

SQL, a domain-specific language, can be used in managing RDB and interfacing relational DBMS. Most commercialized RDB management systems (RDBMS) adopt SQL as the query language. There are alternatives, but they are rarely used compared to SQL. There is a evolving standard on what operations should an RDBMS support using SQL. The latest version as of this writing is `SQL:2023`.

Examples of relational databases include *Oracle*, *MySQL*, *Microsoft SQL Server*, *MariaDB*, *IBM Db2*, *Amazon Redshift*, *Amazon Aurora* and *PostgreSQL*.

TABLE 12.1
An example of a relational database table.

user			
`user_id`*	`user_email`	`membership`	`referee-id`
sunlu	sunlu@xxx.com	premium	NULL
xingzhe	xingzhe@yyy.com	basic	sunlu
...	...	...	...

12.2 Structured Query Language

As of this writing, RDB is the most commonly used database, and SQL is its standard manipulating and querying language. They are introduced below.

12.2.1 Tables and Columns

An example of a table used in RDB is given in Table 12.1. The table has a name `user` and 4 attributes `user_id`, `user_email`, `membership` and `referee_id`. A table should have an attribute (or a set of attributes) defined as the primary key. In this example, `user_id` is assigned to be the primary key as denoted by the asterisk. The primary key is used to uniquely identify a row in the table. A primary key can consist a single column or multiple columns. When a primary key is composed of multiple columns, it is called a composite key. A table should have one and only one primary key.

Depending on the meaning behind the primary key, it can be divided into two types, namely surrogate key and natural key. A surrogate key is like a serial number or an incremental ID, which serves only for recording and distinguishing rows, and does not have a physical meaning. In contrast, a natural key such as a timestamp, email, citizenship IC number, reflects some meaningful information in the real world.

A foreign key is the attribute(s) that link a table to another table. It is often the primary key of another table that in some way links to this table. For example, consider another table `membership_type` as defined in Table 12.2. In the first Table 12.1, column `membership` can be a foreign key which points to `membership_type` in the second Table 12.2.

The introducing of foreign key helps to maintain the consistency and integrity of the database. For example, when adding a new member to Table 12.1, the DBMS will first check whether the membership type is registered in Table 12.2. If not, the insert operation will be rejected. This guarantees that all registered members have valid membership types.

Notice that a table can have multiple foreign keys. The foreign key can not only relate a table to another, but also relate a table to itself. For example,

TABLE 12.2
A second database table in the example.

membership		
membership_type	monthly_price	annual_price
none	0	0
basic	5	50
premium	10	80

the "referee-id" attribute in Table 12.1 could be the foreign key that links to "user-id" of the same table.

12.2.2 RDB Naming Conventions

It is recommended that the following naming conventions be applied to databases, tables and columns.

- General rules:
  - Use natural collective terms over plurals, for example, "staff" over "employees".
  - Use only letters, numbers, and underscores.
  - Begin with a letter and may not end with an underscore.
  - Avoid using abbreviations unless commonly understood.
  - Avoid using prefixes.
- Table:
  - Do NOT use the same name for a table and one of its columns.
  - Do NOT concatenate two table names to create a third relationship table.
- Column:
  - Use singular name for columns.
  - Avoid using over simplified terms such as "id".
  - Use only lowercase if possible.
- Alias:
  - Use keyword `AS` to indicate an alias.
  - The correlation name should be the first letter of each word of the object name.
  - If there is already the same correlation name, append a number.

- Stored procedure: always contain a verb in the name of a stored procedure.
- Uniform suffixes:
  - `_id`: primary key.
  - `_status`: flag values.
  - `_total`: the total number of a collection of values.
  - `_num`: a number.
  - `_date`: a date.
  - `_name`: the name of a person or a product.

12.2.3 General Introduction to SQL

SQL is the most widely used language for interacting with DBMS for data query and maintenance. SQL is very powerful and flexible in its full capability, and it is hardly possible to cover everything in one section. Hence, in this section only the basic SQL operations are introduced.

Notice that the support of different DBMS to SQL may differ slightly. This is because an DBMS may (in fact, very likely) fail to adopt everything in the latest SQL standard. However, most of the commands especially the widely used ones such as creating tables, inserting rows and most of the querying, shall be universally consistent.

SQL is a hybrid language consisting of the following 4 sub-languages.

- Data query language: query information and metadata of a database.
- Data definition language: define database schema.
- Data control language: control user access and permission to a database.
- Data manipulation language: insert, update and delete data from a database.

SQL supports variety of data types, and different DBMS may cover slightly different data types. Some of the most commonly used data types are summarized in Table 12.3. Nowadays, many DBMS supports more complicated types such as objects (the associated database is also known as the object-relational database), and many more. For the full list of data types that a DBMS supports, check the manuals and documents of that DBMS.

SQL defines reserved keywords for database manipulation. The keywords have specific meanings and cannot be used as user-defined variable names. Commonly used SQL keywords are summarized in Tables 12.4, 12.5 and 12.6.

TABLE 12.3
Widely used SQL data types.

Data Type	Description
INT/INTEGER	Integer, with a range of -2147483648 to 2147483647. When marked "UNSIGNED", the range becomes 0 to 4294967295. Some relevant data types are TINYINT, SMALLINT, MEDIUMINT, and BIGINT, which have different ranges.
DEC/DECIMAL(size,d)	An exact fixed-point number. The total number of digits and the number of digits after decimal point are specified by "size" and "d", respectively. Some relevant data types are DOUBLE(size,d), which can also be used to specify a floating point. Notice that DEC/DECIMAL is usually preferable in most occasions.
CHAR(size)	A fixed length string with the specified length in characters.
VARCHAR(size)	A variable length string, with the specified maximum string length in characters.
BOOL/BOOLEAN	This is essentially a 1-digit integer, where 0 stands for "false" and otherwise "true".
BLOB	A binary large object with maximum 65535 bytes.
DATE	A date by format "YYYY-MM-DD".
TIME	A time by format "hh:mm:ss".
DATETIME	A combination of date and time by format "YYYY-MM-DD hh:mm:ss".
TIMESTAMP	A timestamp that measures the number of seconds since the Unix epoch. The format is "YYYY-MM-DD hh:mm:ss". Unlike DATETIME which is meaningful only if the timezone is known, TIMESTAMP does not rely on timezone.

TABLE 12.4
Widely used SQL keywords (part 1: names).

Keyword	Description
CONSTRAINT	A constraint that limits the value of a column.
DATABASE	A database.
TABLE	A table.
COLUMN	A column (attribute, field) of a table.
VIEW	A view, which is a virtual table that does not store data by itself but only reflects the base tables data.
INDEX	An index, which is a pre-scan of specific column(s) of a table and can be used to speed up future queries related to the column(s). Notice that unlike a view, an index needs to be stored together with the table.
PRIMARY KEY	The primary key of a table.
FOREIGN KEY	A foreign key defined in a table that links to a (different) table.
PROCEDURE	A procedure that defines a list of database operations to be executed one after another

TABLE 12.5
Widely used SQL keywords (part 2: actions).

Keyword	Description
CREATE	Create a database (CREATE DATABASE), a table (CREATE TABLE), a view (CREATE VIEW), an index (CREATE INDEX) or a procedure (CREATE PROCEDURE).
ADD	Add a column in an existing table, or a constraint to an existing column.
ALTER	Modify columns in a table (ALTER TABLE), or a data type of a column (ALTER COLUMN).
SET	Specify the columns and values to be updated in a table.
DROP	Delete a column (DROP COLUMN), a constraint (DROP CONSTRAINT), a database (DROP DATABASE), an index (DROP INDEX), a table (DROP TABLE), or a view (DROP VIEW).
CHECK	Define a constraint that limits the value that can be placed in a column.
DEFAULT	Define a default value for a column.
INSERT INTO	Insert a new row into a table.
UPDATE	Update an existing row in a table.
DELETE	Delete a row from a table.
EXEC	Executes a stored procedure.

TABLE 12.6
Widely used SQL keywords (part 3: queries).

Keyword	Description
SELECT	Query data from a database. Relevant combinations are SELECT DISTINCT which returns only distinct values; SELECT INTO which copies data from one table into another; SELECT TOP which returns part of the results.
AS	Assign an alias to a column or table.
FROM	Specify the table where the query is run.
WHERE	Filter results that fulfill a specified condition.
IN	Specify multiple values in a WHERE clause.
AND	Select rows where both conditions are true.
OR	Select rows where either condition is true.
ALL	Return true if all followed sub-query values meet the condition.
ANY	Return true if any followed sub-query value meet the condition.
BETWEEN	Select values within a given range.
ORDER BY	Sort the results in ascending or descending order.
JOIN	Join tables for query. Relevant combinations are OUTER JOIN, INNER JOIN, LEFT JOIN and RIGHT JOIN.
EXISTS	Tests for the existence of any record in a sub-query.
GROUP BY	Groups the result set when using aggregate functions (COUNT, MAX, MIN, SUM, AVG).
UNION	Combines the result sets of multiple select statements.

12.2.4 General Syntax

Notice that different DBMS may use slightly different syntax for the same or similar function. In the rest of the chapter, unless otherwise mentioned, MySQL/MariaDB syntax is used.

All SQL commands shall end with a semicolon "`;`".

The programming of SQL shall follow the following common practices wherever possible. This helps to maintain the good quality and portability of the code.

- Use standard SQL functions over user-defined functions wherever possible for better portability.
- Do NOT use object-oriented design principles in SQL and database schema wherever possible.
- Use UPPERCASE for keywords.
- Use `/*<comments>*/` to add comments to the code, otherwise precede comments with `-- <comments>` and finish them with a new line.

The naming of database, tables and columns shall follow conventions introduced in Section 12.2.1.

During the coding, follow the following rules.

- Use spaces to align the codes.
- Use a space before and after equals (=), and after commas (,).
- Use BETWEEN and IN, instead of combining multiple AND and OR clauses.

When creating a table, follow the following rules.

- Choose standard SQL data types wherever possible.
- Specify default values and set up constraints, and put them close to the declaration of the associated column name.
- Assign primary key carefully and keep it simple.
- Specify the primary key first right after the CREATE TABLE statement.
- Implement validation. For example, for a numerical value, use CHECK to prevent incorrect values.

12.2.5 Database Manipulation

To list down all the databases running on the server, use

```
SHOW DATABASES;
```

To create a database, use

```
CREATE DATABASE <database-name>;
```

To select a database, use

```
USE <database-name>;
```

To delete a database, use

```
DROP DATABASE <database-name>;
```

12.2.6 Table Manipulation

Tables are the fundamental components in an RDB. Some features of a table have been introduced in Section 12.2.1. An example of creating a table using SQL is given below

```
CREATE TABLE <table_name> (
   PRIMARY KEY (<column_name>),
   <column_name_1>  <data-type>  <constraint>,
   <column_name_2>  <data-type>  <constraint>,
                    CONSTRAINT <constraint-name-1>
                    CHECK(<constraint-rule>),
                    CONSTRAINT <constraint-name-2>
                    CHECK(<constraint-rule>)
);
```

where in this demonstrative table, 2 columns and 2 constraints are defined.

The `<constraint>` that comes after the data type of a column is used to set an additional restriction to the data in the table. When such restriction is violated, an error would raise to stop the operation. For example, if `NOT NULL` is set as a constraint, then when inserting a row to the table later, the user cannot input NULL for that specific column. Notice that the "primary key" can also be set as a constraint named `PRIMARY KEY` using this syntax, although it is a better practice to use `PRIMARY KEY (<column-name>)`.

Commonly used such constraints are summarized into Table 12.7. As shown by Table 12.7, a default value can be assigned to a column by using the `DEFAULT <value>` constraint. When inserting a new row, the column of that row will be assigned to its default value if no other value is assigned. If no such statement is provided for a column, its default value is NULL.

Upon creation of a table, its basic schema can be reviewed using

```
DESCRIBE <table-name>;
```

To list down existing tables, use

TABLE 12.7
Commonly used constraints.

Constraint	Description
NOT NULL	Not allowed to be NULL.
UNIQUE	Not allowed to have duplicated values.
PRIMARY KEY	Set as primary key, thus, must be not NULL and must remain unique.
FOREIGH KEY	Set as foreign key.
DEFAULT <value>	Set a default value.
AUTO_INCREMENT = <value>	Each time a new row is inserted and NULL or 0 is set for this column, instead of set the column to NULL or 0, automatically generate the next sequence number. The starting value is defined by <value> which by default is 1.

```
SHOW TABLES;
```

To delete a table, use

```
DROP TABLE <table-name>;
```

To edit the column of a table, use either of the following

```
ALTER TABLE <table-name>
ADD <column-name> <data-type>; -- add new column
ALTER TABLE <table-name>
DROP COLUMN <column-name>; -- drop column
ALTER TABLE <table-name>
RENAME COLUMN <old-name> TO <new-name>; -- rename column
ALTER TABLE <table-name>
MODIFY COLUMN <column-name> <data-type>; -- modify column data type (
    depending on DBMS, syntax may differ)
```

or

```
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name> CHECK(<constraint-rule>); -- add
    constraint
ALTER TABLE <table-name>
DROP CONSTRAINT <constraint-name>; -- drop constraint
```

Notice that it is possible to change the primary key using the above syntax because essentially the primary key is treated as a constraint named "primary". It is not recommended to do so in general.

The foreign key is a key used to point to another table, in many cases other table's primary key. Therefore, the foreign key can be nominated only after the other table has been created. Declare foreign key upon creation of a table as follows. As mentioned, to do this, the other tables must be created beforehand. Two methods to declare a foreign key are shown below.

```
CREATE TABLE <table-name> (
    PRIMARY KEY (<column-name>),
    <column-1>   <data-type>   <constraint>,
    <column-2>   <data-type>   <constraint>,
                 CONSTRAINT <constraint-name-1>
                 CHECK(<constraint-rule>),
                 CONSTRAINT <constraint-name-2>
                 CHECK(<constraint-rule>),
    FOREIGN KEY (<column-name>) REFERENCES <target-table-name>(<target-
        column-name>), -- method 1
                 CONSTRAINT <constraint-name-3>
                 FOREIGN KEY (<column name>)
                 REFERENCES <target-table-name>(<target-column-name>) --
                      method 2
);
```

where, as can be seen, foreign key is treated as a constraint in the table.

To define a foreign key in an existing table, use

```
ALTER TABLE <table-name>
ADD FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<
    referred-column-name>); -- one way
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name> FOREIGN KEY (<column name>) REFERENCES
     <referred-table-name>(<referred-column-name>); -- another way
```

To drop a foreign key, use

```
ALTER TABLE <table-name>
DROP CONSTRAINT <constraint-name>;
```

There are variety ways of checking the constraints names of a table. An example is given below.

```
SELECT TABLE_NAME, CONSTRAINT_TYPE, CONSTRAINT_NAME
FROM information_schema.table_constraints
WHERE table_name=<table-name>;
```

One thing to notice is that upon creation of a foreign key, the referred column becomes the "parent" and the foreign key becomes a "child". As long as the child exists, the parent cannot be removed from its table. This helps to protect the schema of the database. Should there be any quest to break the schema, this restriction can be overwritten. When defining the foreign key, add an additional claim "ON DELETE SET NULL" or "ON DELETE CASCADE" as follows.

```
FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<referred-
    column-name>) ON DELETE SET NULL
FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<referred-
    column-name>) ON DELETE CASCADE
```

in the first scenario, the child foreign key will be set to NULL, while in the second scenario, the child relevant rows will be removed.

12.2.7 Row Manipulation

To insert a row into a table, use

```
INSERT INTO <table-name> VALUES (<content>, <content>, ...);
```

where the contents shall follow the field sequence as shown by the `DESCRIBE <table-name>` command. To specify the column name while inserting a row, use

```
INSERT INTO <table-name>(<column-name>, <column-name>, ...) VALUES (<
    content>, <content>, ...);
```

Notice that it is also possible to populate multiple rows of a table using one command as follows.

```
INSERT INTO <table-name>(<column-name>, <column-name>, ...)
VALUES (<content>, <content>, ...),
      (<content>, <content>, ...),
      (<content>, <content>, ...);
```

where 3 rows are inserted into the table.

Notice that if a foreign key bound exists between two tables, when inserting a row to the child table, the foreign key value of this row must already be defined in the parent table.

Use the following command to query all items in a table, which can be used to check whether the row is added to the table correctly.

```
SELECT * FROM <table-name>;
```

To modify the attributes of specific row(s), use

```
UPDATE <table-name>
SET <column-name> = <value>, ...
WHERE <filter-criteria>;
```

where `<filter-critera>` is used to filter the rows to which the update is carried out. Commonly used filter criteria are a set of `<cloumn-name> = <value>` separated by `AND` and `OR`. The filter criteria can be set very flexibly and more details are given in later sections. Notice that it is possible to change multiple column values together, by stacking multiple `<column-name> = <value>` separated by ",". Similarly, to delete rows from a table, use

```
DELETE FROM <table-name>
WHERE <filter-criteria>;
```

Notice that if filter criteria is not specified, i.e., if `WHERE` is missing, all items in the table will be affected.

12.2.8 Query

A typical query looks like the following and it returns the data in a table-like format.

```
SELECT <column-or-statistics>
FROM <table-name-or-combination>
GROUP BY <column-name>
WHERE <filter-criteria>
ORDER BY <column-name>, ...
LIMIT <number>;
```

where

- `<column-or-statistics>` describes the columns to be returned, and specifies the returning information format.
- `<table-name-or-combination>` describes the source of the information, either being a table, or a joint of multiple tables.
- `GROUP BY <column-name>` groups rows with the same value of the specified column into "summary rows".
- `<filter-condition>` defines the filter criteria and only rows meet the criteria are returned.
- `ORDER BY <column-name>` allows the items to be returned in a specific order based on ascending/descending order. It is worth mentioning that the `<column-name>` here does not need to appear in the selected returns, and it can be multiple columns separated by ",". Use `ASC` (default) or `DESC` after each `<column-name>` to specify ascending or descending order.
- `LIMIT <number>` restricts the maximum number of rows to be returned.

Notice that `SELECT` and `FROM` statements are compulsory in all queries, `WHERE` statement very widely used, and other statements optional case by case.

More details are given below.

The statement `<column-or-statistics>` mainly controls the information to be returned. Commonly seen selected items in `<column-or-statistics>` are summarized as follows.

- `*` (asterisk): return all columns.
- `<column-name>`: return selected columns. When multiple fields are returned, use bracket `(<col1>, <col2>, ...)`. The same applies to other return formats below.
- `<table-name>.<column-name>`: return selected columns, and to avoid ambiguity, specify table name with the column name. This is useful when the source of data is a joint of multiple tables, some of which share the same column name.

- `<column-name> AS <alias>`: return selected columns, and use alias in the returns.
- `DISTINCT <column-name>`: return only distinct rows.
- `COUNT()`, `SUM()`, `MIN()`, `MAX()`, `AVG()`: return aggregate function of a column instead of all the items in that column. They can be used along with DISTINCT, for example, `COUNT(DISTINCT <column-name>, ...)`.
- Simple calculations to the above result, for example `1.5*<column-name>`. Commonly used arithmetic operations are `+`, `-`, `*`, `/`, `%`, `DIV` (integer division).

When the column is of date time type or interval type, it is possible to use `EXTRACT()` to select a field of the timestamp or interval. For example,

```
SELECT EXTRACT(YEAR FROM birthday) AS year FROM customer;
```

would look into table `customer`, focus on column `birthday` which should be a datetime type, and extract only `year` from the datetime to return the result.

The statement `<table-name-or-combination>` mainly indicates the source table(s). It can be a single table, a joint of multiple tables, or a nest query. More details about joint of multiple tables are illustrated below.

Consider the following example, where two tables are given as follows.

```
> SELECT * FROM test;
+---------+---------+---------+
| test_id | value_1 | value_2 |
+---------+---------+---------+
|       1 | a       |      10 |
|       2 | a       |      20 |
|       3 | a       |      30 |
|       4 | b       |     100 |
|       5 | b       |     200 |
|       6 | c       |    1000 |
|       7 | c       |    2000 |
+---------+---------+---------+

> SELECT * FROM test_join;
+--------------+---------+---------+---------+
| test_join_id | value_1 | value_2 | value_3 |
+--------------+---------+---------+---------+
| a            |      10 |      99 | alpha   |
| b            |     100 |     999 | bravo   |
| d            |   10000 |   99999 | delta   |
+--------------+---------+---------+---------+
```

There are different types of joins, namely "inner join" (or "join"), "left join", "right join" and "cross join". They are introduced as follows.

The most intuitive join is the cross join. It returns everything in the two tables like a Cartesian product (that explains whey cross join is also called

Cartesian join), where the total number of columns are the sum of two tables, the number of rows the product of two tables, as shown below.

```
> SELECT * FROM test CROSS JOIN test_join;
+---------+---------+---------+-------------+---------+---------+---------+

| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
    value_3 |
+---------+---------+---------+-------------+---------+---------+---------+

|       1 | a       |      10 | a            |      10 |      99 | alpha  |
|       1 | a       |      10 | b            |     100 |     999 | bravo  |
|       1 | a       |      10 | d            |   10000 |   99999 | delta  |
|       2 | a       |      20 | a            |      10 |      99 | alpha  |
|       2 | a       |      20 | b            |     100 |     999 | bravo  |
|       2 | a       |      20 | d            |   10000 |   99999 | delta  |
|       3 | a       |      30 | a            |      10 |      99 | alpha  |
|       3 | a       |      30 | b            |     100 |     999 | bravo  |
|       3 | a       |      30 | d            |   10000 |   99999 | delta  |
|       4 | b       |     100 | a            |      10 |      99 | alpha  |
|       4 | b       |     100 | b            |     100 |     999 | bravo  |
|       4 | b       |     100 | d            |   10000 |   99999 | delta  |
|       5 | b       |     200 | a            |      10 |      99 | alpha  |
|       5 | b       |     200 | b            |     100 |     999 | bravo  |
|       5 | b       |     200 | d            |   10000 |   99999 | delta  |
|       6 | c       |    1000 | a            |      10 |      99 | alpha  |
|       6 | c       |    1000 | b            |     100 |     999 | bravo  |
|       6 | c       |    1000 | d            |   10000 |   99999 | delta  |
|       7 | c       |    2000 | a            |      10 |      99 | alpha  |
|       7 | c       |    2000 | b            |     100 |     999 | bravo  |
|       7 | c       |    2000 | d            |   10000 |   99999 | delta  |
+---------+---------+---------+-------------+---------+---------+---------+
```

where notice that `CROSS JOIN` can be replaced by a comma ",".

In this example, `value_1` from table `test` and `test_join_id` from table `test_join` are corresponding with each other. In this context, the rows with inconsistent `test.value_1` and `test_join.test_join_id` is meaningless and shall be removed. This can be achieved by the following code.

```
> SELECT * FROM test CROSS JOIN test_join
   -> where test.value_1 = test_join.test_join_id;
+---------+---------+---------+-------------+---------+---------+---------+

| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
    value_3 |
+---------+---------+---------+-------------+---------+---------+---------+

|       1 | a       |      10 | a            |      10 |      99 | alpha  |
|       2 | a       |      20 | a            |      10 |      99 | alpha  |
```

```
|       3 | a       |      30 | a           |      10 |      99 | alpha   |
|       4 | b       |     100 | b           |     100 |     999 | bravo   |
|       5 | b       |     200 | b           |     100 |     999 | bravo   |
+---------+---------+---------+-------------+---------+---------+---------+
```

and this is equivalent to inner join (or simply, join)

```
> SELECT * FROM test JOIN test_join
    -> ON (test.value_1 = test_join.test_join_id);
+---------+---------+---------+-------------+---------+---------+---------+

| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
   value_3 |
+---------+---------+---------+-------------+---------+---------+---------+

|       1 | a       |      10 | a           |      10 |      99 | alpha   |
|       2 | a       |      20 | a           |      10 |      99 | alpha   |
|       3 | a       |      30 | a           |      10 |      99 | alpha   |
|       4 | b       |     100 | b           |     100 |     999 | bravo   |
|       5 | b       |     200 | b           |     100 |     999 | bravo   |
+---------+---------+---------+-------------+---------+---------+---------+
```

where `ON (<table1.column-name> = <table2.column_name>)` is used to indicate the association. The number of columns remain unchanged compared with cross join, but the number of rows is reduced.

From table `test` perspective, its rows regarding `value_1` equals to “a” and “b” are fully included in the inner join results. However, the two rows regarding `value_1=c` is omitted. This is because there is no corresponding row in the other table `test_join` with `test_join_id=c`.

It is possible to “prevent” information loss from table `test` by adding these two rows back, with all the columns from table `test_join` filled with NULL. This can be done by left join as follows.

```
> SELECT * FROM test LEFT JOIN test_join
    -> ON (test.value_1 = test_join.test_join_id);
+---------+---------+---------+-------------+---------+---------+---------+

| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
   value_3 |
+---------+---------+---------+-------------+---------+---------+---------+

|       1 | a       |      10 | a           |      10 |      99 | alpha   |
|       2 | a       |      20 | a           |      10 |      99 | alpha   |
|       3 | a       |      30 | a           |      10 |      99 | alpha   |
|       4 | b       |     100 | b           |     100 |     999 | bravo   |
|       5 | b       |     200 | b           |     100 |     999 | bravo   |
|       6 | c       |    1000 | NULL        |    NULL |    NULL | NULL    |
|       7 | c       |    2000 | NULL        |    NULL |    NULL | NULL    |
```

```
+---------+---------+---------+-------------+---------+---------+---------+
```

One can think of this as temporarily adding a new row to `test_join` with `test_join_id=c` and everything else NULL, before the inner joining.

The same idea applies to right join as well, as shown below.

```
> SELECT * FROM test RIGHT JOIN test_join ON (test.value_1 = test_join.
    test_join_id);
+---------+---------+---------+-------------+---------+---------+---------+

| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
    value_3 |
+---------+---------+---------+-------------+---------+---------+---------+

|       1 | a       |      10 | a           |      10 |      99 | alpha  |
|       2 | a       |      20 | a           |      10 |      99 | alpha  |
|       3 | a       |      30 | a           |      10 |      99 | alpha  |
|       4 | b       |     100 | b           |     100 |     999 | bravo  |
|       5 | b       |     200 | b           |     100 |     999 | bravo  |
|    NULL | NULL    |    NULL | d           |   10000 |   99999 | delta  |
+---------+---------+---------+-------------+---------+---------+---------+
```

Some DBMS supports "outer join", which is basically a union of the left and right join results. More about union is introduced later.

The statement `<filter-condition>` applies filtering to the results. Commonly seen filter criteria `<filter-condition>` are summarized as follows.

- `<column-name> = <value>`, where `=` can be replaced by `<` (less than), `<=` (less than or equal to), `>` (larger than), `>=` (larger than or equal to) and `<>` (not equal to).
- `<column-name> IN (<value>, <value>, ...)`
- `<column-name> BETWEEN <value> AND <value>`
- `<column-name> LIKE <wildcards>`, which compares the column value (usually a string) with a given pattern.
- A combination of the above, with `AND` and `OR` joining everything together.

A bit more about wildcard query is introduced as follows. A wildcard character is a "placeholder" that represents a group of character(s). Most commonly used wildcard characters in the SQL context include

- `_`: any single character.
- `%`: any string of characters (including empty string).
- `[<c1><c2> ...]`: any single character given in the bracket.

- `^[<c1><c2>...]`: any single character not given in the bracket.
- `[<c1>-<c2>]`: any single character given within the range in the bracket.

Wildcard query can be applied to both CHAR and DATE/TIME types, as they can all be characterized as strings.

The `GROUP BY` groups the rows with the same value of the specified column into "summary rows". In each summary row, aggregated information is collected. To further explain this, consider the following example.

```
> SELECT * FROM test;
+---------+---------+---------+
| test_id | value_1 | value_2 |
+---------+---------+---------+
|       1 | a       |      10 |
|       2 | a       |      20 |
|       3 | a       |      30 |
|       4 | b       |     100 |
|       5 | b       |     200 |
|       6 | c       |    1000 |
|       7 | c       |    2000 |
+---------+---------+---------+
```

Running the following command gives

```
> SELECT * FROM test GROUP BY value_1;
+---------+---------+---------+
| test_id | value_1 | value_2 |
+---------+---------+---------+
|       1 | a       |      10 |
|       4 | b       |     100 |
|       6 | c       |    1000 |
+---------+---------+---------+
```

From the result, it can be seen that summary rows have been created using `GROUP BY`. In the returned table, `value_1` has distinguished values. In this example, it simply picks up the first appearance of the rows in the original table that has distinguished `value_1`, and a lot of information seems to be lost. However, it is worth mentioning that although not displayed, the aggregated information is included.

To verify the presence of the aggregated information, consider running the following command.

```
> SELECT COUNT(*) FROM test GROUP BY value_1;
+----------+
| COUNT(*) |
+----------+
|        3 |
|        2 |
|        2 |
+----------+
```

From the result, we can see that the counted number of each summary row is returned. Similarly, the following SQL returns other aggregation information associated with each summary row.

```
> SELECT value_1, COUNT(*), SUM(value_2) FROM test GROUP BY value_1;
+---------+----------+--------------+
| value_1 | COUNT(*) | SUM(value_2) |
+---------+----------+--------------+
| a       |        3 |           60 |
| b       |        2 |          300 |
| c       |        2 |         3000 |
+---------+----------+--------------+
```

Finally, `ORDER BY` and `LIMIT` controls the sequence and maximum number of returned rows, respectively.

The returns of multiple queries might be able to `UNION` together, if they are union-compatible. Union simply means concatenate the tables vertically. To union the results, use

```
SELECT <...>
UNION
SELECT <...>
UNION
SELECT <...>
...
SELECT <...>;
```

where inside `<...>` are the original query statements. Notice that for the queries to be union-compatible, they must have the same number of columns with identical data type for the associated columns. The names of the column in the returns, if different, follow the first query result. Use alias `AS` to change the names if needed. Duplicated rows in the union will be excluded. If duplications need to be included in the result, certain DBMS provides the `UNION ALL` option.

