

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	xiii
Preface	xv
List of Figures	xvii
List of Tables	xix
I Linux Basics	1
1 Brief Introduction to Linux	3
1.1 Linux as an Operating System	3
1.2 A Brief History of Linux	4
1.3 Linux Distributions	5
1.3.1 Red-Hat-Based Distributions	6
1.3.2 Debian-Based Distributions	7
1.4 Linux Graphical Desktop	7
1.5 Linux Installation	8
2 Shell	9
2.1 Shell Basics	9
2.1.1 Types	10
2.1.2 Prompt	11
2.1.3 Basic Commands	12
2.1.4 Command Pipeline	14
2.2 Shell Environment Variables	16
2.3 Customization	18
2.4 Basic Shell Script	19
2.4.1 Script Structure	19
2.4.2 Basic Syntax	19
3 Text File Editing	23
3.1 General Introduction to Vim	24
3.1.1 Main Features	24
3.1.2 Vim Modes	24
3.2 Vim Customization	26
3.2.1 Shortcuts	27

3.2.2	Syntax and Color Scheme	27
3.2.3	Scrolling	28
3.2.4	Spell Check	28
3.2.5	Search Highlight	28
3.2.6	Screen Splitting	29
3.2.7	Plug Tool	29
3.3	Basic Operations	30
3.3.1	General Rule	30
3.3.2	Delete, Cut, Copy and Paste	31
3.3.3	Search in the Text	33
3.3.4	Other Commands	33
3.4	Visual Mode	34
3.5	Vim Macros	35
3.6	File Explorer and Screen Splitting	36
3.7	NeoVim	36
3.8	Other Text Editors	37
4	File Management	41
4.1	Filesystem Hierarchy	41
4.2	Hard and Soft Links	44
4.2.1	Hard Link	45
4.2.2	Soft Link	45
4.3	Commonly Used File Exploring Commands	46
4.3.1	Information Retrieval	46
4.3.2	Manipulation	48
4.3.3	Wildcard Characters	50
4.4	Access Control List	51
4.4.1	Change of Ownership	51
4.4.2	Change of Permissions	52
4.4.3	Change Default Permissions	52
4.5	Location of Command and File	53
4.5.1	Location of Command	53
4.5.2	Location of File	53
4.6	File Archive	55
5	Software Management	57
5.1	Tasks and Approaches	57
5.2	RPM Package	58
5.2.1	Brief Introduction	58
5.2.2	Package Management Using <code>rpm</code>	59
5.2.3	Package Management Using <code>yum</code> and <code>dnf</code>	59
5.3	DEB Package	61
5.4	Linux Kernel Management	62

<i>Contents</i>	vii
6 Process Management	65
6.1 General Introduction to Process	65
6.1.1 Process	65
6.1.2 Thread	67
6.2 Process Management in Linux	68
6.2.1 Monitor	68
6.2.2 Termination	70
6.2.3 Switching between Foreground and Background	70
6.2.4 Process Priority Manipulation	71
II Linux Administration	73
7 User Management	75
7.1 Root User Management	75
7.2 Regular User Management	77
7.3 Remote Login	77
7.3.1 Remote Control	77
7.3.2 Remote File Transfer	78
8 Storage Management	79
8.1 Partitions and Filesystems	79
8.2 Disk Partition Table Manipulation	82
8.2.1 Disk Partition	82
8.2.2 Disk Partition Table Manipulation	82
8.3 Mount, Unmount and Format a Partition	83
9 Services	85
9.1 Booting Process Management	85
9.2 Service Control	86
9.3 Logs	87
III Popular Services	89
10 Git	91
10.1 Introduction	92
10.2 Basic Setup	93
10.3 Local Repository Management	93
10.3.1 Initialization of a Repository	94
10.3.2 Version Tracking	94
10.3.3 Branch Management	98
10.4 Remote Repository Management	101
10.4.1 Clone and Pull: from Remote to Local	101
10.4.2 Push: from Local to Remote	102
10.5 Pull Request	103
10.6 Collaborative Development in Practice	103
10.7 GitHub Actions	105

10.7.1 Workflow Building Blocks	105
10.7.2 Supported Triggers	106
10.7.3 Jobs and Steps	107
10.7.4 Actions	110
10.7.5 Contexts	112
10.7.6 Costs	113
11 Database	115
11.1 Database and Database Management System	115
11.2 Relational Database	116
11.3 Non-Relational Database	117
11.4 Structured Query Language	118
11.4.1 Naming Conventions	118
11.4.2 Datatypes and General Syntax	120
11.4.3 Database Manipulation	123
11.4.4 Table Manipulation	123
11.4.5 Row Manipulation	126
11.4.6 Query	127
11.4.7 Trigger	135
11.4.8 SQL Demonstrative Example	136
11.5 RDB Connectivity	141
12 RDB Example: PostgreSQL	145
12.1 Brief Introduction	145
12.2 Installation and Authentication	146
12.2.1 Installation	146
12.2.2 Regular Authentication	147
12.2.3 Peer Authentication	147
12.2.4 Deployment of PostgreSQL in Container	148
12.3 Data Types	149
12.3.1 PostgreSQL-Specific Data Types	149
12.3.2 User-Defined Data Types	150
12.4 PostgreSQL Stored Procedural and Functions	151
12.5 Manipulation and Query	153
13 NoSQL Database Example: MongoDB	155
13.1 A Brief Introduction	156
13.2 Installation	158
13.3 MongoDB Shell	158
13.3.1 Select Database	159
13.3.2 Create Collection and Document	159
13.3.3 Query	162
13.3.4 Update Document	163
13.3.5 Remove Document	164
13.3.6 Advanced Query	165

<i>Contents</i>	ix
13.4 MongoDB Compass	167
13.5 Sharding and Indexing	167
13.5.1 Sharding	167
13.5.2 Indexing	167
13.6 Third-Party Connection	171
13.7 Aggregation Pipeline	172
13.7.1 Aggregation Pipeline Syntax	173
13.7.2 Stage: <code>\$match</code>	174
13.7.3 Stage: <code>\$group</code>	175
13.7.4 Stage: <code>\$sort</code>	176
13.7.5 Stage: <code>\$limit</code>	176
13.7.6 Stage: <code>\$set</code>	176
13.7.7 Stage: <code>\$count</code>	177
13.7.8 Stage: <code>\$project</code>	178
13.7.9 Stage: <code>\$out</code>	178
13.8 MongoDB Atlas	179
13.8.1 MongoDB Atlas Dashboard	180
13.8.2 Connection String	181
13.8.3 Atlas Search	182
13.8.4 Schema Pattern	184
14 Virtualization and Containerization	185
14.1 Introduction	186
14.2 Virtualization and Containerization Technologies	189
14.2.1 Virtualization	189
14.2.2 Containerization	191
14.3 Container Engine Basics	192
14.3.1 Container Engine Choices	192
14.3.2 Installation	193
14.3.3 Container Manipulation	194
14.3.4 Containerized Application Deployment Example	200
14.4 Docker Volume and Bind Mount	202
14.4.1 Volume	202
14.4.2 Bind Mount	203
14.4.3 Multiple Volumes	204
14.5 Docker Image	204
14.5.1 Dockerfile Programming	204
14.5.2 Build an Image	210
14.5.3 Docker Image Management	211
14.5.4 Docker Image Sharing with Docker Hub	212
14.6 Multi-Container Deployment and Orchestration	212
14.6.1 Communication	213
14.6.2 Docker Compose	215
14.7 Container Cloud Deployment	218
14.7.1 Docker Hosting Providers	219

14.7.2 VM-Based Approach	219
14.7.3 Managed-Service-Based Approach	221
15 Kubernetes	225
15.1 Kubernetes Basics	226
15.1.1 Infrastructure	226
15.1.2 Installation	229
15.1.3 Kubernetes Cluster Manipulation	231
15.2 Basic Kubernetes Objects	237
15.2.1 Pod Object	237
15.2.2 Deployment Object	238
15.2.3 Service Object	241
15.2.4 ConfigMap Object	242
15.2.5 Ephemeral Volume	244
15.2.6 Persistent Volume	247
15.3 Connectivity	250
15.3.1 Pod-Internal Communication	250
15.3.2 Cluster-Internal Communication	251
15.3.3 External Communication	253
15.3.4 Ingress Object	254
15.4 Kubernetes Cloud Deployment	255
15.4.1 Setup and Connect to EKS Cluster	255
15.4.2 Configure Node Group	256
15.4.3 Apply Kubernetes Configuration Files	256
15.4.4 Use Volumes	257
15.5 Kubernetes IDE and Alternatives	257
16 HTTP Server	261
16.1 Brief Introduction to Apache HTTP Server	261
16.2 Installation of Apache HTTP Server	262
16.2.1 Apache HTTP Server Installation on Host Machine .	262
16.2.2 Apache HTTP Server Execution in Container	262
16.3 Apache HTTP Server Configuration and Deployment	263
16.3.1 Virtual Host Configuration	264
16.3.2 Kerberos Authentication Configuration	265
16.4 A Brief Introduction to Website Development	265
16.5 Other Web Servers	265
IV Linux Security	267
17 Introduction to OS Security	269
17.1 Basics	269
17.1.1 Risks and Attacks	269
17.1.2 General Security Architecture	270
17.1.3 Standards and Requirements	272
17.2 Elements of Security	273

17.2.1 Security Policy	273
17.2.2 Security Mechanism	273
17.2.3 Security Assurance	275
17.2.4 Trusted Computing Base	276
17.3 Access Control	277
17.3.1 Discretionary Access Control	278
17.3.2 Mandatory Access Control	279
18 Security of Services and Applications	281
18.1 Database Security	281
18.1.1 Database Security Risks Categories	282
18.1.2 Database Access Control	283
18.1.3 Security-Enhanced DBMS Solutions in a Glance	285
18.1.4 Outsourced Database Security	286
18.1.5 Big Data Security	287
18.2 Virtualization Security	288
18.2.1 Security Concerns	288
18.2.2 VMM Security	288
A Scripts	291
A.1 Vim User Profile	291
A.2 NeoVim User Profile in Lua	292
B A Brief Introduction to YAML	295
B.1 Overview	295
B.2 Syntax	296
B.2.1 Key-Value Pair	296
B.2.2 Object and Nested Object	296
B.2.3 List	297
B.2.4 Multi-line String	298
B.3 Commonly Seen YAML Use Cases	299
C Continuous Integration and Continuous Delivery	301
C.1 Models	301
C.1.1 Waterfall	301
C.1.2 Agile	303
C.1.3 Roles in Agile-based Development	304
C.2 CI/CD Workflow	304
C.2.1 Pipeline	305
C.2.2 Continuous Integration	306
C.2.3 Continuous Delivery	306

D Commonly Used Tools and APIs for Application Development	309
D.1 Web Query	309
D.1.1 Google Custom Search	310
D.1.2 Serper Query	310
D.1.3 Playwright	311
D.2 Database Query	311
D.3 Code Execution	311
D.4 Push Notification	311
D.5 Email	311
D.6 User Interface	311
E Cloud Computing	313
E.1 Introduction to Cloud Services	313
Bibliography	315

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) were released, people from photocopy shops would start selling notebooks photocopies that they claimed 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 read than the textbooks. I guess we cannot blame the top scorers for being so smart that they sometimes made 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 were probably my schoolmates. Why would I pay money to a stranger in a photocopy shop for my friends' notebooks, rather than requesting a copy from them 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 (and 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 enough time on other irrelevant but still important and interesting contents.

To make a difference, I enforced myself reading articles beyond my comfort zone, which ended up motivating me to take notes to consolidate the knowledge. I used to work with 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 dozens of these notebooks on my bookshelf. Eventually, it came to me: why not digitizing them, making them accessible online and open-source and letting everyone read and edit it?

Similar with most open-source software, this notebook does not come with any "warranty" of any kind, meaning that there is no guarantee that every-

thing in this notebook is correct, and it is not peer reviewed. **Do NOT cite this notebook in your academic research paper or book!** If you find anything helpful here with your research, please trace back to the origin of the knowledge and confirm by yourself.

This notebook is suitable as:

- a quick reference guide;
- a brief introduction for beginners of an area;
- a “cheat sheet” for students to prepare for the exam or for lecturers to prepare the teaching materials.

This notebook is NOT suitable as:

- a direct research reference;
- a replacement of the textbook.

The notebook is NOT peer reviewed, thus is more of a notebook than a book. It is meant to be easy to read, not to be comprehensive and very rigorous.

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, should you decided to trace the origin of the knowledge.

Some of the figures in this notebook are plotted using Excalidraw, a very convenient tool to emulate hand drawings. The Excalidraw project can be found on GitHub, *excalidraw/excalidraw*. Other figures may come from MATLAB, R, Python, and other computation engines. The source code to reproduce the results are intended to be included in the same repository of the notebook, but there might be exceptions.

This work might have benefited from the assistance of large language models, which are used exclusively for editing purposes such as correcting grammar and rephrasing sentences, without introducing new content, generating novel information, or changing the original intent of the text.

Preface

I have used Microsoft Windows for decades in both daily life and research, and it has served me well. Windows offers an intuitive user interface and supports a wide range of software, to name a few that I use frequently, Adobe, MATLAB, WinEdt, Steam, Edge and Chrome. One could live without ever working on Linux directly and still get along just fine. So, what is the point of learning Linux?

On may argue that Linux is still useful for the “professionals” such as developers, data scientists and server administrators. However, even that is not entirely accurate as of today. Nowadays, many tools used by data scientists and developers, such as database servers and integrated development environments, are readily available on Windows, with some of the best ones even developed by Microsoft. Developers can work productively in countless ways without ever needing to interact with Linux directly. In fact, many industrial computers ship with Windows pre-installed and run reliably. Many front-line factory operators do not even care what operating system they are using. As long as the graphical interfaces work properly, few users are concerned about whether the backend is running Windows or Linux.

That said, learning Linux remains important, especially for researchers and professionals with an engineering or technical background.

Compared to Windows, Linux is open-source, more flexible, and highly customizable. It has a vibrant global community that actively supports its development. For certain tasks, it outperforms Windows. In some cases, software tools are easier to install and use on Linux, thanks to fewer restrictions and a more transparent system architecture.

Even if we do not interact with Linux directly on a daily basis, we frequently rely on software and tools that run on Linux environments. For example, Android is built on the Linux kernel. Containerization technologies such as Docker, Podman and Kubernetes run natively on Linux. When using them on Windows machines, they typically create a hidden Linux virtual machine in the backend. Linux knowledge is critical when developing or maintaining such software.

Beyond these reasons, the main motivation for this notebook is to show that learning Linux is not just about mastering another operating system. It provides a strong foundation for understanding computer operating systems in general, including Windows, macOS, UNIX and more. Gaining a deeper understanding of how computers and digital systems work can genuinely improve productivity. After all, we interact with these systems every day. The

next time your computer slows down, at least you will check your CPU and memory usage before jumping into buying a new one.

A list of key references of this notebook is given below. These materials are frequently cited in the notebook. For convenience, they are not given in the Reference section but listed here instead, whereas otherwise I will have to cite them everywhere in the text.

- The Linux Bible (10th edition)
- Docker & Kubernetes: The Practical Guide (2024 edition), Udemy

The majority of the content in this notebook was tested on a machine with Red Hat Enterprise Linux 9, with some exceptions from Ubuntu.

List of Figures

3.1	Mode switching between normal mode and insert mode, and basic functions associated with the modes.	25
3.2	A flowchart for simple creating, editing and saving of a text file using Vim.	26
3.3	An example of visual mode where a block of text is selected..	34
3.4	Vim (with user's profile customization as introduced in this chapter).	38
3.5	The nano editor.	38
3.6	The emacs editor.	39
3.7	The gedit editor.	39
3.8	Visual Studio Code.	40
4.1	An example of Linux file system hierarchy.	42
4.2	A rough categorization of commonly used directories in Linux file hierarchy standard.	43
4.3	Original file name, hard link and soft link.	45
4.4	List down information of files and subdirectories in the current working directory.	47
6.1	A demonstration of running multiple processes on a single-core CPU.	66
6.2	Fundamental states of a process and their transferring.	67
6.3	A demonstration of multiple threads in a process.	68
6.4	Execution of <code>ps -ef</code> command.	69
6.5	Execution of <code>top</code> command.	69
6.6	Priority levels in Linux.	72
8.1	A demonstration of how a hard drive is used in the OS.	80
8.2	A demonstration of how partitions and filesystems relate to each other.	80
10.1	Git for software development management.	92
10.2	The project directory managed by Git.	95
10.3	Two approaches of integrating branches, <code>git merge</code> VS <code>git rebase</code> . In the <code>git rebase</code> example, two rebase commands are executed consecutively, the first on the feature branch and second on the main branch.	100

10.4 A demonstration of GitHub Actions workflow	106
13.1 A demonstrative example of how MongoDB stores data as (nested) object	157
13.2 A demonstration of MongoDB Atlas dashboard	180
13.3 A demonstration of MongoDB Atlas dashboard where the databases and collections under “Project 0”, “Cluster0” are browsed. Database “ <code>sample_analytics</code> ”, collection “ <code>customers</code> ” is selected. There are 500 documents under this collection	181
13.4 Atlas search set up search index using the dashboard	183
13.5 Atlas search test	183
14.1 System architectures of PC, VM and container	187
14.2 PC implementation: a cook in a kitchen	188
14.3 VM implementation: many cooks in a kitchen, each with a different cookbook	188
14.4 Container implementation: one cook in a kitchen, handling multiple dishes, each has a cookbook and stays in its own pan	189
14.5 An example of running alpine container, with interactive TTY and name <i>test-alpine</i>	195
14.6 List the running container <i>test-alpine</i>	196
14.7 List the exited container <i>test-alpine</i>	196
14.8 A demonstration of how Dockerfile, image and container link to each other	205
14.9 A demonstration of docker image layer structure	206
15.1 Kubernetes cluster and its key components	227
15.2 An example of ingress service framework	255
15.3 Portainer login page to create admin user	259
15.4 Portainer dashboard overview of docker servers	259
15.5 Portainer dashboard overview in a docker server	260
15.6 Portainer dashboard list down of all running containers	260
16.1 Apache HTTP server test page on RHEL	263
17.1 Layer structure of an operating system	271
17.2 Reference Monitor Architecture	274
17.3 Reference Monitor Architecture [1]	276
17.4 A demonstration of access control matrix	278
C.1 Waterfall model	302
C.2 Agile model	303
C.3 Integration and delivery of a new feature. The integration corresponds with the development and delivery, operation, in the figure respectively	305
C.4 CI and CD	308

List of Tables

2.1	Commonly seen items in shell prompt as in PS1.	11
2.2	Commonly used shell environment variables.	17
2.3	Shell configuration files.	18
3.1	Commonly used modes in Vim.	25
3.2	Commonly used shortcuts to switch from normal mode to insert mode.	26
3.3	Commonly used operators related to delete/cut, change, copy and paste.	32
3.4	Commonly used motions.	32
4.1	Introduction to commonly used directories in Linux file hierarchy standard.	44
4.2	Commonly used commands to navigate in the Linux file system.	46
4.3	Commonly used arguments and their effects for <i>ls</i> command.	48
4.4	Commonly used arguments and their effects for <i>mv</i> and <i>cp</i> commands.	49
4.5	Commonly used arguments and their effects for <i>rm</i> command.	50
4.6	Commonly used wildcard characters.	50
4.7	Three types of permissions.	51
4.8	Commonly used file archive tools.	56
6.1	Some attributes of a PCB.	66
7.1	The columns in <i>/etc/passwd</i>	76
9.1	Commonly seen terminologies regarding service control.	87
10.1	Different file status in a Git managed project.	95
10.2	Commonly seen configurations in GitHub Actions workflow.	109
10.3	Commonly seen configurations under <i>jobs.<id></i>	110
10.4	Commonly seen configurations under <i>jobs.<id>.steps[*]</i>	111
11.1	An example of a relational database table.	117
11.2	A second database table in the example.	117
11.3	Widely used SQL data types.	120
11.4	Widely used SQL keywords (part 1: names).	121

11.5 Widely used SQL keywords (part 2: actions).	122
11.6 Widely used SQL keywords (part 3: queries).	123
11.7 Commonly used constraints.	124
12.1 Widely used psql commands.	154
13.1 MongoDB basic commands.	159
13.2 MongoDB basic query operators.	163
13.3 MongoDB basic update operators..	164
13.4 MongoDB arithmetic aggregation functions.	177
14.1 Commonly used docker commands to launch a container. . .	197
14.2 Critical keywords used in a Dockerfile.	208
15.1 Key components in Kubernetes master and nodes.	228
15.2 Commonly used Kubernetes object types.	238
15.3 Commonly used access modes for PVs.	248
18.1 DB security risks categories and associated security methods.	282
C.1 Roles in Agile model.	304
E.1 Levels of cloud services and the corresponding responsibilities shared between the cloud service provider and the user. . . .	314

Part I

Linux Basics



1

Brief Introduction to Linux

CONTENTS

1.1	Linux as an Operating System	3
1.2	A Brief History of Linux	4
1.3	Linux Distributions	5
1.3.1	Red-Hat-Based Distributions	6
1.3.2	Debian-Based Distributions	7
1.4	Linux Graphical Desktop	7
1.5	Linux Installation	7

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

1.1 Linux as an Operating System

Linux is one of the most widely used operating systems. An **operating system** (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 handle at least the following tasks.

- Detect and prepare hardware
- Manage process
- Manage memory
- Manage filesystem and storage
- Provide user interface and Application Programming Interface (API)
- Provide Software Development Kit (SDK) for software development

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 listed below.

- 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.

- Virtualization

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 they are 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, Microsoft Windows and macOS are all successful OSes, yet they differ in many ways. Among the three OSes, Linux is the only completely open-source OS, and can be deployed free-of-charge and customized as requested.

1.2 A Brief History of Linux

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

Many modern OSes including macOS and Linux are inspired by UNIX. UNIX 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 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)”. As AT&T was not allowed to sell computers to the end users, the only thing it could do was to license the source code to other computer manufacturers with a high price. This has prevented end users from obtaining UNIX source code. 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 (GNU)”, 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 Torvalds 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 it “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 often come with corresponding installation packages friendly to amateur users. They share the same Linux OS kernel, but differ from each other in the add-on such as software management tools and default user interfaces.

Today, there are hundreds of Linux distributions in the community. The most famous two categories of distributions are as follows.

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

- CentOS
- Debian-Based Distributions
 - Debian
 - Ubuntu
 - Raspberry Pi OS

Notice that although the source code of all the distributions above is publicly available as required by GPL (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.

The two types of distributions differ in many aspects, the most important one being the package management methods. More details are introduced as follows.

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. As of this writing, YUM is considered legacy and **Dandified YUM** (DNF) is used as its replacement. Compared with YUM, DNF has a vastly improved package dependency resolution, hence is more efficient.

RHEL is a commercial, stable and well-supported OS that can host mission-critical applications for enterprises 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 paid 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 plays as the testbed for Red Hat to interact with the community and test new

features. 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. APT 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 a larger software pool than Fedora. Ubuntu and its associated software usually have a longer lifespan than Fedora because Ubuntu serves 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.

More about RPM, YUM, DNF and APT are introduced in Chapter 5.

1.4 Linux Graphical Desktop

Graphical user interface is not necessary to run Linux OS. Nevertheless, many Linux distributions support graphical desktops for the convenience of end users. When installing these distributions, the user has the option to install graphical desktop environments along with the OS.

The most popular graphical desktop environment is probably 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. GNOME adopts a more Linux/macOS style desktop environment. KDE, on the other hand, adopts the “Windows 7” style. LXDE and Xfce are more simple in graphics presentations since 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.

Instructions of installing Ubuntu is given by [2]. Instructions of installing Fedora is given by [3]. For the use of RHEL, consult with Red Hat at [4]. 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

2.1	Shell Basics	9
2.1.1	Types	9
2.1.2	Prompt	10
2.1.3	Basic Commands	12
2.1.4	Command Pipeline	14
2.2	Shell Environment Variables	16
2.3	Customization	18
2.4	Basic Shell Script	18
2.4.1	Script Structure	19
2.4.2	Basic Syntax	19

Linux command line interface, usually known as the “shell” or “terminal”, is the most available, flexible and powerful tool for the users and programs to interact with the OS and perform actions.

Notice that the shell is not compulsory for casual users when graphical desktop environment is available. Still, it is strongly recommended that the users shall understand at least the basics of the shell since it is more flexible and powerful than the graphical desktop and hence can become handy from time to time when configuring certain software.

Linux shell will be used frequently in the remainder of this notebook.

2.1 Shell Basics

Linux’s default **Command Line Interface** (CLI), known as the **shell** or **terminal**, 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 shell is critical.

Notice that different Linux distributions may have different default configurations or built-in tools for the shell. That said, the majority shell commands introduced in this notebook can be used across commonly seen Linux distributions.

2.1.1 Types

There are different types of shells for Linux. The most widely used shell across commonly seen Linux distributions is **Bourne Again Shell**, also known as “**bash**”. It is derived from the “Bourne Shell” used in UNIX. Unless otherwise mentioned, the shell we refer to in the remainder of the notebook is bash.

An example of a shell script that calculates Fibonacci series is given below. Notice that in practice shell is mainly used for interaction with the OS and not for software development, data processing or numerical calculation. The example is only for demonstration.

```
#!/bin/bash

read -p "Enter the number of terms: " n

a=0
b=1

echo "Fibonacci series up to $n terms:"

for (( i=0; i<n; i++ ))
do
    echo -n "$a "
    fn=$((a + b))
    a=$b
    b=$fn
done

echo
```

To execute the code, either use **bash** as follows

```
$bash <filename>
```

or make the file executable and call it directly using

```
$chmod +x <filename>
$<filename>
```

The result is shown below. Here 10 is entered when it prompts for input.

```
Enter a number: 10
Fibonacci sequence up to 10 terms:
0 1 1 2 3 5 8 13 21 34
```

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 bash. In case where a Linux distribution does not come with a particular shell, the user can install it just like installing any other software.

TABLE 2.1

Commonly seen items in shell prompt as in PS1.

Variable Symbol	Description
\!	Index of the current command in the command history.
\#	Index of the current command in the active shell.
\\$	Prompt sign, “\$” for a regular user, or “#” for the root user.
\w	Path to current working directory.
\W	Base name of current working directory.
\h	Host name.
\d	Current date.
\t	Current time.
\u	Username.
\s	Shell name, for example “bash”.

2.1.2 Prompt

With a newly started shell, a string (usually containing username, hostname, current working directory, etc.) followed by either “\$” or “#” should appear. Following the string, the user can key in shell commands. The string may appear differently on different Linux distributions. An example is given below.

```
[<username>@<hostname> <working directory>]<prompt sign>
```

such as

```
[sunlu@sunlu-dev-rhel sh]$
```

The above string is called a **command prompt**, indicating the start of a user command. By default, for regular users 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 2.2 and 2.3 for details about environment variables and shell customization methods respectively. Commonly seen items are summarized in Table 2.1.

For the term **root user** or root for short, 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. For security purposes, root user shall not be used on a daily basis. For this reason, the root user’s authentication is often deactivated by default. This can be done by setting its login password to invalid.