SQL uses nest queries to add more flexibility. Nest queries plays as the intermediate steps to provide a temporary searching result, from which another query can be executed. Wherever a table name appears in the query, it can be replaced by a `SELECT` statement nested in a bracket "()". A demonstrative example is given below. Consider the same tables `test` and `test_join` as follows.

```
> SELECT * FROM test;
+---------+---------+---------+
| test_id | value_1 | value_2 |
+---------+---------+---------+
|       1 | a       |      10 |
|       2 | a       |      20 |
|       3 | a       |      30 |
|       4 | b       |     100 |
|       5 | b       |     200 |
```

```
|      6 | c      |    1000 |
|      7 | c      |    2000 |
+---------+---------+---------+

> SELECT * FROM test_join;
+--------------+---------+---------+---------+
| test_join_id | value_1 | value_2 | value_3 |
+--------------+---------+---------+---------+
| a            |      10 |      99 | alpha   |
| b            |     100 |     999 | bravo   |
| d            |   10000 |   99999 | delta   |
+--------------+---------+---------+---------+
```

A inner join is provided to the above tables. However, for each `value_1` in the first table, the sum of the associated `value_2`, instead of each individual row, is used. This can be achieved using

```
> SELECT temp.value_1 AS type,
   ->        temp.sum_value_2 AS total_value,
   ->        test_join.value_1 AS minval,
   ->        test_join.value_2 AS maxval,
   ->        test_join.value_3 AS abbrev
   -> FROM (SELECT value_1,
   ->             SUM(value_2) AS sum_value_2
   ->      FROM test GROUP BY value_1) AS temp
   -> JOIN test_join
   -> ON (temp.value_1 = test_join.test_join_id);
+------+-------------+--------+--------+--------+
| type | total_value | minval | maxval | abbrev |
+------+-------------+--------+--------+--------+
| a    |          60 |     10 |     99 | alpha  |
| b    |         300 |    100 |    999 | bravo  |
+------+-------------+--------+--------+--------+
```

where notice that alias are quite some times to clarify the logics.

Nest queries can be popular in table joins as well as filter criteria, where the boundary of a variable can be obtained from a nest query.

12.2.9 Trigger

A trigger defines a set of operations to be carried out automatically when something happens to specified tables. For example, in any case a new row is added to a table, a trigger can automatically insert an associated record into another table.

There are mainly 3 types of triggers: DML trigger (triggered by `INSERT`, `UPDATE`, `DELETE`, etc.), DDL trigger (triggered by `CREATE`, `ALTER`, `DROP`, `GRANT`, `DENY`, `REVOKE`, etc.), and CLR trigger (triggered by LOGON event).

A quick DML trigger can be defined as follows.

```
CREATE TRIGGER <trigger-name>
    [BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON <table-name>
    FOR EACH ROW <operation>;
```

where `BRFORE` is often used to validate and modify data to be added to `<table-name>`, and `AFTER` is often used to trigger other changes consequent to this change.

In case multiple operations need to be defined, consider using

```
DELIMITER $$
CREATE TRIGGER <trigger-name>
    [BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON <table-name>
    FOR EACH ROW BEGIN
        <operation>;
        ...
        <operation>;
    END$$
DELIMITER ;
```

where `DELIMITER $$` and `DELIMITER ;` is used to temporarily change the delimiter for the `BEGIN...END` statement. It is possible to build slightly complicated logics in the operations, for example to build conditional statements.

Use `NEW` in the operation(s) to represent the rows that is added/updated/deleted from the `table-name`.

Use the following to drop a trigger.

```
DROP TRIGGER <trigger-name>;
```

12.2.10 SQL Demonstrative Example

An an example to demonstrate the use of SQL, a database is created from scratch. MariaDB is used as the DBMS in this example. More about MariaDB is introduced in later Section 12.3. The database is used in the smart home project to track the resources obtained and consumed by the user. The resources in this context may refer to groceries bought from the supermarket, books purchased online, subscriptions of magazines and services, etc. For simplicity, the prompt is ignored in the rest of this section.

Check the existing databases as follows.

```
SHOW DATABASES;
```

A database named `smart_home` is created as follows.

```
CREATE DATABASE smart_home;
```

Select the database as follows.

```
USE smart_home;
```

With the above command, `smart_home` is selected as the current database.

Based on the database schema design, a few tables need to be created. We shall start with creating `asset`, `accessory`, `consumable` and `subscription` tables as follows.

The `asset` table is used to trace assets in the home. They are often expensive and comes with a serial number or a warranty number, and shall persist for a long time (a few years, at minimum). Examples of assets include beds, televisions, computers, printers, game consoles. The `accessory` table is used to trace relatively cheaper accessories than assets. Though they are designed to last long, they may not have an serial number. Examples of accessories include books, charging cables, coffee cups. The `consumable` table is used to trace items that is meant to be used up or expire. Examples of consumable items include food, shampoo, A4 printing paper. And finally the `subscription` table is used to trace subscriptions of services. Examples of these services include software license (either permanent license or annual subscription license), magazine subscriptions, membership subscriptions, and digital procurement of a movie.

The serial number or warranty number for assets are used as the primary key of `asset` table. For the other three tables, surrogate keys are used. Each table has a column `product_type_id` that specifies the type of the item, such as "television", "cooker", "fruit", "software". The types in these tables are given by integer indices. A separate `product_type` relates the indices with their associated meanings. The same applies to `product_brand_id` and `payment_method_id`.

Create `asset` table as follows.

```
CREATE TABLE asset (
       PRIMARY KEY (serial_num),
       serial_num              VARCHAR(50)   NOT NULL,
       product_type_id         INT(5),
       product_brand_id        INT(5),
       product_name            VARCHAR(50)   NOT NULL,
       receipt_num             VARCHAR(50),
       procured_date           DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       procured_price          DECIMAL(10,2),
       payment_method_id       INT(5),
       warranty_date_1         DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       warranty_date_2         DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       expire_date             DATE          NOT NULL DEFAULT
           '9999-12-31',
       CONSTRAINT warranty_after_procured
       CHECK(warranty_date_1 >= procured_date AND warranty_date_2 >=
           warranty_date_1),
       CONSTRAINT expire_after_procured
       CHECK(expire_date >= procured_date)
);
```

where

- `serial_num`: the serial number, MAC number or registration ID that can be used to uniquely identify the asset.
- `product_type_id`: type index.
- `product_brand_id`: brand index.
- `product_name`: full name of the product that can uniquely specify the asset on the market.
- `receipt_num`: receipt and/or warranty number.
- `procured_date`: date of procurement.
- `procured_price`: price of the product as procured.
- `payment_method_id`: payment method.
- `warranty_date_1`: warranty expiration date (free replace or repair); leave it as the procured date if no such warranty is issued.
- `warranty_date_2`: second warranty expiration date (partially covered repair); leave it as the procured date if no such warranty is issued.
- `expire_date`: the date when the asset expires or needs to be returned. For example, in Singapore a car "expires" in 10 years from the day of procurement.

Notice that constraints and default values have been added to the table creation. An SQL script is used contain the code, and

```
$ mariadb -u <user-name> -p < <script-name>
```

is used to execute the script, which is more convinient than typing all the lines in the MariaDB console.

Similarly, create the rest 3 tables for the resources as follows.

```
CREATE TABLE accessory (
       PRIMARY KEY (item_id),
       item_id                  INT(5)         AUTO_INCREMENT,
       product_type_id          INT(5),
       product_brand_id         INT(5),
       product_name             VARCHAR(50)    NOT NULL,
       receipt_num              VARCHAR(50),
       procured_date            DATE           NOT NULL DEFAULT (
           CURRENT_DATE),
       procured_number          DECIMAL(10,2)  NOT NULL DEFAULT 1.00,
       procured_unit_price      DECIMAL(10,2),
       procured_price           DECIMAL(10,2),
       payment_method_id        INT(5),
```

```
       expire_date              DATE          NOT NULL DEFAULT
           '9999-12-31',
       CONSTRAINT expire_after_procured
       CHECK(expire_date >= procured_date)
);

CREATE TABLE consumable (
       PRIMARY KEY (item_id),
       item_id                  INT(5)        AUTO_INCREMENT,
       product_type_id          INT(5),
       product_brand_id         INT(5),
       product_name             VARCHAR(50)   NOT NULL,
       receipt_num              VARCHAR(50),
       procured_date            DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       procured_number          DECIMAL(10,2) NOT NULL DEFAULT 1.00,
       procured_unit_price      DECIMAL(10,2),
       procured_price           DECIMAL(10,2),
       payment_method_id        INT(5),
       expire_date              DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       CONSTRAINT expire_after_procured
       CHECK(expire_date >= procured_date)
);

CREATE TABLE subscription (
       PRIMARY KEY (item_id),
       item_id                  INT(5)        AUTO_INCREMENT,
       product_type_id          INT(5),
       product_brand_id         INT(5),
       product_name             VARCHAR(50)   NOT NULL,
       receipt_num              VARCHAR(50),
       procured_date            DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       procured_price           DECIMAL(10,2),
       payment_method_id        INT(5),
       expire_date              DATE          NOT NULL DEFAULT (
           CURRENT_DATE),
       CONSTRAINT expire_after_procured
       CHECK(expire_date >= procured_date)
);
```

Create the tables for users, product types, product brands and payment methods as follows.

```
CREATE TABLE user (
       PRIMARY KEY (user_id),
       user_id                  INT(5),
       first_name               VARCHAR(50)   NOT NULL,
       last_name                VARCHAR(50)   NOT NULL,
```

```
       email                    VARCHAR(50)   NOT NULL UNIQUE
);

CREATE TABLE product_type (
       PRIMARY KEY (product_type_id),
       product_type_id          INT(5)        AUTO_INCREMENT,
       product_type_name        VARCHAR(50)   NOT NULL UNIQUE,
       product_type_name_sub    VARCHAR(50)   NOT NULL DEFAULT ('na')
);

CREATE TABLE product_brand (
       PRIMARY KEY (product_brand_id),
       product_brand_id         INT(5)        AUTO_INCREMENT,
       product_brand_name       VARCHAR(50)   NOT NULL UNIQUE
);

CREATE TABLE payment_method (
       PRIMARY KEY (payment_method_id),
       payment_method_id        INT(50)       AUTO_INCREMENT,
       user_id                  INT(5),
       payment_method_name      VARCHAR(50)   NOT NULL
);
```

Finally, create foreign keys as follows.

```
ALTER TABLE payment_method
ADD FOREIGN KEY (user_id)
REFERENCES user(user_id);
ALTER TABLE asset
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE asset
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE asset
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE accessory
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE accessory
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE accessory
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE consumable
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE consumable
```

```
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE consumable
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE subscription
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE subscription
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE subscription
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
```

12.3 RDB Example: MySQL and MariaDB

Commonly seen DBMS examples include Oracle Database, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, MariaDB, and many more. Some of them are briefly introduced in this section. Here are some examples.

MariaDB and MySQL are two widely used relational DBMS. MariaDB is initially a fork of MySQL, and in this sense they share many similarities. While MySQL moves towards a dual license approach (free community license and paid enterprise license with proprietary code), MariaDB is designed to be fully open-source under GNU license, and plays as a replacement of MySQL.

In general, MariaDB supports a larger varieties of data engines and new features, and it is claimed to be faster, more powerful and advanced than MySQL. However, it lacks some of the enterprise features provided by MySQL. The users can gain some of these features by using open-source plugins. The most recent new features in MySQL and MariaDB are also diverging.

Some of the main features of MySQL and MariaDB are summarized as follows.

- Good performance (very fast) in general, for a medium size database. Hence, it is popular in many web applications.
- Ease of use.
- Supports in-memory tables to handle read-heavy write-lite tasks.
- Not very flexible, as compared to PostgreSQL where more complex data types, queries, and functions add-ons are supported.

Different databases may propose different minimum system requirements. There is no standard on how these minimum requirements are calculated.

TABLE 12.8
An estimation of DBMS hardware requirements.

	OS	Minimum Requirements		
		CPU	RAM	Disk*
MySQL (E)	All*	2 core	2 GB	1.3 GB
MySQL (C)	All	1 core	1 GB	1.3 GB
MariaDB	All	1 core	400MB	660 MB
PostgreSQL	All	1 core	1 GB	40MB
Firebird	All	1 core	12MB	15MB

* Some installations may come with default test database and logs. The disk usage can be reduced if those files are removed after installation.
** "All" refers to Linux, MacOS and Windows. Different platforms may have different minimum requirements.

Therefore, it is often unfair to directly compare the requirements of different databases. Nevertheless, a summary table is given in Table 12.8. Notice that this is merely an estimation and can differ largely from practice.

In the remaining of this section, MariaDB is used for demonstration.

12.3.1 MariaDB Installation

To install MariaDB, follow the instruction on the official website. Different OS, such as RHEL and Ubuntu, may require different ways of installation. For example, on RHEL

```
$ sudo yum update
$ sudo yum install mariadb-server
```

and on Ubuntu,

```
$ sudo apt update
$ sudo apt install mariadb-server
```

Notice that MySQL can be installed instead by replacing `mariadb-server` with `mysql-server`. Similarly, replacing `mariadb` with `mysql` in the rest of this section, wherever applicable, would work for MySQL.

MariaDB server can be controlled using `systemctl` which is introduced in Section 10.1. For example, to start MariaDB, use

```
$ sudo systemctl start mariadb.service
```

and to check its status, use

```
$ sudo systemctl status mariadb
```

12.3.2 MariaDB Basic Configuration

After installation of and starting MariaDB, use

```
$ sudo mysql_secure_installation
```

to run a quick security-related configuration such as creating password for the root user, and deleting test database.

Login to MariaDB console using

```
$ sudo mariadb
```

Notice that when *sudo* privilege is used, the user should be brought to the root account of the DBMS. Without sudo privilege, a user name and a password is required to login to the DBMS as follows.

```
$ mariadb -u <user-name> -p
Enter Password:
```

Depending on the setup, remote login to the database from other machine, especially using root account, might be forbidden.

12.3.3 MariaDB Console

After login to MariaDB console, a prompt that looks like the following would show up.

```
MariaDB [(none)]>
```

from where an admin account can be created as follows.

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@'
    localhost' IDENTIFIED BY '<user-password>' WITH GRANT OPTION;
MariaDB [(none)]> FLUSH PRIVILEGES;
```

By using an admin account instead of the root account in daily operations, the security risk is reduced.

Notice that remote access to a database is by default forbidden. A command similar with the above is required to enable remote access. Wildcard expression can be used for the IP address, if necessary, to allow a user to access the database from multiple machines.

To check existing users and their IP addresses to whom access has been granted, use

```
SELECT host, user FROM mysql.user;
```

Finally, use

```
MariaDB [(none)]> exit
```

to quite MariaDB console.

12.3.4 Execute SQL File

In many occasions, instead of executing SQL commands line by line, an SQL file is prepared in advance and the DBMS is asked to execute the SQL file as a whole. In the Linux shell, execute the following command and MariaDB shall be able to execute an SQL file on the specified host, on behalf of the user, on the selected database.

```
$ mariadb --host="mysql_server" --user="user_name" --database="
    database_name" --password="user_password" < "file.sql"
```

If the user already logs into the MariaDB console, use the following command instead.

```
MariaDB [(none)]> source file.sql
```

12.4 RDB Example: PostgreSQL

PostgreSQL (or Postgres) is claimed to be the world's most advanced open-source RDB, and it has some great features. It is a object-relational database where the user can define customized data types in the form of objects. The database functions are modularized, meaning that it has a small base installation (only 40MB) as given in Table 12.8, and the user can expand its capability by installing additional modules per required. It complies very well with the latest SQL standard, with additional powerful add-ons.

Though powerful and efficient, PostgreSQL has not grown as popular as some other databases such as Oracle, Microsoft SQL server, and MySQL. One of the reasons for this is the lack of enterprise tier support. Being under PostgreSQL license (an open-source license similar to BSD and MIT), the available support is mostly from the community. With that being said, PostgreSQL is mature and has been adopted in some large enterprises. Of course the IT department of these companies need to work hard to maintain the database. In addition, great flexibility often means a steeper learning curve, making mastering PostgreSQL generally more difficult than other aforementioned databases.

However, with the populating use of microservices where the database efficiency and performance becomes absolutely critical, PostgreSQL is gaining more and more attentions. It is especially popular when using inside containers.

Some of the main features of PostgreSQL are summarized as follows.

- Object-relational database.
- Excellent adherence to SQL standard.

- Excellent performance and scalability.
- Multi-version concurrency control.
- Modularization of features, i.e., light weight base installation and scalable add-ons.
- Disk-based database, and does not support in-memory tables. As a compensation, it has robust caching mechanisms that speed up reading and writing.
- Steep learning curve.

12.4.1 Installation

PostgreSQL supports a large range of operating systems including Linux, MacOS, Windows, and more. The source code and very detailed documentations of PostgreSQL are available on its websites and can be downloaded to a local machine freely. Due to the lite weight of the base installation, PostgreSQL can be used conveniently in containers. The associated images can be found on Docker Hub.

To install PostgreSQL directly on a host machine, simply follow the instructions on the official website, where command line tools are provided for Linux users, and installer for windows users. For example, to install PostgreSQL version 15 on RHEL version 9 running on `x86_64` and setup initial configurations, simply use

```
# Install the repository RPM:
$ sudo dnf install -y https://download.postgresql.org/pub/repos/yum/
    reporpms/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# Disable the built-in PostgreSQL module:
$ sudo dnf -qy module disable postgresql

# Install PostgreSQL:
$ sudo dnf install -y postgresql15-server

# Optionally initialize the database and enable automatic start:
$ sudo /usr/pgsql-15/bin/postgresql-15-setup initdb
$ sudo systemctl enable postgresql-15
$ sudo systemctl start postgresql-15
```

After installation, use command `pg_ctl` to control the server, and `psql` to login to the server. Notice that PostgreSQL installation may come with PgAdmin, the GUI for the DBMS, which can also be used to interact with the databases.

To use PostgreSQL in containers, either download and run PostgreSQL images from Docker Hub, or create a Dockerfile like the following that installs and configures PostgreSQL on top of an Alpine base image. An example of such Dockerfiles is given below.

```
FROM alpine
RUN apk update
# install postgresql
RUN apk add postgresql
RUN mkdir /run/postgresql
RUN chown postgres:postgres /run/postgresql/
USER postgres
WORKDIR /var/lib/postgresql
RUN mkdir /var/lib/postgresql/data
RUN chmod 0700 /var/lib/postgresql/data
RUN initdb -D /var/lib/postgresql/data
# prepare user scripts
RUN mkdir /var/lib/postgresql/user-scripts
RUN chmod 0700 /var/lib/postgresql/user-scripts
COPY ./start.sh /var/lib/postgresql/user-scripts
COPY ./setup_db.sql /var/lib/postgresql/user-scripts
COPY ./populate_db.py /var/lib/postgresql/user-scripts
# prepare user data
RUN mkdir /var/lib/postgresql/user-data
RUN chmod 0700 /var/lib/postgresql/user-data
COPY ./google_stock_price.csv /var/lib/postgresql/user-data
#
CMD ["/bin/sh", "/var/lib/postgresql/user-scripts/start.sh"]
```

12.4.2 PostgreSQL-Specific New Data Types

PostgreSQL supports a large range of data types, more than other databases such as MySQL and MariaDB. In addition to the commonly seen numeric types, character types, date and time types and boolean type, the following data types are supported.

- Currency (monetary) types.
- Geometric types, including points, line segments, boxes, paths, polygons and circles.
- Network address types, such as IPv4 and IPv6 addresses and MAC addresses.
- Array types and associated functions such as accessing, modifying and searching arrays.
- Composite types and associated functions.
- Many more.

12.4.3 PostgreSQL User-Defined Data Types

User-defined data types can be used to demonstrate the object-relational database aspect of PostgreSQL. It essentially means that the attribute of an element can be an object, and can have some complex and comprehensive features.

To create customized data types, use the following syntax

```
/*Composite Types*/
CREATE TYPE name AS
( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] );

/*Enumerated Types*/
CREATE TYPE name AS ENUM
( [ 'label' [, ... ] ] );

/*Range Types*/
CREATE TYPE name AS RANGE (
SUBTYPE = subtype
[ , SUBTYPE_OPCLASS = subtype_operator_class ]
[ , COLLATION = collation ]
[ , CANONICAL = canonical_function ]
[ , SUBTYPE_DIFF = subtype_diff_function ]
[ , MULTIRANGE_TYPE_NAME = multirange_type_name ]
);

/*Base Types*/
CREATE TYPE name (
INPUT = input_function,
OUTPUT = output_function
[ , RECEIVE = receive_function ]
[ , SEND = send_function ]
[ , TYPMOD_IN = type_modifier_input_function ]
[ , TYPMOD_OUT = type_modifier_output_function ]
[ , ANALYZE = analyze_function ]
[ , SUBSCRIPT = subscript_function ]
[ , INTERNALLENGTH = { internallength | VARIABLE } ]
[ , PASSEDBYVALUE ]
[ , ALIGNMENT = alignment ]
[ , STORAGE = storage ]
[ , LIKE = like_type ]
[ , CATEGORY = category ]
[ , PREFERRED = preferred ]
[ , DEFAULT = default ]
[ , ELEMENT = element ]
[ , DELIMITER = delimiter ]
[ , COLLATABLE = collatable ]
);
```

For example,

```
CREATE TYPE sex_type AS
enum ('M', 'F');
```

which creates a new data type called `sex_type`, and it can take enumerated value of either `M` or `F`.

12.4.4 PostgreSQL Stored Procedural and Functions

Many databases including Oracle SQL, Microsoft SQL Server, MySQL, MariaDB and PostgreSQL, support stored procedures and user defined functions. This feature has been there for a very long time, but it is often not introduced in a typical introductory course. Though useful, they may introduce performance and portability issues, hence need to be handled with caution. Besides, nowadays it is often regarded as a better practice to implement the logics in the application layer, not in DBMS. Nevertheless, they are briefly introduced as follows.

The following is an example to define a function using SQL.

```
CREATE OR REPLACE FUNCTION add_int(int, int)
RETURNS int AS
'
SELECT $1+$2;
'
LANGUAGE SQL
```

where notice that the input variable types are given in the bracket (use `()` if there is no input), the output following `RETURNS` (use `void` if there is no output), and the SQL operations in between quotations `'`, which is a deliminator and can be replaced by something else, such as the following

```
CREATE OR REPLACE FUNCTION add_int(int, int)
RETURNS int AS
$body$
SELECT $1+$2;
$body$
LANGUAGE SQL
```

Instead of using `$1`, `$2` to refer to a input, names can be assigned together with types as follows.

```
CREATE OR REPLACE FUNCTION add_int(var1 int, var2 int)
RETURNS int AS
$body$
SELECT var1+var2;
$body$
LANGUAGE SQL
```

Notice that so far we have been using SQL as the programming language for the functions, as indicated by `LANGUAGE SQL`. Notice that PostgreSQL also supports other languages, such as PL/pgSQL, which is a procedural programming language supported by PostgreSQL. It closely resembles Oracle's PL/SQL language. "PL" in these terms represents "Procedural Language".

The following is a list of languages supported by PostgreSQL.

- Naive installation:
  - PL/pgSQL
  - SQL
  - C
- Extension:
  - PL/Python
  - PL/Perl
  - PL/Java
  - PL/R
  - and more.

SQL is sufficient to carry out simple and straight forward tasks such as adding two numbers, as shown in the earlier example. However, when comes to handling conditional statements and loops, etc., procedural language is required. When the function is complex, it is sometimes impossible or inefficient to implement it using SQL, and PL/pgSQL and other procedural languages can solve this problem. An example is given below.

```
CREATE OR REPLACE FUNCTION increment_value(value INT, increment INT)
RETURNS INT AS $$
DECLARE
	result INT;
BEGIN
	IF increment > 0 THEN
		result := value + increment;
	ELSE
		RAISE EXCEPTION 'Increment must be positive';
	END IF;
	RETURN result;
END;
$$ LANGUAGE plpgsql;
```

With the above been said, though convenient and powerful it might be in some use cases, it is often a good practice to keep complex logic in application layer, for logic consistency and database portability.

12.4.5 Manipulation and Query

PostgreSQL adopts SQL for database manipulation and query. SQL has been introduced in earlier sections, hence it is not repeated here. Only selected unique features to PostgreSQL are introduced.

While PostgreSQL server is running, enter its console using `psql` from the shell. One can tell PostgreSQL console by its prompt which looks something like

```
postgres-#
```

or

```
postgres->
```

with "postgres" the current selected database. It is also possible to specify user name, default database, and other configurations when connecting to PostgreSQL server. Instead of running `psql` as admin, use

```
$ psql -h <host> -p <port> -U <username> <default_database>
```

Once in PostgreSQL console, use `help` to display the basic commands, including `\copyright` that shows the distribution terms, `\h` to check SQL commands, `\?` to check psql commands, and `\q` to quit PostgreSQL console, etc. Notice that both SQL and psql commands can be used in PostgreSQL console.

Some widely used psql commands are summarized in Table 12.9. Most, if not all, psql commands start with a back slash "\".

SQL commands, such as creating a database, have been introduced earlier, hence is not repeated here.

12.5 Database API

This section discusses the tools or program interfaces used along with the DBMS. Many software programs provide interface or toolkit to connect to a database. For example, MATLAB uses database toolbox to connect to SQLite or Microsoft SQL server. Python, as a "glue" language, also provide variety of packages to connect to different databases. IDEs such as VSCode can connect to databases using extensions.

Some of these tools and packages are introduced in this section.

12.5.1 RDB Interface with Python

Python provides variety of libraries to access RDS, many of which use embedded SQL codes to interact with the DBMS. Depending on the DBMS,

TABLE 12.9
Widely used psql commands.

Command	Description
SELECT VERSION();	Check PostgreSQL version.
\l	List databases.
\c <database>	Switch databases.
\d	Describe items. By default, it lists the tables in the current database, and describe each of them.
\dt	List tables.
\dv	List views.
\dn	List schema.
\df	List functions.
\du	List users.
\d <table>	Describe a table.
\s	Show command history.
\h	Show help.
\?	Show psql commands.
\!cls	Clear screen.
\q	Quit DBMS shell.

different libraries and commands can be used, some of which more general and the other more specific to a particular DBMS.

In this section, both `pandas` and `mariadb` libraries are introduced. The `pandas` library provides data manipulation and analysis tools, and it provides `pandas.io.sql` that allows connecting to a DBMS and embedding SQL commands into the python code. The `mariadb` library, on the other hand, is dedicated for MariaDB connection. Like `pandas.io.sql`, it also allows embedding SQL commands to interface the DMBS.

As pre-requisitions, make sure that the following has been done.

- The DBMS has been configured to allow remote access.
- Make sure that an account has been registered in DBMS that has the privilege of operation from a remote machine.
- Make sure that the firewall configuration is correct.

For DBMS configuration, in the case of MariaDB, use the following code in the shell to check the location of the configuration files.

```
$ mysqld --help --verbose
```

Typical locations of the configuration files are `/etc/my.cnf` and `/etc/mysql/my.cnf`. In the configuration file, use the following to disable binding address.

```
[mysqld]
```

```
skip-networking=0
skip-bind-address
```

For account setup, in the DBMS console, use something like

```
> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@'<ip-address>'
    IDENTIFIED BY '<user-password>' WITH GRANT OPTION;
```

where `'<ip-address>'` is the remote machine that runs the Python program. If the Python codes are running locally, simply use `'localhost'`.

It might be necessary to install MariaDB database development files in the DMBS host machine for the Python libraries to be introduced to function properly. Install the development as follows.

```
$ sudo apt install libmariadb-dev
```

PANDAS Library

Python library `pandas` is one of the essential libraries for data analysis. It provides flexible interfaces and tools for data reading and processing and works very well with different data formats and engines including CSV, EXCEL and DBMS. This section focuses mainly on the interaction of `pandas` with DBMS. Therefore, the detailed use of `pandas` for data analysis, etc., are not covered in this section.

A class `pandas.DataFrame` is defined in `pandas` as the backbone to store and process data. The data attribute of `pandas.DataFrame` is a `numpy` array. Many functions are provided to read different data formats into `pandas` data frame, which makes reading data easy and convenient. An example of reading a CSV file is given below.

```
import pandas as pd
df = pd.read_csv(<file-name>)
print(df)
print(df.head(<number>))
print(df.tail(<number>))
print(df.info())
```

where `head()`, `tail()` gives the first and last rows of the data frame, and `info()` checks the data frame basic information including shape and data types of the columns. Check `df.columns` for all the columns of the data frame. Details specific to a column can be accessed via `df.[<column-name>]`. Many functions are provided to further abstract the details, such as grouping and counting. Use `df.loc(<row-index-list>, <column-name-list>)` to check the content of specified rows and columns.

With `pandas` and other relavent libraries, Python can connect to a database and execute a query. An example of using `pandas` to connect to an Microsoft SQL server and implement a query is given below. Notice that different DBMS may require different database connectivity driver standards, and there are mainly two of them, namely open database connectivity (ODBC) and Java database connectivity (JDBC). Microsoft adopts ODBC, and a separate package is required in the Python program to connect to the Microsoft SQL server.

```
import pyodbc
import pandas.io.sql as psql

server = "<server-url>,<port>"
database = "<database>"
uid = "<uid>"
pwd = "<pwd>"
driver = "<driver>" # such as "{ODBC Driver 17 for SQL Server}"

# connect to database
conn = pyodbc.connect(
       server = server,
       database = database,
       uid = uid,
       pwd = pwd,
       driver = driver
)

# get cursor
cursor = conn.cursor()

# execute sql command
query = """<query>"""
runs = psql.read_sql_query(query, conn)
```

The above codes returns a data frame corresponding to the result set of the query string, which is saved in `runs`.

MARIADB Library

Use the following code to test connectivity from Python to the database.