Notice that the root user is different from a **sudoer**, the latter of which is basically a regular user equipped with **sudo privilege**. Depending on the configuration, a sudoer may temporarily switch to the root user using **sudo su** as follows.

```
<username>@<hostname>:$ sudo su
```

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

More about sudo privilege are introduced in Chapter 7.

2.1.3 Basic Commands

A typical Linux shell command looks like the following

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

The shell comes with built-in basic commands. Commonly seen commands are introduced below. Notice that applications installed on the machine may introduce additional commands as part of their CLI interfaces. Commonly used applications and their associated shell commands will be introduced in later chapters.

When login to an unfamiliar system, the first step is often to check the basic system information, such as hardware configuration, OS version, username, hostname, etc. The following commands display the basic information of the login 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 content from files or folders, in this case `<username>` from `/etc/passwd` file. The `/etc/passwd` file stores basic user information. This should return the username, the password (for encrypted password, an "x" is returned), UID, Group ID (GID), General Electric Comprehensive Operating System (GECOS) field (used to amend user information, such as address, email, etc.), home directory and default shell location of the 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. Notice that sudo privilege is recommended when using this command, to give more detailed and accurate information of the system. Sometimes displayed information can be too detailed. Use `-short` argument with the command to shorten the output.

```
$ sudo lshw
<hostname>
    description: <something>
```

```
product: <something>
vendor: <something>
...
```

The list goes on and on to include detailed information about the hardware, including mother board, CPU, cache and memory, and many more.

To retrieve OS information, consider using

```
$ cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="9.6 (Plow)"
...
```

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 -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --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 `~/.bashrc`, which is a piece of shell script that is automatically executed each time a new shell is started by the user. It is sometimes known as the **user script**.

Some commands are powerful yet difficult to use. Consider using the following two methods to check the user manual about a command.

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

Use `#` to lead a single-line comment. For multi-line comment, use `: ... :``. Examples are given below.

```
# this is a single-line comment
:'

this is a multi-line comment
this is a multi-line comment
this is a multi-line comment
,'
```

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.

Use \ to break a single-line command into multiple rows when the command is very long.

2.1.4 Command Pipeline

A truly powerful feature of the shell is its ability to redirect inputs and outputs of commands, thus enabling the commands to be chained together. Meta-characters pipe (`|`), ampersand (`&`), semicolon (`;`), dollar (`$`), parenthesis (`()`), square bracket (`[]`), less than sign (`<`), greater than sign (`>`), double greater than sign (`>>`) and error greater than sign (`>>`) are commonly used special characters in shell commands that directs the outputs of commands. 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 argument. The content in `()` is sometimes called sub-shell. For example, to count the number of files/folders in the current directory, use

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

where **echo**, as its name suggests, is used to repeat its input as the output, and **wc** counts the number of lines, words or bytes in a file.

The dollar sign \$ can also be used for simple arithmetic calculation. An example is given below.

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

The dollar sign \$ is also used to expand the value of an environment variable, either system-defined or self-defined. For this feature, more details will be given in Section 2.2. An example is \$PATH which returns the value of the PATH environment variable.

The less than sign < and greater than sign > are used to reroute the input and output of a command 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. This can be done by

```
$ echo -e "a\nc\nb\nng\nne\nf\nnd\nn" > before_sort
```

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.

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

2.2 Shell Environment Variables

Shell Environmental Variables are either system-defined or user-defined variables that the shell can access and keep track. Commonly used environmental variables are introduced below.

Variable `PATH` is one of the most important shell environmental 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 `PATH`, the environment variable that traces the locations of commands, so that they can be executed anytime from any working directory. The `PATH` environment variable is a collection of directories in the system, and it is initialized automatically when a shell is started. Check `PATH` by

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

where \$PATH gives the PATH environment of the current shell. The dollar sign \$ is used to retrieve the content of an environment variable.

The default PATH environment variable often contains the following directories.

- /bin, /usr/bin: Commonly used Linux built-in commands.
- /sbin, /usr/sbin: Commonly used Linux built-in commands for administrators.
- /home/<username>/bin: Commands defined by a user.

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. The syntax is demonstrated below.

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

There are many useful environment variables just like PATH. A summary is given in Table 2.2. Command `echo $<variable name>` can be used to check the values of these variables. Use command `env` to check a list of environment variables in the shell.

TABLE 2.2

Commonly used shell environment variables.

Variable	Description
BASH	Full pathname of the <code>bash</code> command.
BASH_VERSION	Current version of the <code>bash</code> command.
EUID	Effective user ID number of the current user, which is assigned when the shell starts, based on the user's entry in <code>/etc/passwd</code> .
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.

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` shell environmental variable, and export that variable to the current shell. 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 customizing the shell. Shell customization is introduced in Section 2.3.

2.3 Customization

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 2.3.

TABLE 2.3

Shell configuration files.

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

To customize the shell behavior for a user, the most common method is to edit their `~/.bashrc` which is executed automatically each time the user starts a shell. The file is human-readable and self-explanatory.

2.4 Basic Shell Script

We have been discussing single-line shell commands for one-time usage so far. This section briefly introduces shell script programming. A shell script is a collection of reusable shell commands to serve a specific purpose. Shell script can be used both as a standalone commands pipeline and a function that takes into arguments, processes them, and generates results.

In practice, shell scripts are mainly used for system configuration and software management. It is rarely used for data processing, as there are more handy tools such as C/C++ and Python.

2.4.1 Script Structure

A typical shell script looks like the following

```
#!/bin/bash

# <introduction to the script>

<variable 1>=<value 1>
<variable 2>=<value 2>
...

<main body>
```

where `#!/bin/bash` is known as the **shebang**, which tells the shell what program to use to execute the script. With shebang, once the script is made executable, the user can execute the script with only the script name and not specifying the program. In this example, `/bin/bash` is used as the interpreter of the script.

Item `# <comment>` adds comment to the script. It is often a good practice to briefly introduce the script in the beginning. Items `<VARIABLE>=<value>` assigns global variables used by the script and they should be listed before the main body of the script.

2.4.2 Basic Syntax

Conditional statements in bash shell script include `if` and `case`, the syntax of which is given below.

```
if [<statement>]
then
    <body>
elif [<statement>]
then
    <body>
```

```

else
    <body>
fi

case <expression> in
    <pattern> )
        <body>
        ;;
    <pattern> | <pattern> | ... )
        <body>
        ;;
    <pattern> | <pattern> | ... )
        <body>
        ;;
    *)
        <body>
        ;;
esac

```

Notice that the syntax of `case` in bash shell script looks different from the most other programming languages such as C or Java. For example, it uses single right bracket `)` to separate the case pattern and body instead of `{}` or `:`, and `;;` to terminate a body instead of `break`. This is because the syntax evolved from the older Unix shell conventions.

Commonly used loops include `while` and `for`. The syntax is given below.

```

while [<statement>]
do
    <body>
done

for <item> in <item 1> <item 2> <item 3> # general syntax
do
    <body>
done

for <index> in {<start>..<end>..<interval>} # number ranges
do
    <body>
done

```

The `for` loop can also be used to navigate files in a directory. For example,

```

for <file> in <regular expression>
do
    <body>
done

```

loops over all the files in the current directory, with the regular expression used to filter the applicable files.

A user-defined function can be created as follows:

```
<function_name>() {  
    <body>  
    [return <status code>]  
}
```

The function is called by its name.

Within the function body, \$1, \$2, and so on represent the input parameters. Note that \$0 always refers to the script name. If the function needs to return data, use `echo` or modify variables, because the `return` command is limited to exit status codes (0–255).

To call the function with arguments:

```
<function_name> <value for $1> <value for $2> ...
```



3

Text File Editing

CONTENTS

3.1	General Introduction to Vim	23
3.1.1	Main Features	24
3.1.2	Vim Modes	24
3.2	Vim Customization	26
3.2.1	Shortcuts	27
3.2.2	Syntax and Color Scheme	27
3.2.3	Scrolling	28
3.2.4	Spell Check	28
3.2.5	Search Highlight	28
3.2.6	Screen Splitting	29
3.2.7	Plug Tool	29
3.3	Basic Operations	30
3.3.1	General Rule	30
3.3.2	Delete, Cut, Copy and Paste	31
3.3.3	Search in the Text	33
3.3.4	Other Commands	33
3.4	Visual Mode	33
3.5	Vim Macros	34
3.6	File Explorer and Screen Splitting	36
3.7	NeoVim	36
3.8	Other Text Editors	37

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 even provide integrated development environment for a large set of programming languages.

Among the vast number of choices, Vim and its variations are probably the most popular text editors. Vim works perfectly in a shell environment without relying on graphical desktops, thus is adopted by many Linux distributions as the built-in default text editor. NeoVim is a popular fork of Vim, and it is introduced briefly in the end of the chapter. The majority of features introduced about Vim also apply to NeoVim.

3.1 General Introduction to Vim

Vi IMproved (Vim) is a free and open-source software developed by Bram Moolenaar et al. It is an expansion to the Vi text editor to include features such as syntax highlighting, etc., and has become the default text editor of many Unix/Linux based operating systems.

Bram Moolenaar, the original inventor of Vim, passed away on August 3, 2023 at the age of 62. His project Vim has been taken over by the contributors from the community.

3.1.1 Main Features

Some people claim Vim to be the most powerful text file editor and Integrated Development Environment (IDE) 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. It is light in size and is suitable to run even on an embedded system.
- Vim operations are done mostly via mode switching and shortcut keys without requiring mouse movement, and hence the operation efficiency can be improved.
- Vim is highly flexible and can be customized according to the user's habit, and can be integrated with third-party tools (for example, GitHub Copilot) which boost its capability.

Vim can become very powerful and convenient for the user if the user becomes proficient with it. However, Vim is not as intuitive as other text editors with graphical interfaces, and there might be a learning curve for the beginners.

3.1.2 Vim Modes

Unlike other text editors, Vim defines four different modes, namely **normal mode**, **insert mode**, **visual mode** and **command line mode**, as summarized in Table 3.1.

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
```

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.

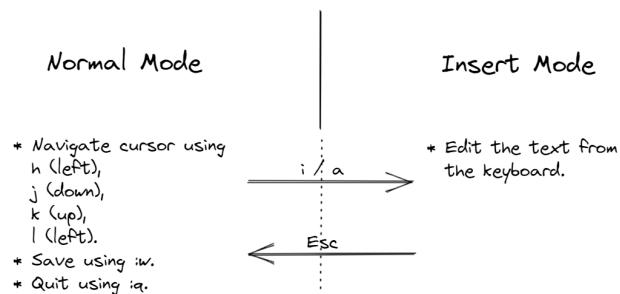
TABLE 3.1

Commonly used modes in Vim.

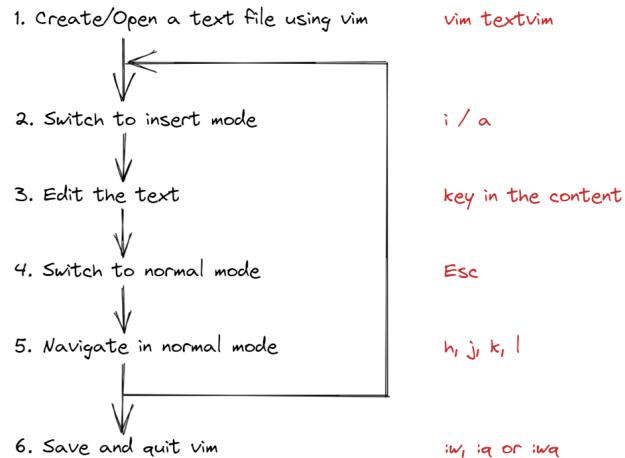
Mode	Description
Normal	Default mode. It is used to navigate the cursor in the text, search and replace text pieces, and run basic text operations such as undo, redo, cut (delete), copy and paste.
Insert	It is used to insert keyboard inputs into the text, just like commonly used text editors today.
Visual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

In the opened file, use `Esc` and `i/a` (insert/append) to switch between normal mode and insert mode. Notice that in the normal mode, the cursor is on a character. When using the insert command, the insertion cursor is put to the left of that character, whereas when using the append command, the insertion cursor is put to the right of that character. In the normal mode, use `h`, `j`, `k`, `l` to navigate the position of the cursor. If a modern keyboard is used, the left, right, up and down arrows can also be used to navigate the position of the cursor in both normal and insert modes. 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 for creating/opening, editing, saving, and quitting a text file using the aforementioned commands is given in Fig. 3.2.

**FIGURE 3.1**

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

**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.

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

3.2 Vim Customization

With the basic operations introduced in Section 3.1.2, we are able to create and edit text files of any kind. Though at this point the advantages of using Vim over other text editors are not obvious yet, the Vim editor is at least useable.

We can now customize the user profile for better user experience. Notice

that the customization is completely optional and personal. This section only introduces the idea and basic methods 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`, which will serve as the Vim configuration file. Each time Vim is started, it will automatically access this file and apply the configurations inside.

There are many readily available `vimrc` profiles shared in the community. Feel free to use them as a reference when creating a new one.

3.2.1 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 quickly (notice that consequent “`jj`” is rarely used in English). The mapping of keys and keys combinations can be done as follows in `vimrc`.

```
inoremap jj <Esc>
```

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.

3.2.2 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
```

3.2.3 Scrolling

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
```

3.2.4 Spell Check

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.

3.2.5 Search Highlight

The following customization in `vimrc` shall give a better searching experience. When searching in the text for a particular word or phrase (more about searching in the text will be introduced in later sections), to make the search results 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. This can be confusing sometimes. To tackle the issue, use command `exec "nohlsearch"` (`exec` in `vimrc` executes a command when starting a new session) after `set hlsearch` to force Vim to clear its searching memory on a new session.

To quit searching, use `:nohlsearch` in cmdline mode. For convenience, consider mapping it with a customized shortcut key such as `Space`.

Set `ignorecase` to ignore case-sensitive during the searching.

Keys `n` and `N` are used to navigate through the search results, and `zz` is used to center the cursor position on the screen. For convenience, they are mapped to `=` and `-`.

3.2.6 Screen Splitting

Screen splitting and resizing can be done in command line mode, and more details are introduced in the remainder of the chapter. Useful mappings are given below.

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 sh :set nosplitright<CR>:vsplit<CR>
noremap sl :set splitright<CR>:vsplit<CR>
noremap sk :set nosplitbelow<CR>:split<CR>
noremap sj :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-h> <C-w>h
noremap <C-l> <C-w>l
noremap <C-k> <C-w>k
noremap <C-j> <C-w>j
```

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

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

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

3.2.7 Plug Tool

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 install `curl` on Red-Hat-based distributions, use

```
$ sudo dnf install curl
```

and on debian-based distributions, use `apt` instead of `dnf`.

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.

3.3 Basic Operations

This section introduces basic Vim operations in normal mode.

3.3.1 General Rule

Many Vim commands in normal mode follow a common pattern: an operator command directly followed by a motion command, without spaces in between. For some operators, the motion argument is mandatory. The pattern is shown below.

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

The operator specifies the action (for example, delete, yank, or change), while the motion specifies the range to which the action applies (for example, a character, a word, a line, or the entire file). Some operators may work without an explicit motion, or they may have a default motion.

For example, consider the `u` command for undo. It is not an operator-motion command, but a standalone command that does not take a motion argument.

To repeat an operation multiple times, prepend the command with a number indicating how many times it should be executed.

3.3.2 Delete, Cut, Copy and Paste

Delete, cut, copy and paste are mostly done in normal mode as follows.

Quick Delete by Character

Use `x` to delete the character at the cursor, or `X` to delete the character before the cursor. To delete multiple characters, one option is to press `x` or `X` repeatedly (or hold the key). Alternatively, Vim allows automatic repetition by prefixing a count. For example, `20x` deletes 20 characters starting from the cursor. The same principle applies to other operators and motions. For example, `10l` moves the cursor 10 characters to the right.

Delete and Cut

In Vim, cutting text is equivalent to deleting it, since deleted text is stored in a register and can be pasted. Thus, all delete operations can be considered cut operations, and they are more flexible than simply using `x` or `X`.

The operator `d` deletes text and it requires a motion to define the range. For example, `d1` deletes one character to the right (the character at the cursor), while `dh` deletes one character to the left. Similarly, `d20l` deletes 20 characters to the right, where “`20l`” serves as the motion. A combination such as `5d4l` also works, producing the same result as `d20l`.

The operator `d` becomes more powerful when combined with word motions. For example, `w` is a word motion and its range is to the first character of the next word. Consider sentence “William Shakespear (c. 23 April 1564[b] – 23 April 1616) was an English playwright, poet and actor.” with the position of the cursor at “S” in “Shakespear”. Operation `dw` deletes “Shakespear”. When the cursor is in the middle of a word, `dw` deletes from the cursor to the beginning of the next word. For instance, if the cursor is on the “k” in “Shakespeare”, `dw` deletes “kespeare”. Motion `b` works similarly, except that it traces words backward to the left. To delete the whole word regardless of cursor position, use the “inner word” motion `iw`. For example, `diw` deletes the entire word under the cursor but keeps the following space.

In addition to character motions (`h, l`) and word motions (`b, w`), there are sentence motions (`(` and `)`, and paragraph motions (`{` and `}`). There are also

“inner” motions such as `is` (inner sentence), `ip` (inner paragraph), `i'`, `i"`, `i`` (inner quotations), and `i(`, `i<`, `i{` (inner blocks). They all work in a similar manner.

TABLE 3.3

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

Operator	Description
<code>x</code>	Delete (cut) the character at the cursor.
<code>X</code>	Delete (cut) the character before the cursor.
<code>dd</code>	Delete (cut) the entire current line.
<code>d</code>	Delete (cut) text according to a motion.
<code>cc</code>	Change (replace) the entire current line.
<code>c</code>	Change text according to a motion.
<code>yy</code>	Copy (yank) the entire current line.
<code>y</code>	Copy (yank) text according to a motion.
<code>p</code>	Paste the most recently yanked or deleted text after the cursor.

TABLE 3.4

Commonly used motions.

Motion	Description
<code>h, l</code>	One character left or right.
<code>j, k</code>	One line down or up.
<code>b</code>	Beginning of the current word, or the beginning of the previous word if the cursor is already at the beginning of a word.
<code>e</code>	End of the current word.
<code>w</code>	Beginning of the next word.
<code>(,)</code>	One sentence backward or forward.
<code>{, }</code>	One paragraph backward or forward.
<code>iw, is, ip</code>	Inner word, inner sentence, inner paragraph.
<code>aw, as, ap</code>	A word, a sentence, a paragraph (including trailing space or blank line).
<code>i(, i<, i[, i]</code>	Inner block for different types of brackets.
<code>i', i", i`</code>	Inner quotation (single, double, backtick).
<code>0 (zero)</code>	Beginning of the current line.
<code>\$</code>	End of the current line.
<code>gg</code>	Beginning of the file.
<code>G</code>	End of the file.

The operator `c` works similarly to `d`, except that it automatically switches to insert mode after removing the selected text.

To copy text into a register, use `y` (short for “yank”), followed by a motion

to indicate the range of text. The motions follow Table 3.4. To paste the most recently yanked or deleted text after the cursor, use `p`. No motion is required.

In addition to the motions listed in Table 3.4, another commonly used type of motion is “find by character.” For example, consider the following line of text, with the cursor starting on the letter “A”:

```
ABCDEFG;HIJKLMNOP;RST;UVW;XYZ
```

In normal mode, typing `f` followed by a character moves the cursor to the next occurrence of that character on the same line. For example, `fG` moves the cursor to the letter “G”. Similarly, `f;` moves the cursor to the first “;” after “G”. Typing `f;` again moves the cursor to the next “;” after “N”. From there, `2f;` moves the cursor to the “;” between “T” and “U”, since it repeats the motion twice.

If the command `df;` is used when the cursor is at the letter “A”, Vim deletes everything from “A” through the first “;”, resulting in the removal of “ABCDEFG;”.

3.3.3 Search in the Text

In normal mode, use `/<keyword>` followed by `Enter` to search the file for a keyword. Keys `n` and `N` are used to navigate through the search results (`n` repeats the search in the same direction, while `N` reverses it). The command `zz` recenters the current cursor line in the middle of the screen.

When a search is active, `n` and `N` can also be used as motions together with delete, change, or yank (copy) operators, as shown in Table 3.3.

3.3.4 Other Commands

Use `Ctrl+o` and `Ctrl+i` to jump backward and forward through the cursor position history, respectively. These commands only change the cursor position; they do not modify the text.

To perform a “save as,” use `:w <new path>` in command-line mode.

In command-line mode, the prefix `!` allows interaction with the shell. For example, consider the case of editing a read-only file when Vim was not started with `sudo`. A simple `:w` will fail. Instead, use:

```
:w !sudo tee %
```

Here, `tee` is a Linux command that takes standard input and writes it to a file, and `%` refers to the current file.

Another example is inserting the contents of an external file into the current buffer. Move the cursor to the desired location, then use:

```
:r !cat <filename>
```

3.4 Visual Mode

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 (default)**, **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. 3.3. 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.

```

1. Verified disks
4 $ lsblk -d -o name,rota,size,model
5 - nvme0n1 (119 GB NVMe SSD, ROTA=0) -> OS installed here
6 - sda (931 GB Toshiba HDD, ROTA=1) - empty
7

8 2. Partitioned HDD with GPT
9 $ sudo parted /dev/sda -- mklabel gpt
10 $ sudo parted -a optimal /dev/sda -- mkpart primary xfs 0% 100%
11

12 3. Formatted HDD
13 $ sudo mkfs.xfs -f /dev/sdal
14

15 4. Mounted HDD at /data
16 $ sudo mkdir /data
17 $ sudo mount /dev/sdal /data
18 $ sudo chown -R sunlu:sunlu /data
19

```

FIGURE 3.3

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-**. This will insert **prefix-** to all the lines, as if **0iprefix-** is executed in normal mode to all the lines separately, where **0** (zero) navigates to the beginning of the line, **i** switches to insert mode, and **prefix-** is entered as the content.

Alternatively, 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 exiting the insert mode using **<Esc>**, the changes will apply to all selected rows. Similar effect can be achieved.

3.5 Vim Macros

Vim macros allows the user to define a sequence of keyboard operations that can be recorded and triggered repeatedly. 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 containing the following content

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

and we would like to change the content to

```
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>Oimage: '<Esc>A'<Enter>ttl: 5<Esc>jq
```

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

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

and the cursor should be 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 file should appear as

```
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.

3.6 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.

Use `:split` and `:vsplit` for horizontal and vertical screen splitting respectively. A second split window would show up with the same text file opened. Command `splitright` and `splitbelow` is used to setup the default cursor position after splitting the screen. To navigate the cursor across different split windows, use `Ctrl+w` followed by `h`, `j`, `k` and `l`. Resize the selected split window using `:res+<number>`, `:res-<number>`, `:vertical resize+<number>`, `:vertical resize-<number>`.

3.7 NeoVim

Vim being an open-source project allows other to fork and build new projects on top of it. **NeoVim** is one of those projects. Comparing with Vim whose

code is almost all from Bram Moolenaar, NeoVim is more of a community-driven project with diversified contributors.

A potential problem with project codes coming from a single contributor is that the code is often a bit more messy than if it were written and cross-checked by multiple contributors, and that has been the main criticism Vim has received. NeoVim, on the other hand, has cleaner code base.

NeoVim is often regarded as a more cutting-edge version of Vim in the sense that new features usually come sooner in NeoVim than in Vim. NeoVim also has a better and more native support for Lua language. The configuration file for NeoVim, i.e., the counterpart of `vimrc`, can be built by Lua files. Vim is initially a single-thread application, which is fine when it is used as a text editor. However, it limits the ability of Vim calling other shell services. The user needs to wait for the shell services to end before he can start editing the document again. NeoVim, on the other hand, uses multi-thread framework, hence does not pose this issue. Later on Vim added the support to allow plug-ins to trigger multi-thread processes.

The drawback of NeoVim is that it being a younger and more collaborative project seems to be less stable than Vim.

Both Vim and NeoVim are in-development projects, and new commits are being update every day.

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 shell script that calculates the first 10 elements of Fibonacci series.

```

1 #!/usr/bin/bash
2 n=10
3 function fib
4 {
5     x=1; y=1
6     i=2
7     echo "$x"
8     echo "$y"
9     while [ $i -lt $n ]
10    do
11        i=`expr $i + 1 `
12        z= expr $x + $y `
13        echo "$z"
14        x=$y
15        y=$z
16    done
17 }
18 r=`fib $n`
19 echo "$r"
20

```

NORMAL | calculate_fib.sh sh 5% ln:1/20=61
"calculate_fib.sh" 20L, 220C

FIGURE 3.4

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

```

GNU nano 4.8          calculate_fib.sh
#!/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"

[ Read 20 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text ^T To Spell ^L Go To Line

```

FIGURE 3.5

The nano editor.

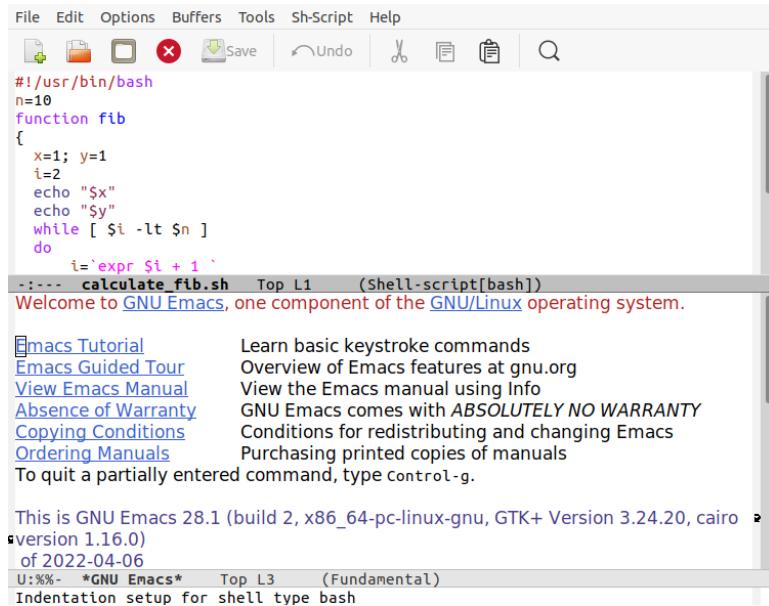


FIGURE 3.6
The emacs editor.

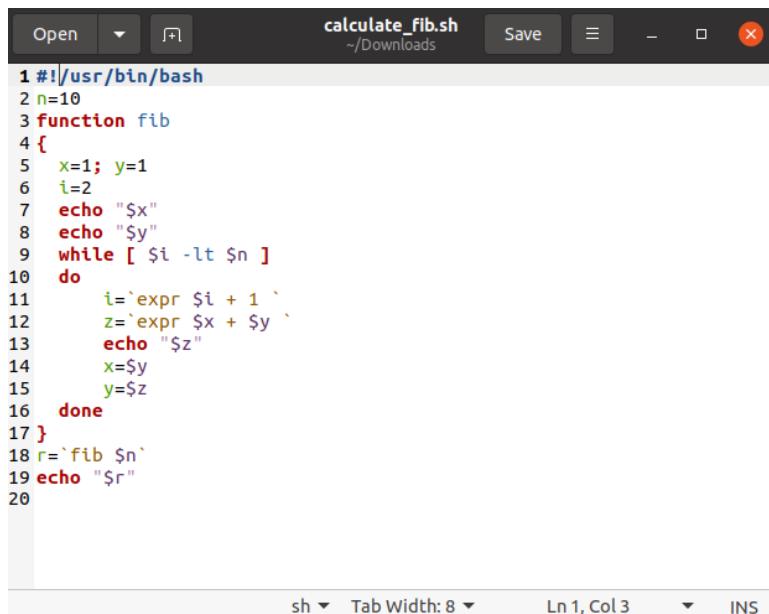
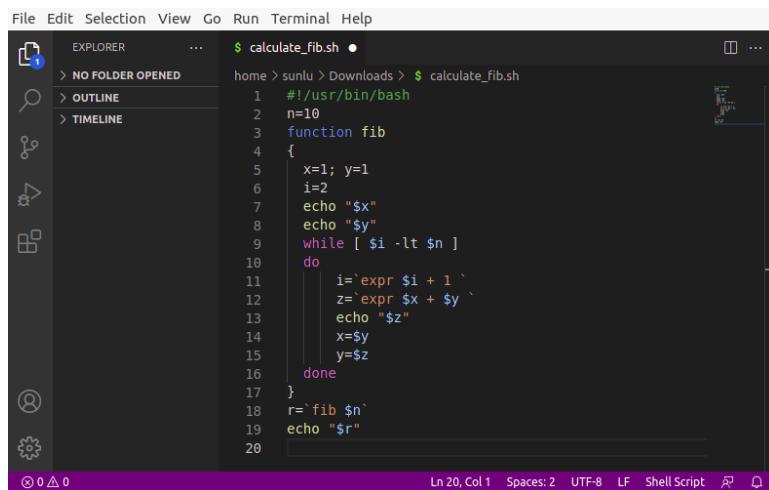


FIGURE 3.7
The gedit editor.



The screenshot shows a terminal window in Visual Studio Code displaying a shell script named `calculate_fib.sh`. The script calculates the nth Fibonacci number using a loop and arithmetic expressions. The terminal output shows the script's path and the command used to run it.

```
$ calculate_fib.sh ●  
home > sunlu > Downloads > $ calculate_fib.sh  
1 #!/usr/bin/bash  
2 n=10  
3 function fib  
4 {  
5     x=1; y=1  
6     i=2  
7     echo "$x"  
8     echo "$y"  
9     while [ $i -lt $n ]  
10    do  
11        i=`expr $i + 1`  
12        z=`expr $x + $y`  
13        echo "$z"  
14        x=$y  
15        y=$z  
16    done  
17 }  
18 r=fib $n  
19 echo "$r"  
20
```

At the bottom of the terminal window, status information is displayed: Ln 20, Col 1, Spaces: 2, UTF-8, LF, Shell Script.

FIGURE 3.8
Visual Studio Code.

4

File Management

CONTENTS

4.1	Filesystem Hierarchy	41
4.2	Hard and Soft Links	43
4.2.1	Hard Link	44
4.2.2	Soft Link	45
4.3	Commonly Used File Exploring Commands	46
4.3.1	Information Retrieval	46
4.3.2	Manipulation	48
4.3.3	Wildcard Characters	50
4.4	Access Control List	50
4.4.1	Change of Ownership	51
4.4.2	Change of Permissions	52
4.4.3	Change Default Permissions	52
4.5	Location of Command and File	52
4.5.1	Location of Command	53
4.5.2	Location of File	53
4.6	File Archive	55

File management is a big portion of OS. 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

Linux system has the **root directory** which 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>/<directory>/.../<directory or file>
```

where the first / represents the root directory, and sequential <directory>/ represents a subdirectory under its root.

Upon Linux installation, a **filesystem hierarchy** is created. 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 to store root user related documents. They are two different directories.

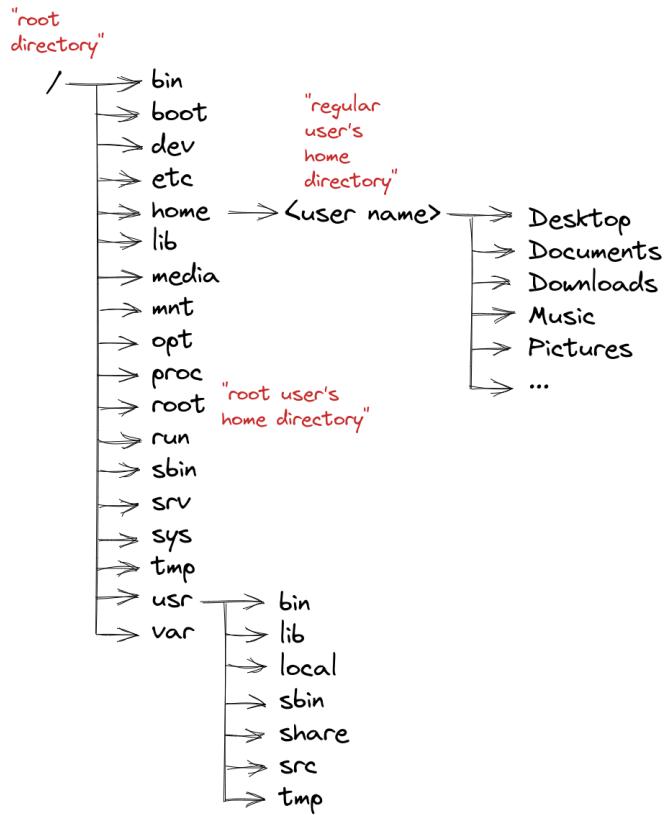


FIGURE 4.1

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 stored in the `HOME` environmental variable and can be retrieved by `$HOME`. A shortcut to `$HOME` is given by the tilde `~` for convenience. Hence, for example `ls ~` lists down the files and directories under his home directory.

As can be seen from Fig. 4.1, the hierarchy contains quite a few pre-determined subdirectories, each serving a different purpose. For the ease of

illustration, these subdirectories are roughly categorized by functionalities and accessibility shown in Fig. 4.2.

	"Used by the OS"		"Used by all users"		"Used by a specific user"
	Administration (System) Level		All-Users Level		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/src	/usr/share	/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 the 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 indicate file types, while Windows often uses suffixes to tell file types.

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. Distinguishing file types using magic numbers can be more reliable than using suffixes, though a bit less intuitive.

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.

4.2 Hard and Soft Links

The files are stored in the physical disk, and the name (including path) of the file is nothing but a “link” to that storage. The user and the OS can reach and manipulate the storage using the file name.

In addition to the original file name, Linux allows the user to create **hard link** or **soft link** (also known as **symbolic link**) to refer to a file. A demonstrative plot is given in Fig. 4.3. More details are introduced in the rest of the section.

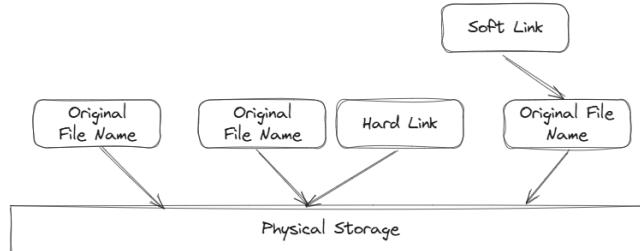


FIGURE 4.3
Original file name, hard link and soft link.

4.2.1 Hard Link

As shown in Fig. 4.3, a hard link is an alternative to the file name that directly points to the stored data. Both the original file name and the hard link share the same **inode number**. The inode number is a unique identifier of a file that Linux uses to map a file name to the stored data. From this sense, a hard link is equivalent to the original file name, and the original file name can be taken as the “default hard link”.

If a file is associated with additional hard links, removing the original file does not erase the data, and the data is still accessible via the remaining hard links. Only when all the hard links are removed, should the OS mark the storage as “free” and the data might be erased in the future when new data comes in. This is because the OS traces the link account of an inode number (the number of hard links to an inode number, the original file name included), and will consider the data removable only if it drops to 0.

Hard links can be used as a back up of the original file just in case its original file name is deleted by accident. Notice that a hard link does not create a duplication of the data referred by the original file name.

Use

```
$ ln <file> <hard link>
```

to create a hard link for a file.

4.2.2 Soft Link

As shown in Fig. 4.3, a soft link or a symbolic link is a second-layer point of the data. The soft link does not point to the data in the physical storage directly, but points to another link and forms a chain. If any links in the chain is broken, the soft link becomes a dangling link and loses the reference to the data. Notice that there is a limit to the length of the chain which effectively avoids looped chain. A larger overhead of processing time is expected as the chain becomes longer.

Soft links are useful to create shortcuts or cross-file system links.

Use

```
$ ln -s <file> <soft link>
```

to create a soft link for a file.

4.3 Commonly Used File Exploring Commands

Some of the most widely used file exploring and managing commands are summarized in Table 4.2.

In Table 4.2, `chmod` and `chown` are administration related commands that change the accessibility of a directory or a file and they will be introduced in detail in Sections 4.4.1 and 4.4.2 as part of Linux permission system. The rest of the commands are categorized into two types, query (read only) and manipulation (read and write), and are introduced in the remainder of the section.

TABLE 4.2

Commonly used commands to navigate in the Linux file system.

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

4.3.1 Information Retrieval

As given in Table 2.2, `$PWD` is the environmental variables to store the current working directory of the shell. Therefore, to print the current working directory in the console, use command

```
$ echo $PWD
```

Alternatively, use `pwd` as follows which has the same effect.

```
$ pwd
```

As one of the most frequently used commands, `ls` lists down information about the files and subdirectories in the selected directory, and by default sort the entries alphabetically. The syntax is given below.

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

An example is given in Fig. 4.4. As shown in the example, command `ls` alone shows only the name of files and subdirectories excluding hidden items and details of each item. With the additional arguments in the option field, the returns can be customized with more details displayed. For example in Fig. 4.4, the `-l` argument displays the information in long listing form, 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.

```
[sunlu@sunlu-dev-rhel ~]$ ls -l
total 16
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Desktop
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Documents
drwxr-xr-x. 2 sunlu sunlu      6 Sep  6 22:11 Downloads
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Music
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Pictures
drwxr-xr-x. 5 sunlu sunlu    75 Sep  9 13:35 Projects
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Public
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Templates
-rw-r--r--. 1 sunlu sunlu 15162 Sep  6 01:15 user.logs
drwxr-xr-x. 2 sunlu sunlu      6 Aug 26 21:53 Videos
[sunlu@sunlu-dev-rhel ~]$
```

FIGURE 4.4

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

More information can be found in 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 combine the options. For example, `ls -al` aggregates the effects of using `ls -a` and `ls -l`.

Notice that some Linux distributions may come with aliases 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.

Use `tree` to check the subdirectories structures of a directory as follows.

```
$ tree <directory>
```

Use `cat` to display the content of a text file in the console. When the text files are long, the relevant information can be difficult to find. It is recommended to use `grep` or `egrep` to query or filter the text files as follows.

```
$ cat <file / wildcards> | grep "<keyword>"
```

TABLE 4.3Commonly 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.

or

```
$ grep "<keyword>" <file / wildcards>
```

4.3.2 Manipulation

The **touch** command is used to update the timestamps of a file. If the file does not exist, **touch** creates an empty file. To do so, simply run **touch** followed by the file path:

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

For example:

```
$ touch ~/test
```

creates an empty file named **test** in the user's home directory. If only a filename is given, the file is created in the current working directory. Note that if a filename begins with “.”, it is treated as a hidden file.

Command **touch** can also create multiple files in a single command. For example:

```
$ touch test1 test2
```

creates both **test1** and **test2** in the current working directory.

To create a file containing a single line of text, the **echo** command with **>** redirection can be used:

```
$ echo '<content>' > <file>
```

For example

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

creates a simple static HTML file named `test.html` in the user home directory that displays “Hello world!” when opened in a browser.

Be aware that using `>` overwrites the file if it already exists. To append to an existing file, use `>>` instead. For appending multiple lines, use a here-document

```
$ cat >> <file> << "EOF"
<body>
...
EOF
```

where `EOF` is used as the end mark in the example, and practically it can be any string that does not appear in the body of the text.

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

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

Specifically, `-p` option of `mkdir` allows it to create nested directories along the given path if the directories do not exist.

To move a file or a directory from an existing directory to another, 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 change 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 different name.

Some commonly used arguments of `mv` is summarized in Table 4.4, many of which handling existing file with the same name in the destination directory.

TABLE 4.4

Commonly used arguments and their effects for `mv` and `cp` commands.

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

The copy-and-paste command `cp` works similarly to 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, when using **rm** the OS will keep prompting messages asking user to confirm whether to permanently delete an item. In some OS setups, it is by default forbidden to delete a directory unless it is empty. The following arguments in Table 4.5 can be used to temporarily change the behavior.

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.

It is possible, though, that items removed using **rm** can be recovered by experts. 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

```
$ shred --help
```

4.3.3 Wildcard Characters

When performing actions such as listing, moving, copying, removing, wildcard characters can be used in the path. For example, **ls a*** lists all items in the current directory that starts with letter “a”. Commonly used meta-characters are summarized in Table 4.6.

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.

4.4 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). ACL prevents unauthorized entities from accessing an item illegally. The ACL of a file can be viewed using `ls -l`. An example has been given in Fig. 4.4 in the earlier section.

The first column of the output in Fig. 4.4 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 the item type, the ACL of the item is given in the form of 9-bit permission that looks 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 accessible 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 any bit in the 9-bit permission is overwritten by a dash `-`, it means that the associated permission for the associated users is banned.

TABLE 4.7

Three types of permissions.

Directory	Description
<code>r</code>	View what is in the file or directory.
<code>w</code>	Change file contents; rename file; delete file. Add or remove files or subdirectories in a directory.
<code>x</code>	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.

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

4.4.1 Change of Ownership

Administrative privileges are required to run the `chown` command, which changes the ownership and group of a file or directory:

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

For example,

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

changes the ownership and group of the file `calculate_fib.sh` from `sunlu` to `root`. Elevated privileges are required; otherwise, the request will be rejected.

4.4.2 Change of Permissions

Both the owner and users with administrative privileges can change the permissions of a file or directory using `chmod`:

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

The new mode can be expressed symbolically. For example, `g-w` removes write permission from the group while leaving other permissions unchanged. Similarly, `go+w` adds write permission to both group and others. Here, `u`, `g`, and `o` represent “user” (owner), “group,” and “others,” while `r`, `w`, and `x` stand for “read,” “write,” and “execute.”

Alternatively, a 3-digit octal number can represent permissions. For example, 664 corresponds to the following.

- First digit (6 = binary 110): user permissions `rw-`
- Second digit (6 = binary 110): group permissions `rw-`
- Third digit (4 = binary 100): others permissions `r--`

Hence, 664 assigns permissions `rw-rw-r--` to the file.

4.4.3 Change Default Permissions

The default ACL for a newly created file or directory is defined by `umask`. Check its value by

```
$ umask
```

And change its value temporarily in the opening shell by

```
$ umask <new value>
```

The value of `umask`, after converting to binary, represents the bits in the 9-bit permission system that is disabled. For example, a `umask` value of 002 represents `rwxrwxr-x` in the 9-bit permission system, because the binary form of 002, 000000010, blocks the 8-th bit in the permission.

The default `umask` values for the root and the regular users in RHEL are 002 and 022 respectively. This is defined in `/etc/bashrc`. A user can permanently change the default value of `umask` by overwriting its value in `.bashrc` as introduced in Section 2.3.

4.5 Location of Command and File

The most frequently used 3 searching actions are as follows.

- 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

Many approaches can be used to achieve the goals, some of which are introduced as follows.

4.5.1 Location of 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'
```

4.5.2 Location of File

The package `mlocate` can be installed to enable the `locate` command, which quickly searches for files by path. As long as a file or directory's full path contains the search term, it may appear in the results.

```
$ sudo dnf install mlocate
$ sudo updatedb
$ locate <file or path fragment>
```

Here, `updatedb` must be run once after installing `mlocate` to initialize the database.

The mechanism of `locate` works as follows. The system periodically runs `updatedb` (usually once per day) to refresh an internal database containing file names and paths. The `locate` command searches this database rather than the live filesystem, making it very fast. However, this means that newly created files may not be found until the database is updated. Furthermore, not all files are included by default; the configuration file `/etc/updatedb.conf`

determines which paths are scanned and which are excluded. The first run of `updatedb` may take some time, since it must index the entire filesystem.

Users may be confused by the commands `locate` and `mlocate`. In practice, they are nearly identical for basic usage. On many systems, `locate` is provided as a wrapper or alias for `mlocate`. In the scope of this discussion, no distinction is made between them.

For security and privacy reasons, `locate` only shows files that the invoking user has permission to access. For example, a regular user cannot use `locate` to see files under `/root` or in another user's home directory.

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

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

The `<options>` argument can be used to configure how the symbolic links should be handled. With default option `-P`, the symbolic links are treated as the links themselves but not the files they are linking to, whereas `-L` does the opposite. Debugging modes and query optimization can also be enabled from `<options>`. The `<path>` argument specifies the directory from where the query is conducted. The `<expression>` argument specifies the keyword and the query method.

The following are examples of using `find` in different scenarios. The examples are taken from RedHat at [5].

Find Files by Name

Use `-name` or `-iname` to search files by their names as follows. Case sensitive

```
$ find / -name "*Foo*txt" 2>/dev/null
/home/seth/Documents/Foo.txt
```

Case insensitive

```
$ find / -iname "*Foo*txt" 2>/dev/null
/home/seth/Documents/Foo.txt
/home/seth/Documents/foo.txt
/home/seth/Documents/foobar.txt
```

Wildcards such as `*` can be used. Commonly used wildcards are given in Table 4.6.

Find Files by Content

Use `-exec` followed by `grep` to find files with specific contents.

```
$ find ~/Documents/ -name "*txt" -exec grep -Hi penguin {} \;
/home/seth/Documents/Foo.txt:I like penguins.
/home/seth/Documents/foo.txt:Penguins are fun.
```

where `-exec` allows executing a command to the findings. Notice that `grep` by itself can also be used to search files by contents.

Find Files by Type

Use `-type` to filter files by their types. To find all files,

```
$ find ~ -type f
/home/seth/.bash_logout
/home/seth/.bash_profile
/home/seth/.bashrc
/home/seth/.emacs
/home/seth/.local/share/keyrings/login.keyring
/home/seth/.local/share/keyrings/user.keystore
/home/seth/.local/share/gnome-shell/gnome-overrides-migrated
```

Specifically, to find empty files,

```
$ find ~ -type f -empty
random.idea.txt
```

To find all subdirectories,

```
$ find ~/Public -type d
find ~/Public/ -type d
/home/seth/Public/
/home/seth/Public/example.com
/home/seth/Public/example.com/www
/home/seth/Public/example.com/www/img
/home/seth/Public/example.com/www/font
/home/seth/Public/example.com/www/style
```

Use `-maxdepth` to set the searching depth as follows.

```
$ find ~/Public/ -maxdepth 1 -type d
/home/seth/Public/
/home/seth/Public/example.com
```

Find Files by Timestamp

To find files whose latest modified time is at least 30 days ago (30 days ago or earlier), use

```
find /var/log -mtime +30
```

where `-atime`, `-ctime` and `-mtime` search based on the number of days since each file was accessed, changed or had its metadata changed respectively. The alternatives `-amin`, `-cmin` and `-mmin` work similarly in minutes. The `+` indicates “at least” whereas `-` indicates “not more than”.

4.6 File Archive

The `tar`, `gzip` and `zip` commands can all be used for files archive. Each of them has unique features as given in Table 4.8.

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.

Only **tar** command is introduced here, as it is the most commonly used one among the three, and it 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

```
$ 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.

5

Software Management

CONTENTS

5.1	Tasks and Approaches	57
5.2	RPM Package	58
5.2.1	Brief Introduction	58
5.2.2	Package Management Using <code>rpm</code>	59
5.2.3	Package Management Using <code>yum</code> and <code>dnf</code>	59
5.3	DEB Package	61
5.4	Linux Kernel Management	62

Linux as an OS monitors and maintains the software installed on the system. Different Linux distributions may use different tools to manage the software.

5.1 Tasks and Approaches

When it comes to software management, an OS should support at least the following tasks.

- Install software
 - Install software from the installation package
 - If possible, automatically find and download the installation package
 - If possible, automatically find, download and install all the dependencies
- Query software
 - Query the software already installed on the system
 - If possible, query software online
- Remove software
- Update software

In Linux, software information (dependencies, version, installation instructions, etc.) is stored in special data structures known as packages. When installing software, the OS locates the packages and installs them according to the instructions that come with the packages. The OS also tracks the software it has installed, monitors their behavior, and performs updates or uninstalls as requested by the user. Different Linux distributions may use different package formats and package management tools. The most well-known ones include:

- RPM
- Debian Package Manager (DEB)

Note that RPM and DEB can refer to both the package formats and the package managing tools. There are proponents on both sides of RPM versus DEB, and both of them are very popular tools in different Linux distributions.

Linux kernel is also a software and needs to be monitored and updated from time to time. Linux kernel management is also briefly introduced in this chapter. Notice that advanced kernel management such as kernel customization goes beyond the scope of this notebook, hence are not included in this chapter.

5.2 RPM Package

RPM (`.rpm`) is the preferred package format for Red-Hat-based distributions such as RHEL, Fedora, CentOS and Oracle Linux. It uses `rpm` command to manage RPMs. In a later stage, other tools such as `yum` and `dnf` have been added to the library for better user experience and enhanced RPM facility.

5.2.1 Brief Introduction

RPM contains rich information, including not only the payload of the software such as commands, configuration files, libraries and documentation, but also metadata such as the source of the package, dependencies, etc.

The name of an RPM should follow the protocol and by itself contains important information about the software. An example is given below. Use `rpm -q` followed by the software name `python3` to query the package as follows. If the package is installed, its full name should be returned.

```
$ rpm -q python3
python3-3.9.18-3.el9_4.1.x86_64
```

where in this example the name of the software is `python3`, the version `3.9.18`, the release (build) `3.el9_4.1` with release number `3`, distribution `el9_4` and patch number `1`, and finally architecture `x86_64`. When the package is stored on the machine, it should also come with the suffix `.rpm`.

To retrieve more details, use `rpm -qi` instead and

```
$ rpm -qi python3
Name      : python3
Version   : 3.9.18
Release   : 3.el9_4.1
Architecture: x86_64
Install Date: Tue 09 Jul 2024 12:48:28 PM +08
Group     : Unspecified
Size      : 33021
License    : Python
Signature  : RSA/SHA256, Tue 21 May 2024 06:30:22 AM +08, Key ID 199
             e2f91fd431d51
Source RPM : python3.9-3.9.18-3.el9_4.1.src.rpm
Build Date : Fri 17 May 2024 10:49:06 PM +08
Build Host : x86-64-03.build.eng.rdu2.redhat.com
Packager   : Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>
Vendor     : Red Hat, Inc.
URL       : https://www.python.org/
Summary    : Python 3.9 interpreter
Description :
Python 3.9 is an accessible, high-level, dynamically typed, interpreted
programming language, designed with an emphasis on code readability.
It includes an extensive standard library, and has a vast ecosystem of
third-party libraries.
...
```

can be retrieved.

5.2.2 Package Management Using rpm

An RPM package is required to install the software. An RPM package comes from the upstream software provider, who collects source codes, binary files, and other data, builds the software, and signs the package.

The command `rpm` can be used to install the software from an RPM package. Note that installing software using `rpm` requires the precise location of the RPM package and all dependencies to be installed in advance, which can be inconvenient. The `rpm` command can also be used to validate, query, update, and remove software.

When a package is installed on the machine, its information is saved in the RPM database managed by the OS.

5.2.3 Package Management Using yum and dnf

Due to the drawbacks of `rpm` mentioned earlier, other tools have been developed to handle package upstream repositories and dependencies more conveniently. These tools are YUM and its next generation DNF, with their corresponding commands `yum` and `dnf` respectively.

YUM and DNF introduce the concept of repositories, allowing RPM packages to be part of a larger software “package tree”. When installing Linux, a list of default repositories are added to the system. The repositories list depends on the version and license type of the OS. The user can modify the repositories list manually. More will be introduced later. YUM and DNF automatically find required packages and their dependencies from the repositories when installing software. In this sense, YUM and DNF can be taken as an extension of `rpm` that come with additional features such as repository resolution and package retrieval. They run `rpm` internally when installing packages.

DNF is a further improvement of YUM with better dependency resolution and less memory usage. In many occasions, command `yum` can be replaced by `dnf` seamlessly. In some Linux distributions such as RHEL 9, `yum` is used as an alias for `dnf`. In the remaining part of the section, only DNF is discussed.

The basic syntax of DNF is

```
$ dnf <operation> <package>
```

where `<operation>` include

- `install`: install a package
- `remove`: remove a package
- `search`: search the repositories for a package
- `autoremove`: remove dependency packages not relevant to currently installed programs
- `check-update`: check updates of packages from the repositories without downloading or installing the updates
- `downgrade`: revert to a previous version
- `info`: provide the basic information of a package
- `reinstall`: reinstall a package
- `upgrade`: upgrade packages
- `exclude`: exclude a package from transaction

Each time when `dnf` starts, it checks its configuration file which is located at `/etc/dnf/dnf.conf`. Users can edit the configuration file to change the behavior of DNF, for example, to enable / disable GNU Privacy Guard (GPG) key check, the maximum number of different versions of the same software the system can keep, whether to delete dependencies when removing software, where to keep cache and log files, etc.

DNF keeps a list of repositories at `/etc/yum.repos.d` directory as its upstream software provider. The stored repositories are essentially text files named with suffix `.repo`, inside which are the name of the repository, the

URL, the GPG key, etc. To install software whose repository is not stored by default in this folder (this could happen when the software provider is a third-party and its software is not registered in the default repositories), users may need to add the repository and GPG key manually. By adding the repository configuration file and importing the GPG key, the user enables DNF to use the new repository to install and update software packages.

Other commonly used commands include

- `dnf list --installed [<package>, ...]`: list installed packages
- `dnf list --available [<package>, ...]`: list available packages
- `dnf list --upgrades [<package>, ...]`: list upgrades available for the installed packages
- `dnf list --autoremove`: list dependencies not used by any software
- `dnf -q`: query repository for information; this is a powerful and rich command whose details are too many to be covered here
- `dnf deplist <package>`: query dependencies of a package
- `dnf history`: check historical transactions
- `dnf clean all`: clean all the files from cache

A full list of `dnf` commands and their explanations can be found elsewhere at [6].

5.3 DEB Package

DEB (.deb) package is developed by Debian GNU/Linux project, and it is used by Debian and other Debian-based distributions such as Ubuntu and Linux Mint. The basic tool to install, remove and query DEB is `dpkg`, but the end users often use `apt*` commands to perform package management, rather than using `dpkg` directly.