```
import mariadb
import sys

user = "<user>"
password = "<password>"
host = "<server-url>"
port = "<port>" # MariaDB default: 3306
database = "<database>"

# connect to database
try:
       conn = mariadb.connect(
              user = user,
              password = password,
              host = host,
```

```
              port = port,
              database = database
       )
except mariadb.Error as e:
       print(f"Error connecting to MariaDB Platform: {e}")
       sys.exit(1)

# get cursor
cur = conn.cursor()

# execute sql command
cur.execute("<sql-command>")
```

Notice that for query, the result is stored in the cursor object. Use a for loop to view the results.

13

Non-Relational Database

CONTENTS

Relational database has been introduced in the earlier chapter. This chapter discusses non-relational database.

13.1 Brief Introduction to Non-Relational Database

Non-relational databases (NoSQL databases) gained its popularity in the 2000s. In contrast to RDB, NoSQL databases do not store data in tables the way RDB does, but in key-value pairs, graphs, documents, or other formats, and it is more flexible, efficient and easier to use in some applications than the conventional RDBs. Examples of NoSQL databases include *Redis*, *Azure CosmosDB*, *Oracle NoSQL Database*, *Amazon DynamoDB*, *MongoDB*, *AllegroGraph* and many more.

Unlike SQL which applies to almost all RDBMS, there is no universally adopted language for NoSQL DBMS. Each NoSQL database often has its unique query language tailored to its specific data model. We use "NoSQL" to refer to a collective set of languages used for NoSQL database management. For simple applications, different RDB may perform similarly, with some subtle differences in the performance. However, this is different for NoSQL databases, which may function completely differently.

There are many types of non-relational databases including document-oriented databases, directed graph databases, etc. Each of them has some unique features over other databases, and may adopt its own database manipulation language. It is impossible to cover everything in this notebook. Only brief introductions of selected databases examples are given here. More details might be found on other relevant notebooks, for example graphical database in *A Notebook on Probability, Statistics and Data Science* as part of the semantic web.

Database services, both RDB and NoSQL, have become critical to our daily life and they are massively deployed on servers. In many applications they work together to deliver the service.

13.2 Non-RDB Example: MongoDB

MongoDB is a source-available, cross-platform, general-purpose, document-oriented database program. Classified as a NoSQL database program, MongoDB uses BSON (an extension of JSON) documents with flexible schema to store data. This is not surprising, as MongoDB itself is powered by a JavaScript engine. MongoDB, together with other JavaScript-relevant software such as *Express.js*, *React.js*, *Node.js*, etc., is often used to create database-powered web applications.

Comparing with conventional RDBs, MongoDB is better at

- Massive data storage (in the order of TBs and PBs);
- Frequent and parallel operations when performing insert and query;
- Flexible scalability and high availability.

Notice that MongoDB as well as many other NoSQL databases are not suitable to handle financial transactions due to the consistency issue that many NoSQL databases suffer. But things might change as new technologies become handy.

Examples of MongoDB-suitable applications include

- Posts and streaming management on social media websites or APPs;
- Online gaming;
- Logistics industry and supply chain management;
- IoT data management.

As a document-oriented database, MongoDB stores data in documents and collections instead of rows and tables. A document is the basic unit of the storage. A collection is a group of documents on the same/similar topic. Each document in the same collection can adopt its own schema, and it should be self-contained so that when querying the data the user does not join collections and map documents. Figure 13.1 gives a demonstrative example of how MongoDB stores and organizes data.

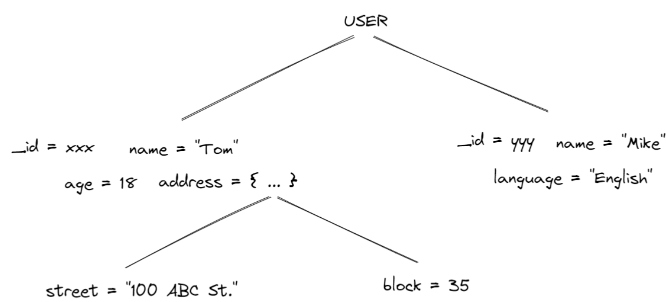


FIGURE 13.1
A demonstrative example of how MongoDB stores data as (nested) object.

In Fig. 13.1, a collection "USER" is defined. The collection contains two documents. Each document has a few properties. Different document may share common properties such as "name" in this example. Each of them can also have unique properties of its own such as "age", "address" and "language". Different properties may have different data types. For example, a property can be string, numeric, or a nested object, or an array of the above. When querying MongoDB, the DBMS is able to return selected properties of documents that meet specific criteria, and sort them in the required order.

13.2.1 Installation

MongoDB, like many other DBMS, has both community and enterprise versions. MongoDB community server can be installed following the instructions in the official website

```
https://www.mongodb.com/try/download/community
```

To interact with MongoDB DBMS, the quickest way is to use MongoDB shell (also known as "mongosh"). A description of the shell can be found at

```
https://www.mongodb.com/try/download/shell
```

Notice that when installing MongoDB server, it is possible that the server comes with MongoDB Compass, the GUI for MongoDB DBMS. The GUI can also be used to interact with the databases.

TABLE 13.1
MongoDB basic commands.

Command	Description
`db.help()`	Show a list of methods of `db` object.
`db.version()`	Show database server version.
`db.getUsers()`	Show users.
`db.createUser(<content>)`	Create a user. The username, password, roles, etc., needs to be included in the `<content>` area.
`db.dropUser(<username>)`	Drop a user.
`db.dropDatabase()`	Drop current database.
`db.status()`	Show the basic status of the currently selected database, such as its name, number of collections, storage size, etc.

Different versions of MongoDB are available. Choose the correct MongoDB version depending on the CPU and the OS of the machine. MongoDB community server installation size is about 500MB.

MongoDB provides enterprise service, MongoDB Atlas, as part of its cloud solution where user can deploy clusters to host databases. With proper gateway setup, the user can connect to MongoDB Atlas clusters from his local server to retrieve data or to manipulate the databases. MongoDB Atlas is not the covered in this notebook.

13.2.2 Basic Global Operations

After installing both MongoDB server and MonghDB shell, use

```
$ mongosh
```

in the command line to login to the DBMS. JavaScript-like commands are used to manipulate the database, including creating databases and inserting data.

The object `db` contains many methods using which the user can access and modify the basic configurations of the database. For example,

```
> db.version()
```

gives the version of the database server. More commands can be found using `db.help()`. Some commonly used commands are listed in Table 13.1.

To display the existing databases, use

```
> show dbs
```

On a clean installation, the above should return the 3 default databases, `admin`, `config` and `local`. To change to a particular database, use

```
> use <database_name>
```

Notice that there is no "create database" command in MongoDB. To create a database, switch to that database using the above command (even though it does not exist yet), and create some data such as a collection there. The database will be automatically created. This is again a feature closely related to JavaScript.

13.2.3 Data Format

MongoDB is a document-oriented database program. The data is stored in the format of "binary JSON" (BSON), which can be taken as an extension of JSON that supports more data types. Since BSON and JSON have a strong connection to the JavaScript object datatype, which looks very similar to a "dictionary", one may claim that MongoDB uses key-value pairs to store data. That statement is partially correct because BSON indeed adopts key-value pair structure to store data, but it may over simplifies the reality and be misleading sometimes. In fact, BSON supports much more complicated data types other than string-to-string, such as array and nested objects.

JSON supports 6 datatypes: number, string, boolean (true or false), object, array and null. Notice that JSON is text-based, which is essentially a string. It is a notation of data. It does not concern how the data would be stored and used in the program that takes JSON as the input file. For example, from JSON's point of view, it does not distinguish different types of number, being integer or float or double. In JSON, it is just a notation of number, say "108". It is the program's responsibility to decide how to treat the number, either as an integer, or as a 32-bit float, or a 64-bit float.

BSON, on the other hand, is binary-based, which is essentially a list of binary numbers. This makes BSON more efficient (but less flexible and more difficult to use sometimes) than JSON. In BSON, more datatypes are supported, including: double (64-bit float), string, object, array, binary data (a binary string), object id, boolean, date, null, regular expression, JavaScript code, 32-bit integer, 64-bit integer, timestamp, etc. When using BSON, the user needs to be more specific on how the data should be stored. For example, for a number "108", the user needs to specify whether to store it as a 32-bit integer or a 64-bit integer, or maybe as a 64-bit float.

Both JSON and BSON are intuitive and human-readable. MongoDB uses BSON due to its enhanced capability.

13.2.4 Create Collection and Document

A MongoDB database contains multiple collections. A collection is similar to a table in an RDB in the sense that it is the "host" of similar data. However, unlike tables where schematics such as column names and datatypes are enforced upon creation of the table, a collection does not enforce fields and data types. An example of creating a collection inside a database is given below.

```
> use testdb;
> db.createCollection("users");
```

Use `show collections` to show the collections in the current database.

Should I use semicolon in the end of each MongoDB command?

If you are using MongoDB shell, then technically speaking, you don't have to. It works both ways. However, if you are using JavaScript environment such as *Node.js* and integrating MongoDB commands as part of the program, you should use semicolon.

For example, to create a new database, in MongoDB shell

```
> use myNewDatabase
```

or

```
> use myNewDatabase;
```

would both work just fine. However, in *Node.js*,

```
const newDb = client.db("myNewDatabase");
```

the semicolon is required.

As a conclusion, semicolon is recommended mostly, just to follow the general JavaScript good practice. Although in JavaScript semicolon is also optional due to Automatic Semicolon Insertion (ASI), it is still widely recommended to use semicolon anyway.

Data can be installed into a collection. An entry to be inserted to a collection, corresponding with a row of a table in RDB, is called a document. It is possible to insert one or multiple entries at a time. To insert documents, first prepare the document in BSON format. For example, consider the following documents.

```
{
  name: "Alice",
  age: 20,
  address: "123 Center Park",
  hobbies: ["football", "reading"],
  parents: {
    father: "Chris",
    mother: "Kite"
  }
}

{
  name: "Bob",
  age: Long.fromNumber(25),
  address: "135 Center Park",
  hobbies: BSON.Array(["basketball", "jogging"])
```

```
}
```

Notice that user "Alice" and "Bob" are represented by JSON and BSON, respectively. MongoDB uses BSON internally, but it can also take JSON as input, in which case MongoDB driver converts JSON to BSON.

Then use the following syntax to insert the document into the collection.

```
db.<collection_name>.insertOne({...});
db.<collection_name>.insertMany([{...}, {...}, {...}]);
```

where `{...}` is the document in JSON or BSON format as shown earlier. To make it more readable, consider do the following instead.

```
const doc = {...};
db.<collection_name>.insertOne(doc);
```

which first store the document in `doc`, then pass it to `insertOne()` method. Upon successful insertion, an insert id (also known as object id) will be created automatically.

13.2.5 Query

MongoDB uses `find({...})`, `findOne({...})` to find documents, where `{...}` is a query argument in the form of JavaScript object. Details are given below.

To obtain all the document under a collection, use

```
db.getCollection('<collection_name>').find({});
```

where an empty query simply matches all documents in the collection. Of course, when `findOne()` is used, it will return only one document. When an object is given in the query, MongoDB returns only the documents containing the same fields.

Notice that

```
db.collection_name.some_function()
```

is equivalent with

```
db.getCollection('<collection_name>').find({});
```

The first implementation is more convenient while the second more flexible as it supports dynamic naming, i.e., something like

```
collectionName = 'some_collection';
db.getCollection(collectionName).some_function();
```

Several examples are given below. To find documents that has a certain field with certain values, use the following

```
db.getCollection('posts').find({comments: 1})
```

The above query search documents with field `comments` whose value is 1 under collection `posts`. To get those posts with at least 1 comment, use the following

TABLE 13.2
MongoDB basic query operators.

Operator	Description	Example
`$gt`	Greater than	`{age: {$gt: 18}}`
`$lt`	Less than	`{age: {$lt: 18}}`
`$and`	And	`{$and: [{age: {$gt: 18}}, {sex: 'M'}]}`
`$or`	Or	`{$or: [{age: 18}, {age: 21}]}`
`$in`	In	`{name: {$in: ['Alice', 'Bob']}}`
`$nin`	Not in	`{name: {$nin: ['Alice', 'Bob']}}`

```
db.getCollection('posts').find({comments: {$gt: 0}})
```

where `{$gt: 0}` stands for any value greater than 0. Commonly seen query operators are given in Table 13.2, each with an example.

13.2.6 Update and Remove Document

Two functions, `updateOne(...)` and `updateMany(...)`, are provided to update documents. Details are given below via an example.

```
db.getCollection('users').updateOne(
      {userId: '0015'},
      {$set: {age: 25}}
);
```

The above code query `users` collection, find the document with `userId` being `'0015'`, and change its `age` field to 25. From this example, it can be seen that the document updating function contains a query and a set of updating operators. Commonly seen updating operators are given in Table 13.3.

Use either `deleteOne({<query>})` or `deleteMany({<query>})` to remove documents.

13.2.7 Sharding and Indexing

Given that MongoDB is often used with massive data storage, efficient query from massive storage becomes critical. MongoDB implements a few technologies to speed up the query, including sharding and indexing.

Sharding is a method of distributing data across multiple servers or instances. In MongoDB, a shard consists of a subset of the total dataset, and each shard is responsible for managing a portion of the data. This distribution allows MongoDB to scale horizontally by adding more servers, thereby spreading the load and the data volume across a cluster. When a query is executed, it only needs to be processed by the shards that contain relevant data, rather than the entire dataset. This can significantly reduce query times in a large, distributed system. Sharding is particularly useful for very large

TABLE 13.3
MongoDB basic update operators..

Operator	Description & Example
$set	Replace field value {<query>}, {$set: {age: 30 } }
$unset	Remove field {<query>}, {$unset: {age: ""} }
$inc	Increment a field by a value {<query>}, {$inc: {age: 1} }
$rename	Rename field {<query>}, {$rename: {"age": "yearsOld"}}
$currentDate	Set field value to current date {{<query>}, {$currentDate: {lastModified: true}}}
$addToSet	Add element to array field {<query>}, {$addToSet: {hobbies: "reading"}}

datasets and high throughput operations, where a single server would not be sufficient to store the data or provide acceptable performance.

Indexing is a technique used to speed up the retrieval of documents within a database. MongoDB uses indexes to quickly locate data without having to scan every document in a collection. Indexes are a critical component of database optimization, as they can drastically reduce the amount of data MongoDB needs to look through to find documents that match a query.

MongoDB primarily uses B-Tree data structures for its indexes. A B-Tree is a self-balancing tree data structure that maintains sorted data in a way that allows searches, sequential access, insertions, and deletions in logarithmic time. The “B” in B-Tree stands for “balanced” and indicates that the tree is designed to keep the data balanced, ensuring that operations are efficient even as the dataset grows. The B-Tree structure allows MongoDB to perform efficient searches. Instead of scanning every document in a collection, MongoDB can use the B-Tree index to quickly navigate through a small subset of the data to find the documents that match the query criteria.

There are many ways to define the index. For example, it is possible to use a single field to form a single field index, or to use multiple fields to form a compound index. Index by itself is also an argumented data structure and it consumes disk space. Each time there is a write operation, the index needs to be updated. Therefore, a very complicated index schema slows down data insertion. It is critical to design appropriate index to optimize the overall performance of the database.

MongoDB provides commands to check, set and remove indexes. To check the indexes of a system, use

```
db.collection_name.getIndexes()
```

13.2.8 Other Features

MongoDB uses aggregation framework provides way to perform complex data transformations in a "pipeline". The data is processed step-by-step.

When a document is added to a collection, a default field `_id` is added. Querying via `_id` is usually faster than with other criteria because otherwise MongoDB has to search for the entire collection to find relevant documents.

Data in MongoDB can be exported or imported from files, such as BSON and JSON files.

MongoDB, like many other DBMS especially cloud-based enterprise tier ones, provide replica setting to tackle read-heavy applications. A primary server is replicated to multiple replicas. Writing is allowed only to the primary server, but reading is allowed to all replicas as well as the primary server. In the case of MongoDB, when primary server fails, one of the replicas is automatically promoted to be the primary server.

13.3 Non-RDB Example: MongoDB Atlas

MongoDB Atlas is the MongoDB cloud-based serverless solution. It allows the user to deploy MongoDB on the cloud. The user has the freedom to choose the base cloud service provider(s) including AWS, Azure and Google Cloud, and more. Thanks to Altas, the user has the flexibility to seamlessly change cloud service providers and service tiers without downtime.

In addition to just a host of the database, Atlas provides varieties of tools that helps the developer to develop applications using the database, such as a centralized cloud-based dashboard, autonomous synchronization with edge devices, etc. Some other examples include Atlas data lake which optimized for analytical queries. Big data can be stored in Atlas data lake, from where analytical information can be retrieved. Atlas federation allows seamlessly query, transform, and aggregate data from one or more MongoDB Atlas databases and cloud object storage offerings. Atlas chart provides rich tools for MongoDB data visualization. There are many more tools.

As of this writing, Atlas allows the user to choose from 3 different storage tiers:

- Shared: the database is stored on shared servers; the user can select the server provider from AWS, Azure and Google Cloud
  - M0: free, 512MB
  - M2: 2GB, $9 / month
  - M5: 5GB, $25 / month
- Dedicated: the database is stored on dedicated servers with dedicated stor-

age, RAM and multi-core CPUs; 10GB to 4TB; $0.08 / hour - $33.26 / hour

- Serverless: the database is provided as a microservice, and it automatically scales up and down based on use; up to 1TB; the cost depends on the amount of stored data, the number of read and write operations, the computational cost of backups, etc.

13.3.1 MongoDB Atlas Dashboard

To use MongoDB Atlas, register an account with MongoDB Atlas. T

Create a database cluster. Select the pricing tier for the database cluster, create a user with admin role and assign him a password, and configure the database cluster access gateway (i.e., a list of IP address that can access the database cluster). Upon creation of the database cluster, the user gains the access to MongoDB Atlas dashboard. The browser-based MongoDB Atlas dashboard can be used to view, add, edit and remove documents, collections and databases.

The newly created database cluster is emply and has no databases, collections or documents. For tutorial purpose, Altas provides a sample database cluster. One can import the data in the sample database cluster into the created empty database cluster. MongoDB Atlas dashboard provides data explorer tool that allows the user to view and edit the database directly, without using CLI or code. One can also filter documents from a collection by filtering the fields of the documents.

The following screenshot in Fig. 13.2 gives MongoDB Atlas dashboard. The database cluster, user and gateway can be viewed and configured from the dashboard.

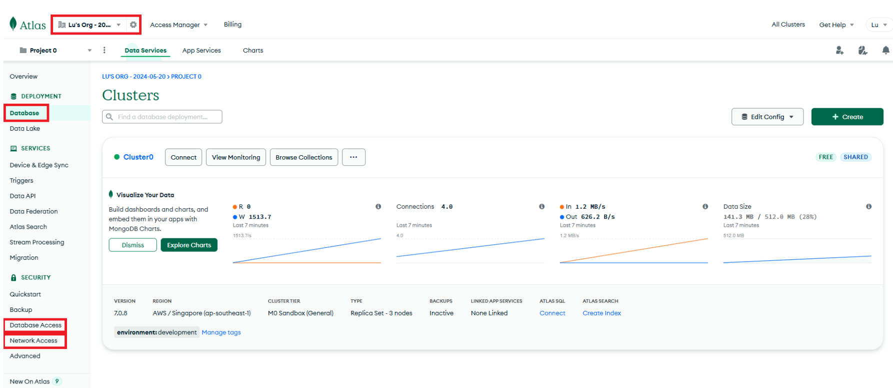


FIGURE 13.2
A demonstration of MongoDB Atlas dashboard.

Click "Browse Collections" of the cluster in Fig. 13.2. This sample database cluster contains multiple databases, each database with several collections, as shown by Fig. 13.3.

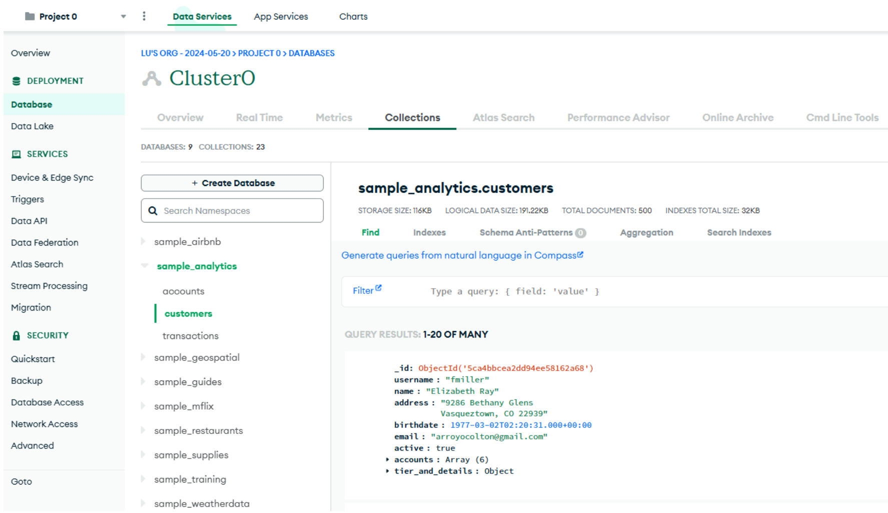


FIGURE 13.3
A demonstration of MongoDB Atlas dashboard where the databases and collections under "`Project 0`", "`Cluster0`" are browsed. Database "`sample_analytics`", collection "`customers`" is selected. There are 500 documents under this collection.

13.3.2 Connection String

Database connection string allows a client to connect to a database server such as a MongoDB Atlas database cluster. MongoDB Atlas provides two connection string formats, namely the standard format and the DNS seed list format. The connection string of MongoDB Atlas can be retrieved from the dashboard by clicking "Connect" in Fig. 13.2. It is possible to connect to MongoDB Atlas from:

- MongoDB Shell (MongoDB's CLI)
- MongoDB Compass (MongoDB's desktop client GUI)
- User applications

A connection string may look like the following

```
mongodb+srv://<username>:<password>@<server-dns>/?authSource=admin&
    replicaSet=myRepl
```

13.4 Non-RDB Example: Redis

Redis, short for *REmote DIctionary Server*, is an open-source in-memory distributed key-value database. It is often used as a lightweight database, cache tool, or message broker. Some key features are listed below.

- In-memory storage. This speeds up reading and writing operations.
- Persistence. While primarily it is an in-memory database, it also offers variety of ways to persist data on disk without compromising a lot on performance.
- Complex data structures and associated atomic operations. Though Redis is key-value store, it supports more complex data structures than that.
- High availability via replicas.
- Distributed storage via horizontal partitioning.
- Lightweight.

13.5 Non-RDB Example: AllegroGraph

Graph-based database is elsewhere introduced in details in notebook *A Notebook on Probability, Statistics and Data Science* under semantic web. Check that notebook for more details.

14

Virtualization and Containerization

CONTENTS

Virtualization and containerization are widely appreciated technologies for distributing multiple instances of applications on single or multiple physical servers. The primary objective of these technologies is to enhance resource utilization efficiency while ensuring isolation between applications.

14.1 Introduction

One of the major differences between a server and a PC is that the former is usually shared among multiple users or applications at the same time. Though working on the same machine, a user would usually want a private working environment not interrupted by other users. In other words, a user would want to "virtually" work on an independent machine with his own CPU, RAM, I/O, OS, drivers and storage, despite that the actual hardware is shared with others. This can be achieved through *virtualization*, which enables running multiple operating systems on a single physical server in an uninterrupted and logically separated manner. The virtually independent computer of such kind is often called a *virtual machine* (VM).

Deploying a new VM generally consumes a considerably large amount of time and resources. This is because different VMs on the same server are separated at the OS level, with each VM requiring its own OS installation. Consider a scenario where there are hundreds of small applications (microservices), each requiring a similar but separate environment. Launching VMs for each and every of them is resource-intensive, particularly when these applications could have shared the same OS kernel and operated within their own isolated workspace.

A more efficient approach would be to deploy a single VM and place each application in a "container" with its own customized drivers and configurations. A container is similar to a VM in the sense that it provides a degree of isolation from others, but it is typically "lighter" than a VM because it doesn't need to virtualize or duplicate the whole OS as VMs do. This makes containers cheaper to launch and manage.

The technology used to deploy and manage containers is known as *containerization*. A container contains all the configuration and requirement information of an application. Running a container on different platforms would consistently generate the same expected result. This has made the sharing and rapid deployment of containers remarkably easy and convenient. The similarities and differences of personal PCs, VMs, and container applications are summarized in Fig. 14.1.

As an analogy, think of running an APP as asking a kitchen to prepare a dish. The hardware is corresponding with the physical resources in the kitchen, such as the cooktop and gas. The OS is corresponding with the person who manages the kitchen, say a cook. The OS needs associated drivers and libraries to run the APP correctly. The drivers and libraries correspond with the skill sets or specific cookers for the dish. Finally, the APP is corresponding with the expected dish.

In the most simple configuration, a dedicated machine is used to run an APP. This is like constructing a dedicated kitchen and hiring a dedicated cook for each dish. The cook is trained to master all necessary skills required for that specific dish. This is shown in Fig. 14.2.

In a VM implementation, a larger and more capable kitchen is setup in advance as shown in Fig. 14.3. For each dish, a cook is hired. Each cook is trained with the skills necessary for his assigned dish. All cooks share the same kitchen. This implementation is more efficient than Fig. 14.2, as there is no need to scale up the kitchen for a new dish. By sharing the resources among the cooks, the kitchen can be utilized more efficiently.

While Fig. 14.3 might be a popular practice in many restaurants, it is still too costly to hire a new cook for each dish. In a containerization implementation, a cook usually handles a category of dishes, as shown in Fig. 14.4. Of course, each dish will stay in its own fry-pan in an isolated way. For each dish, its receipt is provided that gives all information required to prepare the dish

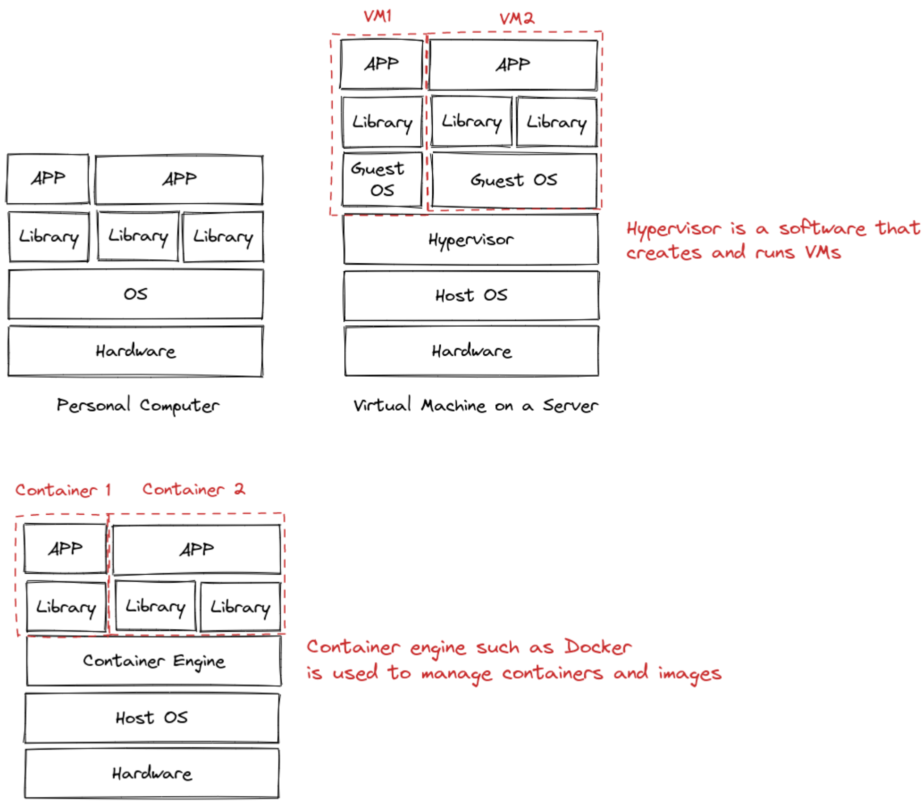


FIGURE 14.1
System architectures of PC, VM and container.

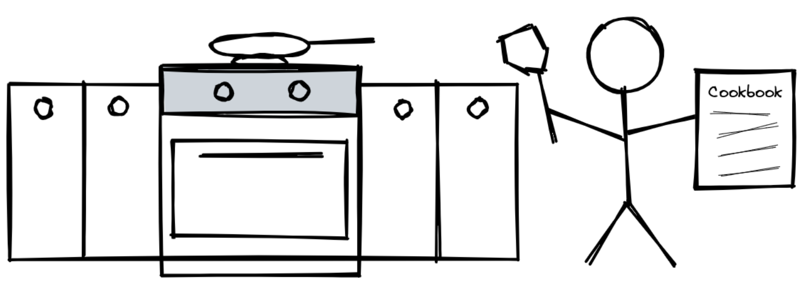


FIGURE 14.2
PC implementation: a cook in a kitchen.

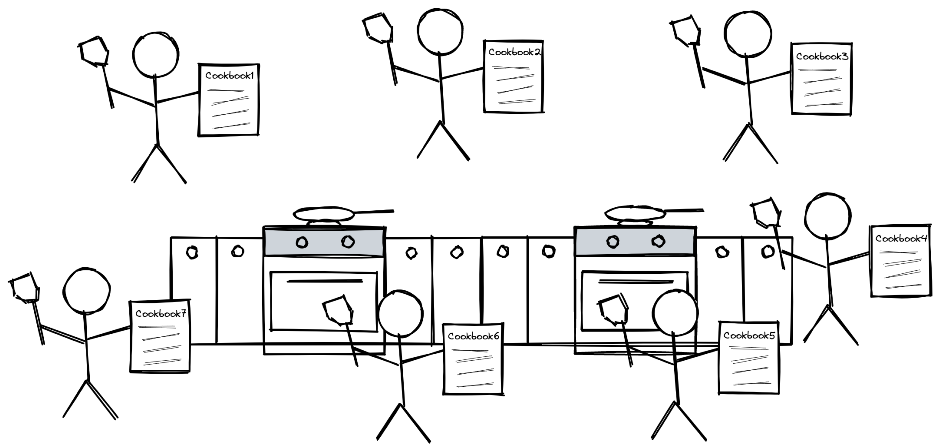


FIGURE 14.3
VM implementation: many cooks in a kitchen, each with a different cookbook.

consistently. As long as the cook is good at multi-tasking, Fig. 14.4 is a more efficient implementation than Fig. 14.2.

Just like a receipt guaranteeing the consistency of dishes, in containerization, an "image" guarantees the consistent performance of container instances. An image is basically a collection of prerequisites and configurations to run a container quickly, efficiently and consistently. Images can be shared among machines to replicate the containers even if the machines adopt different underlying infrastructure such as OS.

14.2 Virtualization and Containerization

Virtualization and containerization have been studied for decades. This section gives a brief introduction to both technologies.

14.2.1 Virtualization

In a conventional data center, multiple physical servers are deployed, and each server has a single associated application. Servers may share the same LAN and network-attached storage (NAS) for data exchanging. A big issue of this implementation is the utilization efficiency of the servers and the unevenly distributed loads: some of the servers may not be utilized efficiently, while others may be overwhelmed. To deploy a new application, a new server must

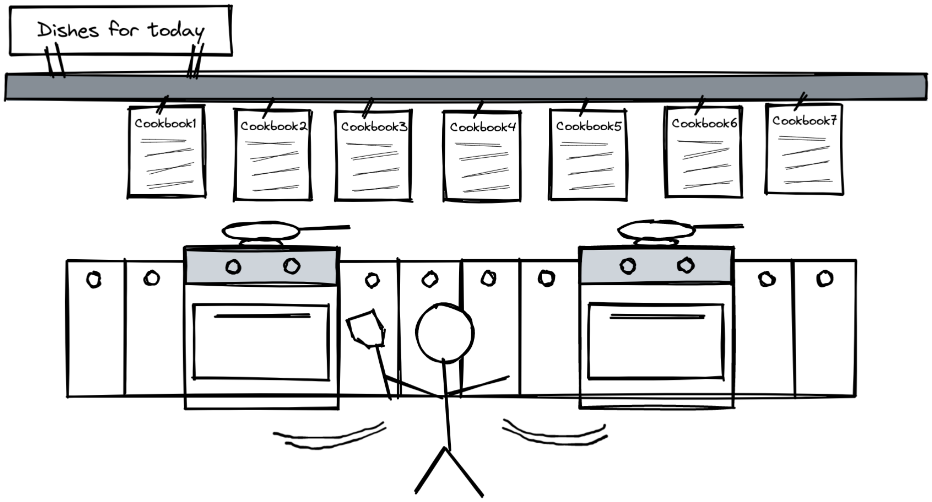


FIGURE 14.4
Container implementation: one cook in a kitchen, handling multiple dishes, each has a cookbook and stays in its own pan.

be purchased, which can take months of time. With more and more servers, the management and IT cost may grow exponentially.

To solve this problem, we need a systematical and automated way of integrating the resources in a data center and re-distributed them to the applications in an efficient manner. Virtualization is one of the most important technology used in this framework. Virtualization is essentially about running a system on a "virtualized machine", where by saying "virtualized" we mean that the machine is not a physical machine, but a running environment (known as the virtual execution environment) that is emulated and managed by a special software running on the actual machine (known as the "hypervisor"). The setup should be transparent to the applications, in the sense that the applications would think that they are running on a physical machine, not a VM.

Depending on the items to be virtualized, virtualization can be divided into the following categories.

- Virtualization of equipment, mainly network resources and storage. This results in virtual local area network (VLAN), virtual private network (VPN), NAS and storage area network (SAN).
- Virtualization of operating system. This results in the well-known VM and virtual desktop.
- Virtualization of software, mainly programming language and applica-

tions. An example is Java virtual machine (JVM) that allows Java to generate consistent result while running on different machines.

There are different virtualization architectures. Widely seen components in the virtualization architectures include

- Host OS. The OS that hosts everything else.
- Virtual Machine Monitor (VMM), also known as the Hypervisor. This is a software that runs on the host OS, managing all the guest OS, providing them interface to the host OS and the hardware.
- VM, a virtual system that runs in an virtualized isolated environment.

Host OS, VMM and VM relationship is shown in Fig. 14.2.

Different virtualization techniques are used in different types of VMMs. They can be widely divided into 3 categories.

- Full virtualization. The VMM virtualizes everything including the hardware. The guest OS can run on the VM without modification or adaptation, just like running on any other machine.
- Paravirtualization. The VMM does not virtualize hardware. The guest OS is modified to certain extent so that it can run on the virtual machine.
- Hardware-assisted virtualization. The physical processors provide some specific services to handle some of the functionality of VMM. Some functions requested by the guest OS, after passing to VMM, is directly run on the hardware, thus enhancing system performance.

VMM is able to virtualize hardware resources such as CPU, memory and I/O. For the virtualization of CPU, the key is to virtualize privileged instructions (a set of instructions that can only be executed by software running in privileged mode, usually the OS) for the guest OS, so that the guest OS would think it is directly talking to a CPU. This is challenging because guest OS usually does not possess the privilege due to the VM architecture. If the request of the guest OS contain privileged instructions, the CPU will deny the instructions. When that happens, an exception will be raised to the VMM which will take care of the privileged instructions sequentially. Since hypervisor runs in privileged mode, it can execute those instructions.

For the virtualization of memory, VMM uses shadow page table to assign memory pages to different VMs. The maximum memory allocated to a VM can change dynamically.

For the virtualization of I/O, the development trend is that I/O operations will be less and less rely on software and OS, and more and more on hardware. The VMM directly maps the I/O hardware interface to the VMs.

Hypervisor VS Host OS, Who is the Boss?

Both the hypervisor and the host OS can run in privilege mode (also known as “Ring 0” in x86 architecture). However, notice that at one time there can be only one boss, i.e., there can be only one entity that runs in Ring 0 at the same time.

Depending on the type of the hypervisor configuration, this entity can be either the hypervisor or the host OS. In a type 1 hypervisor configuration, the hypervisor itself runs in Ring 0 and manages all hardware resources directly. In a type 2 hypervisor configuration, the host OS runs in Ring 0, and the hypervisor runs in a lower privileged ring.

14.2.2 Containerization

Containerization is an alternative virtualization approach to VM that can also virtualize an independent workspace for an application. Comparing with VM, containers are often lighter, hence more efficient for massive microservices deployment. Depending on the context, the term “container” may refer to one or more of the following 3 components: container runtime, container engine, and container orchestration.

Container runtime refers to the backend software that actually runs the containers. Examples of widely used container runtimes are “containerd”, “runc” and “cri-o”. Container engine is the interface for a user or software to manage images and containers. Examples of container engines include “docker” and “podman”. Finally, container orchestration is the software that smartly deploy, monitor, restart, and terminate the containers on servers. It is often able to balance API calls, automatically scale up and down the number of running containers, and restart containers upon failure. One of the most widely used container orchestrations is “kubernetes”.

Container runtime is definitely vital in every containerization implementation. Container engines and container orchestrations must have container runtime built-in. However, the users rarely directly talk to the container runtime. Container engine is the interface and management tool of the container runtime, from where the operator can monitor and control the containers manually. Container engines are widely used especially in small projects or in the development environment. For massive deployment of containers in enterprise-level applications, container orchestration can become handy. In such a case, the container orchestration calls the container engine via API (nowadays a container orchestration may also talk to the container runtime directly), instructing it on how the containers should be deployed.

14.3 Docker Container

This section introduces docker. Notice that although docker is famous for docker container engine, docker as a company or community provides many revolutionary container related tools and services that go far beyond a container engine, many of which even used in its opponent container engines.

This section focuses mostly on the introduction of docker engine.

14.3.1 Docker Engine VS Alternatives

Docker engine is the most popular container engines available on the market as of 2023, and it is free of charge for open-source, personal and small business usage. More details of docker can be found at *https://docs.docker.com/*.

But does that mean docker engine is the absolutely best and perfect container engine solution?

Docker has surely revolutionized how we use containerization technology in software development and deployment, and it has been one of the most popular and beloved container engine solutions. However, it is worth mentioning that docker engine is not the only available container engine. For example, as explained earlier podman is an alternative to docker. It supports the same interface (as if podman is an alias) as docker in its CLI, and claims to have better performance and security. As a matter of fact, RHEL already started the transition from docker to podman from RHEL 8. Nowadays, installing docker on the latest versions of RHEL is possible but tedious. On the other hand, installation of podman (it might be built-in to the OS installation) on RHEL can be done simply by