Like RPM, DEB contains both the metadata of the software known as the control files as well as the payload of the software. A control file may look like the following [7]

```
Package: hello
Version: 2.9-2+deb8u1
Architecture: amd64
Maintainer: Santiago Vila <sanvila@debian.org>
Installed-Size: 145
Depends: libc6 (>= 2.14)
```

```
Conflicts: hello-traditional
Breaks: hello-debhelper (<< 2.9)
Replaces: hello-debhelper (<< 2.9), hello-traditional
Section: devel
Priority: optional
Homepage: https://www.gnu.org/software/hello/
Description: example package based on GNU hello
```

which is very similar to RPM package information.

More details about DEB as well as the use of `apt*` can be found at [7].

5.4 Linux Kernel Management

Different Linux distributions, though essentially using the same OS kernel, may differ when it comes to how they update and maintain the kernel. In the scope of this notebook, only RHEL is introduced in a very brief manner. A more detailed explanation to how RHEL manages kernel can be found at [8], which is not only a handbook of how RHEL manages Linux kernel, but also a good learning material for OS kernels in general.

Notice that RHEL kernel is a modified version of the upstream Linux kernel by Red Hat engineers. Therefore, it may differ from kernels used in other Linux distributions.

Package `kernel*` such as `kernel-core` are the main package of Linux kernel. Just like other packages, `dnf` can be used to query and update that package. It is also possible to install multiple kernels with different versions on the same physical machine, in which case there will be a boot entry for each kernel and the user can select which kernel to boot upon restarting the system.

To query, install and update the kernel, use

```
$ sudo dnf -q kernel-core
$ sudo dnf install kernel-<version>
$ sudo dnf update kernel
```

respectively.

Kernel modules are scripts used to extend OS functionality. The user can develop and deploy their own modules to customize the OS.

To list existing modules and to query information of these modules, with `kmod` installed, use

```
$ lsmod
$ modinfo <module name>
```

respectively.

With `kernel-devel`, `gcc` and `elfutils-libelf-devel` installed, the user can develop, test and deploy customized modules.

Linux kernel parameters are tunable values relevant to OS behavior. They can be changed in run-time or when system reboots. It is possible to address the kernel parameters via the following approach.

- Use `sysctl` command
- Edit `/proc/sys/` to change kernel parameters temporarily
- Edit configuration files in `/etc/sysctl.d/`

For example,

```
$ sysctl -a
```

displays all kernel parameters values.



6

Process Management

CONTENTS

6.1	General Introduction to Process	65
6.1.1	Process	65
6.1.2	Thread	67
6.2	Process Management in Linux	68
6.2.1	Monitor	68
6.2.2	Termination	70
6.2.3	Switching between Foreground and Background	70
6.2.4	Process Priority Manipulation	71

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, a graphical tool, to check and manage all the running processes. In a Linux system, the user can manage processes in the prompt console using bash commands.

6.1 General Introduction to Process

A **process**, also known as **task**, represents an instance of a program or an application that is being executed on the machine. The OS manages the applications and their corresponding required resources by managing the processes.

6.1.1 Process

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

The status and context of a process are stored in its **Process Control Block** (PCB). The PCB is a special data structure used to describe the meta-

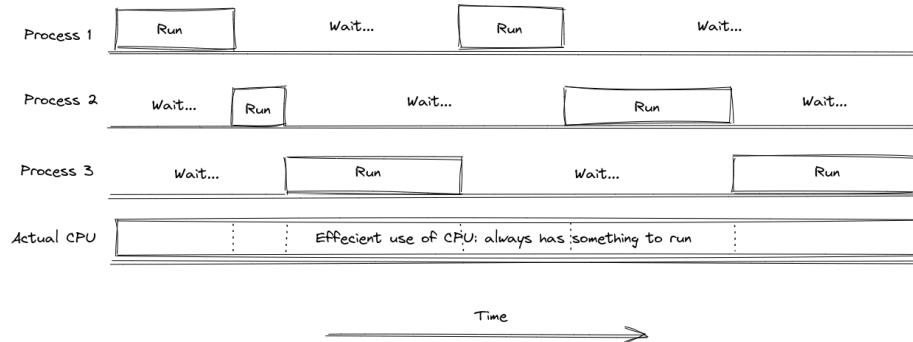


FIGURE 6.1

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

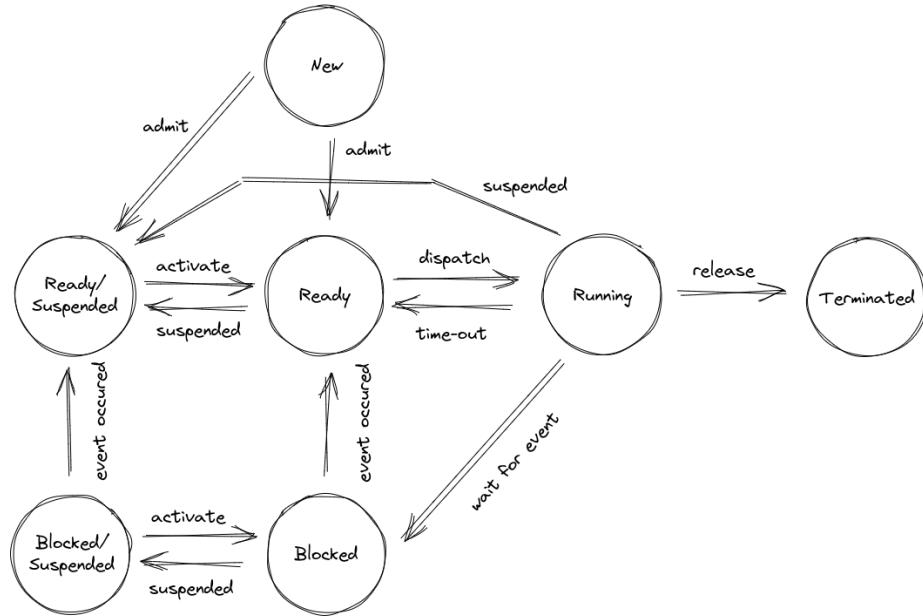
data and the dynamic of a process. The OS manages the PCBs and control the processes accordingly. Some of the attributes of a PCB are summarized in Table 6.1.

TABLE 6.1

Some attributes of a PCB.

Name	Description
Process ID	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.

Fig. 6.2 shows the possible states that a process might be at. Note the differences between “blocked” and “suspended”. “Block” 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 put on hold for some reasons. When a process is offloaded from the CPU, its context is moved from the CPU registers to the PCB of the process.

**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 have a higher priority. Based on the running environment, there are foreground processes and background processes. Based on the resources used, there are CPU processes and I/O processes.

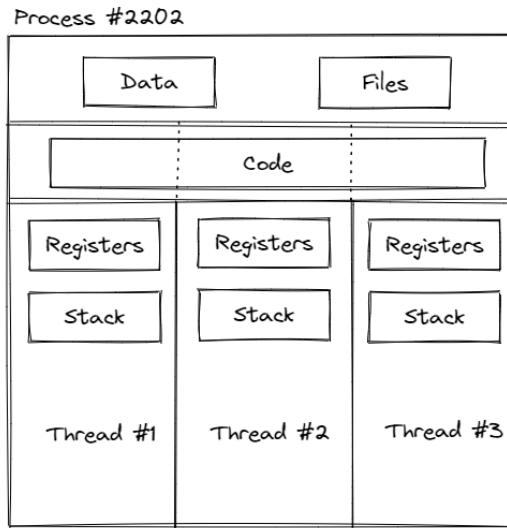
A process usually has a relatively isolated environment, and it does not share memory storage with other processes. Special inter-process communication mechanisms, which are often referred to as “pipes”, are required for processes to talk to each other. Inter-process communication requires OS level controls.

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

Basic process management commands including monitoring and terminating a process are given. Switching a process between foreground and background is also introduced.

6.2.1 Monitor

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 as follows.

```
$ ps [options]
```

or

```
$ top
```

The major difference between these two commands is that `ps` provides a snapshot (in a text format) of a list of processes, each with its name, Process ID (PID) and owner, etc., while `top` provides a dynamic and 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 -ef` 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,  0.2 sy, 86.4 id,  0.2 wa,  0.0 hi,  0.0 st,  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 inter-

acting 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.

Some commonly used options for `ps` are summarized below.

- `-e`: list all processes
- `-f`: list details of the processes
- `-C <command name>` filter by command
- `-u <username>` filter by user

6.2.2 Termination

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` (SIGKILL) and `kill -15` (SIGTERM) followed by the PID. It is mostly recommended to use `kill -15` over `kill -9`, as it allows the process to clean up and terminate properly. Most well designed software defines the protocol to terminate the process in a safe way. This is also the default termination option. Command `kill -9`, on the other hand, force the OS to terminate the process immediately. It is used mostly when the process is not responding and cannot be terminated using `kill -15`.

Another command `killall` runs similarly to `kill`, except that it sends kill signals based on the command name, not the process ID. As a result, it is possible to use it to kill multiple processes with the same command name at a time. Use it with cautions.

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.

6.2.3 Switching between Foreground and Background

To start a command as a background process, use `&` in the end of the command as follows

```
$ <command> &
```

This is useful if a command is time consuming and it does not need additional user interaction while running.

To check commands running in the background, use

```
$ jobs
[1] <status> <command>
[2] <status> <command>
[3]+ <status> <command>
[4]- <status> <command>
...
...
```

A plus sign “+” indicates that the command was the most recent one placed in the background, and the minus sign “-”, the second most recent one. To bring a background command to the foreground, use

```
$ fg %<job number>
```

where `<job number>` is the index given in `jobs` outputs. If no job number is given, the most recent job will be brought to the foreground.

6.2.4 Process Priority Manipulation

The **priority** (PR) of a process is given by an integer between -100 and 39 , the smaller the more prioritized. This value can be viewed using `top`, as shown in Fig. 6.5 under column “PR”. The process with priority -100 is displayed as `rt`, indicating the highest priority.

The entire priority range from -100 to 39 is divided into two portions. The priority between -100 and -1 is known as “real-time”, and between 0 and 39 , “normal”. In practice, real-time processes are mainly managed by the root user and are not accessible to regular users. A regular user’s process typically falls within the normal priority range of 0 to 39 .

A “NICE value” is assigned to the priorities. The priority 0 corresponds with NICE value of -20 , and 39 with 19 . The user can adjust the NICE value of his process and hence change its priority. This is demonstrated by Fig. 6.6.

For normal processes, their NICE values can be viewed using `top` as shown in Fig. 6.5 under column “NI”. Real-time processes do not use NICE value, hence NI are displayed as 0 for these processes.

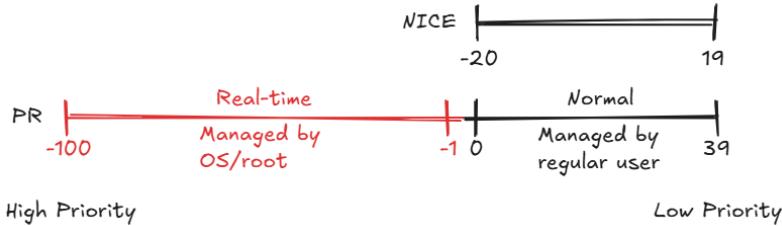
A properly designed OS should have built-in algorithms to manage process priorities and scheduling. However, tools like `nice` and `renice` are provided to give users and administrators the ability to manipulate these priorities in specific situations. Examples are given below.

To run a command with a specified NICE value, use

```
$ nice -n <increment> <command>
```

and `<increment>` will be the NICE value of the command.

To change the NICE value of an existing process, use

**FIGURE 6.6**

Priority levels in Linux.

```
$ renice -n <increment> -p <process id>
```

where **<increment>**, being either positive or negative, will be added to the current NICE value of the process with the specified process ID.

It is also possible to manage the priorities not by individual process ID, but by a group of processes. This is useful when a group of processes belongs to a similar service or to a group of users, and we want to change the priority of the service or users all together.

Part II

Linux Administration



7

User Management

CONTENTS

7.1	Root User Management	75
7.2	Regular User Management	77
7.3	Remote Login	77
7.3.1	Remote Control	77
7.3.2	Remote File Transfer	78

This chapter discusses user account management such as assigning roles and enabling access to resources.

7.1 Root User 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 it comes to system administration. A more detailed explanation is given below.

The root user is created during the 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 `$`, the latter of which is used by regular users. The root user has administrative privileges and can do almost everything without being denied or questioned by the system.

As a widely appreciated safety practice, the password for the root user is usually disabled. Therefore, nobody can login to the system as the root user using username and password authentication. In the case when administrative tasks need to be performed by the root user account, a sudoer may be able to temporarily switch to the root user using `sudo su`.

Notice that the root user does not necessarily need to use the username “root”, although it is a default convention. The root user privilege comes from the UID of 0, not the username. Theoretically speaking, it is even possible to

create multiple accounts with root user privilege by letting them share the same UID of 0, though it is very rare and not recommended.

To check the root user information, use

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

File */etc/passwd* contains basic user information of all user accounts including the root user. There are 7 columns in the file separated by :. Detailed explanations to these columns are given in Table. 7.1. It can be seen that the

TABLE 7.1

The columns in */etc/passwd*.

Index	Name	Comments
1	Username	
2	Password	The x signifies that the password stored in the file is encrypted.
3	UID	
4	GID	
5	GECOS	A comment field that stores additional information of the user, such as its full name, address, telephone, etc.
6	Home Directory	
7	Login Shell	

UID and GID of the root user are both 0.

For comparison, a regular user would have 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 regular users who can temporarily elevate their privileges and execute administration commands using

```
$ sudo <privileged command>
```

There can be multiple sudoers in the system, each with a different set of allowed privileged commands. A sudoer may be able to switch to the root user temporarily using *sudo su* and quit using *exit*.

The privileges of sudoers come 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 sudoer file, usually `/etc/sudoers`, records all the sudoers and their privileges. Notice that all the sudoers can run `sudo <command>`, but the supported commands for each sudoer can be different according to `/etc/sudoers`. The syntax is

```
<username> <host>=(<execute as user>:<execute as group>) <allowed  
commands>
```

and

```
%<group name> <host>=(<execute as user>:<execute as group>) <allowed  
commands>
```

For example,

```
# 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
```

which gives privileges to sudo group to execute all commands on every host on behalf everyone and every group.

As explained earlier, the privileges of the root user come from its UID, regardless of the sudoer file. That corresponding line with the root user in the sudoer file is only for clarity and consistency.

7.2 Regular User Management

xxx

7.3 Remote Login

This section discusses remote operations on Linux servers.

7.3.1 Remote Control

Secure Shell Protocol (SSH) is a commonly used protocol to remote login to a Linux server. To do so, SSH remote login must be enabled on the server side as a prerequisite. Use

```
ssh <user name>@<server ip / name>
```

to remote login to the server. Usually a prompt for password will pop up for authentication.

One can use SSH key pairs to login to the system without password authentication. In the client, use

```
ssh-keygen
ssh-copy-id <user name>@<server ip / name>
```

to generate an SSH key pair and copy the public key to the Linux server. Notice that the user name must already exist on the server. This process requires password authentication, but password will no longer be required in the sequential login.

7.3.2 Remote File Transfer

Secure File Transfer Protocol (SFTP) can be used to quickly transfer files between the client machine and the remote Linux server. For that, SFTP must be enabled on the server side. Use

```
sftp <user name>@<server ip / name>
```

to connect to the remote Linux server. A prompt

```
sftp>
```

shall pop up after the connection has been built.

The SFTP prompt is similar with the BASH shell prompt in terms of many supported commands, such as `pwd`, `cd` and `ls` and `help`. With these commands, the user can navigate through the server.

To fetch files from the server to the client, use

```
sftp> get <file1> <file2>
```

To fetch a directory, use

```
sftp> get -r <directory>
```

To upload files from the client to the server, replace `get` with `put` in the above commands.

In any case the file transfer fails due to internet interruption, use `reget`, `reput` to resume file transfer.

The SFTP prompt supports `df` to check disk usage, `mkdir` to create directory, `rm` to remove files, `rmdir` to remove directory, `rename` to rename files, and `chmod` to change permission and `chown` to change group of a file.

8

Storage Management

CONTENTS

8.1	Partitions and Filesystems	79
8.2	Disk Partition Table Manipulation	81
8.2.1	Disk Partition	82
8.2.2	Disk Partition Table Manipulation	82
8.3	Mount, Unmount and Format a Partition	83

Upon installation of the system, the OS shall scan the machine, look for hard drives, partition them, and mount the partitions to different locations in the OS such as / (root directory), /home/, swap space, etc.

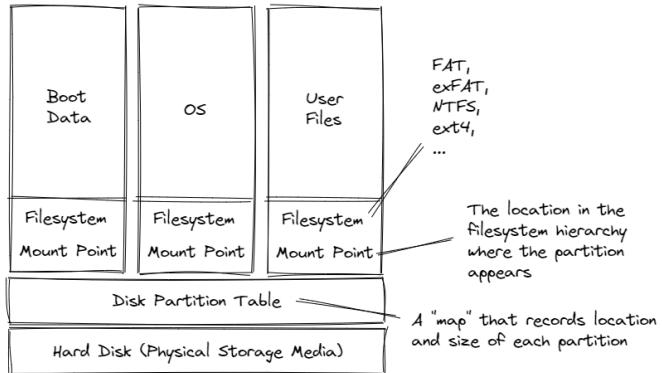
It is often not necessary for a casual user to manage the partitions and mountings by himself as there is the option to let Linux installation handle them automatically. However, when it comes to Linux servers where hardware is scaled up and down frequently, it is recommended that the administrator shall understand how the storage can be managed. Linux provides flexible tools to monitor and manage the storage of the machine, including manipulating partition table, format partition, and managing its mounting point.

8.1 Partitions and Filesystems

A **partition** is a logical slice of the hard drive managed by the OS via partition table. A **filesystem**, on the other hand, describes how OS formats data in the partition and where the partitions are mounted in the directory hierarchy. This is demonstrated by Fig. 8.1.

Usually, each filesystem is associated with a partition. The partition focuses on the logical separation of the hard drive, while the filesystem focuses on how the operating system manages the data within that partition. Ideally there should be a clear correspondence between a partition and a filesystem. However, it is possible that

- A filesystem is sized smaller than its corresponding partition, in which case the remaining space in the partition is wasted.

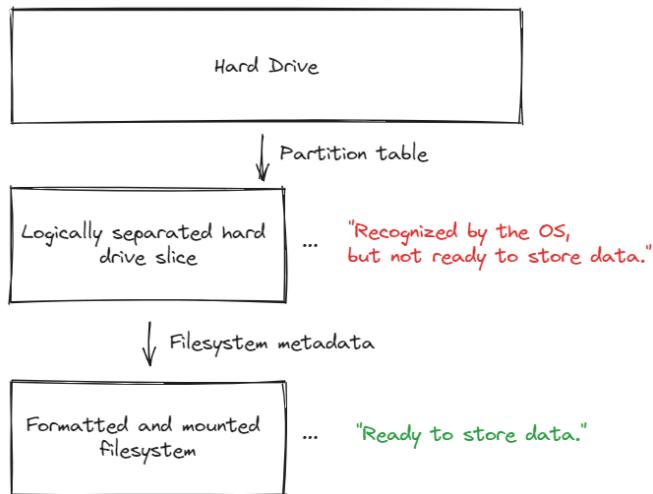
**FIGURE 8.1**

A demonstration of how a hard drive is used in the OS.

- A partition does not have its corresponding filesystem, in which case the space is wasted and cannot be used by the OS.

A filesystem must always have its corresponding partition. This is because a filesystem essentially describes how a partition is used by the OS, hence only meaningful with the partition.

A demonstration of partitions versus filesystems is given in Fig. 8.2.

**FIGURE 8.2**

A demonstration of how partitions and filesystems relate to each other.

Use `df` to monitor the status of the mounted storage, including the filesys-

tem 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
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 *916G*, is mounted on the root directory. Of this filesystem *818G* is empty and available to use.

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
          "
```

8.2 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.

8.2.1 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.

8.2.2 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.

8.3 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.



9

Services

CONTENTS

9.1	Booting Process Management	85
9.2	Service Control	86
9.3	Logs	87

This chapter discusses the booting options and services control.

9.1 Booting Process Management

When booting Linux OS during the system's startup, the following services are started sequentially.

1. **Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI)**

The BIOS or UEFI is the firmware stored on the motherboard. It is the first software executed when a computer is powered on. It checks hardware integrity and initializes system components such as the CPU, RAM, and disk controllers.

2. Bootloader (including MBR for BIOS or EFI partition for UEFI)

The bootloader is responsible for loading the operating system kernel into memory.

3. OS kernel and drivers.

Once the bootloader executes, it loads the Linux kernel into memory. The following steps then occur:

- The kernel initializes and detects hardware components.
- If necessary, an initial RAM filesystem is loaded to provide essential drivers before mounting the root filesystem.
- The root filesystem is mounted, and the kernel executes the first user-space process.

4. `systemd`

After the kernel completes its initialization, it starts the first user-space process, which is traditionally called `init`. On modern Linux distributions, this is typically `systemd`, which is responsible for managing system services, mounting filesystems, and configuring the runtime environment.

5. User-space services and `default.target`.

The final stage of the boot process involves bringing up user-space services and reaching the desired system state, known as `targets` in `systemd`. Some common `systemd` targets include:

- `multi-user.target` – A multi-user command-line environment.
- `graphical.target` – Starts the graphical user interface (GUI).
- `rescue.target` – A minimal rescue shell for troubleshooting.

To check or change the target, use the following commands respectively.

```
$ systemctl get-default
$ sudo systemctl set-default <target>
```

9.2 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 9.1 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>
```

TABLE 9.1

Commonly seen terminologies regarding service control.

Term / Tool name	Description
<code>systemd</code>	The <code>systemd</code> , i.e., <i>system daemon</i> , 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.
<code>systemctl</code>	The <code>systemctl</code> command interacts with the <code>systemd</code> service manager to manage the services. Contrary to <code>service</code> command, it manages the services by interacting with the <code>Systemd</code> process instead of running the <code>init</code> script.
<code>service</code>	The <code>service</code> 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 <code>/etc/init.d</code> scripts, Upstart's <code>initctl</code> command, and also <code>systemctl</code> .

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

9.3 Logs

System logs are critical in troubleshooting and auditing. By default, the system stores logs under `/var/log/`, and they can be managed just like managing any other text files.

Modern Linux systems also use `journalctl` utility to manage system journals. Examples include

```
$ journalctl -b # check journals from the latest reboot  
$ journalctl --since <time> --until <time> # check journals in interval  
$ journalctl -u <unit> # check journals of a particular service
```

The behaviour of `journalctl` can be configured from `/etc/systemd/journald.conf`. The default location of journals is `run/log/journal/`.

Part III

Popular Services



10

Git

CONTENTS

10.1	Introduction	91
10.2	Basic Setup	92
10.3	Local Repository Management	93
10.3.1	Initialization of a Repository	93
10.3.2	Version Tracking	94
10.3.3	Branch Management	98
10.4	Remote Repository Management	100
10.4.1	Clone and Pull: from Remote to Local	101
10.4.2	Push: from Local to Remote	101
10.5	Pull Request	103
10.6	Collaborative Development in Practice	103
10.7	GitHub Actions	105
10.7.1	Workflow Building Blocks	105
10.7.2	Supported Triggers	106
10.7.3	Jobs and Steps	107
10.7.4	Actions	109
10.7.5	Contexts	112
10.7.6	Costs	112

Git is a distributed version control system for tracking changes during the software development, and it can be used on multiple 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.

Git-based version control is widely used in association with Continuous Integration and Continuous Deployment (CI/CD). CI/CD is an important concept in nowadays software development and deployment. GitHub Actions is GitHub's CI/CD solution and it is also briefly introduced.

10.1 Introduction

Git, initially created by Linus Torvalds 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 software in distributed non-linear workflows. Git is free and open-source under GNU general public license.

Git introduces **Git repository**, which can be interpreted as the “ecosystem” Git uses to manage a project. This refers to the `.git` file associated with each Git-managed project, using which Git traces the development of the project.

With Git, all computers participating in the software development store a copy of the full-fledged Git repository locally with complete history, and they can synchronize with centralized remote servers. It introduces **branch** to manage the concurrent development of different features of the project, where the main branch (also known as the master branch) is the stable and shared repository among everyone, and the feature branches are copies of the main branch where individual features can be developed. For a feature branch, once its developed feature is approved, it can be merged back to the main branch. A demonstration is given in Fig. 10.1.

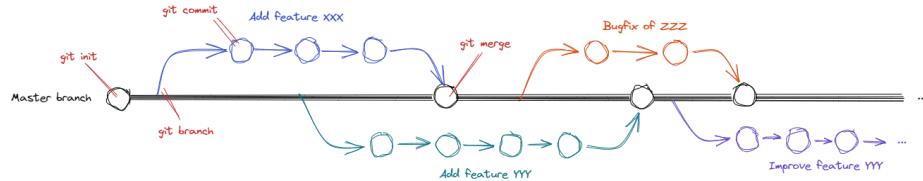


FIGURE 10.1
Git for software development management.

It is possible that multiple branches work on relevant parts in a project and cause conflicts. When merging branches, the system must be tested and the conflicts must be resolved.

CLI is often used to manage a Git repository. Some commonly used commands are shown in Fig. 10.1 with more to be introduced later. Notice that a graphical user interface is also available. However, in the scope of this notebook, command line is mostly used.

10.2 Basic Setup

Git and its relevant documents can be obtained from its official website [9]. In many Linux distributions, Git is pre-installed. If Git is missing, follow the instructions on the official website to install it. The installation procedure may differ with different Linux distributions.

To install Git on RHEL, simply use

```
$ sudo dnf install git-all
```

Upon successful installation, it is recommended to use `git config` for the necessary basic configurations such as the username and user email. Without these configurations, Git cannot run normally.

Notice that there are two types of configurations, namely the global configurations which apply to the machine and the user, and the repository configurations which apply to a particular Git repository. By default, the global level configurations are stored under `~/.gitconfig` and the repository configurations `./.git/config` of the repository, respectively. It is recommended to use Git CLI to setup the configurations, rather than modifying 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 [10].

10.3 Local Repository Management

Git CLI can manage both local and remote repositories, and can synchronize them bidirectionally. This section studies local repository management.

10.3.1 Initialization of a Repository

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

```
$ git init
```

When the above command is applied, Git creates `.git` directory in the working directory. From this point onward, Git monitors everything that happens inside this directory and its sub-directories and tries to track any change to the files, unless otherwise configured specifically. Many Git commands such as `git status` become available. More details are introduced in the remainder of the section.

10.3.2 Version Tracking

For simplicity, assume that there is only one branch in the repository, namely the main 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.

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 techniques, 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 consecutive versions.

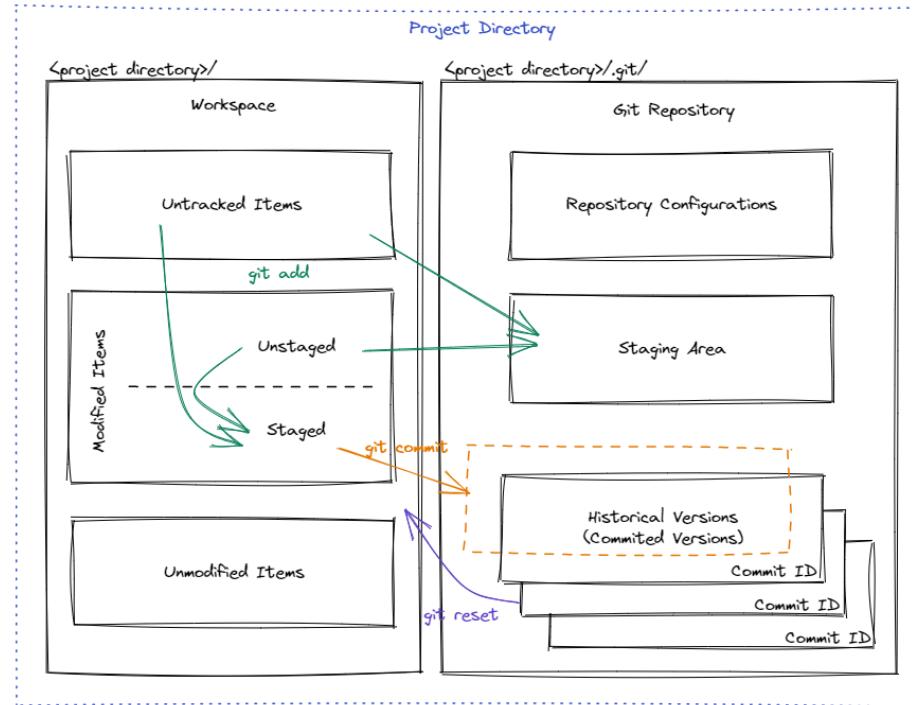
Figure 10.2 gives a demonstration of how Git manages the project directory. Git categorizes files in the workspace into the following states: untracked, modified (unstaged), staged, and unmodified. This is shown by Fig. 10.2. A brief explanation of each state is given in Table 10.1. More details are given later.

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

```
$ git status
On branch master

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

Changes not staged for commit:
```

**FIGURE 10.2**

The project directory managed by Git.

TABLE 10.1

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 staging area.
Modified (Staged)	Modified items from the last version that has been registered in the staging area. Notice that an untracked item can be staged directly, skipping “modified (unstaged)” step. Use <code>git add</code> to stage items.
Unmodified	Unmodified items from the last 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. Notice that unmodified files are not shown.

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. 10.2. 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 [10].

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. Their differences are introduced below.

Command `git reset` erases the past commits in the local branch. This should not pose any problem if the erased commits have not spread everywhere else. Therefore, `git reset` is helpful for local repositories without upstreams, or for modifications that have not been pushed to the upstreams.

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 shortcut `HEAD` the current position of the `HEAD` (commit pointer), usually the latest commit.

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` may pose an issue if the commits to be erased have already spread to the upstreams. This is because the erased commits in the local repository by `git reset` will be reverted back the next time the local repository synchronizes with its upstream, where the commits in the upstream will be considered more up to date, hence pulled to the local repository.

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.

10.3.3 Branch Management

Git branch is a core concept and feature 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. Remote branch is often associated with local branches, and play as the upstream mirror of the later for sharing and synchronizing the project development across machines. This section focuses on local branches. Remote branches will be introduced in Section 10.4.

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. A use case of the command is to merge the features developed in a feature branch to the main branch, after a pull request is raised by the feature branch. To do that, checkout to the main branch, and use

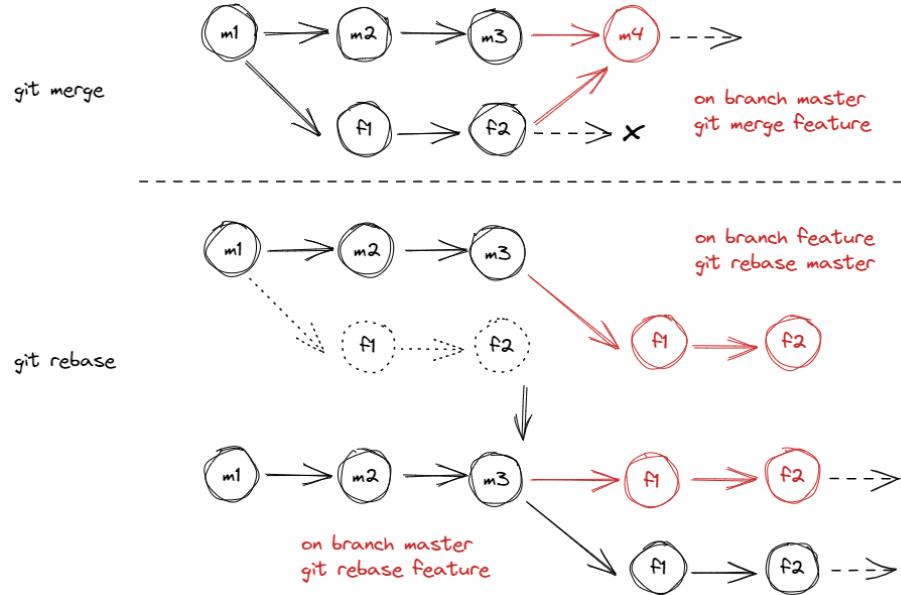
```
$ git merge <feature branch name>
```

By default, Git automatically deletes the merged branch. This behavior can be changed in the configuration. More about pull request are introduced in Section 10.5.

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. 10.3 explains the differences between `git merge` and `git rebase`. It is clear from Fig. 10.3 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`.

**FIGURE 10.3**

Two approaches of integrating branches, `git merge` VS `git rebase`. In the `git rebase` example, two rebase commands are executed consecutively, the first on the feature branch and second the main branch.

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, \dots, 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, \dots, A_n on the reset *A* in a sequential manner. In the case where there are conflicts when reapplying A_1, A_2, \dots, A_n , the user needs to jump in and resolve them 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.

10.4 Remote Repository Management

Git hosting servers provide platforms for the developers to share their Git repositories and collaboratively develop projects. There are many cloud-based public Git hosting service providers such as GitHub, GitLab and Gitee. In this section, GitHub is studied. Notice that under the scope of this section, only basic functions are introduced and they should apply similarly to other Git hosting service providers.

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.

10.4.1 Clone and Pull: from Remote to Local

Consider the case where there is already a remote repository, either under the user's GitHub account or from the community. The user can clone the repository, i.e., make a local copy of the repository, as follows.

Use

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

where `repository URL` can be the HTTPS URL of the repository.

Git is able to trace the URL from which a local repository is cloned. To maintain the local repository up-to-date with the remote repository, regularly use

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

to pull the latest update.

Should I use `git pull` or `git fetch`?

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.

With that being said, if the user simply wants to keep a well-maintained community project up to date, `git pull` should work just alright.

However, this does not guarantee that the user can perform a `git push` command to push changes made in the local repository back to the remote source. For that, additional authentication is often required. More about `git push` and remote repository authentication methods are introduced in later sections.

10.4.2 Push: from Local to Remote

Consider a scenario where you have a well-developed local Git repository, and you have just created an empty remote repository on GitHub. The goal is to link the remote repository with the local repository so that changes can be synchronized.

To establish a connection between the local and remote repositories, navigate to the local repository and run

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

This command registers the remote repository's URL in the local Git configuration. A commonly used remote name is "origin".

Note that if the local repository was originally created by cloning an existing remote repository, then "origin" is likely already set to the source of the clone. Use `git remote -v` to check the registered remote source.

The next step is to associate the current working branch with a branch in the remote repository. This ensures that `git pull` and `git push` operate on the correct branch by default. This can be done as follows.

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

Alternatively, the upstream branch can be set when pushing it for the first time using

```
$ git push --set-upstream <remote-name> <branch-name>
```

Once the upstream is configured, the user can use

```
$ git push
```

to push the changes in the local repository to the remote upstream. If there is no remote branch with the same branch name as the local repository, a remote branch with that name will be created when the local branch is pushed.

The Names of Corresponding Local and Remote Branches

A local branch and its corresponding remote branch usually have the same name by default, but this is not necessarily always the case. To set an upstream with a different name, use

```
$ git branch --set-upstream-to=\<remote-name>/<remote-branch-name> <local-branch-name>
```

or

```
$ git push <remote-name> <local-branch-name>:<remote-branch-name>
```

When performing `git push` for the first time, authentication is required. This allows GitHub or another hosting service to verify the identity of the user before accepting changes. Common authentication methods include

- Username and password authentication. Notice that this has been deprecated by GitHub in favor of token authentication.
 - Personal access tokens.
 - SSH key authentication.
-

10.5 Pull Request

In a collaborative project, the main branch is managed very carefully because everything on that branch will affect the project in the production environment. There is often a group of senior developers maintaining the main branch. Any code to be added to the main branch must be reviewed by one or a few of them.

When a feature branch wants the features developed on that branch to be merged to the main branch, it does not push the changes to the main branch directly. Instead, the owner of the feature branch needs to raise a “pull request” from the GitHub dashboard. As its name suggests, it requests the main branch to pull the updates from the feature branch. The main branch maintenance team will then view and check the feature branch, making sure that everything is correct before they approve the merging.

If the pull request is approved, GitHub will perform the merge in the cloud. Neither the owner of the feature branch nor the maintenance team needs to run the merge manually.

10.6 Collaborative Development in Practice

This section introduces the common procedures an individual contributor follows to make improvements to a community project. We assume that an individual contributor has found a community project hosted on GitHub that interests him and believes he can update the code to improve it.

The following is a general flow where an individual contributor helps with improving an open-source project.

1. The contributor forks the project to his own GitHub account.
2. The contributor clones a copy of the project from his own GitHub account to his own machine.
3. The contributor creates a new branch about the feature he wants to add or improve.

4. The contributor develops the feature on the branch.
5. The contributor commits the development and pushes the commit to the forked repository.
6. The contributor submits a pull request to the maintenance team of the community project.
7. The maintenance team receives the pull request and reviews the changes made to the code in the developer's forked repository.
8. The maintenance team either sends feedback to the contributor for further clarification, or pulls the changes from the developer's forked repository and merge the new feature branch with the existing branches.

As a first step, the contributor forks the project into his own GitHub account. This creates a copy of the community project under the contributor's account, giving him full control over the forked repository. He is free to make or test any modifications without affecting the original project.

Although a fork is treated as a separate repository in many ways, GitHub tracks the relationship between the fork and the original project. This allows the contributor to later request that his modifications be merged back into the original community repository. Using the fork approach also acknowledges the contributions of the original authors, making it a more respectful and transparent process compared to simply downloading the project and uploading it as a new repository under the contributor's account.

After forking, the contributor can clone the project from his GitHub account to the local computer. He can then modify and test the code locally. Once the changes are made, he can upload the updated code to their forked repository using

```
$ git push
```

Is it possible to push to the original repository directly?

It is worth mentioning that if `git fork` were not used to fork the repository to the contributor's GitHub account, this `git push` will run into a permission issue. This is because the contributor does not possess ownership to the original repository, hence prevented from modifying it.

It is not recommended, but possible, that the contributor can contact the owner of the repository and ask for permission. The owner of the original repository can invite the contributor as a collaborator of the project. In that case, the contributor will be able to create and push branches to the original repository. Notice that likely the contributor cannot merge his branch to the main branch and still needs pull request to do so.

When the forked repository has been updated with improvements, the

contributor can create a pull request in the original community repository. The maintainers of the community project will review the pull request, evaluate the modifications, and either reject it, provide feedback, or approve it for merging.

It is possible that while the contributor is working on their changes, new features or updates are introduced in the original community project. To keep their fork up-to-date, the contributor can add the original repository as a second remote, commonly named “upstream”. The forked repository under their account is typically referred to as “origin”. To synchronize with the original repository, the contributor can execute the following commands.

```
$ git fetch upstream  
$ git checkout master  
$ git rebase upstream/master
```

By following this procedure, the contributor ensures that his work remains aligned with the latest developments in the community project, reducing the chances of conflicts and making their contributions easier to integrate.

10.7 GitHub Actions

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 fast and convenient integration and deployment. Lately, GitHub introduced **GitHub Actions**, its own CI/CD solution, as a response to developers’ requests.

GitHub Actions is essentially a workflow automation service that allows the user to define a sequence of actions that can be triggered by timers or repository-related events. The workflow can be used for both CI/CD tasks such as code building, testing and deployment, and for managing the repository such as adding labels. Machine-readable instruction files are used to configure the workflow and they are included in the repository and managed together with the source code.

This section gives a brief introduction to GitHub Actions. More details are given in [10].

10.7.1 Workflow Building Blocks

The basic building blocks of GitHub Actions include

- Workflows
- Jobs

- Steps

A demonstrative plot is given in Fig. 10.4.

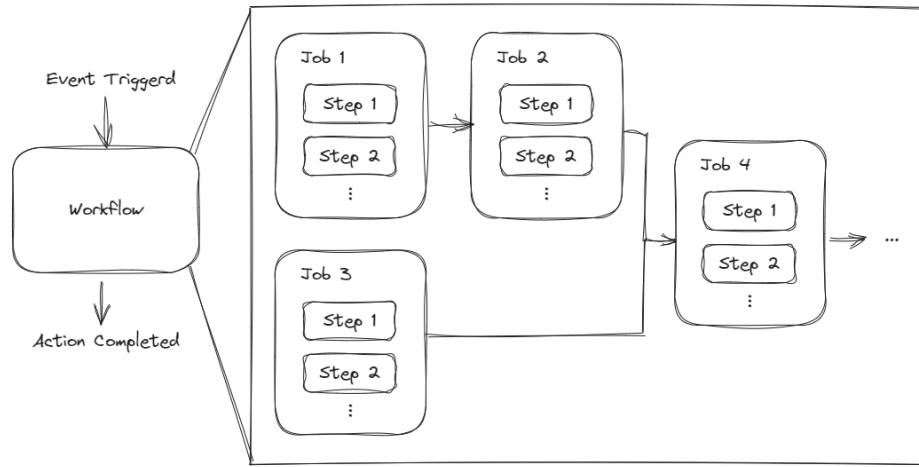


FIGURE 10.4

A demonstration of GitHub Actions workflow.

A workflow consists of jobs to be executed when a trigger happens. The jobs are executed in parallel by default (see “job 1” and “job 3” in Fig. 10.4), or in sequence when there are dependencies (see “job 2” and “job 4” in Fig. 10.4). Each job is executed in a dedicated VM or container known as a “runner”. The running environment of a runner, such as OS, libraries, environment variables, etc., can be configured separately. In each job multiple steps can be defined. A step often corresponds with an action, such as executing a shell script. The steps in a job are executed in sequence in the configured order. Both steps and jobs can be triggered conditionally. There can be any number (at least 1) of steps in a job, any number (at least 1) of jobs in a workflow, and any number of workflows attached to a branch.

The user needs to formulate the pipelines to be executed automatically into GitHub Actions workflows.

10.7.2 Supported Triggers

It is worth mentioning that GitHub Actions can be triggered not only by repository-related events. Commonly seen triggers include

- Repository-related events, such as push, pull request (most commonly seen).
- External events.

- Schedulers or timers.
- The user who triggers the workflow manually.

GitHub Actions provides workflow templates of different task types. There are at least the following workflow types as of this writing. They can be used for but not limited to CI/CD.

- CI
- Deployment
- Testing
- Code quality
- Code review
- Dependency management
- Monitoring
- Automation
- Utilities
- Pages
- Hugo

A full list of GitHub events that can trigger GitHub Actions is given in [10].

10.7.3 Jobs and Steps

The workflows configuration files are YAML files that the user can either prepare from scratch or modify from existing templates on GitHub Marketplace, a platform where commonly used workflows are shared.

A workflow configuration file contains at least the following information:

- The trigger of the workflow.
- The environment to execute the workflow.
- The actions in the workflow.

The YAML files should follow the syntax required by GitHub and be saved under `.github/workflows/` so that GitHub is able to detect them as workflows and execute them accordingly.

An official demonstrative example of a GitHub workflow YAML file is given below [10].

```

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 }}."

```

Save the above YAML file under `.github/workflows/` and GitHub would recognize the workflow. Click “Actions” on GitHub dashboard to check the details of the workflow.

A brief introduction to YAML file syntax for GitHub Actions is given below.

In the workflow of the demonstrative example only one job, namely “Explore-GitHub-Actions”, is defined. In practice, multiple jobs can be defined in a workflow. An example from [10] is given below where 3 jobs are defined.

```

jobs:
  setup:
    runs-on: ubuntu-latest
    steps:
      - run: ./setup_server.sh
  build:
    needs: setup
    runs-on: ubuntu-latest
    steps:
      - run: ./build_server.sh
  test:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - run: ./test_server.sh

```

Jobs are by default independent and can run in parallel unless `needs` field is used just like in the above example, where “build” depends on “setup”, and “test” on “build”.

Commonly seen configurations in a workflow YAML file are given in Table 10.2.

TABLE 10.2

Commonly seen configurations in GitHub Actions workflow.

Field	Description
<code>name</code>	The static name of the workflow.
<code>run-name</code>	The name of an instance of run generated by the workflow.
<code>on</code>	The event(s) that triggers the workflow. A full list of supported events are given in [10]. Multiple triggers are supported. The trigger can be configured very flexibly.
<code>permissions</code>	Allowed r/w/x permissions for the workflow or for different triggers defined by the workflow.
<code>env</code>	Environment variables to be used in the jobs descriptions.
<code>defaults</code>	Default settings for all jobs.
<code>concurrency</code>	Concurrency group of the workflow. There can be only one workflow running concurrently within a concurrency group.
<code>jobs</code>	A dictionary of jobs. Each entry corresponds with a job.

The `jobs` field is a dictionary of jobs that run in parallel (unless `needs` is used in the job description). Each entry in the dictionary corresponds with a job. The key of the entry is referred as `<id>` of the job. A demonstrative example is given below.

```
jobs:
  job1:
    name: <something>
    runs-on: <something>
    steps:
      - <something>
      - <something>
  job2:
    name: <something>
    runs-on: <something>
    steps:
      - <something>
      - <something>
```

Commonly seen fields defined under `jobs.<id>` are summarized in Table 10.3.

A specific action is defined under `jobs.<id>.steps[*]`. Notice that field “steps” is a list, whereas each item in the list is a dictionary. Commonly seen fields of the dictionaries are summarized in Table 10.4.

TABLE 10.3Commonly seen configurations under `jobs.<id>`.

Field	Description
<code>name</code>	Name of the job.
<code>needs</code>	Dependencies of the job. The job executes only if its dependencies finish successfully.
<code>if</code>	Prerequisite of the job.
<code>runs-on</code>	The type of machine to host the job, such as <code>ubuntu-latest</code> , <code>windows-latest</code> , <code>macos-latest</code> .
<code>environment</code>	Environment that the job references.
<code>concurrency</code>	Concurrency group of the job.
<code>outputs</code>	A map of outputs of the job to variables which can be passed to downstream jobs.
<code>env</code>	A map of variables to be shared by all steps in the job.
<code>timeout-minutes</code>	Maximum number of minutes of the job.
<code>strategy</code>	Testing the job on multiple machines with different configurations specified by <code>strategy</code> , for example, different OSs or versions.
<code>continue-on-error</code>	Preventing the workflow from failing with the job, if set to “true”.
<code>container</code>	Container configurations. If specified, some of the steps can run in the container; otherwise, all steps will run directly on the host specified by <code>runs-on</code> . Supported only if Linux OS is used.
<code>steps</code>	A list of steps or actions to be executed in the job.

10.7.4 Actions

From earlier examples, we have seen `run` used in a step to execute a command. There is a limitation to the complexity of what a command can achieve. Therefore, action is introduced. In short, action is a user-defined application that performs a complex and frequently repeated task, and it is well appreciated in GitHub Actions. The basic syntax of calling an action in a step is given below.

```
steps:
  - uses: <action id and version>
    with:
      <configurations>
```

where `with` is usually an optional argument that states the configuration of the action. Its contents differ from action to action.

It is worth mentioning that since actions are essentially applications, they can be packaged and shared in the community. The user can build his own

TABLE 10.4Commonly seen configurations under `jobs.<id>.steps[*]`.

Field	Description
<code>name</code>	The name of the step.
<code>run</code>	The command to executed.
<code>uses</code>	Alternative to <code>run</code> . A reusable application (known as “action”) to be executed.
<code>working-directory</code>	Working directory to run the command.
<code>shell</code>	Shell to run the command. If not specified, <code>bash</code> is used on a Linux machine.
<code>with</code>	A map of input variables which will be used as environment variables in the step.
<code>env</code>	Environment variables of the step.
<code>continue-on-error</code>	Preventing the job from failing with the step, if set to “true”.
<code>timeout-minutes</code>	Maximum number of minutes of the step.

actions, and he can also look for official and unofficial actions developed by other developers. For example, a collection of actions can be found at [11].

An example of an action, `checkout`, is given below. It downloads the source codes in the repository to the runner. The action used in the example is provided by GitHub officials.

```

name:
on:
jobs:
job1:
runs-on: ubuntu-latest
steps:
- name: get code
uses: actions/checkout@v4
with:
repository:
ref:
token:
ssh-key:
ssh-known-hosts:
ssh-strict:
ssh-user:
persist-credentials:
path:
clean:
filter:
sparse-checkout:
sparse-checkout-cone-mode:
fetch-depth:

```

```
fetch-tags:
show-progress:
lfs:
submodules:
set-safe-directory:
github-server-url:
```

where `uses` is used to indicate an action, and `actions/checkout@v4` is the identifier and version of the action. The `with` field (optional) is then used to configure the action if needed. The content of the configuration depends on the specific action in use. The details of the configurations about `checkout@v4` can be found at [11] and are not covered in this notebook.

There are many tasks that can be done via actions. To just name a few,

- Retrieve code, such as `checkout`.
- Upload or download artifact, such as `upload-artifact`, `dowload-artifact`.
- Install software, such as `setup-node`, `setup-python`, `setup-dotnet`.

There are many more wonderful actions on the market, many of which can be used free-of-charge.

10.7.5 Contexts

The following syntax

```
${{ <context>.<information> }}
```

can be used to access GitHub Actions contexts in the workflow configuration files. Examples have been given in earlier sections. **GitHub Actions contexts** are collections of read-only information available to the workflow configuration. There are many contexts defined and they cover a large range of topics regarding the workflow, the job, the runner, the step, environment variables, secrets, and many more.

As a recap, in the earlier example the following information is retrieved.

- `${{ github.event_name }}`: the trigger of the event.
- `${{ runner.os }}`: the OS of the runner.
- `${{ github.ref }}`: branch name.
- `${{ github.repository }}`: repository name.

The information from contexts can be used in if-else statements to direct the workflow. Another common practice is to use an expression, such as

```
${{ toJSON(github) }}
```

to convert contexts into JSON strings, cache them in environment variables, and use them in the commands.

10.7.6 Costs

Last but not least, GitHub Actions may generate additional costs. This is because the workflows are executed on runners based on GitHub's servers, consuming its CPU, memory and storage. As of this writing, additional cost is generated only if all of the below are true:

- Private repository.
- The workflows are executed on GitHub servers. Notice that it is possible to execute workflows using on-premises servers, in which case GitHub Actions becomes a free service.
- CPU, memory and storage usages go beyond the free-tier threshold.

If all the above criteria are met, the user pays GitHub for 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.



11

Database

CONTENTS

11.1	Database and Database Management System	115
11.2	Relational Database	116
11.3	Non-Relational Database	117
11.4	Structured Query Language	118
11.4.1	Naming Conventions	118
11.4.2	Datatypes and General Syntax	119
11.4.3	Database Manipulation	122
11.4.4	Table Manipulation	123
11.4.5	Row Manipulation	126
11.4.6	Query	127
11.4.7	Trigger	135
11.4.8	SQL Demonstrative Example	136
11.5	RDB Connectivity	140

Different databases and database manage systems are introduced in this and the next few chapters. Both relational and non-relational databases are covered. Tools and commands to manipulate databases are introduced.

11.1 Database and Database Management System

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 with small data size.

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 **database manage system** (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 DBMSes, and different DBMSes may provide different interfaces for the users.

11.2 Relational Database

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

Relational database (RDB) 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.

Structured Query Language (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 an 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, and many more.

An example of a table used in RDB is given in Table 11.1. 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,

TABLE 11.1

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
...

TABLE 11.2

A second database table in the example.

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

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 11.2. In the first Table 11.1, column `membership` can be a foreign key which points to `membership_type` in the second Table 11.2.

The introducing of foreign key helps to maintain the consistency and integrity of the database. For example, when adding a new member to Table 11.1, the DBMS will first check whether the membership type is registered in Table 11.2. 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, the “referee-id” attribute in Table 11.1 could be the foreign key that links to “user-id” of the same table.

11.3 Non-Relational Database

Non-relational database (NoSQL database) gained popularity in the 2000s. In contrast to RDBs, NoSQL databases do not store data in tables, but in key-value pairs, graphs, documents or other formats. Compared with RDBs, they are more flexible and may be easier to use in certain types of applications. Ex-

amples of NoSQL databases include Redis, Azure CosmosDB, Oracle NoSQL Database, Amazon DynamoDB, MongoDB, AllegroGraph, ChromaDB and many more.

Unlike SQL which applies to almost all RDBMSes, there is no universally adopted language for NoSQL DBMSes. This is mainly because each type of NoSQL databases has a different data structure, hence each of which requiring distinct query language. We use “NoSQL” to refer to a collective set of (very different and not interchangeable) languages used for NoSQL databases management. It is impossible to cover all the different types of NoSQL databases and their corresponding DBMS manipulation languages in this notebook. Only selected NoSQL databases are briefly introduced in later chapters as examples.

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.

11.4 Structured Query Language

SQL is the most widely used language for interacting with RDBMS for data query and maintenance. SQL is very powerful and flexible in its full capability, and it is hardly possible to cover everything concisely. 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.

Details of SQL is introduced in the remaining of this section.

11.4.1 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 and not self-explanatory terms.
 - 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.

TABLE 11.3

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)	Decimal number, with fixed-precision. The total number of digits and the number of digits after the decimal point are specified by “size” and “d” respectively. Since the precision is fixed and can be customized, DEC/DECIMAL is recommended in finance applications.
DOUBLE and FLOAT	Decimal number, with approximated precision. FLOAT can be used for general purposes, while DOUBLE for scientific data where higher resolution is expected.
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.

11.4.2 Datatypes and General Syntax

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 11.3. Nowadays, many DBMS supports more complicated types such as objects (the associated database is also known as the object-relational

database). 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 11.4, 11.5 and 11.6.

TABLE 11.4

Widely used SQL keywords (part 1: names).

Keyword	Description
CONSTRAINT	A constraint that can be setup for 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

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 11.4.1.

During the coding, follow the following rules.

TABLE 11.5

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.

- 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.

TABLE 11.6

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.

11.4.3 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>;
```

11.4.4 Table Manipulation

Tables are the fundamental components in an RDB. 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 constraints are summarized in Table 11.7. As shown

TABLE 11.7
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 unique and not NULL.
FOREIGN 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.

by Table 11.7, 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

```
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 practice.

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.

11.4.5 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-criteria>** is used to filter the rows to which the update is carried out. Commonly used filter criteria are a set of **<column-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.

11.4.6 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 why 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.

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.