```
$ sudo yum install podman
```

Kubernetes, a famous container orchestration, is also dropping docker support, as they say “docker support in the kubelet is now deprecated and will be removed in a future release”. Some may even argue that “containers are alive, but the role that docker plays is shrinking”. Many open-source initiatives such as podman are gaining popularity.

Docker has some disadvantages indeed. For one thing, docker uses docker server (docker daemon), a single piece of software running in the backend of the system, to support all the services. This creates a single-point-of-failure of the system. Docker requires root privileges, and it starts a container on behalf of the root user. This means that the program running inside the container and the users in the docker group can potentially bypass the OS access control and gain root access, which introduces security risk. These shortages are to some extent addressed by other container engines such as podman which is daemon-less and does not necessarily need to run on root user’s behalf.

This is not to say that Docker is falling behind as a whole. Some key techniques that Docker introduced are widely used in all different types and brands of container runtimes and engines. It is just that people do not like some of the features of Docker engine, and alternative tools are being developed to fix these problems, the latter of which starting drawing more and more attention. Docker still enjoys widespread usage and support due to its massive community, wealth of online resources, and extensive compatibility with numerous tools and platforms. Nevertheless, for demonstration purpose, for the remaining sections docker is used throughout this notebook. Since podman provides the same interface, it is probable that podman can be used likewise to replicate all the results.

As of 2023, Docker is still the dominating container market engine, with a market share of over 80%.

14.3.2 Docker Installation

To install Docker on a Linux machine, go to *https://www.docker.com/* to look for the instruction. The installation steps differ depending on the host machine. As an example, consider installing Docker engine on Ubuntu. Some of the key steps are summarized as follows.

Remove existing Docker engine, if any.

```
$ sudo apt-get remove docker docker-engine docker.io
$ sudo apt-get remove containerd runc
```

Add Docker's official GPG key and set up the repository.

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg lsb-release
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
    dearmor -o /etc/apt/keyrings/docker.gpg
$ echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/
      docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.
      list > /dev/null
```

Install Docker.

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
    compose-plugin
```

To test whether docker is installed correctly, run

```
$ sudo docker run hello-world
```

and if everything is done correctly, a message started with "Hello from Docker!" will be displayed in the console, together with a brief introduction to how docker works.

Notice that to use docker commands, sudo privilege is required. To avoid typing `sudo` each time running a docker command, add the user to the docker group as follows. In the rest of the section, `sudo` is neglected for docker commands.

```
$ sudo usermod <user name> -aG docker
```

Docker installs at least two piece of software on the machine, namely Docker CLI and docker server (also known as docker daemon). The CLI is the interface to the user, and the server is the actual tool that manages images and containers. Docker runs natively on Linux OS. If docker is installed on non-Linux system such as Windows or macOS, "docker desktop" is used, which includes a Linux VM to host the docker daemon and run Linux-based containers.

14.3.3 Docker Container Management

Container Manipulation

To create and run a container from an image, simply use

```
$ docker run <configuration> <image>
```

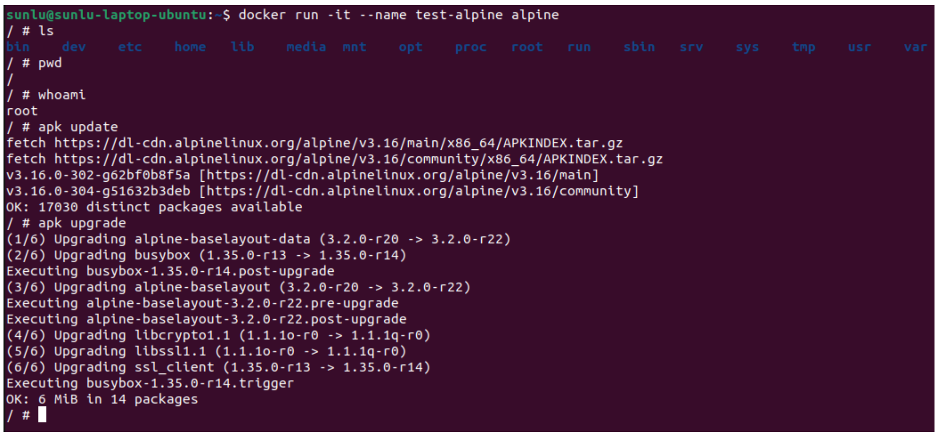
which is a combination of `docker create` and `docker start`. Docker will search the local and remote repositories for the image, download the image if necessary, and start a container from that image. By default, after successful execution and completion of all the tasks, the container will enter "Exited" status. For example, consider running a container of *alpine* as follows. A screen shot is given in Fig. 14.5.

```
$ docker run -it --name test-alpine alpine
```

where `-i` stands for "interactive", which keeps the container's standard input (i.e., the console in this example) open so that the user can actively interact with the container. Option `-t` allocates a pseudo-TTY to the container. TTY stands for "TeleTYpewriter", which enforces the I/O of the container following the typical terminal format and allows the user to interact with the container like a traditional terminal, hence making the interactive interface a bit more user-friendly. Finally, `--name` assigns a name to the container. Without an assigned name, docker will assign a random name to the container.

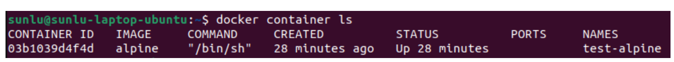
It can be seen from Fig. 14.5 that once the container is started, the user can interact with the container via shell, and perform actions such as listing items in the current directory in the container. While keeping the container running, open another terminal and use `docker container ls`. The container `test-alpine` shall appear in the list, as shown in Fig. 14.6. After exiting from Fig. 14.5 (by using `exit` in *alpine*), the container will transfer its status from "running" to "exited", as shown in Fig. 14.7.

As introduced earlier, `docker run` contains 2 steps in the backend, namely `docker create` and `docker start`, where `docker create` creates a container, with all its file system setup, and `docker start` starts the container



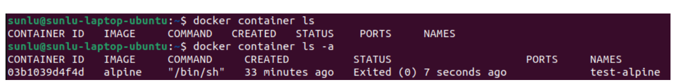
```
sunlu@sunlu-laptop-ubuntu:~$ docker run -it --name test-alpine alpine
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ # pwd
/
/ # whoami
root
/ # apk update
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/community/x86_64/APKINDEX.tar.gz
v3.16.0-302-g62bf0b8f5a [https://dl-cdn.alpinelinux.org/alpine/v3.16/main]
v3.16.0-304-g51632b3deb [https://dl-cdn.alpinelinux.org/alpine/v3.16/community]
OK: 17030 distinct packages available
/ # apk upgrade
(1/6) Upgrading alpine-baselayout-data (3.2.0-r20 -> 3.2.0-r22)
(2/6) Upgrading busybox (1.35.0-r13 -> 1.35.0-r14)
Executing busybox-1.35.0-r14.post-upgrade
(3/6) Upgrading alpine-baselayout (3.2.0-r20 -> 3.2.0-r22)
Executing alpine-baselayout-3.2.0-r22.pre-upgrade
Executing alpine-baselayout-3.2.0-r22.post-upgrade
(4/6) Upgrading libcrypto1.1 (1.1.1o-r0 -> 1.1.1q-r0)
(5/6) Upgrading libssl1.1 (1.1.1o-r0 -> 1.1.1q-r0)
(6/6) Upgrading ssl_client (1.35.0-r13 -> 1.35.0-r14)
Executing busybox-1.35.0-r14.trigger
OK: 6 MiB in 14 packages
/ #
```

FIGURE 14.5
An example of running *apline* container, with interactive TTY and name *test-apline*.



```
sunlu@sunlu-laptop-ubuntu:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS          PORTS     NAMES
03b1039d4f4d   alpine    "/bin/sh"   28 minutes ago   Up 28 minutes             test-alpine
```

FIGURE 14.6
List the running container *test-apline*.



```
sunlu@sunlu-laptop-ubuntu:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
sunlu@sunlu-laptop-ubuntu:~$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                     PORTS     NAMES
03b1039d4f4d   alpine    "/bin/sh"   33 minutes ago   Exited (0) 7 seconds ago             test-alpine
```

FIGURE 14.7
List the exited container *test-apline*.

by executing the startup script defined in the image. These two commands can be used separately. For example, to start an exited container, use

```
$ docker start <container id>
```

This command starts the container with the startup command, and keep it running in the backend. If `-a` is used in front of the container id, the container's IO would be attached to the terminal. If the startup command has been overwritten during the creation of the container, the new startup command will be used. The container id and the revised startup command can be obtained using `docker container ls -a`.

It is also possible to launch a container and let it run in the backend using `-d` flag which stands for "detached mode". An example is given below.

```
$ docker run -dt --name test-background-alpine alpine
```

By changing `-i` to `-d`, the container runs in the backend silently. The status of the container, after executing the above command, will stay running and can be displayed by `docker container ls`.

Commonly used commands regarding launching a container are given in Tables 14.1, and 14.2.

It is worth mentioning that file system snapshot and startup commands are defined inside an image. See Section 14.5 for more details. When a container starts, the file system snapshot is pasted to the container file system, and the startup commands executed. It is possible to overwrite the startup commands when starting a container, simply by amending the revised startup commands to the image name as follows.

```
$ docker run <image> <revised command>
```

Notice that the command needs to be defined in the image file system, otherwise an error will be raised.

For a container running in the backend, use `docker exec` to execute a shell command in that container as follows.

```
$ docker exec <container> <command>
```

To enable the TTY shell of a container running in the backend, use

```
$ docker exec -it <container> <command>
```

If `<command>` is replaced by the shell name used in the container, this would open the terminal of the container. Notice that the shell used by the application running inside the container may differ from the one used in the host machine. In the case of an *alpine* image based container, `ash` is the default shell. For a *ubuntu* image based container, `bash` is often used. To exit from the TTY shell while keep the container running in the backend, use shortcut key `Ctrl+p+q`. An alternative way to interact with containers running in the backend is to use

```
$ docker attach <container>
```

TABLE 14.1
Commonly used docker commands to launch a container.

Command	Flag	Description
`docker run`	—	Launch a container of the image. If the image cannot be found locally, it downloads the image from the remote repository automatically.
`docker run`	`-i`	Keep the standard input of the container open when launching the container.
`docker run`	`-d`	Launch the container in the backend and keep it running.
`docker run`	`--rm`	Automatically remove the container when exiting. The removed container will not be listed in `docker container ls -a`. This is usually used with temporary containers during debugging.
`docker run`	`-t`	Allocate a pseudo-TTY. The flag usually comes with the flags `-i` or `-d`, to form `-it` or `-dt`.
`docker run`	`--restart`	Enforce restart of the container upon exiting. This is usually used on containers running in the backend. Commonly used restart configurations include `--restart no` (do not restart), --restart on-failure[:<max retries>] (restart if exits with an error flag), `--restart always` (always restart when exists).
`docker run`	`--name`	Assign a name to the container.

TABLE 14.2
Commonly used docker commands to display local images and containers.

Command	Flag	Description
`docker image ls`	—	List local images.
`docker container ls`	—	List running containers.
`docker container ls`	`-a`	List all containers.

to attach local standard input, output, and error streams to a running container. Similar with the previously introduced `docker exec -it` command, `docker attach` also starts the shell of the application running in the container. Use `Ctrl-C` to quite the shell.

To stop, kill or restart a container running in the backend, use

```
$ docker stop <container>
$ docker kill <container>
$ docker restart <container>
```

respectively. The difference between `docker stop` and `docker kill` concerns with how the OS manages process. When `docker stop` is used, a `SIGTERM` signal is sent to the main process that the container runs. The container still has a little bit of time (maximum 10 seconds) to terminate the job and clean up. When `docker kill` is used, `SIGKILL` signal is sent to the process, and the process is terminated immediately. When a container is stopped, it enters exited status.

To remove an exited container or all exited containers, use

```
$ docker container rm <container>
```

or

```
$ docker container prune
```

respectively. Notice that there is a more powerful command

```
$ docker system prune
```

which removes not only stopped containers but also unused network configurations, dangling images and cache.

To rename a container (without changing its container ID or anything else), use

```
$ docker rename <container-old-name> <container-new-name>
```

Container Monitoring

Use

```
$ docker container ls
```

to check the list of running containers, and

```
$ docker container ls -a
```

the list of all containers, running or exited. Alternatively, `docker ps`, `docker ps -a` can also be used to list down containers just like `docker container ls`, `docker container ls -a`.

To check the processes that is running in the container, use

```
$ docker top <container id>
```

To quickly check container status including resource consumption (CPU, memory usage, etc.), use

```
$ docker stats [<container id>]
```

where the user can choose to list down all containers or a specified container. To show more detailed information of a container, including its status, gateway, IP address, etc., use

```
$ docker inspect <container id>
```

Finally, to check the logs of a container (e.g., its standard output to the console), use

```
$ docker logs <container id>
```

File Exchange between Container and Host Machine

There are multiple ways and protocols to access the files in a container, depending on the I/O setup of the container. For a container running locally, `docker cp` can be used for file transfer between the container and the host machine as follows. From container to host machine:

```
$ docker cp <container>:<source> <destination>
```

and from host machine to container:

```
$ docker cp <source> <container>:<destination>
```

where `<source>` and `<destination>` refer to the path to the source and destination, respectively, located in the host machine or the container.

Commit Container to Image

A container is usually generated from an image. It is also possible to do vise versa, i.e., packaging a container into an image. Notice that this is generally not recommended. Images shall be created mostly from Dockerfile, as will be introduced in Section 14.5.

To create an image from a container, use

```
$ docker commit <container> <image>
```

or

```
$ docker commit -c 'CMD ["<startup command>"]' <container> <image>
```

where `docker commit` command saves the container's file changes or settings into a new image, which allows easier populating containers or debugging in a later stage. Notice that `docker commit` does not save everything of the container into the image, and it is not the only way an image is created.

A container can be configured accessible from not only the host machine but also other computers in LAN or the Internet. In a typical web service application, the host machine and the containers are often designed following the architecture similar with Fig. 14.8. Notice that in practice, the load balancer and the containers may or may not run on the same physical machine.

The load balancer is a container orchestration that monitors the status of the containers, manages the data flow, and scales up and down the number of containers depending on the total load. In case where there are multiple physical servers, the load balancer is run on a "master" server, and the containers are distributed on multiple "slave" servers, each of which is called a "worker node", or "node" for simplicity. Container engines are installed on each and every node. The load balancer talks with the container engines on each node to deploy containers. The APP instances shall run in the containers distributively.

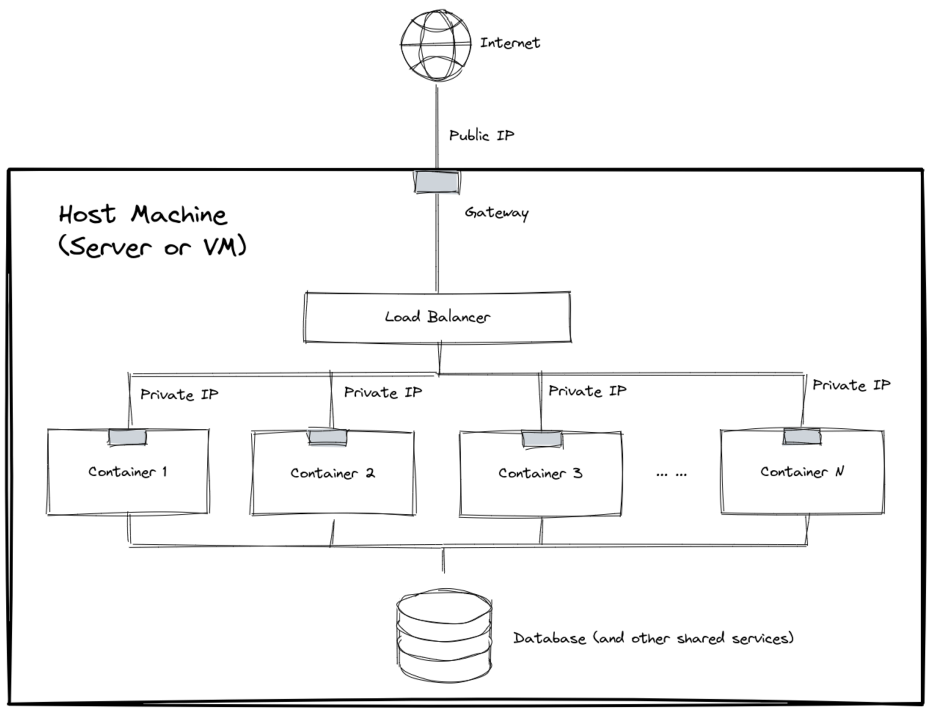


FIGURE 14.8
A simplified architecture where containers are used to host the web service.

An example of setting up a web server in containers from scratch is given in this section. For simplicity, everything happens on a single physical server. Only one container is used and the load balancer and the shared services are not included in the example.

As a first step, create a container from the official *nginx* image as follows. Notice that it is also possible to create a container from *apline*, and install *nginx* on *apline*.

```
$ docker run -dt --name simple-web nginx
```

Next, create the configuration file for *nginx*, and also the *html* files to be used as the static web page. For convenience, the files are created and edited in the host machine, then copied to the container. The following *default.conf* and *index.html* have been created, respectively. The configuration file *default.conf* is given below.

```
server {
       listen 80 default_server;
       listen [::]:80 default_server;
       root /var/www/html/;
}
```

The *html* file *index.html* is given below.

```
<html>
       <body>
              <h1>Hello World!</h1>
       </body>
</html>
```

Use `docker copy` to copy the two files to the designed locations in the container as follows.

```
$ docker exec simple-web mkdir -p /var/www/html
$ docker cp default.conf simple-web:/etc/nginx/conf.d/default.conf
$ docker cp index.html simple-web:/var/www/html/index.html
```

where `mkdir -p` creates the directories along the given path, if not exist. Notice that the file name in the destination can be ignored if it is the same with the source, i.e., the copy commands can be replaced by

```
$ docker cp default.conf simple-web:/etc/nginx/conf.d/
$ docker cp index.html simple-web:/var/www/html/
```

Change the ownership of the *html* file as follows, so that the current user *nginx* is able to access that file.

```
$ docker exec simple-web chown -R nginx:nginx /var/www/html
```

Finally, reload and configuration file and restart the web server as follows.

```
$ docker exec simple-web nginx -s reload
```

To test the web server running inside the container, obtain the IP address of the container using

```
$ docker inspect simple-web | grep IPAddress
```

and open a browser to key in the obtained IP address. If everything is done correctly, the browser should try to access port 80 of the container, and the "Hello World!" web page shall show up.

For easy sharing and populating of the container, commit the container into a new image using `docker commit` as follows. The new image can be used to populate the web server, just like "web01" container given below.

```
$ docker commit simple-web simple-web-image
$ docker run -dt --name web01 -p 80:80 simple-web-image
```

where `-p <host machine port>:<container port>` is used to map ports. Notice that different from the previous container "*simple-web*", the new container "*web01*" IP address port 80 is mapped with the port 80 of the host machine. Therefore, the web page hosted in "*web01*" can be accessed not only by the host machine, but also by other machines in the same network with the host machine.

14.4 Docker Volume

As introduced earlier, `docker cp` can be used to transfer data into and out of a container. An alternative way of accessing docker container data from the host machine is to use docker volume. Docker volume is used to mount host machine hard drive to container storage. Details are introduced below.

To create a docker volume, use

```
$ docker volume create <volume>
```

To list down volumes and to inspect a volume, use

```
$ docker volume ls
$ docker volume inspect <volume>
```

respectively. Finally to remove a volume or all volumes, use

```
$ docker volume rm <volume>
$ docker volume prune
```

respectively.

When starting a container from an image, volumes can be mapped with the internal storage inside the container by using `-v` flag as follows

```
$ docker run -v <volume>:<container-internal-path>[:ro] <image>
```

which should mount `<volumn>` to `<container internal path>`. The optional `:ro` can be specified if it is a read-only volume. Instead of using a volume name, the path to a directory in the host machine can also be used, in which case the specified directories in the host machine and in the container should be synchronized.

Docker volume guarantees data persistence in containerized applications. When a docker container is removed, any data written to the container's writable layer is lost. Docker volumes, however, are stored outside of the container's writable layer, allowing data to persist even after the container is removed. This persistence is particularly important for applications that require permanent storage, such as databases.

Moreover, Docker volumes can be shared among multiple containers, which facilitates data exchange and allows containers to work on the same dataset. Therefore, Docker volumes are not just a tool for data persistence, but also an effective mechanism for data sharing and collaboration among containers.

The data persists in the Docker host, and from the container's perspective, it is treated as a mounted volume. Hence, the data is not duplicated physically.

14.5 Docker Image

Container image related operations are introduced in this section.

14.5.1 Docker Image Creation from Dockerfile

In earlier Section 14.3, images were used to create containers. An image performs like a blueprint that encapsulates all the necessary information needed to spawn a container. It includes initial configurations, requisite libraries, and other pertinent metadata. Docker images are highly portable and can be shared across various machines and platforms. This section delves deeper into the construction and functionality of Docker images.

An image shall contain everything needed to create and initialize a container. This include but not limited to:

- Necessary steps (also known as "layers") to create a container.
- Application code that is to run in the containers
- Files to support the application
- Libraries, tools and dependencies

In addition, an image shall be designed and organized in such a way that it is migratable, reusable and light, and can be used to easily populate large number of containers. For better inheritability, an image might be based on another existing image, which is called its parent image. An image with no parent, such as the official *hello-world* image from Docker Hub, is called a base image.

The Dockerfile is a text document that serves as the blueprint for constructing a Docker image. Generally speaking, a Dockerfile follows the following flow:

(1) Specify base image.

(2) Add additional configurations.

   - Setup file system.

- Install dependencies and other programs.
- Setup working directory.
- Handle access control.
- ...

(3) Specify startup command.

A more detailed step-by-step guidance example is given later. The relationship between Dockerfile, image and container is illustrated in Fig. 14.9. Notably, the Dockerfile itself isn't included within the resulting image.

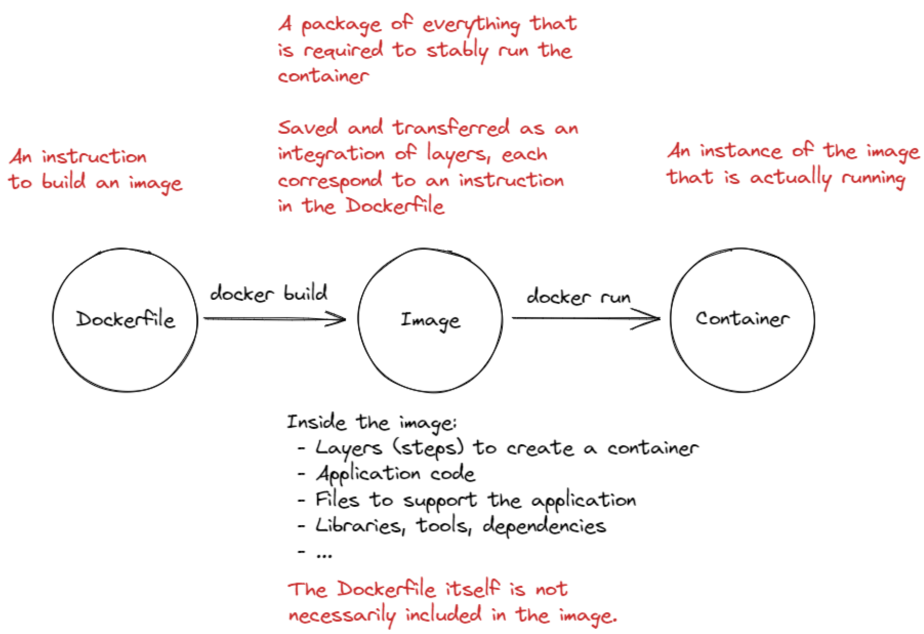


FIGURE 14.9
A demonstration of how Dockerfile, image and container link to each other.

Since a Docker image's primary role is to serve as a template for containers, many Dockerfile commands appear like a "step-by-step" recipe for container creation. Each instruction corresponds to an "image layer". An image is, in essence, an amalgamation of these layers. It is stored and distributed in this format. If images share layers (for instance, different versions of the same app), these shared layers aren't saved or transferred redundantly, resulting in significantly reduced image sizes. More details can be found at *https://docs.docker.com/storage/storagedriver/*.

Docker containers employ a special file system known as the Union File System (UFS), which is well suited to the "layer" concept. UFS facilitates file sharing between the container and the host machine, along with combining read-only upper layers and writable lower layers, among other functions. A demonstration to illustrate this concept is given later in Fig. 14.10.

Just as a quick example, the Dockerfile to build the official *hello-world* image from Docker Hub looks like the following.

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

Like other computer languages, Dockerfiles have reserved keywords, environment variables, and syntax rules. Only the basics of constructing a Dockerfile are introduced in this section. More details can be found in the Docker reference on the official website. In the example above, `FROM scratch` signifies that this image is a base image without a parent. The `COPY hello /` instruction copies the *hello* binary script from the image to the root directory of the container. Lastly, `CMD ["/hello"]` runs the *hello* binary script.

In general, a typical Dockerfile includes the following instructions to build an image. These instructions allow the image to know how to create a container, automatically construct the file system directory structure, install necessary packages, and run the app:

(1) Define parent image.

(2) Create filesystem directory.

(3) Set working directory.

(4) Copy files.

(5) Configure registry.

(6) Install packages.

(7) Copy more files after package installation.

(8) Switch to the correct user.

(9) Expose port.

(10) Run the APP.

The keywords to be used in a Dockerfile to realize the above instructions, such as `FROM`, `RUN`, and many more, are explained in Table 14.3.

Besides Table 14.3, there are other Dockerfile keywords that can significantly simplify the design and maintenance of the image. For example, `ENV <key>=<value>` assign a value to an environmental variable; `LABEL <key>="<value>"` assigns a tag to the image, which can be displayed when `docker inspect <container>` is used.

Examples of Dockerfiles are given below, one from *docs.docker.com* and the

TABLE 14.3
Critical keywords used in a Dockerfile.

Syntax	Description
FROM <image>	Define the parent image. A Dockerfile must start with a FROM instruction. A Dockerfile can contain multiple FROM instructions, in which case the last FROM statement is the final base image and the earlier FROM instructions creates intermediate images that can be used in the final image. An optional :<tag> following <image> can be used to specify the version of the image to use as the base. By default, the latest version of the image is used.
RUN <command>	Execute a shell command using /bin/sh -c.
WORKDIR <path>	Set the working directory from the point onward. This prepares the working directory for the upcoming RUN, COPY, etc., commands.
ADD <src> <dest>	Add (Copy) <src>, either a directory/-file or URL, to <dest>. An optional [--chown=<user>:<group>] can be used to specify the owner and group of the added files.
COPY <src> <dest>	Copy <src>, a directory/file, to <dest>. An optional [--chown=<user>:<group>] can be used to specify the owner and group of the added files. Notice that COPY is similar with ADD. COPY is easier but less powerful than ADD. It cannot handle tar or URL.
USER <user>	Switch user for the instructions beyond this point.
EXPOSE <port>	Specifies the ports that the container shall listen to. An optional /<protocol> following <port> can be used to specify the protocol for communication.
CMD ["<exe>", "p1", ...]	The user-script command. This is the last instruction in the docker file that usually starts the APP. Notice that a Dockerfile can only contain one CMD instruction. The executable command name and the parameters are put into a list. This user-script can be overwritten by docker run <image> <other commands> when running the container.

other from Linux Academy. The docker image layer structure of the second example is given in Fig. 14.10 as a demonstration. Notice that in Fig. 14.10, `bootfs` refers to the "boot file system", including the bootloader and the Linux kernel. Upon run, a container layer will be added to the image, as shown by the blue dashed box in Fig. 14.10. In the container, all the changes made is saved into the container layer.

To generate a new image to include the changes made in the container, use `docker commit`, which essentially commits the container layer as the latest image layer in the new image, as shown by the green dashed box in Fig. 14.10.

```
# First Example
FROM golang:1.16
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go ./
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]
```

```
# Second Example
FROM node:10-alpine
RUN mkdir -p /home/node/app/node_modules && chown -R node:node /home/
    node/app
WORKDIR /home/node/app
COPY package*.json ./
RUN npm config set registry http://registry.npmjs.org/
RUN npm install
COPY --chown=node:node . .
USER node
EXPOSE 8080
CMD ["node", "index.js"]
```

With the Dockerfile ready, use `docker build` to build an image. An example is given as follows.

```
$ docker build <path/url> -t <image name>
```

where `<path/url>` is the path or URL to the directory where the Dockerfile locates (does not need to contain "`/Dockerfile`" in its end), and `-t` gives a tag, in this case an image name, to the image to build.

14.5.2 Docker Image Management

The most commonly used image operations can be categorized as follows.

- Create an image.

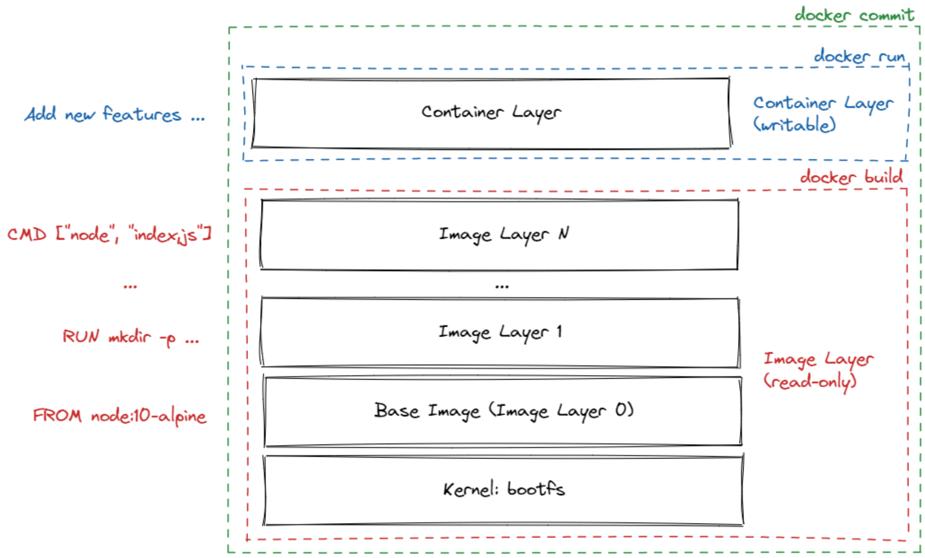


FIGURE 14.10
A demonstration of docker image layer structure using the aforementioned example.

- Create a container from an image.
- Upload and download an image from a remote server.
- Manage local images, such as listing down all images, deleting an image, etc.

The first two operations have been introduced in earlier sections. The third and last ones are introduced below. Use the following command to search for an image on the default remote repository server (Docker Hub).

```
$ docker search <image name>
```

Use the following command to download or update an image from the default remote repository server as follows. Notice that different from `docker run`, this command will not start a container from the image.

```
$ docker pull <image name>
```

Notice that since images are stored by layers, if two images share common layers, it is unnecessary to pull the shared layers repeatedly when downloading the second image, if the first image already exists in the host machine. Command `docker pull` is smart enough to automatically detect shared layers, and avoid duplicating download of layers.

Use the following commands to list down or remove images.

```
$ docker image ls
$ docker image rm <image name>
$ docker image prune # remove all problematic images
$ docker image prune -a # remove all unused images
```

where `prune` removes all problematic images, and `prune -a` removes all unused images from local.

Use the following command to inspect an image, and list down its metadata details.

```
$ docker image inspect <image name>
```

14.6 Docker Hub

Docker Hub is a commonly used server for storing and sharing docker images. It is also the default remote repository server of Docker engine. However, do notice that Docker Hub is not the only remote docker image server. Some alternatives are Amazon Elastic Container Registry, Red hat Quay, Azure Container Registry, Google Container Registry, etc.

After registering an account on Docker Hub, use the following command to login to the docker hub from your local machine.

```
$ docker login --username=<user name>
Passowrd:
```

Assume that there is an image in the local machine, and an empty repository on Docker Hub. In order to push the local image to the Docker Hub, the first step is to add the remote repository and the "*RepoTags*" in the local image as follows.

```
$ docker tag <image name> <user name>/<repository name>:<version
   >
```

where `<version>` is a tag usually used to distinguish the different branches or versions of the images on Docker Hub. For the first image upload, it can simply be `latest`.

Use the following command to push the image to Docker Hub.

```
$ docker push <user name>/<repository name>
```

14.7 Portainer

As introduced earlier, docker engine can be used to build and share images as well as start, monitor, and stop containers. It can be difficult for a user to manage containers manually when a lot of them are deployed. Container orchestrators such as Protainer and Kubernetes are helpful with managing containers. Many of these tools are able to automatically adjust the number of containers and balance their loads.

Portainer is an open-source container management tool. It has a web-based dashboard user interface. Notice that Portainer itself also runs in a container.

Before starting a Portainer container, it is a good practice to first create a Docker volume for Portainer to store the database. Use the following command to create such Docker volume.

```
$ docker volume create portainer_data
```

Then run a Portainer container using

```
$ docker run -d -p 8000:8000 -p 9000:9000 -p 9443:9443 --name portainer
     --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v
    portainer_data:/data portainer/portainer-ce
```

where ports 8000, 9000 and 9443 are used for hosting HTTP traffic in development environments, hosting web interface, and hosting HTTPS or SSL-secured services, respectively. The `docker.sock` is the socket that enables the docker server-side daemon to communicate with its command-line interface. The image name for Portainer community edition (distinguished form the business edition) is `portainer/portainer-ce`.

Use `https://localhost:9443` to login to the container. The following page in Fig. 14.11 should pop up in the first-time login, asking the user to create and administration user.

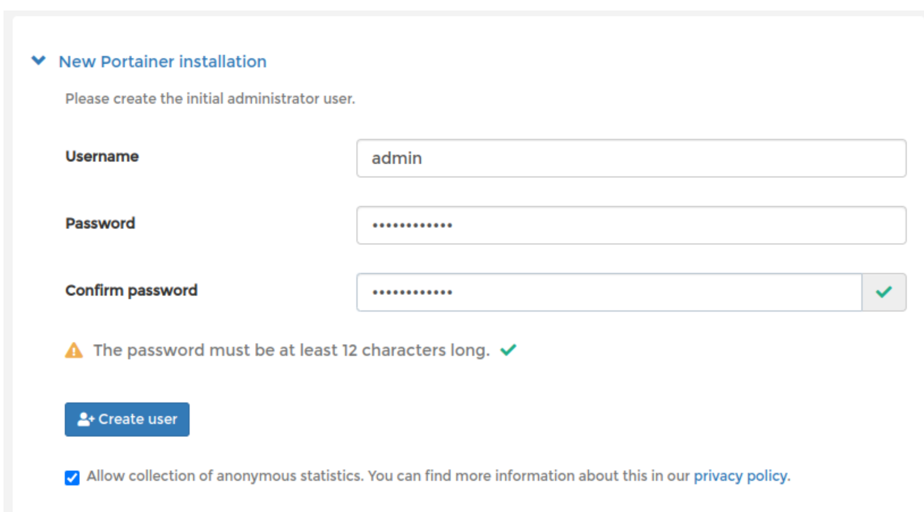


FIGURE 14.11
Portainer login page to create admin user.

After creating the admin user and logging in, the status of images, containers and many more can be monitored via the dashboard, as shown in Figs. 14.12, 14.13 and 14.14. Notice that in Fig. 14.14, using the "quick action" buttons, the user can check the specifics of the container and interact with its console, just like using `docker container inspect` and `docker exec`

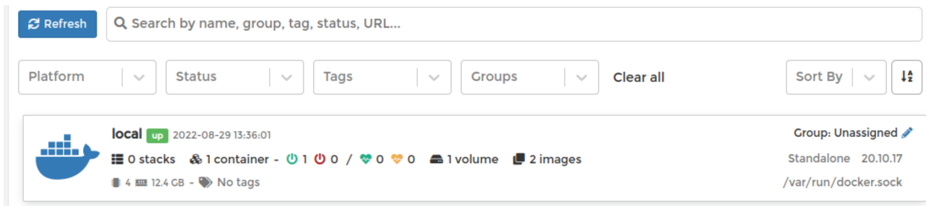


FIGURE 14.12
Portainer dashboard overview of docker servers.

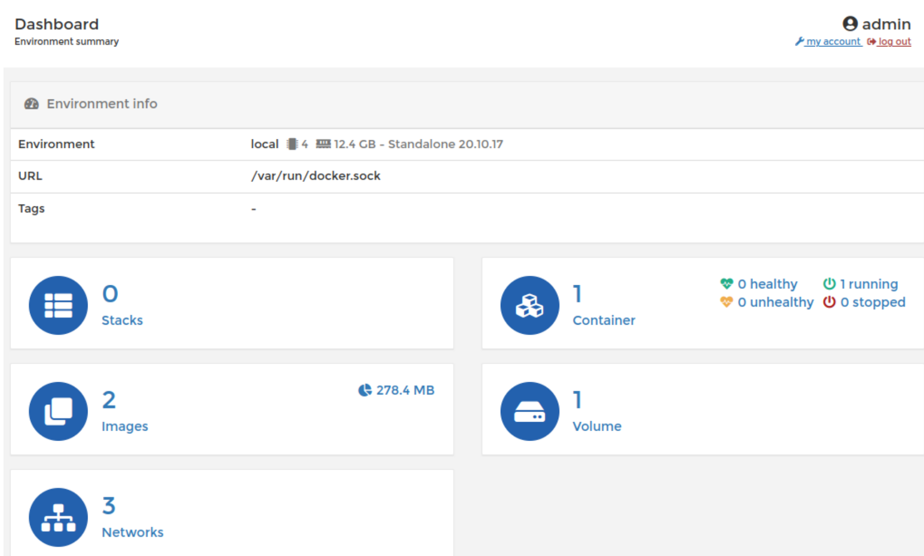


FIGURE 14.13
Portainer dashboard overview in a docker server.

In summary, Portainer is an easy-to-use container management tool with clean graphical interface that a user can quickly get used to without a steep learning curve.

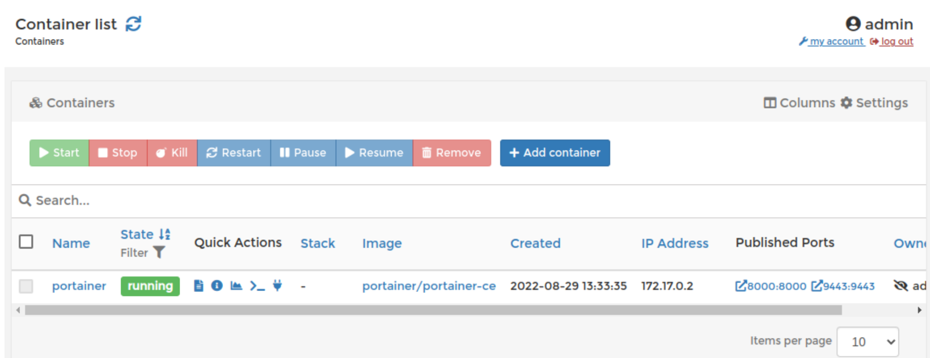


FIGURE 14.14
Portainer dashboard list down of all running containers.

15

Kubernetes

CONTENTS

Kubernetes is one of the most widely used container orchestration tools. Many cloud platforms provide Kubernetes support. Many opponent container orchestration tools are built on top of Kubernetes.

15.1 Basic Kubernetes

Kubernetes, also known as *k8s*, is an open-source container orchestration system originally developed by Google. It automates the deployment, scaling, and management of containerized applications. While Kubernetes is more flexible than Portainer, it's also more complex and has a steeper learning curve.

Notice that running Kubernetes on a local server differs significantly from running it on a cloud platform that often offers managed Kubernetes services with additional features and simplified managing interface. For example, AWS has Elastic Kubernetes Service (EKS) and Google cloud Google Kubernetes Engine (GKE). Both of them have developed their own tools and interfaces for interacting with Kubernetes. So, while learning Kubernetes can be beneficial, one might not need to learn all the low-level details if he is using one of the above platforms instead of DIY everything.

The majority part of this chapter focuses on introducing running Kubernetes on a local server, in which case *kubectl* is used as the user interface to Kubernetes and *minikube* the software to manage the host machine. Minikube is an open-source software developed by the Kubernetes community to run a single-node Kubernetes cluster on a local machine, which is suitable for developers to learn and test different things in development environment.

Notice that in the past days, docker is the default container engine built-in to Kubernetes, and Kubernetes uses a special program "dockershim" that talks to the docker engine. As of now, docker support is deprecated in Kubernetes and dockershim is removed from the installation.

15.1.1 Infrastructure

Figure 15.1 demonstrates the key components Kubernetes has inside its cluster. As shown in Fig. 15.1, Kubernetes manages containers in a centralized

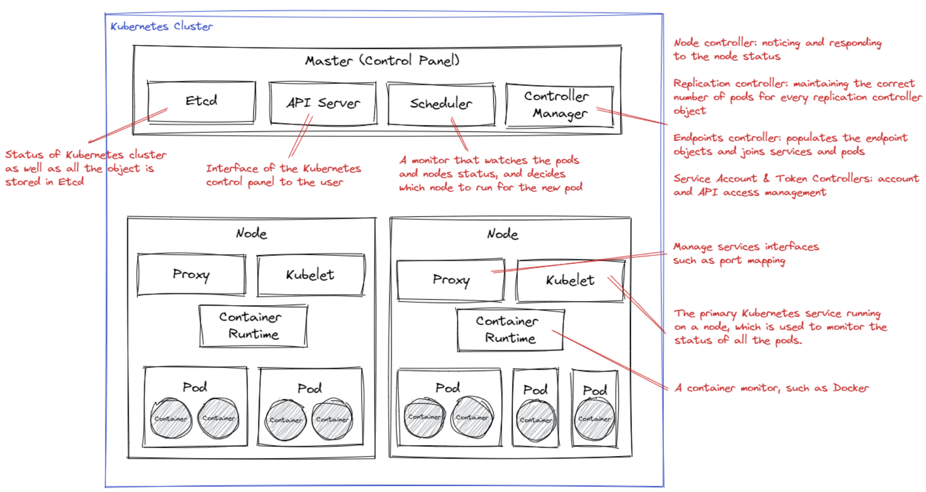


FIGURE 15.1
Kubernetes cluster and its key components.

"master-worker" manner, where the master plays as the control panel to interact with a user, and the worker nodes (nodes, for short) process the data. Each node can host multiple pods. Inside each pod is a container or a group of containers that work together closely (dependently, in many cases). Notice that in Kubernetes, containers never run directly in a node. They are always grouped into pods. A pod should be the minimal unit that can deliver a basic function.

The master and nodes can run on cross servers or VMs. Kubernetes packages need to be installed on each and every server or VM. Kubernetes provides variety of tools to distribute the loads to different servers or VMs, or to add redundancy to the system for high availability.

15.1.2 Installation

The installation guidance of Kubernetes can be found at its official website *kubernetes.io*. Notice that different OS adopts different ways of installing and using Kubernetes. The installation procedures introduced in this section applies to Linux OS only. For Windows users, Kubernetes can be installed from Docker desktop. For macOS users, other tools are used to install the tools.

As introduced earlier, *kubectl* is used to interact with Kubernetes. In addition, since we are in development environment, we will also install *minikube* which is used to setup a small Kubernetes cluster in the local machine. They can be installed separately. See following links for more details.

```
https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/
https://minikube.sigs.k8s.io/docs/start/
```

and do not forget to start *minikube* using the following command. Notice that when both *minikube* and *kubectl* are installed, *minikube* needs to be started before *kubectl* can work properly.

```
$ minikube start
```

Upon the start of *minikube*, use the following command to verify the successful running of *kubectl*.

```
$ kubectl cluster-info
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-
    system/services/kube-dns:dns/proxy
```

Notice that when starting *minikube*, it would launch a VM using Vertual Box, and install *kubectl* as a built-in. Therefore, it is probable to use that built-in *kubectl* instead of manually installing one separately. In that case, use the following command to verify the successful running of *kubectl*

```
$ minikube kubectl cluster-info
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-
    system/services/kube-dns:dns/proxy
```

in which case `alias kubectl="minikube kubectl --"` may make things easier.

It is possible to run Kubernetes cluster directly on a host machine OS without VM if that machine is running Linux. However, this is not recommended for reasons pertaining to access control, security, and isolation. It is generally a better practice to deploy Kubernetes clusters in a VM.

The `kubectl` command is a command-line interface (CLI) for interacting with Kubernetes clusters. Its behavior is governed by a configuration file, typically located at `~/.kube/config`, which determines which Kubernetes cluster `kubectl` communicates with. This could be a cluster running on the host machine, or a cluster running in a VM managed by tools like Minikube.

15.1.3 Kubernetes Configuration Files

Kubernetes requires that all images to be used are pre-built. When using Kubernetes, multiple configuration files are required, each file corresponding with an object to be created. Notice that an object is not necessarily a container. It can be a pod, a replica controller, a service, or any item in the Kubernetes framework. There are manual setups of networking. Details are introduced in the remaining of this section.

The following two configuration files are given as examples to demonstrate how the configuration files of Kubernetes look like. This example comes from Udemy course *Docker and Kubernetes: The Complete Guide* by Stephen Grinder [2]. They are both written in YAML. Configuration file to setup a pod:

```
apiVersion: v1
kind: Pod
metadata:
name: client-pod
labels:
component: web
spec:
containers:
- name: client
image: <image-name>
ports:
- containerPort: 3000
```

Configuration file to setup the networking service:

```
apiVersion: v1
kind: Service
metadata:
name: client-node-port
spec:
type: NodePort
ports:
- port: 3050
targetPort: 3000
```

```
nodePort: 31515
selector:
component: web
```

Commonly used Kubernetes object types are summarized in Table 15.1. Some highlights of the above configuration files are as follows.

- `apiVersion` plays as the prefix that decides what configuration types are supported. For example, under the scope of `v1`, `Pod`, `Service`, `configMap`, `Namespace`, etc., are supported. In a different `apps/v1`, a different set of configuration types `ControllerRevision`, `StatefulSet`, etc., are supported. It's important to choose the correct apiVersion for the Kubernetes API version to ensure the compatibility and availability of the desired configuration options.

- `kind` This indicates the type of the object that the configuration file describes. For example, `Pod` represents a pod that is used to host containers, and `Service` the primary object type that defines networking, with subtypes `NodePort` (see example above), `ClusterIP`, `LoadBalancer` and `Ingress`.

- `metadata` indicates the name and labels of the object. For example, `component: web` is defined as a label of the pod. This information is passed to the networking service under `selector`, so that the networking service knows which object it should link to.

- `port`, `targetPort` and `nodePort` are used to specify ports used in the service. The `targetPort` indicates which port in the pod should be exposed to the service, and it is consistent with `containerPort` defined in the pod. Assume that there is another pod in the node who needs to talk to this pod via the service. The other pod's port that communicates with the service is fed into `port`. Finally, `nodePort` is the port with value between 30000 and 32767 that is exposed from the service to outside the node. If `nodePort` is not assigned, a random number within the range will be assigned. This is shown in Fig. 15.2.

Notice that Kubernetes networking using `kind: Service` is more complicated than shown in the above example. More details of it is given later in a dedicated Section 15.2.2.

15.1.4 Cluster Deployment

With the image and the configuration files ready, the next step is to deploy the nodes, pods, and containers. *kubectl* command line interface is used to instruct Kubernetes to deploy the objects as follows.

```
$ kubectl apply -f <configuration file>
```

TABLE 15.1
Commonly used Kubernetes object types.

Object Type	Description
Pod	The smallest and most basic unit in the Kubernetes object model. It represents a single instance of a process running on the cluster.
Deployment	Manages the deployment and scaling of a set of identical pods, ensuring the desired number of replicas are running and providing rolling updates for seamless application upgrades.
Service	Enables network access to a set of pods using a stable IP address and DNS name. It provides load balancing across multiple pod replicas and allows external traffic to be directed to the appropriate pods.
ConfigMap	Stores configuration data in key-value pairs, which can be consumed by pods as environment variables, command-line arguments, or mounted as files.
Secret	Similar to a ConfigMap, but specifically designed to store sensitive data, such as passwords, API keys, and TLS certificates. Secrets are encrypted at rest and can be mounted into pods as files or exposed as environment variables.
PersistentVolume	Provides a way to provision and manage persistent storage resources in a cluster. It decouples the storage from the underlying infrastructure and allows data to persist beyond the lifecycle of individual pods.
PersistentVolumeClaim	Requests a specific amount of storage from a PersistentVolume. It acts as a request for a specific storage resource and provides an abstraction layer for managing persistent storage in a cluster.
Ingress	Manages external access to services within a cluster. It acts as a reverse proxy and exposes HTTP and HTTPS routes to route traffic to the appropriate services based on hostnames, paths, or other rules.

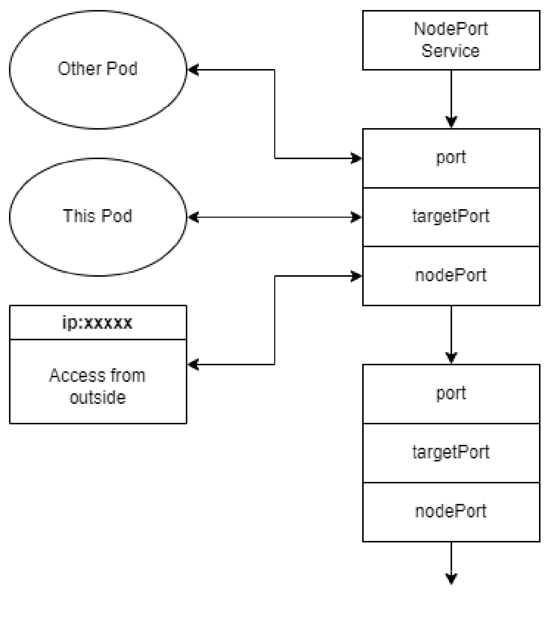


FIGURE 15.2
The NodePort networking service.

This essentially asks the master node in the Kubernetes cluster to start taking actions according to the configuration files, such as to inform the nodes to start creating pods and containers. The master node also keeps monitoring the status of each work node, to make sure that everything is running as planned. If there is a container failure, etc., the master node will guide the associated note to restart the container.

It is worth mentioning here that by default Kubernetes uses declarative deployment instead of imperative deployment, meaning that the developer does not need to specifically tell Kubernetes what to do in each step. The developer only tells the overall objectives, and Kubernetes master node will try to figure out the steps to realize that goal. It is possible to enforce Kubernetes master node to practice specific details via configuration files, but it is almost always recommended to use the default declarative approach with Kubernetes.

To retrieve information, such as the status, of a group of objects, use

```
$ kubectl get <object type>
```

where `<object type>` can be `pods`, `services`, etc. For more details of a specific object, use

```
$ kubectl describe <object type> <object name>
```

for example, to check the containers running in a pod. If `<object name>` is neglected, Kubernetes returns detailed information of all objects of the given object type. For a running object, use

```
$ kubectl logs <object name>
```

to check the log file of that object.

With the above been done, open a browser and use `<ip>:<port>` to access the application running in the container, where `<ip>` is the IP address of the VM (not `localhost`) that *minikube* created, and `<port>` the port configured in NodePort service under `nodePort`. The IP address can be found by running `minikube ip`.

To apply a group of configuration files all together, provide the directory name of all the configuration files to Kubernetes instead of feeding each configuration file one at a time.

```
$ kubectl apply -f <directory>
```

When directory is given instead of a file, Kubernetes will try to apply all the configuration files in that directory.

It is possible to consolidate the configuration files of objects into one conjunctive configuration file. To do that, use `---` to split the configurations for each object in the conjunctive configuration file as follows. It is of personal preference whether to use conjunctive configuration files or separate configuration files for all objects.