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 |
+-----+-----+-----+-----+
```

An 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.

11.4.7 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 BEFORE 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/updat-ed/deleted from the **table-name**.

Use the following to drop a trigger.

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

11.4.8 SQL Demonstrative Example

An example to demonstrate the use of SQL, a database is created from scratch. MariaDB is used as the DBMS in this example. 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 re-pair); 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);
```

11.5 RDB Connectivity

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 provides variety of packages to connect to different databases. IDEs such as VSCode can connect to databases using extensions.

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

As pre-requisites, 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 DBMS 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 relevant 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.

12

RDB Example: PostgreSQL

CONTENTS

12.1	Brief Introduction	145
12.2	Installation and Authentication	146
12.2.1	Installation	146
12.2.2	Regular Authentication	147
12.2.3	Peer Authentication	147
12.2.4	Deployment of PostgreSQL in Container	148
12.3	Data Types	149
12.3.1	PostgreSQL-Specific Data Types	149
12.3.2	User-Defined Data Types	150
12.4	PostgreSQL Stored Procedural and Functions	151
12.5	Manipulation and Query	153

PostgreSQL (or Postgres) is a powerful open-source relational database project. It is gaining a lot of popularity recently.

12.1 Brief Introduction

PostgreSQL is claimed to be the world's most advanced open-source RDB, and it has some great features. It is an 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 of this writing), 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.

12.2 Installation and Authentication

The installation of PostgreSQL depends on the OS. As of this writing, the latest version of RHEL 9.4 is used as an example. Both installation and authentication to the database are introduced.

12.2.1 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 light 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 and enable PostgreSQL 16 on RHEL 9.4 running on `x86_64`, use

```
$ sudo dnf module install postgresql:16/server
$ sudo postgresql-setup --initdb
$ sudo systemctl start postgresql.service
$ sudo systemctl enable postgresql.service
```

After installation, use command `pg_ctl` to control the server, and `psql` to log in 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.

12.2.2 Regular Authentication

When installing Postgres, it automatically does the following.

- Create an OS user `postgres`. This OS user has full access to the database.
- Create a database role user `postgres`. This role has full access to the database.
- Create a database `postgres`.

A sudoer can log in to the database as the OS user `postgres` as follows. Then he does that, he is automatically assigned the database role `postgres`. As mentioned earlier, this role has administrative access over the database.

```
$ sudo -u postgres psql postgres
postgres=#
```

where `postgres=#` is the prompt of the DBMS console.

After logging in to the database as `postgres`, the user can create users and databases using

```
postgres=# CREATE ROLE <username> LOGIN PASSWORD '<password>';
postgres=# CREATE DATABASE <database name> WITH OWNER = <username>;
```

and log in to the database as the user by

```
$ psql -h localhost -d <database name> -U <username> -p <port>
```

where `<port>` is defined in the configuration file and it is 5432 by default.

12.2.3 Peer Authentication

Given that the user already logged in to the OS, it is possible to use peer authentication if the login is from localhost and the PostgreSQL role is the same with the OS username.

Quickly create a user in PostgreSQL with the same username as OS using

```
$ sudo -u postgres createuser -s $USER
```

which creates a database user `$USER`, with the same username as the OS user. With the user created, create a database for the user using

```
$ createdb <database name>
```

Then log in to PostgreSQL using peer authentication as follows

```
$ psql -d <database name>
<username>=>
```

where `<username>=>` is the prompt to a regular user

If the database name happens to be the same with `$USER`, simply use

```
$ psql
<username>=>
```

to login to the database.

After logging in to the database, use `\du` and `\l` to check the users and the databases in PostgreSQL, and use `SHOW port`; to check the port that PostgreSQL is running on which by default should be 5432.

12.2.4 Deployment of PostgreSQL in Container

To use PostgreSQL in containers, you can either download and run official PostgreSQL images from Docker Hub, or build a custom Dockerfile that installs and configures PostgreSQL on top of a base image such as Alpine.

The following example shows how to launch PostgreSQL in a container using the Docker Hub image. In this example, a persistent volume is mounted to preserve user and database data, so that if the container is restarted, previously created users and databases are not lost. Environment variables such as the username and password are stored in an `.env` file.

```
#!/bin/bash

source ~/.env

if podman container exists "$PG_CONTAINER"; then
    status=$(podman inspect -f '{{.State.Status}}' "$PG_CONTAINER")
    if [ "$status" = "running" ]; then
        echo "Container $PG_CONTAINER is already running."
    else
        echo "Starting existing container $PG_CONTAINER..."
        podman start "$PG_CONTAINER"
    fi
else
    echo "Creating and starting container $PG_CONTAINER..."
    podman run -d \
        --name "$PG_CONTAINER" \
        -e POSTGRES_USER="$PG_USER" \
        -e POSTGRES_PASSWORD="$PG_PASSWORD" \
        -e POSTGRES_DB="$PG_DB" \
        -v "$PG_DATA_DIR":/var/lib/postgresql/data:Z \
        -p "$PG_PORT":5432 \
```

```
    docker.io/library/postgres  
fi
```

The next example shows a Dockerfile that installs PostgreSQL on Alpine Linux.

```
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"]
```

12.3 Data Types

PostgreSQL supports a large range of data types, more than other databases such as MySQL and MariaDB.

12.3.1 PostgreSQL-Specific Data Types

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.

12.3.2 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_OPCCLASS = 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 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 delimiter 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.

12.5 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 12.1. 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.

TABLE 12.1

Widely used psql commands.

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

13

NoSQL Database Example: MongoDB

CONTENTS

13.1	A Brief Introduction	156
13.2	Installation	158
13.3	MongoDB Shell	158
13.3.1	Select Database	159
13.3.2	Create Collection and Document	159
13.3.3	Query	162
13.3.4	Update Document	163
13.3.5	Remove Document	164
13.3.6	Advanced Query	164
13.4	MongoDB Compass	166
13.5	Sharding and Indexing	167
13.5.1	Sharding	167
13.5.2	Indexing	167
13.6	Third-Party Connection	171
13.7	Aggregation Pipeline	172
13.7.1	Aggregation Pipeline Syntax	173
13.7.2	Stage: <code>\$ match</code>	174
13.7.3	Stage: <code>\$ group</code>	174
13.7.4	Stage: <code>\$ sort</code>	176
13.7.5	Stage: <code>\$ limit</code>	176
13.7.6	Stage: <code>\$ set</code>	176
13.7.7	Stage: <code>\$ count</code>	176
13.7.8	Stage: <code>\$ project</code>	178
13.7.9	Stage: <code>\$ out</code>	178
13.8	MongoDB Atlas	179
13.8.1	MongoDB Atlas Dashboard	179
13.8.2	Connection String	180
13.8.3	Atlas Search	181
13.8.4	Schema Pattern	184

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

t 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.

Data in MongoDB can be exported to or imported from files, such as BSON and JSON files. These files can be ported to other machines or consumed by other programs. MongoDB server also provides APIs for both local and remote access.

13.1 A Brief Introduction

MongoDB is a source-available, cross-platform, general-purpose, document-oriented NoSQL database program. It stores data in documents with flexible schema.

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.

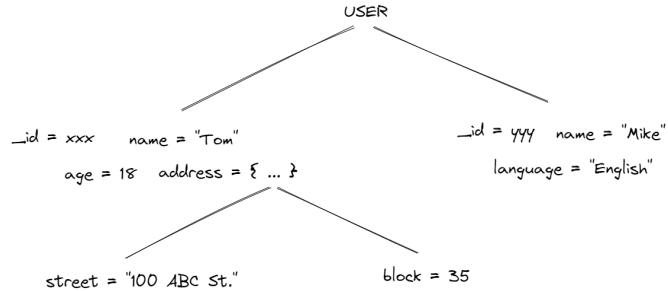
Examples of MongoDB-suitable applications include

- Posts and streaming management on social media websites or APPs;
- Online gaming;
- Logistics industry and supply chain management;
- IoT data management.

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.

MongoDB is a document-oriented database program and it stores data in documents and collections instead of rows and tables. A **MongoDB document** is the basic unit of the storage. A **MongoDB 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 need to join collections and map documents. Figure 13.1 gives a demonstrative example of how MongoDB stores and organizes data.

In the example demonstrated by Fig. 13.1, a collection “USER” is defined, and it contains two documents. Each document has several properties. Different document may share common properties such as “name” in this example.

**FIGURE 13.1**

A demonstrative example of how MongoDB stores data as (nested) object.

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.

The data is stored in the format of **binary JSON** (BSON). BSON is an extension of JSON that supports more data types. MongoDB uses BSON to store and process data due to its enhanced capability, and uses JSON to display the data.

JSON VS BSON

JSON supports the following 6 data types: number, string, boolean, object, array, and null. JSON is text-based, which means that it is essentially a string. It is only a notation of data and does not concern with how the data is interpreted in the program. For example, “0” as a number in a JSON file does not imply whether it is an integer, a 32-bit float, or a 64-bit float.

BSON, on the other hand, is binary-based and hence contains more information. In BSON, more data types are supported, such as different floats, date, timestamp, binary string, regular expression, code, etc. For the same example, “0”, in BSON different number types are compiled and stored differently. This makes BSON more powerful and efficient (but less intuitive) than JSON.

It is strongly recommended to make the data (document and collection) self-contained. The MongoDB architecture design should base on how the data is to be queried. Data that is accessed together should reside together. This concept is known as embedding. Embedding may add redundancy to the data

storage and make documents bigger, but will make query simpler. Notice that when using embedding, make sure the size of the document is bounded and would not go towards infinity. MongoDB has a maximum 16MB size limitation for each document.

MongoDB database, unlike relational DB, is by nature not designed for joining. However, MongoDB does support referencing documents and even performing joins (to some extent) through different mechanisms.

13.2 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>

A MongoDB server can host a MongoDB cluster, inside which are many MongoDB databases. Each database can have its own configurations and policies.

To interact with MongoDB DBMS, the quickest way is to use MongoDB shell (also known as “mongosh”). MongoDB shell and its explanation are available 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.

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 briefly introduced in later Section 13.8.

13.3 MongoDB Shell

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, such as creating databases and inserting data.

13.3.1 Select Database

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.

TABLE 13.1
MongoDB basic commands.

Command	Description
<code>db.help()</code>	Show a list of methods of <code>db</code> object.
<code>db.version()</code>	Show database server version.
<code>db.getUsers()</code>	Show users.
<code>db.createUser(<content>)</code>	Create a user. The username, password, roles, etc., needs to be included in the <code><content></code> area.
<code>db.dropUser(<username>)</code>	Drop a user.
<code>db.dropDatabase()</code>	Drop current database.
<code>db.status()</code>	Show the basic status of the currently selected database, such as its name, number of collections, storage size, etc.

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 switch 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 add some data. The database will be automatically created. This is again a feature closely related to JavaScript.

13.3.2 Create Collection and Document

A MongoDB database contains one or more collections. A collection is similar to a table in an RDB in the sense that it is the collection of data on the same / similar topic. However, unlike tables where schematics such as column names

and data types are enforced upon creation of the table, a collection does not enforce fields and their data types. The syntax is given below.

```
db.createCollection(<name>, <options>)
```

The `<options>` field is optional and allows creating different types of collections, such as time-series collection which is optimized to store time-series sensor data. Probable `<options>` configuration are summarized below.

```
db.createCollection( <name>,
{
    capped: <boolean>,
    timeseries: {                                // Added in MongoDB 5.0
        timeField: <string>,
        metaField: <string>,
        granularity: <string>,
        bucketMaxSpanSeconds: <number>, // Added in MongoDB 6.3
        bucketRoundingSeconds: <number> // Added in MongoDB 6.3
    },
    expireAfterSeconds: <number>,
    clusteredIndex: <document>, // Added in MongoDB 5.3
    changeStreamPreAndPostImages: <document>, // Added in MongoDB
        6.0
    size: <number>,
    max: <number>,
    storageEngine: <document>,
    validator: <document>,
    validationLevel: <string>,
    validationAction: <string>,
    indexOptionDefaults: <document>,
    viewOn: <string>,
    pipeline: <pipeline>,
    collation: <document>,
    writeConcern: <document>
})
```

If left empty, a normal collection will be created.

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 a semicolon in the end of each MongoDB command?

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, it is recommended to do so. An example is given below.

For example, to select a 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 highly recommended by JavaScript. Although in JavaScript semicolon has become optional due to Automatic Semicolon Insertion (ASI), it is still widely recommended to use semicolon anyway.

As a conclusion, semicolon is recommended mostly, just to follow the general JavaScript good practice.

Data can be inserted into a collection in the form of documents. It is possible to insert one or multiple documents at a time. To insert documents, first prepare the document in JSON-like 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" in the above example are represented by JSON and BSON respectively. MongoDB uses BSON internally, but it can

also take JSON as input, in which case MongoDB converts JSON to BSON internally.

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 object id “`_id`” will be created automatically and added to the document as another field. The user can also specify `_id` manually. Notice that `_id` serves as the “primary key”: it must exist and be unique for each document under the same collection.

The above inserting document method creates a collection, if the collection has not been created.

13.3.3 Query

MongoDB uses `find()`, `findOne()` to find documents, where the input is a filter document in the form of a JSON-like string. 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.

Two Ways to Refer to a Collection

Notice that

```
db.<collection_name>.<function>()
```

is equivalent with

```
db.getCollection('<collection_name>').<function>();
```

The first implementation is more convenient while the second more flexible as it supports dynamic naming.

Examples are given below to demonstrate the user of the query dictionary. Assume that there is a collection `posts`, inside which are documents recording the posts by a blogger. Each post may have some comments. The number of comments are recorded in the field `comments` of the documents. 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

```
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.

TABLE 13.2

MongoDB basic query operators.

Operator	Description	Example
<code>\$eq</code>	Equal	<code>{age: {\$eq: 18}}</code>
<code>\$gt</code>	Greater than	<code>{age: {\$gt: 18}}</code>
<code>\$gte</code>	Greater or equal	<code>{age: {\$gte: 18}}</code>
<code>\$lt</code>	Less than	<code>{age: {\$lt: 18}}</code>
<code>\$lte</code>	Less or equal	<code>{age: {\$lte: 18}}</code>
<code>\$and</code>	And	<code>={\$and: [{age: {\$gt: 18}}, {sex: 'M'}]}</code>
<code>\$or</code>	Or	<code>={\$or: [{age: 18}, {age: 21}]}</code>
<code>\$in</code>	In	<code>{name: {\$in: ['Alice', 'Bob']}}</code>
<code>\$nin</code>	Not in	<code>{name: {\$nin: ['Alice', 'Bob']}}</code>

Query can be applied to elements in an array. A commonly used operator is `$elemMatch`.

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

For example,

```
db.scores.find(
    { results: { $elemMatch: { $gte: 80, $lt: 85 } } }
)
```

which returns a document if and only if:

- it has a field `results`, and the field `results` is an array;
- the array field `results` contains at least one element that is greater or equal than 80 and less than 85.

13.3.4 Update Document

Function `updateOne()` is provided to update documents, where the input argument is a tuple containing filtering condition, update and options, following the syntax below.

```
db.<collection_name>.updateOne(filter, update, options)
```

TABLE 13.3

MongoDB basic update operators..

Operator	Description	Example
\$set	Add/update field	<code>{\$set: {age: 30}}</code>
\$unset	Remove field	<code>{\$unset: {age: ""}}</code>
\$push	Append to array	<code>{\$push: {age: 25}}</code>
\$inc	Increase a field	<code>{\$inc: {age: 1}}</code>
\$rename	Rename field	<code>{\$rename: {"age": "yearsOld"}}</code>
\$addToSet	Append to array	<code>{\$addToSet: {hobbies: "reading"}}</code>

Notice that `$addToSet` would not append the element if it is already in the array to avoid duplication, while `$push` will append regardless of whether it exists or not.

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.

Notice that there is another function, `replaceOne()`, that functions similarly with `updateOne()`. The difference is that `replaceOne()` will replace the entire document, removing all the fields (except `_id`) not mentioned in the update, while `updateOne()` allows editing the mentioned update while leaving unmentioned fields untouched.

It is possible to add `{upsert:true}` as the option to `replaceOne()` function, in which case if no document meets the criteria of the filter, it will create an empty document and implement the updates.

To update documents that meet certain criteria all at once, use `updateMany()` instead. Do note that `updateMany()` is not an atomic operation. It is possible that, for unpredictable reasons, only some of the documents are updated and they will not be rolled back. Use it with caution!

13.3.5 Remove Document

Use either `deleteOne(<filter>)` or `deleteMany(<filter>)` to remove documents, with the input the filter document.

13.3.6 Advanced Query

It been introduced earlier how `find()` `findOne()` can be used to retrieve documents. More about query is introduced below, such as sorting and limiting the results, returning only selected fields, returning aggregated information, etc.

It is worth mentioning that `find()` returns a cursor, but not the documents themselves. One can iterate through the cursor to get all the documents matching the query. This is different from `findOne()` which directly returns a document. When viewing the data in MongoDB Compass or Atlas, this does not pose an issue. One can simply take `find()` as a function that returns multiple documents. However, in the programming interface, it is important to note the differences between a cursor and a document.

The below are two demonstrative examples how a Python program should treat the returns from `find()` and `findOne()` differently. The same idea applies to other programming languages as well. More about MongoDB connectivity from Python is introduced in later sections. Here, the examples are only to demonstrate the differences between `find()` and `findOne()`.

```
from pymongo import MongoClient

def find_users():
    client = MongoClient('mongodb://localhost:27017/')
    db = client.mydatabase
    collection = db.users

    cursor = collection.find({ 'age': { '$gte': 18 } })

    # Iterate over the cursor
    for doc in cursor:
        print(doc)

    # Alternatively, convert to a list
    results = list(cursor)
    print(results)

    client.close()
```

```
from pymongo import MongoClient

def find_one_user():
    client = MongoClient('mongodb://localhost:27017/')
    db = client.mydatabase
    collection = db.users

    # Find one document
    user = collection.find_one({ 'name': "John Doe" })
    print(user)
```

```
client.close()
```

Since a cursor, instead of the documents themselves, is returned, we can perform further actions on the return such as sorting the result, etc.

Use the following syntax to retrieve and sort the return.

```
cursor.sort({<field>:<value>})
```

which sorts the result according to `<field>`. The value “1” or “−1” represents the sorting order, either ascending or descending, respectively.

When it is unnecessary to return all the details of all the documents fulfilling the query criteria, it is possible to limit the field number of returned documents which is often helpful with improving the data processing efficiency.

It is possible to limit the number of documents in the returns to improve the query efficiency. This is often used with the `sort()` method. The syntax is as follows.

```
cursor.sort({<field>:<value>}).limit(<number>)
```

where `<number>` specifies the number of returns.

Notice that the full syntax of `find` is

```
db.collection.find( <query>, <projection>, <options> )
```

The second argument, `<projection>`, is used to indicate which fields of the documents should be returned. If left blank as by the default, all the fields of the documents will be returned. Notice that the same applies to `findOne()` method.

For example,

```
cursor = collection.find({'age': {'$gte': 18}}, {name:1, age: 1})
```

will return the names and ages of all the users whose age is greater than or equal to 18. By using

```
{ <field1>: <value>, <field2>: <value> ... }
```

with `<value>` be either 1 or 0, one can control which fields to be included or excluded respectively. The value 0 is often used with the `_id` field which is always returned unless specifically required not to. Notice that the `<field>` in the syntax can also contain sub fields, in which case quotation marks "`<field>. <subfield>`" must be used.

With some more advanced setups, `<projection>` can also be used to specify what elements in an array to return, and more. For details, check the user manual of `db.collection.find()` function.

Consider counting the number of documents fulfilling a particular filter document. Use

```
db.<collection>.countDocuments( <query>, <options> )
```

13.4 MongoDB Compass

Should the user decides to use GUI instead of MongoDB Shell to interact with the MongoDB server, he may consider MongoDB Compass, the official GUI client for MongoDB.

MongoDB Compass is the desktop GUI developed by MongoDB. As a MongoDB client, it can connect to a MongoDB server, either local, remote or MongoDB Atlas, using the connection string. It provides an intuitive user interface that can be used to view and edit the data on the server.

MongoDB Compass comes with analytical tools and data visualization tools that help the user understand the insight of the data as well as the database structure. It allows the user to define and compose aggregation pipelines that automatically abstract useful aggregated information from the data.

13.5 Sharding and Indexing

Given that MongoDB is often used with big data such as web data or IoT data, efficient query from massive data pool becomes critical. MongoDB implements a few technologies to speed up the query such as sharding and indexing.

13.5.1 Sharding

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 data set. This can significantly reduce query times in a large, distributed system. Sharding is particularly useful for very large datasets and high throughput operations, where a single server would not be sufficient to store the data or provide acceptable performance.

13.5.2 Indexing

Indexing is a technique used to speed up the retrieval of documents within a database. MongoDB uses indexes to quickly locate a document 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 access the documents for a query.

When a filter such as `find()`, `findOne()`, or in an aggregation pipeline, `$match` are used, MongoDB automatically utilizes the available indexes to speed up the query, making it more efficient.

When projection is used to limit the fields to be returned, and if it happens that all the returned fields are part of the index, MongoDB would not need to revisit the original documents. In this case, the query speed can be significantly faster.

MongoDB's B-Tree

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 as the dataset grows. The B-Tree structure allows MongoDB to perform efficient query. 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.

However, index by itself is also an argument data structure and it consumes disk and memory 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.

The default index for a collection is “`_id`”, a compulsory and unique field required by MongoDB. The user can also define index of different types. User-defined index can be unique or non-unique. The index of a collection can be created at any time, and it does not have to be when the collection is empty.

A MongoDB collection can have multiple indexes, and when you perform a query, MongoDB will choose the most suitable index to use based on the query's criteria.

Index related operations are introduced below.

Check Index

To check existing indexes, use

```
db.collection_name.getIndexes()
```

The index can also be viewed from the graphical UI.

Create Index

There are many options when creating the user-defined index. The following function

```
db.collection.createIndex(<keys>, <options>, <commitQuorum>)
```

creates a user-defined index. A single or a compound of multiple fields in a collection can be used as the index using

```
db.collection.createIndex(  
  {  
    <field1>: 1,  
    <field2>: -1,  
    ...  
  })
```

Those fields assigned with “1” or “-1” is listed as the ascending and descending index. The choice of order determines how the index sorts the indexed field’s values, which can affect the performance of certain queries.

The data types of the selected field(s) can be scalar, string, or even a nested object. MongoDB has an internal mechanism that can convert different data types into a format that can be indexed and sorted efficiently.

The order of fields in a compound index can significantly impact query performance in MongoDB. Following the principle of “equality, sort, range” when defining compound indexes is a good practice to optimize queries.

- Equality. Fields that are used in equality conditions (=) should come first in the index. These fields are used to narrow down the search space quickly.
- Sort. Fields that are used for sorting the results should come next. Sorting fields in the index help MongoDB to efficiently retrieve documents in the desired order without additional sorting.
- Range. Fields that are used in range queries (<, <=, >, >=, \$in, etc.) should be placed last. Range queries can benefit from being at the end of the index because they can take advantage of the already reduced search space provided by the equality and sort fields.

It is possible to use the <options> argument to set additional constraints to the index, for example, to enforce its value to be unique.

When defining index using a field of array, the index is known as multi-key index. Like the scalar fields, multi-key index can also be both single or compound. It is also possible to mix scalar keys with array keys.

Syntax wise, defining a multi-key index looks similar with defining a regular single or compound index as follows.

```
db.collection.createIndex(  
  {  
    <scalar field1>: 1,  
    <scalar field2>: 1,
```

```

        <array field>: 1
    }
)

```

The only limitation is that there can be only one array key per index.

Behind the screen, MongoDB treats each element in the array key as a separate index value. Likewise, each element in the array of the index can be scalar, string, or nested object.

Hide and Remove Index

It is not recommended to modify the index of a collection frequently, as deleting and recreating index take time and resources. Sometimes we do need to remove a redundant index if it is adding too much writing cost.

It is possible to test the query performance if an index were to be removed before actually removing it. This can be done by temporarily hide the index. MongoDB would not use hidden index in the query, though it will still update the index when new data is written in.

To hide an index, use

```
db.<collection>.hideIndex(<index>)
```

where **<index>** can be either a string or a document that indicates the status of the index. An example is given below.

Consider a restaurant collection, with the following user-defined index:

```
db.restaurants.createIndex( { borough: 1, ratings: 1 } );
```

Now the collection should have 2 indexes, the first the default index with **_id** field, the second the compound index composed of **borough** and **ratings** fields.

The user-defined compound index can be hidden by one of the two syntax below.

```
db.restaurants.hideIndex( "borough_1_ratings_1" ); // option 1
db.restaurants.hideIndex( { borough: 1, ratings: 1 } ); // option 2
```

After hiding the user-defined index, the index status becomes

```
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_"
  },
  {
    "v" : 2,
    "key" : {
      "borough" : 1,
      "ratings" : 1
    }
  }
]
```

```

    },
    "name" : "borough_1_ratings_1",
    "hidden" : true
}
]

```

To remove an index instead of hiding it, use `dropIndex()` instead of `hideIndex()`. A similar-looking command `dropIndexes()` allows removing multiple indexes at the same time.

Evaluate Performance

Consider using `explain()` function as follows when carrying out a find operation.

```
db.<collection>.explain().find()
```

which will list down the down-to-the-ground stages to carry out the query. The user can use this to test the same document with different index setups, and compare their performance.

13.6 Third-Party Connection

Both MongoDB shell CLI and MongoDB Compass GUI can connect to MongoDB as introduced in earlier sections. Third-party applications can also connect to MongoDB. As of this writing, MongoDB supports libraries for the following developing languages:

- C/C++
- C#
- Go
- Java
- Kotlin
- Node.js
- PHP
- Python
- Ruby
- Rust
- Scala
- TypeScript
- Swift

A full list can be found at [12]

Here is a quick example to connect to MongoDB using Python library PyMongo.

```

import pymongo
from pymongo import MongoClient

def initialize_mongodb_collection(connection_string, database_name,
    collection_name, index_field):
    client = MongoClient(connection_string)
    db = client[database_name]
    collection = db[collection_name]

```

```

        collection.create_index(index_field)

def push_to_mongodb(connection_string, database_name, collection_name,
                     data):
    client = MongoClient(connection_string)
    try:
        db = client[database_name]
        collection = db[collection_name]
        if isinstance(data, dict):
            result = collection.insert_one(data)
            print(result.acknowledged)
        elif isinstance(data, list):
            result = collection.insert_many(data)
            print(result.acknowledged)
        else:
            raise Exception("Data should be a dictionary or a
                            list of dictionaries")
        client.close()
    except Exception as e:
        raise Exception("Unable to insert the document: ", e)

def pull_from_mongodb(connection_string, database_name, collection_name
                      , query_dict):
    results = None
    client = MongoClient(connection_string)
    try:
        db = client[database_name]
        collection = db[collection_name]
        results = collection.find(query_dict) # return iterative
                                                object
        client.close()
    except Exception as e:
        raise Exception("Unable to insert the document: ", e)
    return results

```

13.7 Aggregation Pipeline

In the context of MongoDB, aggregation is the analysis and summary of data. MongoDB, like many other databases, provides powerful tools to perform aggregation functions such as counting the number of documents, calculating the sum or average of a particular field of filtered documents, etc. Beyond that, MongoDB allows the user to define a pipeline composed of a sequence of data processing procedures for aggregation. This enables autonomous data processing and analysis, and it is useful especially for big data applications.

This feature, known as the data aggregation framework or data **aggregation pipeline**, is one of the most powerful and useful features that MongoDB offers.

Aggregation pipeline is supported in all MongoDB tiers, including MongoDB Community, Enterprise and Atlas.

13.7.1 Aggregation Pipeline Syntax

To apply different aggregation functions, use `aggregate()` as follows.

```
db.<collection>.aggregate( <pipeline>, <options> )
```

where `<pipeline>` is usually an array that contains a sequence of aggregation operations including filter, sorting, grouping and transforming, which may look like the following

```
db.<collection>.aggregate([
  {
    $<stage>: {<expression>}
  },
  {
    $<stage>: {<expression>}
  },
  {
    $<stage>: {<expression>}
  }
])
```

An example is given blow.

```
db.orders.insertMany( [
  { _id: 0, name: "Pepperoni", size: "small", price: 19,
    quantity: 10, date: ISODate( "2021-03-13T08:14:30Z" ) },
  { _id: 1, name: "Pepperoni", size: "medium", price: 20,
    quantity: 20, date : ISODate( "2021-03-13T09:13:24Z" ) },
  { _id: 2, name: "Pepperoni", size: "large", price: 21,
    quantity: 30, date : ISODate( "2021-03-17T09:22:12Z" ) },
  { _id: 3, name: "Cheese", size: "small", price: 12,
    quantity: 15, date : ISODate( "2021-03-13T11:21:39.736Z"
      ) },
  { _id: 4, name: "Cheese", size: "medium", price: 13,
    quantity:50, date : ISODate( "2022-01-12T21:23:13.331Z"
      ) },
  { _id: 5, name: "Cheese", size: "large", price: 14,
    quantity: 10, date : ISODate( "2022-01-12T05:08:13Z" ) },
  { _id: 6, name: "Vegan", size: "small", price: 17,
    quantity: 10, date : ISODate( "2021-01-13T05:08:13Z" ) },
]
```

```

        { _id: 7, name: "Vegan", size: "medium", price: 18,
          quantity: 10, date : ISODate( "2021-01-13T05:10:13Z" ) }
    ] )

db.orders.aggregate(
    // Stage 1: Filter pizza order documents by pizza size
    {
        $match: { size: "medium" }
    },
    // Stage 2: Group remaining documents by pizza name and
    // calculate total quantity
    {
        $group: { _id: "$name", totalQuantity: { $sum: "
            $quantity" } }
    }
) )

```

where both `$match` and `$group` are commonly used stages.

And the return is a cursor pointing to the following array of documents

```

[
    { _id: 'Cheese', totalQuantity: 50 },
    { _id: 'Vegan', totalQuantity: 10 },
    { _id: 'Pepperoni', totalQuantity: 20 }
]

```

The return of an aggregation pipeline can be consumed by a program, or used to create a new collection.

Notice that the design of the pipeline, such as the order of the stages, may affect the execution speed and efficiency of the aggregation pipeline. There are many supported stages, each with a different list of input arguments. A full list of supported stages can be found from MongoDB's user manual at [13] under "Aggregation Stages". Calculations can be performed along with the aggregation pipeline. The calculations are triggered by aggregation operators. A full list of supported aggregation operators can be found from MongoDB's user manual at [14] under "Aggregation Operators".

It is highly recommended that one should read through MongoDB's user manual to learn the aggregation framework. For the convenience of the reader, commonly used stages are introduced as follows.

13.7.2 Stage: `$match`

The `$match` stage filters the documents to pass only the documents that match the specified conditions to the next pipeline stage. A general syntax is given below.

```
{$match: <filter>}
```

The `$match` stage should be used in early steps to make the pipeline more efficient.

13.7.3 Stage: \$group

The `$group` stage separates documents into groups according to a “group key”. The output is one document for each unique group key. Use `null` as the group key to group all documents into a single group. Consequent aggregations are then performed per group.

A general syntax is given below.

```
{
  $group:
  {
    _id: <expression>, // Group key
    <field1>: { <accumulator1> : <expression1> },
    ...
  }
}
```

where `<accumulator>` indicates what to calculate on the grouped information to put into the field. For example, the accumulator can be `$count`.

The following is an example. Let there be a collection “commodity”, inside which each document represents a product, and has the following 3 fields: “item” (item name), “price” (unit price) and “quantity” (the sold number). The target is to calculate the sold total price for each product.

```
{
  $group :
  {
    _id : "$item",
    totalSaleAmount: { $sum: { $multiply: [ "$price", "
      $quantity" ] } }
  }
}
```

Do notice that to refer to the field name in the documents, use `"$<fieldname>"`.

The return of the above may look like the following

```
[
  { _id: "apple", totalSaleAmount: 50 },
  { _id: "banana", totalSaleAmount: 45 }
]
```

If `null` were used as the group key, the return would have been

```
[
  { _id: null, totalSaleAmount: 95 }
]
```

To count the number of documents in each group and put it as an additional field, consider using `$sum` as follows.

```
{
```

```

$group: {
    _id: <expression>, // Group key
    count: { $sum: 1 } // Accumulator to count documents
}

```

13.7.4 Stage: \$sort

The `$sort` stage sorts the documents in either ascending order or descending order for further processing in the pipeline. The `$sort` stage is sometimes used before `$limit` stage.

A general syntax is given below.

```

{
    $sort:
    {
        <field1>: 1 // or -1
    }
}

```

where “1” and “–1” indicate the ascending and descending order with respect to the field.

13.7.5 Stage: \$limit

The `$limit` stage selects and filters the first a few items from the pipeline. The input argument is simply a positive integer, as shown in the below example.

```

{
    $limit: 3
}

```

13.7.6 Stage: \$set

The `$addFields` or `$set` stage (alias of each other) is used to modify or add fields in the pipeline, using the following syntax.

```

{
    $set:
    {
        <field1>: <new value>,
        <field2>: {<expression>},
        ...
    }
}

```

The assignment can be either a value or an expression, such as rounding, multiplying by a gain, etc. A list of supported arithmetic aggregation functions is given in Table 13.4.

TABLE 13.4

MongoDB arithmetic aggregation functions.

Name	Description
\$abs	Returns the absolute value of a number.
\$add	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
\$ceil	Returns the smallest integer greater than or equal to the specified number.
\$divide	Returns the result of dividing the first number by the second. Accepts two argument expressions.
\$exp	Raises e to the specified exponent.
\$floor	Returns the largest integer less than or equal to the specified number.
\$ln	Calculates the natural log of a number.
\$log	Calculates the log of a number in the specified base.
\$log10	Calculates the log base 10 of a number.
\$mod	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
\$multiply	Multiplies numbers to return the product. Accepts any number of argument expressions.
\$pow	Raises a number to the specified exponent.
\$sqrt	Calculates the square root.
\$subtract	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
\$trunc	Truncates a number to its integer.

13.7.7 Stage: \$count

The `&count` stage counts the number of the documents passing through the pipeline, and put the number into a field.

```
{
    $count: <field>
}
```

An example is given below

```
db.scores.aggregate([
```

```
{
    $match: {
        score: {
            $gt: 80
        }
    }
},
{
    $count: "passing_scores"
}
])
```

which count the number of students whose score is greater than 80. Notice that when using `$group` stage, `count: {$sum: 1}` can be used to count the number of elements in each group. Though both methods counts the number of documents, they are used in different contexts.

13.7.8 Stage: `$project`

The `$project` stage changes the shape of the data in the pipeline. It selects which fields to be displayed. In many occasions it is used near the end of the pipeline, right before the data comes out of the pipeline.

A general syntax looks like the following.

```
{
    $project:
    {
        <field1>: 1,
        <field2>: 0,
        <field3>: <new value>,
        ...
    }
}
```

where “1” and “–1” are used to either include or exclude a field. It can also create new fields or overwrite an existing field, in which case the value of the new fields needs to be specified.

13.7.9 Stage: `$out`

The `&out` stage creates a new collection or overwrite an existing collection using the aggregation pipeline result. If `&out` is used, it should be the last stage.

```
{
    $out:
    {
        db: "<database name>",
        coll: "<collection name>"
```

```
    }  
}
```

or

```
{  
    $out: "<collection name>"  
}
```

13.8 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 including AWS, Azure and Google Cloud. 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 is 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 storage, 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.

13.8.1 MongoDB Atlas Dashboard

To use MongoDB Atlas, register an account with MongoDB Atlas.

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 empty 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. Of course, MongoDB Atlas also provides connection string so that the user can connect to it remotely using CLI, from MongoDB Compass, or from third-party applications.

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.

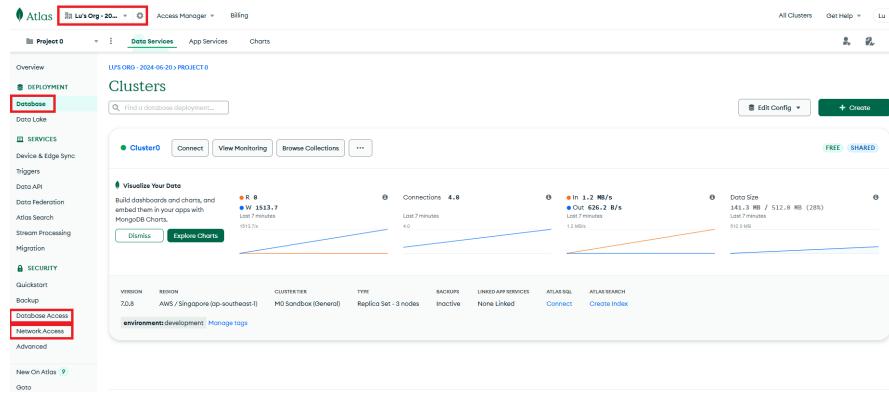


FIGURE 13.2

A demonstration of MongoDB Atlas dashboard.

To view collections and query for documents, click “Browse Collections” in Fig. 13.2 which gives Fig. 13.3. This sample database cluster contains multiple databases, each database with several collections, as shown by Fig. 13.3.

The screenshot displays the MongoDB Cloud interface for the 'PROJECT 0' database. On the left, a sidebar lists services such as Project 0, Data Services, App Services, Charts, Overview, Deployment, Database, Data Lake, and various service-specific sections like Device & Edge Sync, Triggers, Data API, Data Federation, Atlas Search, Stream Processing, Migration, and Security. The 'Security' section is currently selected.

The main content area shows the 'ClusterO' database. It includes tabs for Overview, Real Time, Metrics, Collections (which is selected), Atlas Search, Performance Advisor, Online Archive, and Cmd Line Tools. Below these tabs, it shows 'DATABASES: 9' and 'COLLECTIONS: 23'. A prominent 'sample_analytics.customers' collection is highlighted, showing its storage size (16KB), logical data size (19.2KB), total documents (500), and index size (32KB). Below this, there are buttons for Find, Indexes, Schema Anti-Patterns, Aggregation, and Search Indexes. A natural language query input field says 'Generate queries from natural language in Compose!'. A 'Filter' section allows querying by field and value. Finally, a 'QUERY RESULTS: 1-20 OF MANY' section shows a single document's details:

```
_id: ObjectId("5ca4bbcea2dd94ae58162a68")
username: "failer"
name: "Elizabeth Ray"
address: "#2345 Anthony Glens
city: "Brentwood, CO 22999"
birthdate: 1977-03-02T02:30:11.000+00:00
email: "arrywcolton@gmail.com"
active: true
accounts: Array (6)
tier_and_details: Object
```

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.

13.8.2 Connection String

Database connection string allows a client to connect to a database server. 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

Simply follow the instructions to connect to Atlas from one of the above entries. Notice that as a prerequisite, MongoDB Atlas needs to have the gateway policy to allow such connectivity.

A connection string may look like the following

```
mongodb+srv://<username>:<password>@<server-dns>/?authSource=admin&  
replicaSet=myRepl
```

13.8.3 Atlas Search

Atlas search is a powerful full-text search feature provided by Atlas that can access the documents and search for information based on relevance. It is built on top of Apache Lucene, a high-performance, full-featured text search engine library, and is designed to offer advanced search capabilities beyond the basic query.

Notice that Atlas search is NOT database query based on `find()`, nor AI-encoder-decoder based semantic search. Unlike traditional MongoDB queries which are based on exact matches or range conditions, Atlas Search provides relevance-based search. This means that search results are ranked by how well they match the search criteria, similar to how web search engines work. From this point of view, Atlas search is more business-user friendly, while MongoDB database search is more for developers and applications.

Atlas search, as well as Apache Lucene, uses search index to collect and parse the data to improve search efficiency. Do NOT confuse the search index with the MongoDB collection index introduced in earlier sections, as they are fundamentally different.

The remaining of this section introduces the use of Atlas search. Notice that Atlas search is power and flexible. Only a scratch is covered here.

Create Search Index and Test Atlas Search

Don't confuse Atlas search index or Apache Lucene index with MongoDB collection index. They function fundamentally differently. The purpose of the search index is to provide advanced full-text search capabilities, as so required by Apache Lucene.

MongoDB Atlas dashboard provides a GUI to create and manage search indexes for collections, as shown by the demonstration in Fig. 13.4. All configurations such as index analyzer, search analyzer, etc., are left as default in this example. It will take some time when setting up the search index.

With this setup, we can search the documents as if we were using a web searching engine. An example is given in Fig. 13.5. The most relevant documents with the keyword "big room" is returned. Each returned document corresponds with a score.

The default search index, as shown by the above example, uses dynamic mapping. It index all the fields in the collection. If we have priori knowledge about which field(s) to query, we can use static mapping which usually returns results faster.

To use static mapping, click the button "refine your index" during the search index creation, disable dynamic mapping, and add fields of interest following the instruction.

Use Atlas Search in the Aggregation Pipeline

With Atlas, we can add Atlas search as part in the aggregation pipeline.

The screenshot shows the MongoDB Atlas dashboard under the 'Atlas Search' tab. On the left, a sidebar menu includes 'Overview', 'Deployment', 'Database' (selected), 'Data Lake', 'Services', 'Device & Edge Sync', 'Triggers', 'Data API', 'Data Federation', 'Atlas Search' (selected), 'Stream Processing', and 'Migration'. Under 'Atlas Search', there are 'Index Configurations' and a table:

	Description	Setting
Index Analyzer	Creates searchable terms from data to be indexed.	lucene.standard
Search Analyzer	parse \$search queries into searchable terms.	lucene.standard
Dynamic Mapping	Automatically index common data types in a collection	On
Field Mappings	Define data types and input parameters for specific fields.	None
Stored Source Fields	Make all data available for lookup on Atlas Search side. See use cases and performance Considerations for details.	None
Synonyms Mappings	Enable defined synonym matching for like-terms for more relevant search results.	None

FIGURE 13.4
Atlas search set up search index using the dashboard.

The screenshot shows the 'Search Tester' section of the MongoDB Atlas search interface. A search bar contains the query 'big room'. Below it, a message says 'This search took 0.16 seconds'. Two search results are displayed in cards:

- Score: 17.411001205444336** _id: "235651"


```
listing_url : "https://www.airbnb.com/rooms/235651"
name : "LARGE ARTSY Room w/ Loft Bed 4 DOOGERS!"
summary : "Large room primarily rented monthly with a 20% to 25% discount dependi..."
```
- Score: 16.976694107055664** _id: "15771501"


```
listing_url : "https://www.airbnb.com/rooms/15771501"
name : "獨立房間 位於市中心 private room in the city"
summary : "House with 3 bedrooms and one big living room in Kowloon, 5 mins walk ..."
```

FIGURE 13.5
Atlas search test.

The associated stage is `$search`. It must be used together with Atlas search. The syntax looks like the following.

```
{
    $search: {
        "index": "<index-name>",
        "<operator-name>"|"<collector-name>": {
            <operator-spec>|<collector-spec>
        },
        "highlight": {
            <highlight-options>
        },
        "concurrent": true | false,
        "count": {
            <count-options>
        },
        "searchAfter"|"searchBefore": "<encoded-token>",
        "scoreDetails": true| false,
        "sort": {
            <fields-to-sort>: 1 | -1
        },
        "returnStoredSource": true | false,
        "tracking": {
            <tracking-option>
        }
    }
}
```

which can be defined in the Atlas dashboard under “Collections”, “Aggregation”.

13.8.4 Schema Pattern

The schema pattern is the guidance of designing good MongoDB architecture. Schema anti-pattern, on the other hand, refers to the architecture design that causes sub optimal performance.

Commonly seen schema anti-patterns include:

- Massive arrays.
- Massive number of collections.
- Bloated documents.
- Unnecessary indexes.
- Queries without indexes.
- Data that is accessed together but not stored together.

Atlas provides tools to help identify these schema anti-patterns. There are different ways to tackle each of these issues.

14

Virtualization and Containerization

CONTENTS

14.1	Introduction	185
14.2	Virtualization and Containerization Technologies	189
14.2.1	Virtualization	189
14.2.2	Containerization	191
14.3	Container Engine Basics	192
14.3.1	Container Engine Choices	192
14.3.2	Installation	193
14.3.3	Container Manipulation	194
14.3.4	Containerized Application Deployment Example	200
14.4	Docker Volume and Bind Mount	201
14.4.1	Volume	202
14.4.2	Bind Mount	203
14.4.3	Multiple Volumes	203
14.5	Docker Image	204
14.5.1	Dockerfile Programming	204
14.5.2	Build an Image	209
14.5.3	Docker Image Management	211
14.5.4	Docker Image Sharing with Docker Hub	211
14.6	Multi-Container Deployment and Orchestration	212
14.6.1	Communication	213
14.6.2	Docker Compose	215
14.7	Container Cloud Deployment	218
14.7.1	Docker Hosting Providers	219
14.7.2	VM-Based Approach	219
14.7.3	Managed-Service-Based Approach	221

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. Containerization is also found useful in fast migration and deployment of applications on new platforms.

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 because it virtualizes the infrastructure and the entire OS needs be installed on the newly deployed VM. It is hardly possible to launch a VM for each microservice when there are many of them. In this case, a more efficient approach is to deploy a single VM and place each application in a “container” with its own customized drivers and configurations. The technology used to deploy and manage containers is known as **containerization**.

A container is similar to a VM in the sense that it provides certain degrees 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. 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 cooking a dish. The hardware corresponds with the physical resources in the kitchen such as the cooktop and gas. The OS corresponds with the cook. The OS requires drivers and libraries to run the APP correctly. The drivers and libraries correspond with the specific skills or cooks for the dish. Finally, the APP corresponds 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 large and 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.

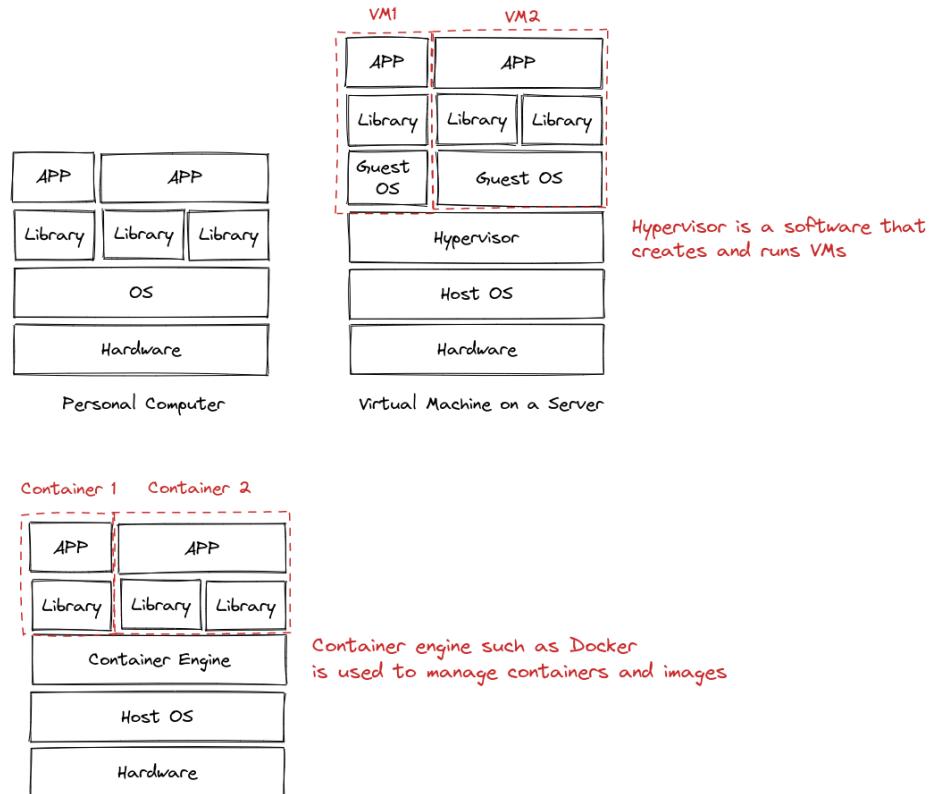


FIGURE 14.1
System architectures of PC, VM and container.

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 recipe is provided that gives all information required to prepare the dish consistently. As long as the cook is good at multi-tasking and has the basic skill sets for general dishes, Fig. 14.4 is usually a more efficient implementation than Figs. 14.2 and 14.3.

In the case of a restaurant, recipes describes the cookers and the raw materials a dish requires. Similarly in containerization, an “image” describes the initialization requirements of a set of containers. An image is basically a collection of prerequisites and configurations to start a container consistently.

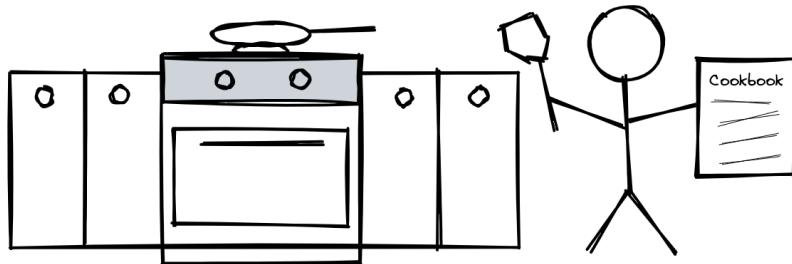


FIGURE 14.2
PC implementation: a cook in a kitchen.

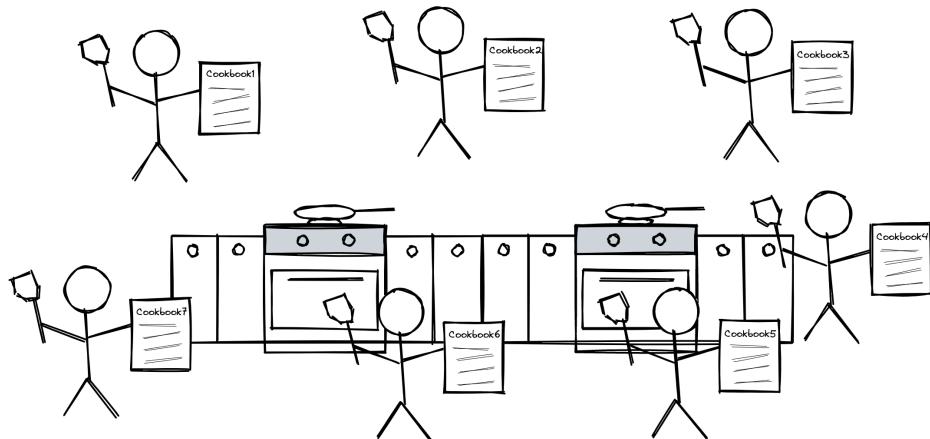
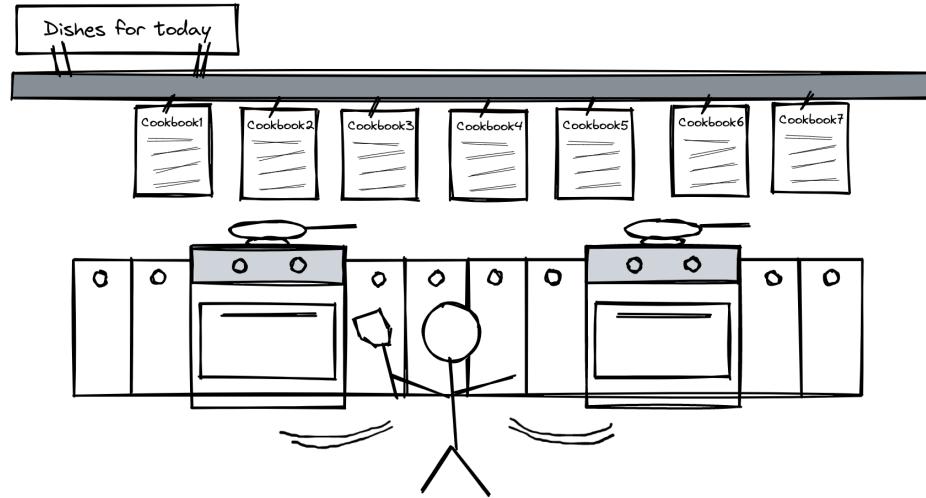


FIGURE 14.3
VM implementation: many cooks in a kitchen, each with a different cookbook.

**FIGURE 14.4**

Container implementation: one cook in a kitchen, handling multiple dishes, each has a cookbook and stays in its own pan.

Images can be shared among machines to replicate the containers initial status even if the machines adopt different underlying infrastructures such as hardware and OS.

14.2 Virtualization and Containerization Technologies

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 local area network (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 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 and re-distributed computational resources to the applications in

an efficient manner. Virtualization is one of the most important technologies 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 virtual environment emulated and managed by a special software running on the actual machine known as the “**Virtual Machine Monitor (VMM)**” or “**hypervisor**”. The setup is often transparent to the applications as they perform the same way as if they were running on a dedicated physical machine.

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 LAN (VLAN), virtual private network (VPN), NAS and storage area network (SAN).
- Virtualization of operating systems. This results in the well-known VM and virtual desktop.
- Virtualization of application running environments. An example is Java virtual machine (JVM) that allows Java to generate consistent results while running on different machines.

This chapter focuses on the virtualization of operating systems and the conventional VM concept.

There are different virtualization architectures. Commonly seen components shared by almost all architectures often include

- Host OS. The OS that resides on the physical machine, on which virtualization tools run.
- VMM. This is a software that runs on the host OS, managing all the guest OSs, providing them interface to the host OS and the hardware.
- VM, the 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 VMs. They can be widely divided into the following 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.
- Para-virtualization. The VMM does not virtualize hardware. The guest OS is modified to certain extent to suit the VMM of this type.
- Hardware-assisted virtualization. The processor of the system is specifically designed to provide certain VM functions. These functions, after passing through VMM, are directed and executed 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, one of the challenges is to enable privileged instructions for the guest OS. This can be difficult because guest OS does not naturally run in privileged mode due to the VM architecture. If the requests of the guest OS contain privileged instructions, the CPU will deny the request. 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 independent workspaces for applications. Comparing with VM, containers are often lighter, hence more efficient for massive microservices deployment. Depending on the context, the term “container” may refer to the following different concepts: container runtime, container engine, and container orchestration.

Container runtime refers to the backend software that actually executes the containers. Examples of widely used container runtimes are `containerd`, `runc` and `cri-o`.

It is often inconvenient for users to talk to container runtimes directly. 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 strategically deploy, monitor, restart, and terminate the containers on servers. It can scale up and down the number of containers based on the demanding, and balance the

API calls each container receives. Container orchestration is very useful in production environment of large-scale services. One of the most widely used container orchestrations is kubernetes.

Notice that though container runtime is always a must-have, container engine and container orchestration are not. In the old days, a container orchestration talks to a container runtime through a container engine. Nowadays, many container orchestrations directly communicate with container runtimes.

14.3 Container Engine Basics

This section introduces docker and podman container engines. Notice that the scope of this notebook does not cover execution technologies behind the screen. When comes to user experience, docker and podman share very similar interfaces, and hence can be introduced as a whole. Unless otherwise mentioned, everything introduced in the remaining of this section can run in a machine with either docker installation or podman installation and `docker` set as its alias.

14.3.1 Container Engine Choices

Docker engine is the most popular container engine available on the market as of 2023 (over 80% market share), and it is free of charge for open-source, personal and small business usage.

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**, developed by Red Hat, is an alternative to docker. It supports the same interface as docker in its CLI (as if podman were an alias), and claims to have better performance and security. RHEL has 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 contrary, podman often comes with RHEL installation by default. Kubernetes, a famous container orchestration, is also dropping docker support. According to the official statement, docker support will eventually 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 in 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.

14.3.2 Installation

As explained earlier, RHEL uses podman as its recommended container engine. This notebook focuses on RHEL, and hence podman will be used in the remaining of the chapter. Notice that docker and podman have similar interfaces. Everything introduced in this chapter should apply to both docker and podman, unless otherwise mentioned, and the command `docker` and `podman` can be used interchangeably. Given the popularity of docker, in this chapter `docker` is used as an alias of `podman`.

Podman ships with recent versions of RHEL, and does not require specific installation. In case the user needs to install podman on a RHEL distribution manually, use

```
$ sudo dnf install podman podman-docker
```

Given the popularity, docker installation on a Debian distribution is introduced below. Ubuntu Linux distribution is used as the host machine.

Firstly, remove existing docker engine and runtime as follows.