```
<configurations-for-object-1>
---
```

```
<configurations-for-object-2>
---
<...>
```

15.1.5 Cluster Update

Without container orchestration tools such as Kubernetes, one of the most challenging tasks is to update the container for a different configuration, for example, changing the underlying image. With the help of Kubernetes declarative approach, it is possible update the cluster simply by revising the configuration files, and pass them to Kubernetes as if the cluster is to be deployed for the first time. Kubernetes automatically checks the names and kinds of the revised configuration files, comparing them with existing running objects, and update them if necessary.

Check the status of the pods using `kubectl get pods`. After updating, the pods are often restarted, hence it is expected to see increment in the "RESTARTS" tag. To double confirm that updates have been made, use `kubectl describe` to check the details of the relevant objects.

However, there is a limitation to the updating of the Kubernetes deployment. For an existing object, only certain fields in the configuration files can be changed. For example, for a pod that runs containers, the image can be changed, but the container port cannot. Sometimes there can be a walk around. For example, in the case of changing container port of pods, consider using a new object type `Deployment` instead of `Pod`, which allows more flexible updating. The `Deployment` in its backend is consist of one or more monitored and managed identical pods.

To revert `kubectl apply`, i.e., to remove a configuration file, use

```
$ kubectl delete -f <configuration file>
```

Kubernetes treats the above delete command as a specific type of update to the cluster, and will action accordingly.

15.2 Advanced Kubernetes

This section introduces advanced commonly used Kubernetes objects, tools and techniques.

15.2.1 Kubernetes Object: Deployment

As introduced in Section 15.1.5, updating pods has some limitations. It is practically more convenient to setup pods using "Deployment" object instead of "pod". The Deployment object servers as an additional layer of Kubernetes infrastructure that manages identical pods. More details of Deployment object is introduced in this section.

As an example, here is a configuration file from Kubernetes manual that deploys a Deployment object.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
labels:
app: nginx
spec:
replicas: 3
selector:
matchLabels:
app: nginx
template:
metadata:
labels:
app: nginx
spec:
containers:
- name: nginx
image: nginx
ports:
- containerPort: 80
```

Some highlights are as follows.

- `replicas` gives the expected number of pods that the Deployment object manages.
- `matchLabels` specifies the pods with which label are to be managed by the Deployment object. In this example, the label is `app: nginx`. When populating pods, the pods would have the same label, as the same label is assigned under `template`, `metadata`, `labels`.
- `template` specifies the template that is used to create the pods.

When a new version of an image becomes available, we may want to update the containers accordingly. Re-apply the same configuration file would not help, as Kubernetes would reject apply request if no change is detected in the configuration file. It would not check whether the image is in its latest version. Kubernetes uses the following imperial command to update images as a walk around, and the developer needs to run this command manually.

```
$ kubectl set image deployment/<Deployment name> <container name>=<
    image name>
```

For example,

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.25.1
```

15.2.2 Kubernetes Object: Service

There are 4 service types defined in Kubernetes. So far "NodePort" service type has been introduced in earlier examples. More types are introduced here.

A summary of different service types are given below.

- ClusterIP: Exposes the service object on a cluster-internal IP. Objects in the cluster can access to the object that a ClusterIP is pointing at.
- NodePort: Assigns a static port with the cluster IP, and exposes the Service object to the internet. This is used mostly used in development environment, not in production environment.
- LoadBalancer: Exposes the service object externally using an external load balancer. Kubernetes does not provide built-in load balancer.
- ExternalName: Maps the service object to the contents of the `externalName` field, such as a host name. This is related to cluster DNS server.

ClusterIP

A ClusterIP configuration file may look like the following.

```
apiVersion: v1
kind: Service
metadata:
name: client-cluster-ip-service
spec:
type: ClusterIP
selector:
<target tag>
ports:
- port: <port for internal comm>
targetPort: <port for internal comm>
```

where the first `port for internal comm` is the port in the ClusterIP service object that opens to other objects in the cluster, and the second the port the target object opens to the ClusterIP service object. They can be set differently, but usually they are just set to the same value.

NodePort

NodePort has already been introduced earlier. It exposes the object to the internet with a static port, and it is used more in a development environment than a production environment.

LoadBalancer

LoadBalancer is a legacy way of getting network traffic into the pods. It is essentially an interface or a tool to bridge an Kubernetes Deployment with an external load balancer. It will try to automatically configure the external load balancer using the configuration provided by the developer, while compromising the external load balancer rule.

Ingress

Ingress is a more commonly used Service type than LoadBalancer to get traffic into the Kubernetes containers. There are different types of Ingress, for example, Nginx Ingress by *github.com/kubernetes/ingress-nginx*. An demonstrative example of ingress service realization is given in Fig. 15.3. In this implementation framework, the configuration file (mainly a set of routing rules) of the object is used to define an "Ingress Controller" which manages the runtime that controls inbound traffic. In some applications such as *kubernetes/ingress-nginx*, the ingress controller and the runtime are integrated together.

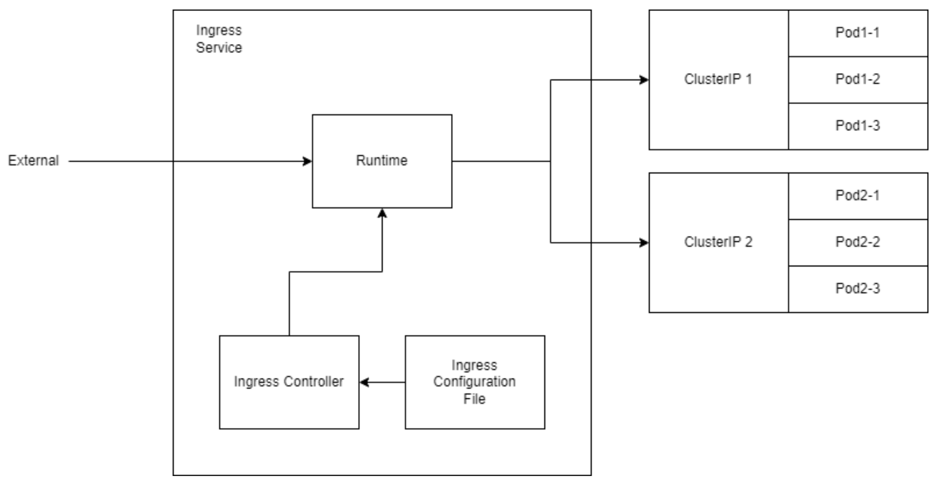


FIGURE 15.3
An example of ingress service framework.

The ingress configuration differs depending on the ingress service type and the platform. Details are not given here.

15.2.3 Kubernetes Object: Persistent Volume Claim, Persistent Volume, and Volume

Docker engine uses volumes to maintain persistent data and share data among containers. Details have been introduced in Section 14.4. Kubernetes volume framework is similar in the sense that it makes sure that the data is saved and managed by the host machine, so that when the pods or containers are shutdown or restarted, the data can be restored safely.

Do notice that when comes to data sharing using volume, it is dangerous to have multiple containers or the host machine accessing the same files simultaneously, without knowing the existence of each other. Usually additional steps need to be setup to ensure data consistency.

It is worth emphasizing the differences of "volume" technology in containerization and Kubernetes volume-related objects: Persistent Volume Claim (PVC). Persistent Volume (PV), and Volume. As a matter of fact, Kubernetes Volume object is usually not what we want. Kubernetes Volume creates the volume tied to a pod, not to the host machine. It survives container failure in the pod, but not pod failure. In summary:

- Kubernetes Volume: A volume tied to the pod. It survives container failure inside the pod, but not pod failure.
- Kubernetes PV: A volume tied to the host machine. It survives pod failure. It can be provisioned either automatically by a StorageClass, or manually by the developer and administrator.
- Kubernetes PVC: It is essentially a request sent from a pod or a container, asking for specific amount of storage from a PV. Kubernetes will find that amount of PV from either existing provisioned static PV, or dynamically provision new ones for the pod or container.

Notice that it is not necessary to claim PV in order to use PVC, as PVC can provision PV dynamically. There is a one-to-one relationship between the provisioned PV and the PVC. If there are multiple pods, each requiring a dedicated PV, then multiple PVCs must be used. The developer can either create those PVCs manually, or use a Volume Claim Template to claim them if they are similar.

An example of claiming Kubernetes PV and PVC is given below. In the remaining part of this section, we will be mostly using PVC instead of PV.

```
# persistent-volume.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
name: my-pv
spec:
storageClassName: standard
capacity:
storage: 10Gi
accessModes:
- ReadWriteOnce
hostPath:
path: /data/my-pv
---
# persistent-volume-claim.yaml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: my-pvc
spec:
storageClassName: standard
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 5Gi
```

To check the PV objects, use `$ kubectl get pv` and `$ kubectl get pvc`.

To add the above Kubernetes PVC to a Kubernetes Deployment, add volumes information to the specs as given in the following example.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: my-app-deployment
spec:
replicas: 1
selector:
matchLabels:
app: my-app
template:
metadata:
labels:
app: my-app
spec:
containers:
- name: my-app-container
image: my-app-image
ports:
- containerPort: 8080
volumeMounts:
- name: data-volume
mountPath: /data
subPath: data-from-container
volumes:
- name: data-volume
persistentVolumeClaim:
claimName: my-pvc
```

where `volumes` defines which Kubernetes PVC is used, and `volumeMounts` tells how it is mounted in the container. The `mountPath` is the path in the container whose data is mounted by the volume. If a `subPath` is given, a sub-folder of its specified name will be created in the host machine in the volume to contain the data.

There are different types of access modes:

- `ReadWriteOnce`: Allow one node to read and write at a time.
- `ReadOnlyMany`: Allow many nodes to read at a time.
- `ReadWriteMany`: Allow many nodes to read and write at a time.

The developer can specify the place for Kubernetes PVs. This is usually the hard drive on a local server, a virtual storage space in the VM. Use the following command to check Kubernetes possible choice of storage.

```
$ kubectl get storageclass
```

and

```
$ kubectl describe storageclass
```

When deploying Kubernetes on the Cloud, the developer needs to do additional configurations as there would be many storage options. Usually, each Cloud provider will have its own default storage space for Kubernetes, such as AWS Elastic Block Store for AWS.

15.2.4 Kubernetes Object: Secrets

Kubernetes Secrets object is used store confidential information, such as the database password, API key, etc. It is often a piece of information that is necessary for the containers, but the developer does not want to present as plain text in the configuration file.

Secrets are not created from configuration files, which is the recommended way of creating other Kubernetes objects. Instead, it is created from one-time imperative command, inside which the confidential information needs to be told to Kubernetes. Use the following command to create a Secret object.

```
$ kubectl create secret <type-of-secret> <secret-name> --from-
    literal <key>=<value>
```

There are 3 types of Secrets: `generic`, `docker-registry` and `tls`.

15.2.5 Kubernetes Environment Variables

Kubernetes environment variables are used to pass or share information among Deployments. Depending on the features of the information, such as whether it is a constant global configuration or a dynamic value, whether it is plain text or confidential encoding, etc., it might be handled differently.

To define constant environment variables in containers, simply specify them in the Deployment configuration file as given in the example below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: my-app-deployment
```

```
spec:
replicas: 1
selector:
matchLabels:
app: my-app
template:
metadata:
labels:
app: my-app
spec:
containers:
- name: my-app-container
image: my-app-image
ports:
- containerPort: 8080
volumeMounts:
- name: data-volume
mountPath: /data
subPath: data-from-container
env:
- name: <name1>
value: <value1>
- name: <name2>
value: <value2>
- name: <name3>
valueFrom:
secretKeyRef:
name: <secret-name>
key: <key>
volumes:
- name: data-volume
persistentVolumeClaim:
claimName: my-pvc
```

where a new tag `env` is defined under template, specifications, containers. Under the `env` tag, a list is defined containing names and values of the environment variables. The value must be a string, not a numerical number.

15.3 Container Deployment in Production Environment

With the tools and methodologies introduced so far, we are able to deploy containers in development environment. This is good enough for testing purpose or for small individual projects. However, when comes to enterprise tier projects or collaborative projects, there is often a CI/CD pipeline that standardize the integration and delivery of the containers in production environment. Container orchestration tools such as Kubernetes is often a must have.

This section briefly introduces the steps to develop and deploy containers in production environment with Kubernetes. Figure 15.4 gives an example of overview of what such deployment may look like. Notice that this example is more towards a community project but not an enterprise project.

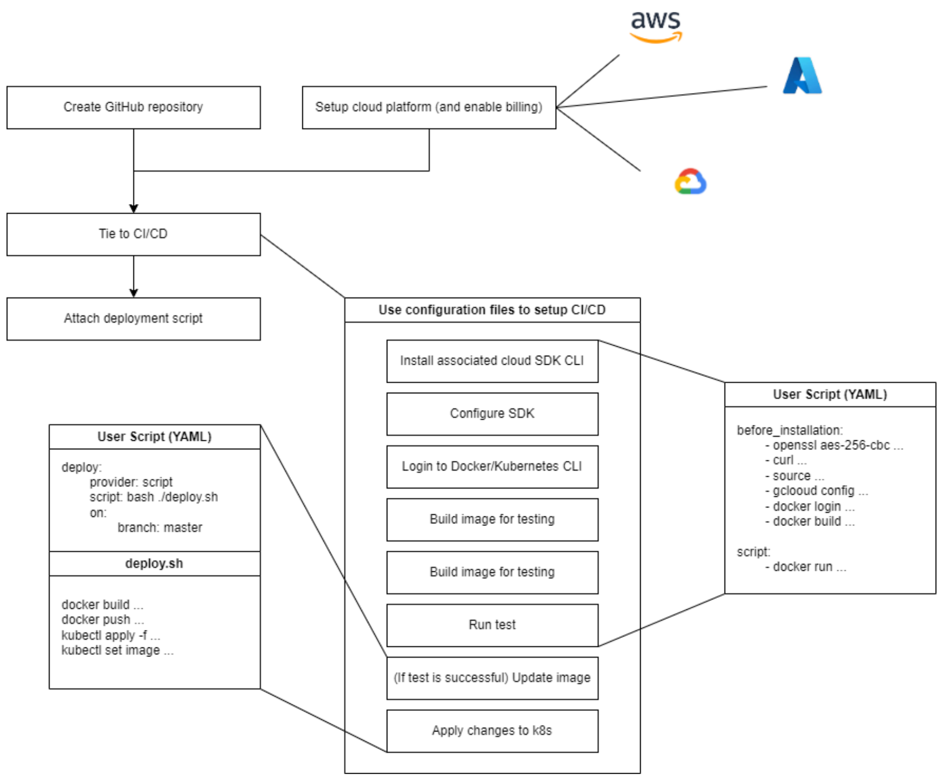


FIGURE 15.4
An example workflow of creating a production environment with Kubernetes.

The example used in this section to demonstrate Kubernetes CI/CD on a cloud platform in production environment is taken from [2]. Following [2], Google Cloud Platform is used as the cloud platform provider.

15.3.1 Setup Cloud Account

Many cloud platforms nowadays have a very good support of Kubernetes. The deployment of containers using Kubernetes can be done via their UIs easily. The developer needs to decide the resources to be used for the deployment. The more nodes and more power machine, the higher the charge. In most cases, the developer does not need to start a VM and install Kubernetes on it by himself. The cloud provider shall have dedicated Kubernetes engine service that would automatically configure the VM per required.

15.3.2 Configure CI/CD

Travis CI is a continuous integration service tool written. It can be deployed on the cloud and linked to a Github repository and a CI/CD platform such as a Kubernetes cluster on Google Cloud Platform.

To run Travis, a machine supporting Ruby programming language is required. For that, a separate container with Ruby installation is deployed to run Travis. Use Github credentials to login to Travis, so that it can link to the Github repositories.

Travis has built-in file encryption function. This function is mainly used to encrypt login credentials and service account credentials (in this example, the service account information to link to Kubernetes clusters on Google Cloud Platform) locally, so that later the unencrypted original credential files can be deleted, and only the encrypted credentials uploaded to the Github repository. When encrypting a file, Travis will also guide the user on how to call the encrypt information in the build script.

Travis uses configuration files to setup CI/CD pipeline.

15.3.3 Deploy Containers

In the Travis configuration file, `deploy` is used to specify the script to run when the testing is successful. A separate bash script `deploy.sh` is defined for this purpose, inside which is a sequence of commands that builds and publishes images, and configures Kubernetes using `kubectl`.

It is particularly worth mention that in `deploy.sh`, when building and applying the latest version of the docker image, tagging using `<image-name>:latest` alone is not going to work for the same reason explained earlier in Section 15.2.1: when the same configuration with `<image-name>:latest` is applied, the system would simply acknowledge it as "no change" and would not actually download the latest version of the image.

The walk around introduced in Section 15.2.1 was to use version number in the configuration file and/or as an imperial command as follows.

```
$ kubectl set image deployment/<Deployment name> <container name
    >=<image name>:<version>
```

so what when the version name changes, Kubernetes would notice the differences and apply the new image. When working with CI/CD using *Git*, this can be further automated. Just use the `$GIT_SHA` as part of the tag as follows.

```
docker build -t <docker-id>/<image-name>:latest -t <docker-id>/<
    image-name>:$GIT_SHA -f <dockerfile> <save-directory>
docker push <docker-id>/<image-name>:latest
```

```
docker push <docker-id>/<image-name>:$GIT_SHA
```

Notice that in addition to `:latest`, `:$GIT_SHA` is used as a secondary tag. When pushing the built image to Docker hub, both `:latest` and `:$GIT_SHA` are pushed (although they have identical content). When setting image, the `$GIT_SHA` is used to identify the image just like the version number.

It is recommended not to remove `:latest` in the building command. This is because if someone wants to pull and test the latest image in his server (without knowing the value of `$GIT_SHA` for the latest commit), he is still able to do so using only the image name.

Notice that `$GIT_SHA` is not a built-in environment variable. The developer needs to set that environmental variable manually in the configuration YAML file as follows. It is possible to replace `$GIT_SHA` with a different name.

```
env:
global:
- GIT_SHA=$(git rev-parse HEAD)
```

With the above setup, `$GIT_SHA` can be used in `deploy.sh` as an environmental variable.

15.3.4 Manage Secrets

Notice that when CI/CD tool is tied to the cloud platform provider, service account authentication is required. It is a good habit to NOT to put the authentication information in the CI/CD configuration file as plain text, or to upload the unencrypted file that contains the authentication information to the public workspace. It is possible that some CI/CD tools provide encryption tools that can be used to encrypt the authentication file. In such case, the developer may need to install the required CLI for that CI/CD tool.

In Section 15.2.4, it has been introduced that Kubernetes uses Secrets object to encrypt secret files. The encrypted secret files can then be safely published online. In the Kubernetes configuration file, an environmental variable can be created to call the secret information.

Many cloud platform providers including Google Cloud Platform provides services to manage secrets.

15.3.5 Helm

To install a software in a Kubernetes pod, the most intuitive way is to commit the installation in the image, and call the image in the Kubernetes configuration file. For commonly used services such as `ingress-nginx`, its installation configuration file is available online as part of the manual. It essentially starts and initializes a branch of Kubernetes objects to enable the service.

Helm is designed as an alternative to manage software installation in Kubernetes clusters. In many occasions, it is more convenient (or even only possible) to use Helm to install a software. More details are given in *github.com/helm/helm*.

We need to first install Helm from script. Helm installation used to contain two parts, the CLI (referred as Helm client) and the server (referred as Tiller server). We could then use Helm CLI to install other third-party software and tools.

Access control is important on cloud platforms. In practice, user accounts are used to identify users, and service accounts to identify pods and programs. Associated role bindings are used to manage what resources can be accessed by a user or program. For example, administrative role over the entire account can be used to bind with the administrative user. The same applies to Helm. The Tiller server required some extent of administrative control over the resources in an account. In many occasions, Tiller server was given the administrative permission to access the entire account, which introduced security risks.

As of Helm version 3, a major change was carried out where Tiller server was removed completely. Helm architecture is more secure and simpler today. The concerns related to Tiller's permission in the Kubernetes cluster are no longer relevant. Helm 3 of course requires permissions to use the resources, which is now managed by Kubernetes role-based access control mechanisms.

Part IV

Linux Security

16

Introduction to OS Security

CONTENTS

Linux, as well as other OSs, uses variety of methods to protect the system and the data. The chapters in this part of the notebook introduce some of these security methods. As cloud computing is getting more and more popular, virtualization security is also discussed.

16.1 Basics

The background, motivation, and basic concepts of computer security are introduced in this section.

16.1.1 Risks and Attacks

No computer or OS are absolutely safe. Risks can be introduced by the subjects listed below.

- Software bugs. The OS and application software may have bugs which leave backdoors to malware.

- Malicious users. A user may perform illegal actions that damage other users sharing the same servers or services.
- Unauthorized access. The user or a program may intentionally or accidentally try to access confidential data that they should not view.

A risk will most likely not turn into an actual disaster by nature. However, when a hacker or a malicious user initiate an attack deliberately taking advantage of the risk, it may cause troubles.

The attacks can be divided into the following categories.

- Malware. The attacker disguises a piece of malware code as a legitimate software. When the software is executed, the malware carries out harmful activities.
- System Penetration. The attacker accesses a protected system bypassing security checks.
- Man-in-the-Middle Attack (MitM). The attacker intercepts communications between legitimate entities, and steal or modify the contents of the communication.
- Denial of Service (DoS). The attacker overwhelms a system and paralyzes its service by sending a lot of requests to the system, more than it can handle.
- Network Sniffing. The attacker passively logs information from the internet, and use them for future attacks.
- TEMPEST (Van Eck phreaking). The attacker collects and analyses data measured from electromagnetic emissions of devices such as mobile phones, and decode information from the measurements.
- Social Engineering. The attacker gathers information of the victims by cheating, phishing emails, etc.

To protect the users from the attacks, we need both computer security and communication security. This notebook focuses on computer security, specifically OS security.

16.1.2 General Security Architecture

It is impractical to secure every component of a system (hardware, OS kernel, OS services, application services, user interface, users, etc.) with a singular universal protection method. Therefore, a more common approach is to implement a layered security architecture as shown in Fig. 16.1. In this paradigm, the system is segregated into distinct layers such as the hardware layer, the OS layer, and the application layer. Each layer employs its own security mechanisms targeting specific vulnerabilities.

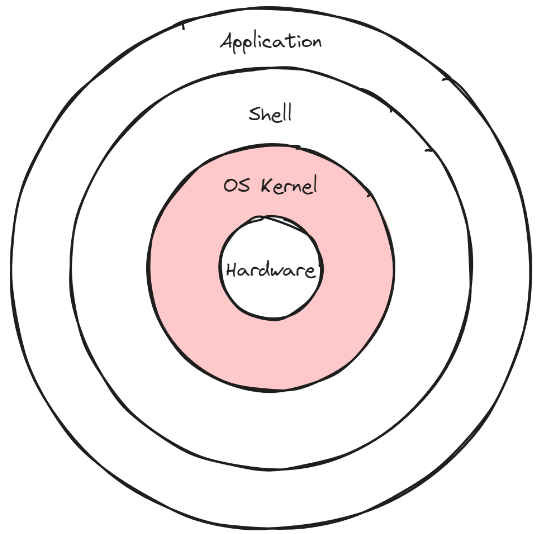


FIGURE 16.1
Layer structure of an operating system.

Among all these layers, securing the OS is particularly crucial for several reasons:

- Breaches in both hardware and application software often exploit vulnerabilities within the OS. By securing the OS, threats to other layers can be substantially mitigated. Even if certain applications are compromised, a robust OS can limit the extent and spread of the damage.
- If the OS is compromised, safeguarding other layers becomes extremely difficult. Most components across layers interact frequently with the OS and they typically operate under the assumption that the OS is trustworthy.

A system is considered secure if the following criteria are met.

- It is secure upon booting, and
- It never perform an action so that it can become insecure from a secure condition.

Notice that just for clarification, there are some slight differences between "secure" and "trusted" as follows. System security is the ultimate goal. A system is either secure or insecure, and we want it to stay secure all the time. Trustworthy, on the other hand, is a graded feature that we use to describe an entity in the system. We can, for example, say which parts (services, users, etc.) of the system is trustworthy, and to what extent they can be trusted.

16.1.3 Standards and Requirements

There are many official standards on computer security. For example, Trusted Computer System Evaluation Criteria (TCSEC) published in 1985, also known as the "orange book", is one of the earliest standards in this domain. It divides system into different tiers in terms of security, including

- A: Verified protection
- B: Mandatory protection
  - B3: Security Domains
  - B2: Structured Protection
  - B1: Labeled Security Protection
- C: Discretionary protection
  - C2: Controlled Access Protection
  - C1: Discretionary Security Protection
- D: Minimal protection

Notice that TCSEC is considered outdated due to the rapid advancement of technology. Nowadays, commercialized PCs and OS such as Windows 11 pro, MacOS, RHEL, implement robust security mechanisms that align with various aspects of TCSEC criteria of different tiers, some of which required by Tier B and even Tier A. While TCSEC remains a classic and milestone, newer standards have been developed and adopted globally by various countries and organizations.

In the scope of Linux, there is an open-project, "Security Enhanced Linux (SELinux)" that enables mandatory access control in Linux. It started as an add-on module of Linux kernel, and today it has become a default module.

In general, the requirements of secure computer systems include

- Confidentiality. Data, as well as the existence of the data, is not leaked to unauthorized entities.
- Integrity. The data can be trusted, and it cannot be modified by unauthorized entities.
- Accountability. It is possible to trace and audit the actions performed by users and programs.
- Availability. The system should be resistant to attacks and consistently provide services.

The primary objective of studying computer security is to ensure that the aforementioned requirements are consistently met and upheld. Regrettably, there is no systematic approach guaranteeing that these requirements are met at all times.

16.2 Elements of Security

Key components in security schema include:

- Security policy. It defines what needs to be protected and what the desired security outcomes are.
- Security mechanism. It defines the tools, methods, and procedures employed to enforce the security policy.
- Security assurance: It defines the means by which we evaluate the efficacy of the security mechanism.

16.2.1 Security Policy

A security policy establishes the standards and objectives that a system must adhere to, outlining the rules that both users and programs are expected to follow. Deviations from these stipulations or breaches of the rules can compromise the security of the system. A security policy often comprises a set of sub-policies, which may be categorized into areas like confidentiality policies, integrity policies, and so on.

Different systems may implement different policies. There are two commonly seen security policies that concern most of the systems. They are

- Confidentiality policies: preventing the unauthorized disclosure of information.
- Integrity policies: preventing the unauthorized alteration of information.

For the sake of clarity and to ensure that no misunderstandings arise, it is crucial for the security policy to be articulated in a precise and consistent manner. Instead of relying on colloquial or vague terminology, we use security policy model and policy language to formally and precisely describe the security policy. Security policy model should be ambiguity-free and easy to comprehend. Though it does not assume or restrict the security mechanism to be used to fulfill the policy, it should give some guidance to how the mechanisms can be designed. At the minimum, it should make sure that the policy is reachable.

More details of security policy, especially security model, is introduced in later sections. There are classic security models such as HRU model that has been proved useful and inspiring.

16.2.2 Security Mechanism

Security mechanisms are the means and tools to fulfill the security policies. Security mechanisms can be widely divided into 3 types:

- Prevention: (most commonly seen) protect system from being damaged.
- Detection: detect potential risks and damages.
- Recovery: recover a compromised system back to a secure system.

Different systems uses different security mechanisms to fulfill different security policies. Some important concepts are introduced below.

One of the most commonly used security mechanisms is access control, which monitors and controls the accessibility of a resource (known as objects) from users or programs (known as subjects). Access control is managed by reference monitor. Reference monitor refers to the combination of hardware and software that practices access control using the architecture given in Fig. 16.2.

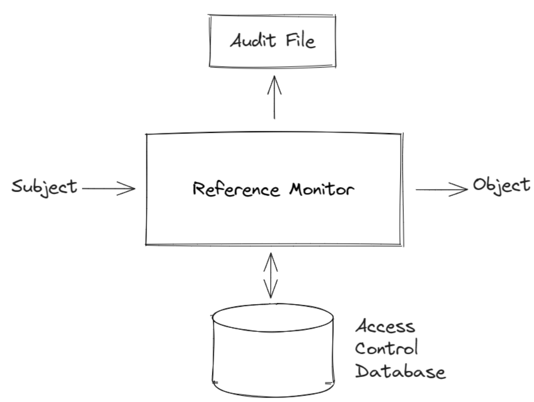


FIGURE 16.2
Reference Monitor Architecture.

Another important concept is security kernel. Security kernel refers to a small piece of code running in the kernel of OS that addresses system security.

Both reference monitor and security kernel shall have the following features:

- Completeness. They must be active all the time, and no process can bypass them.
- Isolation. Their content cannot be modified by unauthorized personals.
- Verification. They can be audited, and their effectiveness can be proved and verified. (This implies that their realization cannot be too complicated.)

16.2.3 Security Assurance

Security assurance refers to the degree of confidence in the security features, practices, procedures, and architecture of an information system. It ensures that the system enforces the security policy effectively. Below are key aspects of security assurance:

- **Verification and Validation**:
  - Verification: Checking that the system complies with specifications and is correctly implemented.
  - Validation: Ensuring the system meets user's needs and its intended purpose.
- **Assessment and Evaluation**:
  - Evaluating the effectiveness of security controls.
  - Assessing compliance with security standards.
- **Testing**:
  - Conducting tests to identify security vulnerabilities.
  - Penetration testing and automated vulnerability scanning.
- **Certification and Accreditation**:
  - Certification: Evaluation of security features of a system.
  - Accreditation: Approval to operate in a secure environment.
- **Risk Management**:
  - Identifying, assessing, and mitigating risks.
- **Audit and Compliance**:
  - Regular audits for policy and standard compliance.
  - Maintaining logs and records for security events.
- **Continuous Monitoring**:
  - Real-time threat detection and response.
- **Documentation**:
  - Maintaining records of security policies, procedures, and changes.
- **Training and Awareness**:
  - Training users and administrators in security best practices.
  - Creating organizational awareness of security threats.

Security assurance is an ongoing process that is critical for ensuring that a system is secure by design, in implementation, and in deployment, adapting to new vulnerabilities and attack vectors over time.

16.2.4 Trusted Computing Base

System boundary refers to the boundary between the system and outside world. Everything in the system is protected by the system following the requirements specified by the security policy.

Security perimeter refers to the imaginary boundary that distinguishes security-relevant components and non-security-relevant components in the system. Security-relevant components include OS, system files, administrators and his files and programs, etc. Non-security-relevant components include user program, user profiles, I/O devices, etc. A demonstrative figure from [1] is given in Fig. 16.3.

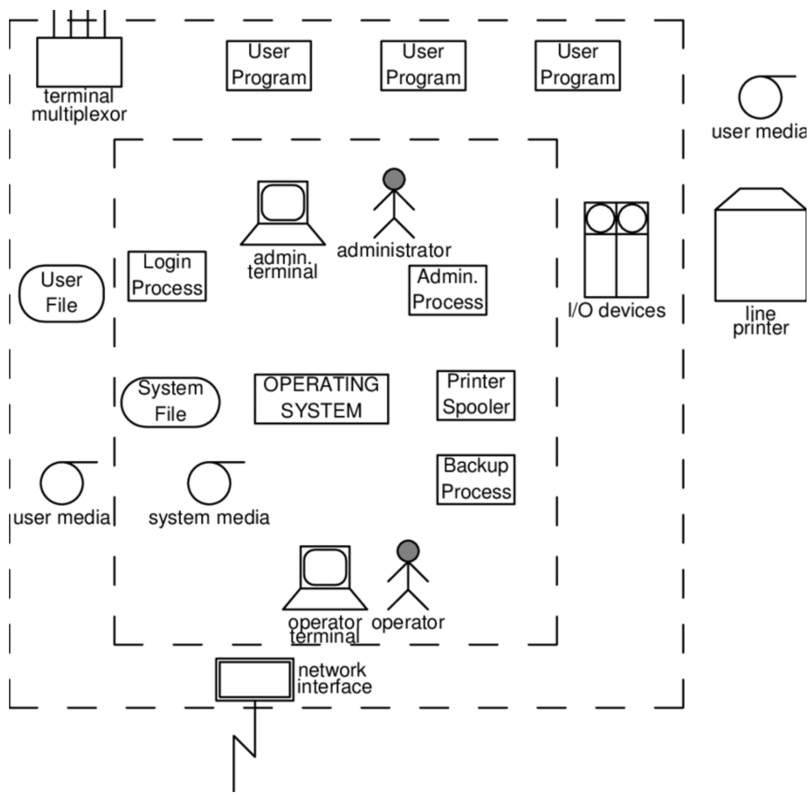


FIGURE 16.3
Reference Monitor Architecture [1].

The Trusted Computing Base (TCB) encompasses all the hardware and software components that are crucial for enforcing a system's security policy. This dual perspective implies that:

- Security Assurance. Efforts must be made to ensure that the TCB components are as secure as possible. Any vulnerability within the TCB could potentially compromise the security of the entire system. Therefore, the integrity, confidentiality, and availability of the TCB are of utmost importance.

- Assumption of Trust. When designing security mechanisms for the system, the TCB is assumed to be inherently secure and trustworthy. Security mechanisms rely on the TCB to operate correctly and to enforce security policies effectively. This assumption is fundamental to the system's architecture, as the TCB underpins the security of all other components.

The TCB can be likened to a police department in a city. Just as citizens rely on the police force to uphold the law and protect public safety, a system relies on the TCB, the "police security bureau" of the system, to enforce its security policies and maintain the integrity of its operations. Similarly, the well-being and readiness of the police officers are paramount to the security of the city. In the case of the TCB, ensuring the security of these trusted components is crucial because a compromise to any part of the TCB could undermine the security of the entire system. The system defenses are only as robust as the TCB's integrity and resilience.

Non-trusted components, while not central to the system security architecture, are analogous to the various elements of a city that the police department does not directly oversee. If these elements are compromised, the immediate risk is localized. However, securing non-trusted components is also important. We always want each and every component in the system to work properly. Maybe more importantly, we want to prevent them from becoming weak links that could be exploited to attack the TCB.

The goal is to keep the TCB, our "police department", as streamlined and strong as possible. A smaller, well-protected TCB simplifies the task of maintaining system security and reduces the potential attack surface.

16.3 Access Control

In the realm of computer security, a "subject" refers to an active entity that initiates an action, typically a user who takes an active role. On the other hand, an "object" refers to a passive entity that receives or is acted upon by the action. In most contexts, objects are files, data, or programs. Processes and threads can simultaneously act as both subjects and objects.

The term "access" is used to describe when a subject performs actions on an object. This can encompass various activities such as creating, reading, executing, editing, or deleting the object. Access control is a mechanism that assists systems in determining whether to grant or deny specific permissions, dictating which subjects can access particular objects and what actions they can undertake with those objects.

Typically, a user who plays subject is required to first identify himself by logging into an account of the system using a valid authentication methods. A user or a program that plays subject needs to be assigned with a "role". The program can only execute if its role possesses the necessary permissions to do so, including access to required resources like CPU, memory, disk space, and databases.

We use access control matrix to describe the association of subject and object. A demonstrative graph is given in Fig. 16.4. In this demonstration, like introduced in Section 4.3, "read", "write" and "execute" permissions are controlled, each using 3-bit "r", "w" and "x", together forming the nine permission bits.

	File/Program 1	File/Program 2	File/Program 3
User 1	rwx	rx	r
User 2	rwx	r	-
User 3	r	r	-

FIGURE 16.4
A demonstration of access control matrix.

The most popular ways to manage the access control matrix include discretionary access control (DAC) and mandatory access control (MAC).

16.3.1 Discretionary Access Control

In the DAC model, the owner of the data, or the group to which the owner belongs, determines the access control matrix for the data. This means that the owner has complete authority over who can access the data and in what manner. Additionally, the owner has the flexibility to delegate full access privileges to others if desired. This level of flexibility makes DAC a popular choice in many operating systems, including Linux.

Under DAC, the operating system can track the capability of each user's access to data. One method involves using a capability list, which outlines all the data a user can access. The users cannot modify their own capability lists. However, they can extend their access capabilities to other users. When a capability list is associated with each subject rather than individual users, it is referred to as a profile. Conversely, an object can be assigned an ACL as introduced in Section 4.3 that records which users or subjects have access. This latter method of assigning ACLs to objects is more widely adopted.

As mentioned in Section 4.3, in Linux access control is managed through a system of "9 permission bits". These bits define the accessibility for three distinct user categories: the owner, the group (the primary group of the owner, or a group the owner specifies), and others. Each category's permissions are represented by three bits, indicating whether the data is readable (`r` or `-`), writable (`w` or `-`) and executable (`x` or `-`).

While this method is straightforward, it does have limitations, notably its simplistic division of subjects into only three categories, which may not offer sufficient flexibility for all use cases. But it suits most of our needs on a personal computer.

16.3.2 Mandatory Access Control

Though flexible, DAC adds risks to the system. A malware such as a Trojan horse is able to change ACL of files on behalf of the data owner without his notice or permission. To tackle this issue, consider MAC instead. MAC is popular on machines with sensitive data, such as government servers.

Unlike DAC where the owner can change the ACLs of his files, when MAC is implemented, the owner cannot change the ACLs of the files. Only the system can change the ACLs of the files according to predefined security policies.

As a special case of mandatory access control, consider multi-label security mechanism. In this implementation, security labels are assigned to both subjects and objects. Only the subject with a equal or higher security level than the object can access that object.

17

Security of Services and Applications

CONTENTS

The previous chapter introduced general OS security mechanisms. This chapter discusses security mechanisms of services and software running on the OS such as database. Notice that network and communication security is a stand alone topic not considered in this chapter.

17.1 Database Security

Many Apps, both local and online, heavily rely on database to manage information. For example, online shopping APP uses database to record transaction history. Online banking uses database to store and manage customer capitals. The data stored in a database is often critical and confidential, and the database service providers need to try their best to prevent data loss and leaking, and to ensure the integrity and availability of the database.

When a database is under attack, the worst scenario may go beyond data damage. Examples are given below.

- Paralyze database service.
- Change and remove data illegally.
- Steel data.

TABLE 17.1
DB security risks categories and associated security methods.

	With Authentication	Without Authentication
Internal	Developer, manager, etc. Access control and audit	Irrelevant employee, etc. Encoding
External	Customer, vendor, etc. Outsourced DB security methods	Hacker, visitor, etc. SQL injection prevention

- Attack and gain unauthorized access to the underlying OS, and damage or control the entire server.
- Deploy Trojan horse program for other servers connected to the database server.

It is possible that the attacker hides and disguises the attacking command into SQL injection to open a back door or to retrieve unauthorized data such as confidential information.

Secure DBMS and Secure OS

It is worth introducing the relationship between a security-enhanced OS and security-enhanced DBMS. In short, they make a better each other, forming a "security chain" together. A secure OS boosts the security of the database. A poor OS, on the other hand, harms the DBMS security level because data is essentially stored on hard drive which can be penetrated if the OS is down.

There are two common ways of forming a security-enhanced DBMS from a normal DBMS. The first way is to upgrade the DBMS kernel codes for additional security features. This can be complicated and requires a high-level mathematics, databases and programming skills, but can provide a safe DBMS. Obviously, this applies to only open-source databases. The second way is to build a "wrapper" for the DBMS to interface with the users and API calls This usually requires less skill sets and can be applied to both open-source and proprietary databases, but can only provide mediocre security.

17.1.1 Database Security Risks Categories

Depending on the identity and the access level of the attacker, database security risks can be divided into the following categories as shown in Table 17.1. Different database security risks categories may tell completely different stories on how an attacker plans his attack. For example, a developer may leave a backdoor in the program which he can use to access confidential data. A vendor may bring the hard drive of the database outside the managed premises, after which he can use variety of tools to crack the database and obtain the data.

To tackle the challenges, different security methods need to be applied to prevent each and every risks category. A high-level summary is given in Table 17.1. Details are given in later sections.

17.1.2 Database Access Control

There are different types of access control, some of which widely adopted to all different types of databases, while others may apply to only high-level secure databases. Some access control schema apply to both database and OS, the most popular ones being DAC and MAC which have already been introduced in earlier sections when discussing OS security. They are re-addressed for database as follows.

DAC restricts access to objects based on the identity of the subject, i.e. the user or the group of the user. The accessibility of an object is determined by the owner of the object. The same idea has been adopted by Linux in file management.

As introduced earlier in database chapters, in DBMS, use syntax that looks like the following to grand and revoke access of an object from a subject.

```
GRANT <privilege> ON <table/view> TO <subject>
REVOKE <privilege> ON <table/view> FROM <subject>
```

In practice, it is common that the database manager sets up different set of views for different user groups, each set of views containing everything that the user group requires. Grant access to only the associated views to the user groups. This can hopefully prevent a user from accessing data confidential to him.

The problem of DAC is that it can "lose control" sometimes, making a user bypassing the restriction. For example, a user who has been revoked from access may still be able to access the data if he had created a procedure that reads the data, and his access to that procedure is not revoked. Many DBMS tries to provide some protection against this, for example, by integrating security labels into the SQL that the user injects. When a user execute an SQL command, in the backend the SQL command is "reformulated" to contain user authentication information. In a good implementation, this security mechanism should be made transparent to the user.

Another challenge is that sometimes the user legitimately asks for aggregated information which, however, can only be derived if he has access to data confidential to him. For example, consider a table that stores the scores of a class. A student wants to check his score as well as the average score of the class, which makes perfect since. However, the scores of all the students are required to derive the average score. If the access of the student is limited to his score alone, he will not be able to get the average score of the class.

Different from DAC where the owner of an object choose his preference of who can access the data and the preference can change case by case, in MAC the rules are enforced by the system administrator consistently and globally. The user cannot overwrite security policy even on his own data. This reduces the chance of human error, hence providing a higher level of security. The cost is the flexibility in the user experience, and the complexity of setting up global rules especially on a large database. MAC is often used in government database where huge amount of confidential and sensitive data is managed with heavy responsibility.

In multi-level security DBMS (MLS), also know as multi-layer DBMS, each piece of data in a database is associated with a security label that reads like "unclassified", "confidential", "secret", "top secret", etc. This security label is a compulsory attribute to all the data. Users also have security labels that determines the level of accessibility. When querying the same database, different users will get different results based on the security level of each piece of data. The higher the security label of the user, the more information he is able to retrieve.

MLS is often used as part of MAC which is introduced earlier in this section.

In addition to query, the security labels also affect how `INSERT` and other database manipulation commands work. Obviously, the user needs to have a higher layer (or at least equal) security label than the data, in order to insert, edit or remove it.

There might be an interesting case where a row with higher security label already exists in the table, and a user with a lower security label who cannot detect that data wants to insert into the table a new row with the same primary key. In a normal database, this operation would have been rejected due to the duplication of primary key. However, in MLS, this action is granted. Otherwise, the user with lower security label would sense the existence of the higher security label data. This technology is known as polyinstantiation, a method used to avoid covert channels by allowing multiple rows with the same primary key but with different data, based on different security levels. Polyinstantiation occurs when multiple rows in table appear to have the same primary key when viewed at different security levels.

Covert channel refers to a "disguised" channel that transfer information between entities while violating the security policy. In many cases, the channel is built from a list of operations, all of which legitimate by itself alone. These operations, when combined together, creates this unexpected bug outbound designer's intention. An example of a convert channel is as follows.

1. Entity A, with higher privileges, encodes secret information in binary format.

2. Periodically, entity A change the permission of a file that can be sensed by entity B. The encoded binary format is used to setup the permissions.

3. Entity B listens to the permission of that file to obtain the binary format data.

4. Entity B decodes the binary format to obtain the secret information.

By doing the above, entity A is able to transfer a secret information to entity B who has a lower layer security label and should not touch the data. Notice that entity A may not be a human traitor, but a malware program.

Covert channel uses unintended system mechanisms for communication, which is often low efficient. As a result, the bandwidth of the covert channel is often much lower than a regular communication channel. Besides, a fast covert channel is likely to be easier to detect, which is something that the hackers want to avoid.

17.1.3 Security-Enhanced DBMS Solutions in a Glance

To wrap up, different security-enhanced DBMS solutions are now available on the market as follows.

Normal DBMS on Security-Enhanced OS

In the early stage, no additional security features is added to the DBMS. It is just that the DBMS is running on a secure OS with MAC enabled. The database is often put into the group with highest sensitivity level. The problem of this implementation is that all users who legitimately access the database have to be in the same highest sensitivity group, which violates the principle of access control. The output of the database, whatever it might be, is considered generated from the highest sensitivity group, and needs to be audited each time before release. This severely adds human cost.

MLS

MLS, also known as "trusted database", adopts security-enhanced DBMS that uses security labels to mark all the data and users. It is secure and flexible, and the details have been introduced in the earlier section. The only obvious problem for MLS is that it is difficult to realize. For third-party MLS, the customer never know whether there is a backdoor, unless he check all the codes (millions of lines of codes) that realize the DBMS by himself, which is enormously tedious and sometimes impossible.

Security Wrapper

An alternative to MLS is use normal OS and DBMS, and add a "filter" between DBMS and the users. This filter serves as a wrapper to the DBMS, and it uses security stamps to manage data transmission. All the data stored in the database can be encoded, and only users with the correct keys can decode them. This forms an encoded database with security wrapper.

Distributed Database

Till this point, it can be seen that the key to database security is to prevent data leakage from high security tier databases to low security tier databases. MLS tries to label the data carefully to isolate high security tier data from low security tier data, while the secure OS and secure wrapper try to prevent low security tier user from accessing high security tier data. Distributed database is another approach trying to further isolate the data of different security tiers: use different database, or even run them on different machines, in the first place.

Some system runs multiple DBMS kernels concurrently on a single server, each kernel managing a security layer of data. The compromise of one kernel does not necessarily mean that other layers are compromised. This may make the DBMS safer, but it will create high computational burden to the system. High security sometimes means low efficiency, and low efficiency can be bad for commercialized databases. It is also a challenge how to balance the trade-off between security, efficiency and cost.

Other system runs multiple DBMS kernels on concurrently on multiple servers, each server in charge of a security tier. Low security tier databases are synchronized to high security tier servers (bot not wise versa). This architecture design is robust to covert channel, but the data consistency and availability becomes a challenge.

17.1.4 Outsourced Database Security

Some enterprise outsources the database management and maintenance to IT companies such as Microsoft, Oracle or other database service providers. Security challenges introduced by cloud/vendor-based database differ largely from on-premises databases mainly because the DBMS itself is not reliable.

User-Encrypted Data: Prevent Data Leakage

There is a risk that the third-party database service provider may leak the data intentionally or unintentionally.

An intuitive solution of this problem is to encrypt the data in the user-end before saving into the database. A downside of this is that when we want to retrieve data from the database using an SQL query, the DBMS may have a difficult time interpreting and filtering the data. "Searchable encryption" is required in this case. It allows filtering of data without decrypting it first.

The result from the outsourced database needs to be audited, mainly to check the authenticity, completeness and freshness of the returned information. Part of the reason is that searchable encryption may fail to return the correct and complete result. More importantly, the underlying assumption is that we cannot entirely trust the DBMS in the first place.

Watermark: Prevent Data Modification

Watermark can be used to identify the owner and authenticity of the data.

It does not affect the normal usage of data, and it is hardly detectable by a third-party, but can be checked and audited conveniently by the party who assigned the watermark. Watermark shall be difficult to remove. The water mark is often added to the least significant bits of a numerical data.

Problems of watermark include

- It allows multiple entities to assign watermark to the same database. It is difficult to tell who the original owner of the database is.
- If a user wants to remove the watermark, he can simply remove all the LSBs of the data.

17.1.5 Big Data Security

As Industry 4.0 and IoT become more and more popular, we are collecting more data than ever in the history. Many activities such as building data driven models rely heavily on the big data. The major cloud service providers offer enterprise-tier databases optimized for big data management, both SQL and NoSQL such as Dynamo DB by Amazon and Cosmos DB by Azure, Microsoft.

There are some unique challenges when comes to the security of big data. This is not surprising as big data is generated, distributed and utilized very differently from the conventional data.

- Access control.
  - Data generated in big data comes from variety of sources by different types of users. It is difficult to track all the sources and all the users to determine the data accessibility and security tiers using traditional method.
  - AI model is used to assist categorizing sources and users, and assigning security tiers.
- Control and maintenance of data when it is distributed.
  - It is difficult to protect the privacy of the owner of the user when his data is published into a big data pool. It is possible to use AI model to find the owner of anonymously published data.
  - The key is to deny suspicious query and prevent a single entity from querying aggregated information.

17.2 Virtualization Security

Virtualization has been introduced in Chapter 14. While having many beneficial features, the use of virtualization brings new challenges to system security.

17.2.1 Security Concerns

Some major security concerns of VMs are listed below.

- Isolation between VMs.
- Migration of VMs.
- VM monitoring and supervisory.

VMs are naturally isolated due to the virtualization mechanism. Ideally, though running on the same physical server, a VM cannot bypass the monitored I/O to talk to another VM. When two VMs need to talk, VMM builds special communication channels for the VMs, usually a message queue or a piece of shared memory space. However, this does not guarantee the absolute isolation of information between two VMs. They may fail to wipe out all the data when handling over the hardware resources to another VM. There are potential covert channels where two VMs may communicate with each other, for example via the utilization rate of a hardware resource.

Migration happens frequently in VM applications due to its frequent scaling up and down. When migrating VMs from one server to the other, it is possible that the undercover malware is also migrated. In such cases, the malware can spread across servers and it will be difficult to trace back to its original source.

Conventionally, the MAC address of the hardware can be used as the unique identity of a machine. This does not apply to VMs. This makes some of the security measures difficult to implement.

When running a VM, the user who deploys the VM usually have administrative access to the VM guest OS. As introduced earlier in Chapter 14, VMM may allow VM guest OS to run some instructions in privileged mode. If there is a flaw in the VMM, the software running in the VM may take advantage of that and use it to attack other VMs hosted on the same server.

17.2.2 VMM Security

The security of VMM is key to the security of VMs as VMM has full control of the hardware and it manages and interacts with all the VMs. If VMM is compromised, all the VMs running on the system is exposed to high risk. In this sense, VMM is the single-point-of-failure to the entire system, hence needs to be monitored at all time.

A secure VMM architecture is helpful. Such architecture can be designed on top of an existing VMM. In practice, the secure VMM architecture may separate critical ("over-powered") functions of VMM into different domains, and provide additional interface for security monitoring of the VMM.

A

Scripts

CONTENTS

This appendix chapter collects the example scripts used in this notebook.

A.1 *Vim* Configuration `vimrc` Used in Section 3

```
call plug#begin()
Plug 'vim-airline/vim-airline'
Plug 'joshdick/onedark.vim'
call plug#end()

inoremap jj <Esc>
noremap j h
noremap k j
noremap i k
noremap h i

noremap s <nop>
noremap S :w<CR>
noremap Q :q<CR>

syntax on
colorscheme onedark

set number
set cursorline
set wrap
set wildmenu

set hlsearch
exec "nohlsearch"
```

```
set incsearch
set ignorecase
noremap <Space> :nohlsearch<CR>
noremap - Nzz
noremap = nzz

noremap sj :set nosplitright<CR>:vsplit<CR>
noremap sl :set splitright<CR>:vsplit<CR>
noremap si :set nosplitbelow<CR>:split<CR>
noremap sk :set splitbelow<CR>:split<CR>
noremap <C-j> <C-w>h
noremap <C-l> <C-w>l
noremap <C-i> <C-w>k
noremap <C-k> <C-w>j
noremap J :vertical resize-2<CR>
noremap L :vertical resize+2<CR>
noremap I :res+2<CR>
noremap K :res-2<CR>

set scrolloff=3
noremap sc :set spell!<CR>
```

A.2 *Neovim* Configuration Files

Neovim is fork from *Vim* and it has gained popularity in the past years. Just like *Vim*, *Neovim* can be customized by user-defined configuration files. These configuration files are usually packed into *lua* file tree, where *lua* is a computer programming language that *Neovim* supports by nature.

B

Continuous Integration and Delivery

CONTENTS

This notebook is mainly about Linux. In this appendix chapter, the boundary of the notebook is slightly expanded to software development, which is quite often how Linux is used in practice.

Continuous integration and delivery (CI/CD) is both a philosophical concept and a bunch of technology that speeds up the development, testing and deployment cycle of a software. It has become a common and beneficial practice that collaborative projects with rapid update implement the practice.

This chapter introduces CI/CD as well as tools to carry out CI/CD. In particular, GitHub, a widely appreciated CI/CD integrated platform to manage and share collaborative projects, is briefly introduced. The introduction of GitHub focuses on GitHub Action, the tool GitHub uses for CI/CD.

Some contents of this chapter come from [3].

B.1 Agile VS Waterfall

Agile and waterfall are both project management methodologies. Both of them are introduced here, starting with the more conventional waterfall model then Agile.

B.1.1 Waterfall

Speaking of project proposal, development, testing and delivery cycle, it is fairly intuitive to follow the procedures below:

1. Understand requirements from the user.
2. Design the architecture of the solution.
3. Develop the solution.
4. Test the solution.
5. Deliver the solution and close the project.
6. (Follow-up) maintain the solution.

The philosophy behind waterfall, as its name indicates, is to "follow the procedures and do not turn back". When a previous step is considered completed, it is completed and should not be revoked or revised. This is demonstrated by Fig. B.1.

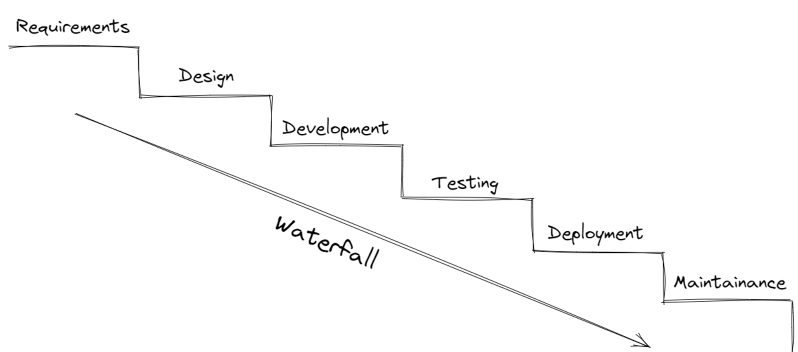


FIGURE B.1
Waterfall model.

With waterfall, a project can be designed, developed and deployed in a relatively efficient manner. However, there is one limitation: everything done in earlier steps cannot be changed in later steps. This sets up a high bar on both the user and the developer. For example, in step 1 "understand requirements from the user", the user needs to illustrate all the requirements to the developer as they cannot be modified in later steps. Similarly, in step 2 "design the architecture", the developer needs to optimize the design to his best, as the architecture cannot be changed later.

Regrettably, with the rapid change in the market and the aggressive advent in technology today, it is challenging even for the smartest user and developer to determine all the requirements and designs in the beginning stage of the project. More likely, the requirements of the user have to change align with the market trend, and so does the design of the solution.

For a new feature to be added into the existing system, it is possible to simply start a new project flow for the new feature, and integrate it into the existing system later. However, integrating the new feature into the existing system can also be challenging when the design is complicated and coupled. The integration often introduces a blackout period of the system. If the system is already deployed, the customer experience would be affected by the blackout.

B.1.2 Agile

Agile is the counterpart of waterfall. It is proposed to tackle the aforementioned issue: rapid change of requirements and adaptations to new technology and tools. It allows continuous integration and deployment of new features into the system in a convenient and systematic way, without introducing blackout.

In agile architecture, each feature is separately modularized. Each feature, before deploying and integrating into production environment, circulates in its own "development and testing circle", where it can be tested and reviewed iteratively by the developers and the users audit team, as shown below in Fig. B.2.

Agile allows rapid change to be made to the requirements and realizations of a feature. Should there be a change, just keep cycling in Fig. B.2 until the change is implemented and tested, before it is pushed back to the master branch.

In parallel development where there are multiple features running in their associated circles, the developer can easily choose which feature branch and circles to prioritize. This gives the developer a clearer overview of what is happening and how to best response to the customers immediate requirements.

B.1.3 Roles in Agile-based Development

The following roles are defined in agile, each role coming with a responsibility. The roles may slightly differ for each project. The most commonly seen approach, namely **scrum**, is introduced in Table. B.1.

In this role assignment, the product owner and scrum master come up with

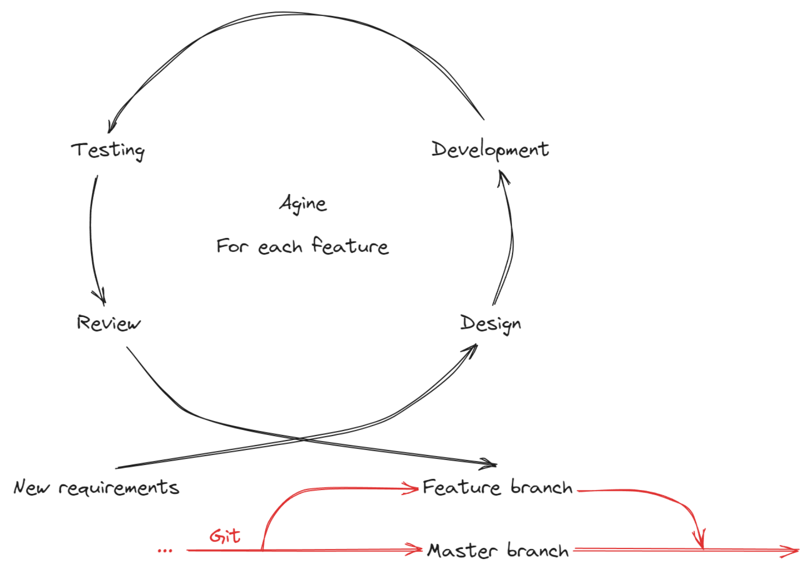


FIGURE B.2
Agile model.

TABLE B.1
Roles in Agile model.

Role	Description
Product owner	Manage the entire program. He understands all the user requirements and progression of all features. He also signs off each feature when they are deployed.
Scrum master	Lead the developer team as team manager or chief developer.
Developer	Based on requirements, program the features.
Tester	Design test cases to verify the efficacy of the developed feature.
Operator/Supporter	Maintain the software

the product backlog, which clarify the sprints (tasks) and their priorities. The team then knows which sprints they shall work on first.

In the case where the features concerns a professional domain (such as economics, medical, etc.) that the developers cannot understand, business analysts are involved who bridge the user and the developer.

For each sprint, sprint planning and sprint backlog are proposed that describes the schedule of the sprint. The team works on the sprint and host daily scrum meetings until the sprint is solved. Upon finishing of a sprint, sprint review is hosted for audition.

B.2 Continuous Integration Continuous Delivery

CI/CD is introduced in this section. As a start, pipeline is introduced. Pipeline is an important concept used in CI/CD.

B.2.1 Pipeline

Pipeline is a set of data processing elements in a queue, where the output of the upper stream process is the input of the down stream process. An example is shown in Fig. B.3. By using a pipeline and let multiple pipelines at different execution phase run in parallel, the efficiency of the system is increased.

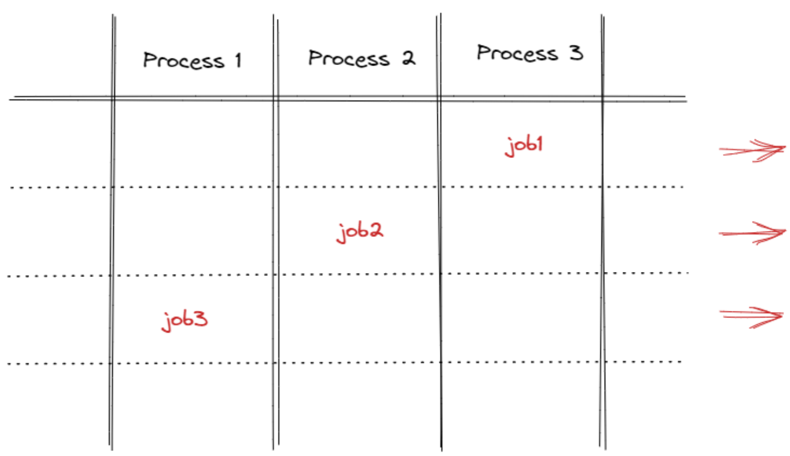


FIGURE B.3
Pipeline.

Pipeline is widely used in OS for process and thread management (recall Fig. 6.1). It is also used in CI/CD.

B.2.2 Continuous Integration

In the development of a sprint, new codes are rapidly developed, and they are rapidly compiled, packaged and tested.

Conventionally, the integration of a new feature requires involvement from multiple parties. An example is given in Fig. B.4. It includes the developer who program the software following users (or business analysts) requests, the integration team who integrates the new feature with the existing system and compile the code into packages, and the operations team who upload the new system into the pre-prod environment for real data testing, and the testers who audit the output of the program running in the pre-prod environment.

Should there be any error along the way, the code is roll back the developer team for trouble shooting. When the new system with the updated code survives pre-prod environment, it is then pushed to the production environment.

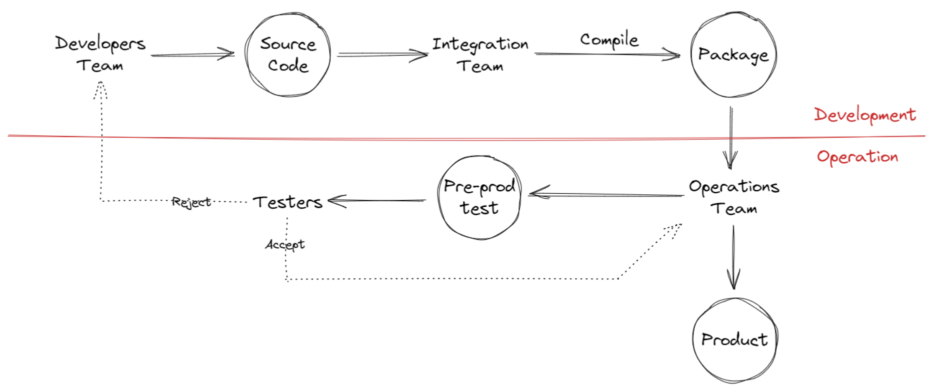


FIGURE B.4
CI of a new feature.

In practice, each cycle in Fig. B.4 can take a few days or even weeks as so many teams have to cooperate to make it happen. It takes time for the integration team to integrate different branches in the source code together, making sure that components from different branches function properly. When there is a defect, the flaws can be spotted only in the last stage of the iteration, i.e., testing. This practically disallows very frequent update of the system in response to the rapid changes in requirements.

CI (together with CD which is introduced in later section) tries to solve the above problems. CI automates the "development" portion in Fig. B.4, while CD automates the "operation" portion.

To speed up the development of new features, CI mainly adopts the following methods.

- Use *Git* to manage features. This simplifies the procedure of managing multiple under-development features and integrating them together. Integration is now managed by *Git* following developers' intention.

- Use a build server to automatically compile the code into ready-for-delivery packages. The scrum master and senior developers can access the server and monitor the progression. Should there be a compiling error, the developer is notified immediately.
- The code, after compiling, is immediately tested in the build server using pre-defined test cases. If the code fails the test cases, the developer is immediately notified.

CI effectively removed "integration team" from the picture. The integration related tasks are split into small pieces and processed by automation tools, feature integration by *Git*, and compiling by build server. During CI, the developer not only generates the codes, but also supervises *Git* and build server. Should there be any error, the developer is notified immediately by the automation tools.

CI speeds up the developing cycle, from the receiver receiving requests from the user, to the point they have new system in the package ready for pre-prod testing.

B.2.3 Continuous Delivery

The package received from the development side contains the latest version of the system where a new feature is integrated. It is, however, not certain at this point whether the new feature works properly and how the system would behave as a whole with this new feature. The sophisticated testing and deployment of the software features are done by the operations team.

Conventionally, operations team and the testers receive the updated package together with an instruction from the developers team. The instruction describes how the package shall be installed, and what test cases to use for auditing. The operations team and the testers need to understand the instructions, and configure the pre-prod environment accordingly for the testing. The testers then uses varieties of scenarios to test the performance of the software. Bugs, if any, are reported to the developers team. If no bugs are spotted, the testers notify the operation team to release the package into production environment.

There are some obvious drawbacks to the conventional approach. The developers team needs to give detailed and precise instruction to the operations team, and the developers may make mistakes or missing something in the instruction, especially regarding environment configuration. Besides, there are too many human interactions, which slow down the process and generates human error. The entire procedure usually take about a whole day.

CD is a software development practice that allows software to be released to production at any time. The idea behind CD is to deploy the code for testing automatically anytime CI provides a new package by adopting the following methods.

- Use machine-readable instruction files for packages installation, and let the server virtualize the execution environment and install the packages automatically.
- Use machine-readable testing scripts, and let the server execute tests and analyze the results automatically.
- The aforementioned machine-readable instruction files and testing scripts are managed the same way as the source code by the developers.

Ideally, as soon as a version of packages is released by CI, CD can automatically have it deployed and tested, and return the testing results to CI without human interaction. This is shown by Fig. B.5. In this CI/CD implementation, the developer is playing a more comprehensive role than what is shown in Fig. B.4.

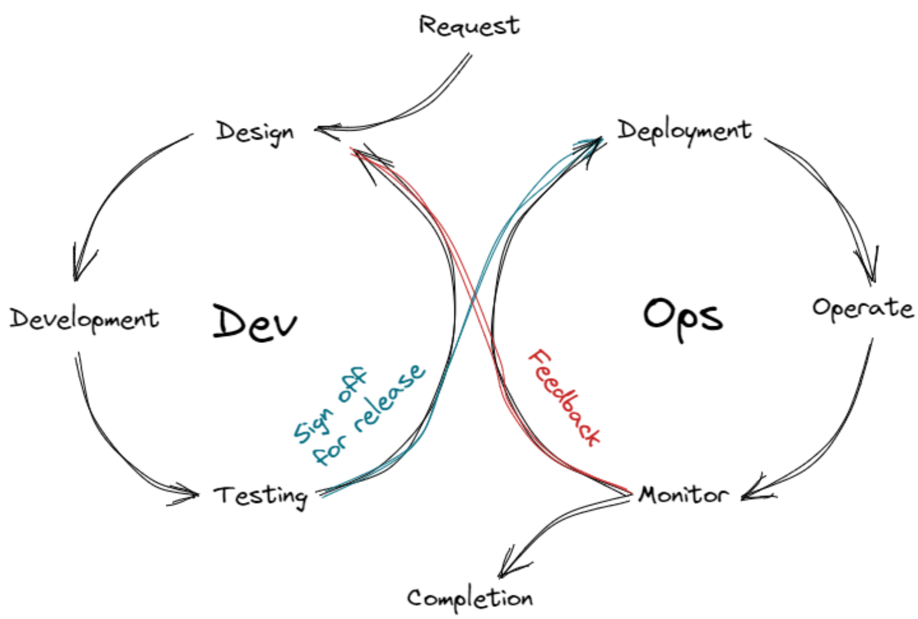


FIGURE B.5
CI and CD.

Under the framework of CI/CD in Fig. B.5, the developer not only develops the new feature, but also integrates the features with the help of version control and branch management tool *Git*, compiles the packages using build server, deploys the new packages in the testing environment by preparing machine-readable instruction files, and finally tests the new features using pre-configured test cases in the machine-readable testing scripts.

Since the operation team joins the developers team, they are now called the DevOps team. The CI/CD framework shown in Fig. B.5 is known as the CI/CD pipeline which represents an end-to-end software development lifecycle (SDLC) within its ecosystem.

This CI/CD framework enables fast deployment of new packages. Large IT companies can make up to dozens of new releases everyday.

B.3 *GitHub* Action (Part I: Basics)

GitHub has been an amazing platform for managing software projects, especially open-source collaborative projects.

In the early days when CI/CD was not enabled in *GitHub*, developers used third-party CI/CD tools such as Jenkins and Travis CI in conjunction with *GitHub* for automatic integration and deployment. Lately, *GitHub* introduced *GitHub Actions* (for short, *Actions*), its own CI/CD solution, as a response to developers' requests.

Actions is essentially a pipeline tool, and it is very useful as part of the practice of CI/CD on GitHub. In short, it allows the developers to create automatic workflows tied to *GitHub* events. *Actions* is widely used to automate the following tasks:

1. Compile the source code. The developer can define how the source code should be compiled.

2. Virtualize the environment to run the compiled packages. The developer can define how to set up the environment.

3. Test the packages on testing scenarios. The developer can define the testing scenarios.

4. Raise an issue if anything occurs during the above procedures.

Machine-readable instruction files are used to guide the automation. They are managed together with the source code in the repository.

Notice that *Actions* might be chargeable for private repositories depending on its computational cost. It is, however, often free of charge for community and public repository projects.

A commonly used scenarios of *Actions* for community projects is that when someone submits a pull request, a series of checks are automatically done, and emails are sent to the project maintenance team to notify them the request. Sometimes a "thank you" email is sent to the contributor.

Pull Request

When an individual developer wants to contribute to an open-source community project of others, the following is the general flow.

1. The developer forks the project to his own *GitHub* account.
2. The developer clones a copy of the project from his own *GitHub* account to his own machine.
3. The developer creates a new branch about the feature he wants to add or improve.
4. The developer develops the feature on the branch.
5. The developer commits the development and pushes the commit to the forked repository.
6. The developer submits a pull request to the maintenance team of the community project.
7. The maintenance team receives the pull request and review the changes made to the code in the developer's forked repository.
8. If no objection, the maintenance team pull the changes from the developer's forked repository, and merge the new feature branch with the existing branches.

With the above, the developer becomes an official contributor to the community project.

Another use case for *Actions* for individual projects is that when a new code is pushed to the repository, the code is automatically compiled and tested. If the test returns an error, create a *GitHub* issue. Otherwise, put out a new release.

B.3.1 Framework

In *Actions*, the developer needs to prepare CI/CD pipelines known as workflows, either by himself or from *Actions* Marketplace (a platform where commonly used workflows are shared). The backbone of a workflow is YAML files that describe the environment, the trigger, and the list of actions to execute. When an event such as a pull request happens, the associated workflow is triggered and executed. A demonstrative plot is given in Fig. B.6.

As illustrated in Fig. B.6, a workflow can contain multiple jobs that can run in parallel (by default) or sequence (by defining dependencies). Each job can contain multiple steps that run in sequence.

For each job, a "runner" is assigned which serves as the environment to run

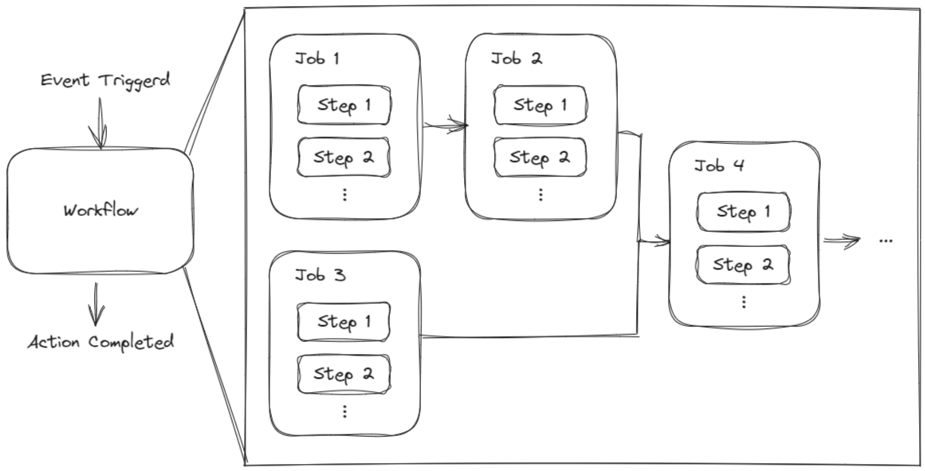


FIGURE B.6
Actions framework.

the job. The developer needs to specify how the runner should be configured such as what software needs to be installed in order to run the associated job.

B.3.2 *Actions* Triggers and Types

It is worth mentioning that *Actions* can be triggered not only by *GitHub* events but also external events, schedules, etc. Commonly seen triggers include

- *GitHub* Events, such as push, pull request, etc. This is the most common setup for CI/CD.
- External events.
- Schedule.
- Manual.

Actions provides workflow templates of different task types. There are at least the following workflow types as of this writing. They are not limited to CI/CD.

- CI
- Deployment
- Testing
- Code quality
- Code review

- Dependency management
- Monitoring
- Automation
- Utilities
- Pages
- Hugo

Some of them will be covered in later sections.

B.3.3 *Actions* Marketplace

Actions marketplace provides commonly seen actions for free, and they can be easily integrated into a workflow. The developer can look for action items in *Actions* marketplace before trying to prepare the entire workflow by himself from scratch.

B.3.4 Costs

Actions may generates additional costs. This is because executing workflows usually consumes *GitHub* servers' storage and CPU. As of this writing, additional cost is generated only if all the following criteria are met:

- Private repository. This is often not the case for community and individual projects, but often true for enterprise repositories.
- The workflows are executed on *GitHub* servers. Notice that it is possible to execute workflows using on-premises servers, in which case *Actions* becomes a free service.
- Storage and/or computational cost goes beyond the free-tier threshold.

When all the criteria is met, the developer pays *GitHub* by the storage and computational consumption. He can select either prepaid mode (fixed bill, limited consumption per month) or postpaid mode (unfixed bill, unlimited consumption per month).

The running environment also affects the cost. It is often more expensive if the workflow needs to run on Windows or MacOS than if it can run on Linux due to the licensing fee. As of this writing, Windows and MacOS introduce a computational cost multiplier of 2 and 10 respectively comparing with Linux.

B.4 ***GitHub Actions*** (Part II: Practice)

As introduced in the previous section, the key to an *Actions* includes defining the triggers and preparing the associated workflows for the triggers. Of course, the developer also needs to map the triggers with the workflows. More details are introduced in this section.

The workflow files are used to define the triggers as well as the associated workflows. They are YAML format files following specific syntax that illustrates environment configurations requirements, jobs, and job steps. They are stored under `.github/workflows` and are managed by the developers just like any other source code. Notice that there are certain events that would trigger only if their associated workflow files exist in the project repository default branch (usually `main` or `master` branch).

As an example, here is a piece of workflow file from *GitHub* document.

```
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "The job was automatically triggered by a ${{ github.
          event_name }} event."
      - run: echo "This job is now running on a ${{ runner.os }} server
           hosted by GitHub!"
      - run: echo "The name of your branch is ${{ github.ref }} and
          your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v4
      - run: echo "The ${{ github.repository }} repository has been
          cloned to the runner."
      - run: echo "The workflow is now ready to test your code on the
          runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "This job's status is ${{ job.status }}."
```

YAML and its syntax are introduced in more details in Appendix C.

B.4.1 Trigger

It is most common that an event triggers a workflow. An event can be defined in several ways such as

- A person or a process does some operations to the *GitHub* repository, such as submitting a pull request or pushing a new commit to the repository.
- Something happened outside *GitHub* which serves as an external trigger that triggers a workflow.
- A schedule that runs a workflow at particular timestamps or periodically.
- Manually starting a workflow.

In a workflow YAML file, the trigger is defined by keyword `on`. For example, one may see

```
on:
  push:
    <...>
```

or

```
on:
  pull_request:
    <...>
```

or

```
on:
  scheduled:
    <...>
```

which define different trigger types. It is possible to include multiple trigger types in the statement using a list, such as

```
on:
  [push, pull_request]:
    <...>
```

B.4.2 Workflow

A workflow contains multiple jobs that run in parallel or in sequence. A job contains multiple steps executed in sequence. Each step is something like a line of command in shell.

Each job is associated with a "runner" which is a physical or a virtual computer or container that execute the job. All the steps defined under a job need to run in the same environment. Of course, each step can call a different software program.

The following is an example given by *GitHub* starter workflows template. Upon pushing a new commit, the workflow create a *ubuntu* environment, install conda, python and associated packages, and finally run `pytest` to check the code quality.

```
name: Python Package using Conda
on: [push]
jobs:
  build-linux:
  runs-on: ubuntu-latest
  strategy:
    max-parallel: 5
  steps:
    - uses: actions/checkout@v3
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: '3.10'
    - name: Add conda to system path
      run: |
        # $CONDA is an environment variable pointing to the root of the
            miniconda directory
        echo $CONDA/bin >> $GITHUB_PATH
    - name: Install dependencies
      run: |
        conda env update --file environment.yml --name base
    - name: Lint with flake8
      run: |
        conda install flake8
        # stop the build if there are Python syntax errors or undefined
            names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --
            statistics
        # exit-zero treats all errors as warnings. The GitHub editor is
            127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-
            length=127 --statistics
    - name: Test with pytest
      run: |
        conda install pytest
        pytest
```

This workflow example defines one job, `build-linux`. The job defines five steps, `Set up Python 3.10`, `Add conda to system path`, `Install dependencies`, `Lint with flake8` and `Test with pytest`. Each step is associated with one `run` that executes a command in the shell, for example,

```
name: Install dependencies
run: conda env update --file environment.yml --name base
```

executes a `conda` command that updates the packages defined in `environment.yml` in the base environment.

In this example, it can be seen that many things besides the commands are configured in the workflow, such as the name of the jobs and steps, the OS, the *Actions* reference, the software version, etc.

Notice that `uses: actions/checkout@v3` as the first step in the list of steps pulls in a predefined action defined at *github.com/actions/checkout*. The domain name *github.com/* can be neglected.

Further details can be configured that are not shown in this example. Just to give one example, by using

```
on:
  push:
    branches:
      - main
      - branch-1
      - branch-2
```

it can be specified which branch(es) to trigger the workflow when push a commit.

B.4.3 Execution

The workflow file needs to be saved under `.github/workflows/` in the repository, with an arbitrary name but it has to end with suffix `.yml` or `.yaml`.

Push the repository with the workflow file to *GitHub*. On *GitHub* repository dashboard under "Actions", select and enable the action so that the workflow would be triggered once the trigger event occurs.

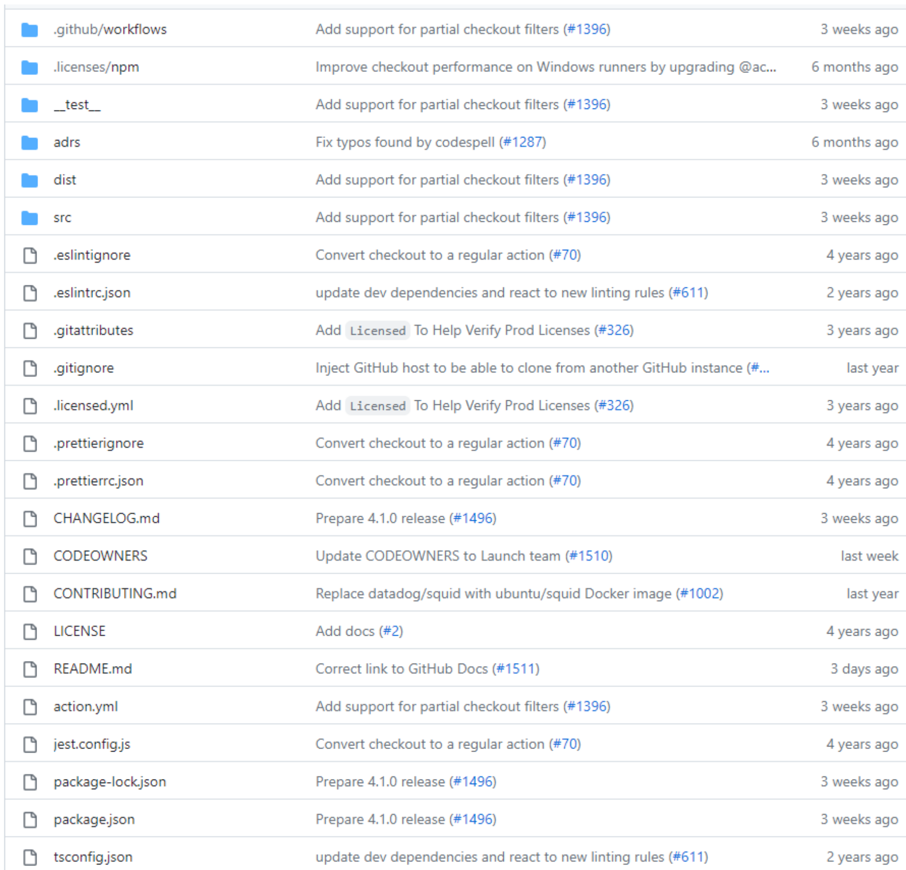
B.4.4 Very Complicated Actions

Some *Actions* workflows can be as simple as a few lines of shell commands, for example, `GitHub Action Demo` given on page 233. Others, however, can be extremely complicated that includes a large set of implementation code, test cases, building, checking for vulnerabilities, packaging, etc. An example of such complicated *Actions* flow is `actions/checkout@v3` which was used in the other example `Python Package using Conda` given in page 234.

More details of the action can be found at *github.com/actions/checkout*. As of this writing, the latest version of the code is `v4.1.0`, and in its repository it has a long list of files in TypeScript (an enhanced superset of JavaScript), YAML, JSON, shell scripts and other languages, and it is still under development. A screenshot is given in Fig. B.7. Like other actions, it stores the workflow files (for this action, there are many of them) under `.github/workflows`. The scripts that does all the actual jobs, in this action, are written in TypeScript and are stored under `src`. Supporting materials such as license, reference to contributors, etc., are also included.

To package such a complicated action, there must be an `action.yml` or `action.yaml` that contains metadata of the action. This file specifies the inputs, outputs and configurations of the action.

It would be too detailed to go through all the files in *github.com/actions/checkout*. The message here is that an action can be designed to be very complicated and powerful.



.github/workflows	Add support for partial checkout filters (#1396)	3 weeks ago
.licenses/npm	Improve checkout performance on Windows runners by upgrading @ac...	6 months ago
__test__	Add support for partial checkout filters (#1396)	3 weeks ago
adrs	Fix typos found by codespell (#1287)	6 months ago
dist	Add support for partial checkout filters (#1396)	3 weeks ago
src	Add support for partial checkout filters (#1396)	3 weeks ago
.eslintignore	Convert checkout to a regular action (#70)	4 years ago
.eslintrc.json	update dev dependencies and react to new linting rules (#611)	2 years ago
.gitattributes	Add `Licensed` To Help Verify Prod Licenses (#326)	3 years ago
.gitignore	Inject GitHub host to be able to clone from another GitHub instance (#...	last year
.licensed.yml	Add `Licensed` To Help Verify Prod Licenses (#326)	3 years ago
.prettierignore	Convert checkout to a regular action (#70)	4 years ago
.prettierrc.json	Convert checkout to a regular action (#70)	4 years ago
CHANGELOG.md	Prepare 4.1.0 release (#1496)	3 weeks ago
CODEOWNERS	Update CODEOWNERS to Launch team (#1510)	last week
CONTRIBUTING.md	Replace datadog/squid with ubuntu/squid Docker image (#1002)	last year
LICENSE	Add docs (#2)	4 years ago
README.md	Correct link to GitHub Docs (#1511)	3 days ago
action.yml	Add support for partial checkout filters (#1396)	3 weeks ago
jest.config.js	Convert checkout to a regular action (#70)	4 years ago
package-lock.json	Prepare 4.1.0 release (#1496)	3 weeks ago
package.json	Prepare 4.1.0 release (#1496)	3 weeks ago
tsconfig.json	update dev dependencies and react to new linting rules (#611)	2 years ago

FIGURE B.7
CI and CD.

C

A Brief Introduction to YAML

CONTENTS

YAML is widely used as configuration files and workflow description files. Under the scope of this notebook, YAML is used at least in *GitHub Actions* and *Kubernetes*. A brief introduction to YAML and its syntax is given here.

C.1 Overview

YAML is a human-readable data serialization language widely used for writing configuration files. The name YAML was initially interpreted as "Yet Another Markup Language" because the motivation for the authors to develop YAML was to simplify XML. However, in the later stage the authors pointed out that YAML is more of a data serialization language (like JSON) than a markup language. Hence, it is now more often known as the recursive acronym "YAML Ain't Markup Language".

Markup VS Serialization Languages

Markup languages focus on the marking up of various elements in a text document. In the early days, the editor of a book often needed to put marks on the manuscripts to show where each line should go, etc., before sending it to the publisher. These marks inspired markup languages, and hence the name.

Data serialization languages, on the other hand, focus on using texts to represent data structures. Data serialization languages like JSON and YAML are able represent "objects" such as Python dictionaries, JavaScript objects, etc., using its textual syntax. Data serialization languages are useful as they maintain the data structures, allowing a machine to decode easily. Therefore, they are widely used in configuration files, data storage, and machine-to-machine data transfer.

It is possible that markup and serialization languages overlaps in some applications. For example, XML, a typical markup language, can also be used to serialize data.

C.2 Syntax

Commonly used YAML file extensions include `.yaml` and `.yml`. When learning YAML syntax, it is helpful to compare it side-by-side with JSON, as both of them are serialization languages and can be translated from one to the other.

Like Python, YAML uses line separation and indentation as part of its syntax. This makes it reader friendly.

YAML uses `#` to lead a comment. YAML is case-sensitive. Use `---` in a new line to separate a single YAML file into multiple logical sectors.

C.2.1 Key-Value Pair

Key-value pair is the most basic syntax in YAML as follows.

```
<key>: <value>
```

For example,

```
name: myApp
port: 9000
version: 1.1
tested: true
```

YAML is able to automatically interpret the data types of different variables. In the above example, for instance, port number 9000 is identified as an integer, while application name `myApp`, a string. To enforce it to interpret values as strings, use quotation marks. Notice that `true/yes/on` and their associated `false/no/off` are regarded as boolean values.

C.2.2 Object and Nested Object

Use indention to indicate object trees. See the example below. Object can be nested.

```
<object name>:
```

```
  <key1>: <value1>
  <key2>: <value2>
  <key3>:
    <key31>: <value31>
    <key32>: <value32>
```

where object `object name` contains 3 key-value pairs. The third key `key3` is associated with a value who is a nested object that contains another 2 key-value pairs.

C.2.3 List

It is possible that the value of a key being a list of items. See the example below. Be careful with the indentation when items in the list are nested objects.

```
<list1>:
  - <item1>
  - <item2>
  - <item3>
<list2>:
  - <item1 key>: <item1 value>
  - <item2 key>: <item2 value>
  - <item3 key>: <item3 value>
<list3>:
  - <item1 key1>: <item1 value1>
    <item1 key2>: <item1 value2>
    <item1 key3>: <item1 value3>
  - <item2 key1>: <item2 value1>
    <item2 key1>: <item2 value2>
    <item2 key1>: <item2 value3>
  - <item3 key1>: <item3 value1>
    <item3 key1>: <item3 value2>
    <item3 key1>: <item3 value3>
```

In the example above, the value of `<list1>` is simply a list of 3 items, `<item1>`, `<item2>`, `<item3>`. The value of `<list2>` is a list of 3 key-value pairs in the form `<itemN key>: <itemN value>`. The value of `<list3>` is a list of 3 objects, each object containing 3 key-value pairs, in the form of `<itemN keyM>: <itemN valueM>`.

Items in a list in YAML do not need to have the same data type or object structure.

For a list whose items are primitive data types, such as integer, float, boolean, string, etc., or a mix of them, it is also possible to use `[item1, item2, ...]`. For example, the following two expressions are equivalent.

```
port:
  - 9000
```

```
  - 9001
  - 9002
```

versus

```
port: [9000, 9001, 9002]
```

where the former and later expressions are known as the block style and flow style respectively.

It is possible to have a list of only one item. Examples are given below.

```
<list1>:
  - <item1>
<list2>: [<item2>]
<list3>:
  - <item3 key>: <item3 value>
<list4>:
  - <item4 key1>: <item4 value1>
    <item4 key2>: <item4 value2>
    <item4 key3>: <item4 value2>
```

In the above example, all 4 lists, `<list1>` to `<list4>`, have 1 items. The first two lists `<list1>` and `<list2>` have primitive items `<item1>` and `<item2>` respectively. List `<list3>` has one item which is a key-value pair. List `<list4>` has one item which is an object that contains 3 key-value pairs.

C.2.4 Multi-line String

Sometimes the value to be stored in YAML can be a long or multi-line string, for example, a command. Use `|` to indicate a multi-line string. An example is given below.

```
<key>: |
  this is the first line of a string
  this is the second line of a string
  this is the third line of a string
```

A practical example is given below.

```
run: |
  ls \
  -la \
  > files.txt
```

where a bash command `ls -la > files.txt` is stored as the value of key `run`.

Do not confuse a multi-line string with a wrapped single-line string. For example,

```
<key>: |
  abc
  def
  ghi
```

is different from

```
<key>: abcdefghi
```

This is because when `|` is used, YAML automatically adds line break characters between different rows.

To express the wrap of a single string, use `>` instead of `|` as follows.

```
<key>: >
  abc
  def
  ghi
```

which is equivalent with

```
<key>: abc def ghi
```

because `>` adds a space instead of a line break character to each row.

Notice that both `|` and `>`, if used alone, will generate a line breaker in the end of the entire text. If no such line breaker is designed, use `|-` and `>-` instead.

To express a very long continuous string without even a space, consider using

```
<key>: "abc\
  def\
  ghi"
```

which is equivalent to

```
<key>: abcdefghi
```

Sometimes for better readability, one may consider storing long strings into lists, then concatenate them later using the program that takes in the YAML file.

C.3 Commonly Seen YAML Use Cases

YAML has gained its popularity among configuration files, especially when containers, cloud service and CI/CD are involved. Some commonly seen services that support YAML are given below. This is only a small portion of all the services and programs that use YAML.

- *GitHub Actions* configuration files. *GitHub Actions* uses YAML as the workflow configuration files.
- *Kubernetes* configuration files. *Kubernetes* pods, images, services, deployments, etc., have to be configured by the developer using YAML.

- AWS CloudFormation. AWS CloudFormation is the manuscript using which AWS can start and configure a service automatically so that the user does not need to do everything using the dashboard. It is useful when the user needs to run similar and repetitive services. The AWS CloudFormation instruction file is written in YAML.
- Azure pipeline, and many other CI/CD services. Many CI/CD services uses YAML for the configuration of pipelines.

Bibliography

[1] M. Gasser, *Building a secure computer system.* Citeseer, 1988.

[2] S. Grinder, “Docker and kubernetes: The complete guide,” 2023.

[3] J. Honai, “CI/CD for beginners,” 2023.