```
$ sudo apt-get remove docker docker-engine docker.io  
$ sudo apt-get remove containerd runc
```

Add docker 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
```

where notice that `docker-compose` is a toolkit to manage containers using CLI. It is a handy tool in the testing environment.

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
```

As for podman, it is designed to be “rootless”. Thus, many podman commands do not require sudo privilege. Yet, there are exceptions. It is possible to change the sudoer rule as follows, so that when executing `sudo podman`, the password is not required. Open sudoer rules by

```
$ sudo visudoer
```

and add

```
<sudoer user name> ALL=(ALL) NOPASSWD: /usr/bin/podman
```

14.3.3 Container Manipulation

This section discusses basic container manipulation such as launching and stopping a container. As mentioned earlier, command `docker` is used as an alias to `podman`.

Launch a Container from an Image

To launch and run a container from an image, simply use

```
$ docker run <configuration> <image>
```

Docker will search the local and default remote repositories for the image, download the image if necessary, and start a container from that image. Customized remote repositories configuration is introduced later.

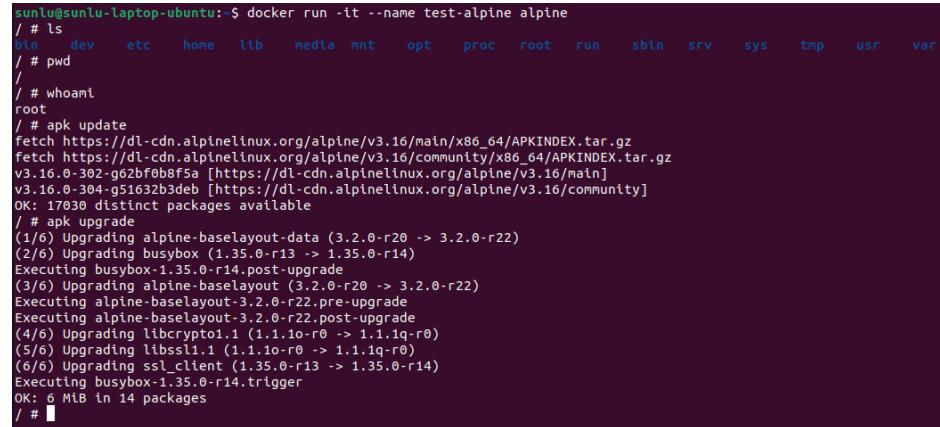
By default, `docker run` starts a container in attached mode (frontend

mode) unless flag `-d` is used. After successful execution and completion of all the tasks, the container will enter “exited” status.

For example, consider running a container of alpine using the command below. 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 “TeleTYewriter”, which enforces the I/O of the container to follow 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. Flags `-it` are often used when running a container in attached mode. Finally, `--name` assigns a name to the container. Without an assigned name, docker will assign a random name to 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: 17830 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 libcrypt01.1 (1.1.10-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 alpine container, with interactive TTY and name *test-alpine*.

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. This is because the container is running in attached mode, and flags `-it` map the the containers input and output with the console.

While keeping the container running, open another terminal and use `docker container ls` to check the status of the container. More about container commands are collectively introduced later. 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.

```
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 14.6List the running container *test-alpine*.

```
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
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 14.7List the exited container *test-alpine*.

Functional wise, **docker run** executes two commands behind the screen, namely **docker create** and **docker start**, where **docker create** creates a container and **docker start** starts the container 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>
```

Differences between **docker run** and **docker start**

Do note the fundamental differences between **docker run** and **docker start**. Firstly, **docker run** starts a container from an image and it always creates a new container, whereas **docker start** starts an existing but exit container. Secondly, **docker run** runs a container in attached mode by default unless **-d** flag is used, while in contrast **docker start** runs a container in detached (backend) mode by default unless **-a** is used.

An example of using **docker run** with **-d** flag to start a container in detached mode is given below.

```
$ docker run -d --name test-background-alpine alpine
```

By changing **-it** 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 Table 14.1.

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 by amending the revised startup commands to the image name as follows.

```
$ docker run <image> <revised command>
```

TABLE 14.1
Commonly used docker commands to launch a container.

Command	Flag	Description
<code>docker run</code>	—	Launch a container from an image in attached mode by default. If the image cannot be found locally, it downloads the image from the remote repository automatically.
<code>docker start</code>	—	Start an existing but exit container in detached mode by default.
<code>docker run</code>	<code>-i</code>	Keep the standard input of the container open when launching the container.
<code>docker run</code>	<code>-d</code>	Launch the container in the backend and keep it running.
<code>docker run</code>	<code>--rm</code>	Automatically remove the container when exiting. The removed container will not be listed in <code>docker container ls -a</code> as an exit container. This is usually used in testings.
<code>docker run</code>	<code>-t</code>	Allocate a pseudo-TTY. The flag usually comes with the flags <code>-i</code> or <code>-d</code> , to form <code>-it</code> or <code>-dt</code> .
<code>docker run</code>	<code>--restart</code>	Enforce restart of the container upon exiting. This is usually used on containers running in the backend. Commonly used restart configurations include <code>--restart no</code> (do not restart), <code>--restart on-failure[:<max retries>]</code> (restart if exits with an error flag), <code>--restart always</code> (always restart when exists).
<code>docker run</code>	<code>--name</code>	Assign a name to the container.

where `<revised command>` must be defined in the image, or otherwise an error will be raised.

Interact with a Container in the Backend

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>
```

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.

Stop and Remove a Container

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, use

```
$ docker (container) rm <container>
```

where `container` can be neglected. Alternatively, use

```
$ docker container prune
```

to remove all exited containers. Notice that there is a more powerful command

```
$ docker system prune
```

which removes not only exited containers but also unused network configurations, dangling images and cache.

Rename a Container

To rename a container (without changing its container ID or anything else), use

```
$ docker rename <container-old-name> <container-new-name>
```

Monitor the Status of a Container

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>
```

To quickly check container status including resource consumption (CPU, memory usage, etc.), use

```
$ docker stats [<container>]
```

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>
```

Finally, to check the logs of a container such as its standard output or error message, use

```
$ docker logs <container>
```

Exchange Files with a Container

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 a Container into an Image

A container is usually generated from an image. It is also possible to do vice 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.

Publish Ports for a Container

By default, containers do not expose any ports to the outside world. A container can be accessed only from its host machine using APIs provided by docker, such as `docker cp` to transfer files and `docker exec` to execute commands, but not network protocols. The user can allow public access from the internet by publishing the ports of the container using

```
$ docker run -p <host machine port>:<container port> <image>
```

when starting a container. To publish multiple ports, use multiple `-p` flags in a command.

More about containerized application connectivity to the internet, to the host machine and to other containerized applications are introduced in later sections.

14.3.4 Containerized Application Deployment Example

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 `alpine`, and install `nginx` on `alpine`.

```
$ 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. Notice that for read-only configuration files, unlike what we are doing now, a good practice is to build them into the read-only image layers. Keep in mind that this serves only as an demonstrative example. More about images are given in later sections.

```
$ 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 and Bind Mount

When a container is running, files inside it can be accessed or copied bidirectionally using the `docker cp` command. However, all data created or modified during the container's life cycle resides in the container's writable layer. More about the concept of layers are introduced in later section. If the container is removed, all data stored in its writable layer is also lost.

To ensure data persistence, it is best practice to use Docker volumes or bind mounts. Both methods link storage on the host machine to the container, allowing data to persist even if the container is removed.

- Docker volumes are fully managed by Docker. The user does not directly interact with the storage on the host machine. There are two types of docker volumes, the anonymous volume and the named volume.
- Bind mounts, on the other hand, allow the user to specify and manage the storage location on the host machine, providing direct control over the data.

In addition to data persistence, named docker volumes and bind mount can also play as a hub to shared data among multiple containers, which facilitates data exchange and allows containers to work on the same dataset.

14.4.1 Volume

In this section, we only consider named docker volume.

To create a docker volume, use

```
$ docker volume create <volume>
```

Notice that docker volume is fully managed by docker. The user cannot, and does not need to specify where the data should be stored in the host machine.

To list down volumes and to inspect a volume, use

```
$ docker volume ls  
$ docker volume inspect <volume>
```

respectively. Notice that when using `docker volume inspect`, docker gives the mount point of the volume. However, the mount point is often a place in a virtual machine that docker creates, therefore, difficult to locate in the host machine. Like said earlier, the user should not access the data of a volume from the host machine directly.

Finally to remove specific an unused volume(s) or all unused volumes, use

```
$ docker volume rm <volume>  
$ docker volume prune
```

respectively. When a volume is removed, all the data is lost.

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 path>[:ro] <image>
```

which should mount `<volume>` to `<container internal path>`. If `<volume>` is not created before hand, docker will create a docker volume with the name. The optional `:ro` can be used if it is a read-only volume, i.e., the container can only read from the volume but not write back to the volume. This can become handy if the volume is shared by multiple containers and only one of them is allowed to writer to the volume while others being only the listener.

Notice that if not specifying the volume name when using flag `-v`, i.e.,

```
$ docker run -v <container-path> <image>
```

an anonymous volume will be created and used. Further more, if `--rm` is used, when the container is stopped, the container together with the anonymous volume will be removed. A use case of anonymous volume is given in the later section. Anonymous volumes can also be set up in the Dockerfile.

14.4.2 Bind Mount

Bind mount works similarly as docker volume, except that the user controls the location on the host machine where the data is persisted. Recall

```
$ docker run -v <volume>:<container-path>[:ro] <image>
```

which associate a named docker volume to the path in the container. Instead of `<volume>`, specify the path in the host machine as follows

```
$ docker run -v <host-machine-path>:<container-path>[:ro] <image>
```

in which case docker will persist and synchronize the data on the paths inside and outside the container. Quotation marks can be used "`<host-machine-path>`" if there are spaces or special characters in the host machine path. Notice that the specified host machine path should be accessible by docker. Likewise, the optional `:ro` can be used to prevent the container from overwriting the host machine.

Bind mount is often used in development stage where the developer wants to map application source code or data on his host machine into the container directly. When the application is ready for shipping, the consolidated source code and data should be built into the image. You cannot expect the end user to have a separate copy of the source code and data on his machine, and to use bind mount each time he starts the application.

Following that spirit, a developer can create a "utility container". He can start a container in `-it` mode, install everything he needs to develop an application in that container, and develop the application there. Using bind mount, all the code he writes will be mirrored to his host machine. With this approach, he does not need to install all the dependencies of the application in his host machine.

14.4.3 Multiple Volumes

It is possible to run a container with multiple `-v` flags, each pointing to a different path in the container. Should there be any clashes, the rules with more specific path wins. For example, consider

```
-v <volume or path 1>:/app -v <volume or path 2>:/app/data
```

used together in a `docker run` command. In this example, `/app/data` will be mounted to `<volume or path 2>` and everything else in `/app` to `<volume or path 1>`.

Consider

```
-v <volume or path 1>:/app -v /app/data
```

which mount an anonymous volume to `/app/data` to protect it from being overwritten by any content in `<volume or path 1>`. This is a good use case of anonymous volume.

14.5 Docker Image

Images are 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.

14.5.1 Dockerfile Programming

An image shall contain everything needed to create and initialize a container. This includes but not limited to:

- Necessary steps to create a container
- 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 portable, 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 most common and flexible way to create a docker image is to build from Dockerfile. 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
 - ...
- (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. 14.8. Notably, the Dockerfile itself isn't included within the resulting image.

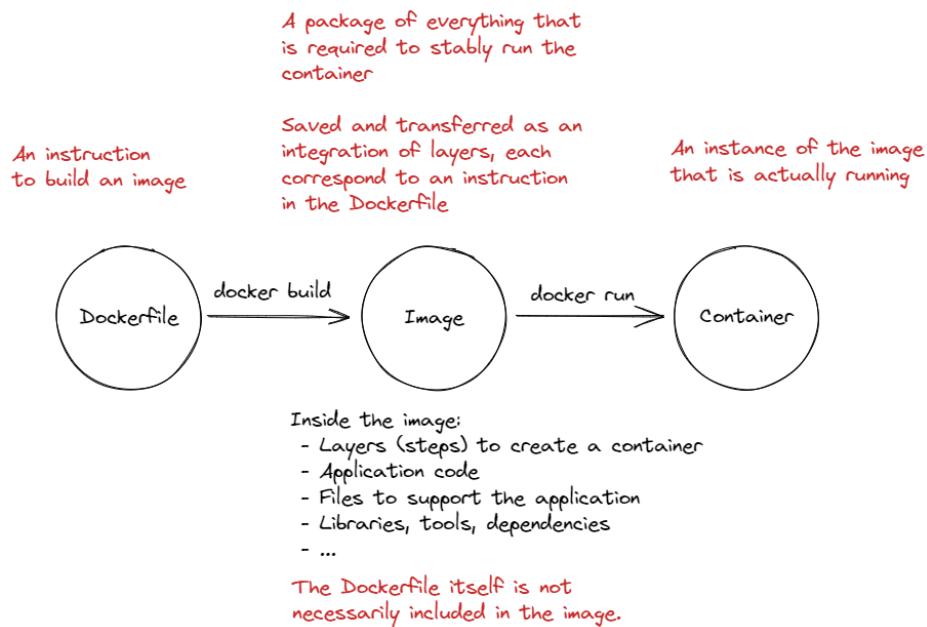


FIGURE 14.8

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 are not saved or transferred redundantly, hence significantly reducing image sizes. More details can be found at <https://docs.docker.com/storage/storagedriver/>.

Docker Image Layers

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 in Fig. 14.9.

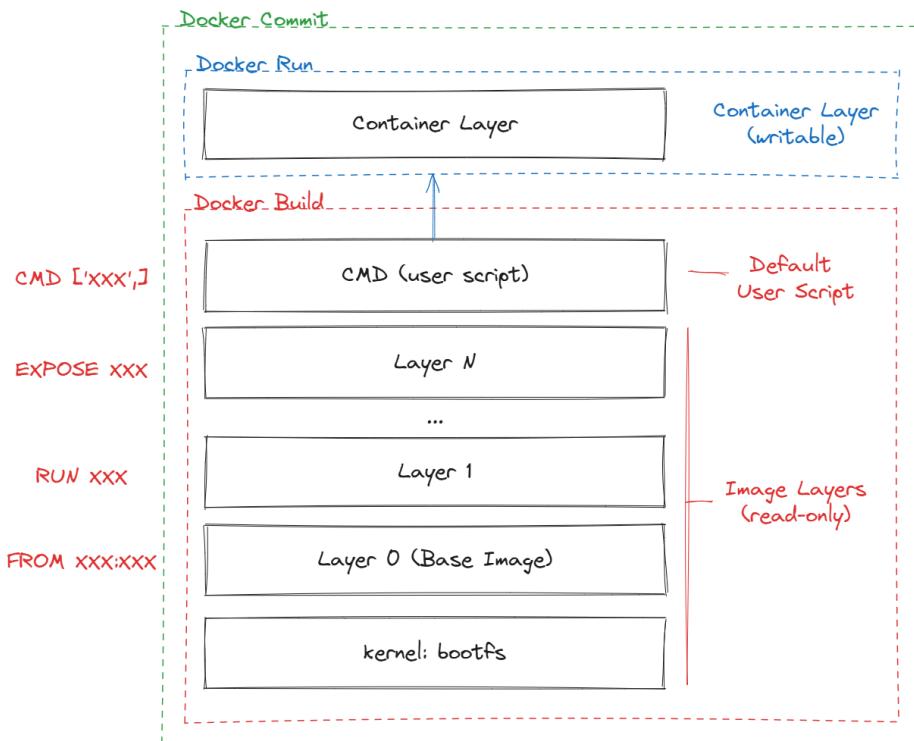


FIGURE 14.9

A demonstration of docker image layer structure.

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"]
```

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 application.

The commonly used keywords to be used in a Dockerfile to realize the above instructions, such as `FROM`, `RUN`, are explained in Table 14.2.

It is worth highlighting the distinction between `RUN` and `CMD`. The `RUN` command is executed during the building of the image, which creates a virtual representation of the environment to run the service. Therefore, `RUN` is used to prepare the environment, for example, to install necessary libraries. On the other hand, command `CMD` simply specifies the default command to run when the container starts. It is built together with the image, but not considered as a read-only image layer. Indeed, the contents in `CMD` can even be overwritten if the user wishes to do so when `docker run` executes.

Besides Table 14.2, 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 which can later be referred as `$<key>` in the Dockerfile. An example is given below.

```
ENV PORT=80  
EXPOSE $PORT # this is interpreted as EXPOSE 80
```

TABLE 14.2

Critical keywords used in a Dockerfile.

Syntax	Description
<code>FROM <image>[:<tag>]</code>	Define the base image. A Dockerfile must start with a <code>FROM</code> instruction. In case of multiple <code>FROM</code> instructions, the last one statement is the base image and the earlier ones intermediate images. An optional <code>:<tag></code> can be used to specify the version of the base image.
<code>RUN <command></code>	Execute a shell command.
<code>WORKDIR <path></code>	Set the working directory from the point onwards.
<code>ADD <src> <dest></code>	Add (Copy) <code><src></code> , either a directory/-file or URL, to <code><dest></code> . An optional <code>[--chown=<user>:<group>]</code> can be used to specify the owner and group of the added files.
<code>COPY <src> <dest></code>	Similar with <code>ADD</code> . <code>COPY</code> is easier but less powerful and cannot handle tar or URL.
<code>USER <user></code>	Switch user from the point onwards.
<code>EXPOSE <port></code>	Specifies the ports that the container shall listen to. An optional <code>/<protocol></code> can be used to specify the protocol for communication.
<code>VOLUME [<path>]</code>	Attaches an anonymous volume to the specified path inside the container.
<code>CMD [<exe>, "<arg>",]</code>	The user-script command. This is the last instruction in the Dockerfile that usually starts the APP. Notice that a Dockerfile can only contain one <code>CMD</code> instruction. The user script is included in the image but not as a the read-only layer.

Notice that environmental variables are accessible by the application code. They can be overwritten during `docker run`. This means that the user can potentially use environmental variable to dynamically change the behavior of the application in the container when using `docker run`. There are several benefits of doing this, for example, to enhance security. For example, if there is a password that the user needs to key in during the start of the container, the developer may want to put it as environmental variable and let the user to fill in, instead of hardcode the password in Dockerfile.

Another useful features of Dockerfile is the Dockerfile argument. It leaves a “space” in the Dockerfile that the developer can later fill in when he builds the image. An example is given below.

```
ARG DEFAULT_PORT[=80]
ENV $DEFAULT_PORT
EXPOSE $PORT
```

The default value [=80] is optional. When building the image, use `--build-arg <variable-name>=<value>` to override the argument value. Notice that unlike environmental variables, Dockerfile arguments are not accessible by the application code or by the `CMD` instruction.

Additionally, `LABEL <key>=<value>` assigns a tag to the image, which can be displayed when `docker inspect <container>` is used.

Dockerfile programming can be very flexible and complicated at the same time. The advanced materials goes beyond the scope of this notebook.

Examples of Dockerfiles are given below, one from *docs.docker.com* and the other from Linux Academy. The docker image layer structure of the second example is given in Fig. 14.9 as a demonstration. Notice that in Fig. 14.9, `bootfs` refers to the “boot file system”, including the bootloader and the Linux kernel. Upon creation of a container using `docker run`, a container layer will be added to the image, as shown by the blue dashed box in Fig. 14.9. 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. 14.9.

```
# 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"]
```

14.5.2 Build an Image

With the Dockerfile ready, use `docker build` to build an image. An example is given as follows.

```
$ docker build <path/url> -t <image name>
```

or

```
$ docker build <path/url> -t <user name>/<image name>:<tag>
```

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. For the convenience of sharing, it is recommended that all images to be pushed to a public registry (such as Docker Hub) shall be tagged with user name and version. Notice that the tag can be changed later after the image is built.

It is worth mentioning that `docker build` builds the images by layers in the Dockerfile, and it will not re-build the same layer to which point no changes are made. This feature suggests that we should put the common and static layers in the early part of Dockerfile, and customized and user-defined layers in the late part, thus making the building (and also sharing, as will be introduced later) of the images more efficient.

Use `.dockerignore` to exclude any files that should not be built into the image. Such files may include caches, the Dockerfile itself, etc.

Dockerfile supports multi-stage image build. An example is given below.

```
# Stage 1: Build
FROM node:18 AS builder
WORKDIR /app
COPY package.json yarn.lock .
RUN yarn install
COPY ..
RUN yarn build

# Stage 2: Runtime
FROM node:18-slim
WORKDIR /app
COPY --from=builder /app/dist /app/dist
CMD ["node", "dist/index.js"]
```

The idea is like the following.

- The earlier stages prepare files. The later stages use `COPY --from` to copy the results of earlier stages to the current stage.
- Only the last stage is included in the final image.

A benefit of using multi-stage build is to reduce the image size. As introduced earlier, the intermediate results of the earlier stages will not be included in the final image unless they are `COPY --from` by the last stage. For example,

the user may want to compile his source code in an earlier stage, and only copy the compiled binary file into the last stage. In this way, the user avoids including the source code in the final stage which he needs to share with others.

The COPY --from syntax is given below.

```
COPY --from=<stage-name-or-index> <source-path> <destination-path>
```

14.5.3 Docker Image Management

The most commonly used image operations can be categorized as follows.

- Create an image.
- 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 rmi <image name> # same as docker image rm
$ 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>
```

14.5.4 Docker Image Sharing with Docker Hub

There are two ways to share an image:

- Share the Dockerfile and APP source code so that the receiver can build docker image from his end.
- Share the image directly.

It is often more convenient to share the built image instead of the Dockerfile and the source code. However, a docker image is stored and managed by layers and cannot be shared simply by file transfer.

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>
```

Notice that when using `docker run` on an image without a specified version and the image is not stored locally, Docker will automatically search and pull the latest version of that image from the remote repository, usually Docker Hub by default. However, if the image already exists locally, Docker simply uses that image and will not automatically check the remote repository for the latest version. To update the local image, manually use `docker pull` to pull the latest or specific version of the image from the remote repository.

14.6 Multi-Container Deployment and Orchestration

This section discusses the deployment and orchestration of multi-container applications. Different methods to enable container-to-host and container-to-

container communications are introduced. Docker compose, a useful tool to automatically configure and start containers of a multi-container application, is also introduced.

14.6.1 Communication

In practice, a containerized application needs to communicate with applications running outside the container to request services such as API calls. Commonly seen use cases are as follows.

- Containerized application requests services from the Internet.
- Containerized application requests services from the host machine, for example, from a database deployed on the host machine.
- Containerized application requests services from other containerized applications.

With proper port mapping using `-p`, the communication with the Internet is automatically enabled. Therefore, the focus of this section is mainly on data exchange and API calls between the container and the host machine or other containers.

The containerized application may request services from the host machine. In this case, the application code needs to be modified to enable communications with the host machine. An example is given below.

Consider a simple scenario where the containerized application needs to send a request to the host machine to trigger an API call of MongoDB. Notice that the MongoDB is installed on the host machine and not in the container.

If the application were running on the host machine instead of in a container, it can communicate with the MongoDB server using a connecting string that looks like the following

```
'mongodb://localhost:27017/<database name>'
```

and it should be part of the application code. However, it would not work if the same code is used in the containerized application. The code needs to be modified as follows.

```
'mongodb://host.docker.internal:27017/<database name>'
```

where `host.docker.internal` is a recognized domain by docker that refers to the host machine. Docker translates the domain to the IP address of the host machine.

Notice that `host.docker.internal` can be used to refer to the host machine not only with MongoDB requests, but also with other requests such as a general HTTP request.

Two containers running on the same machine can always talk to each other via the host machine. However, this may become inefficient in some

applications. Therefore, we need to investigate on direct container-to-container communication.

As a basic solution, cross container communication can be done following the example given below. Notice that this is not a common practice in production environment as it is not robust and efficient.

Consider a simple scenario where there are two containers, one running the application and the other hosting a MongoDB.

The container with MongoDB deployment is started as follows.

```
$ docker run -d --name mongodb mongo
```

where `mongo` is the official docker image for MongoDB on Docker Hub.

The IP address of the MongoDB container can be found using

```
$ docker container inspect mongodb
```

where `mongodb` is the name of the container specified in the earlier command. In the result of the inspection, look for `IPAddress` under `NetworkSettings`.

Then in the application code, replace `localhost` in the connecting string

```
'mongodb://localhost:27017/<database name>'
```

with the IP address of the container.

Notice that each time a container is deployed, its network settings are done by docker and they differ from machine to machine. Therefore, this basic approach only works temporarily in development environment and should not be used in the production environment.

As a more established approach to realize cross container communication, docker allows the user to create container networks. All containers in a container network have reserved IP addresses assigned by docker and they can talk to each other.

To create a container network, use

```
$ docker network create <network name>
```

Then to associate an container with that container network, use `--network` as follows

```
& docker run --network <network name> <image name>
```

If two containers are put under the same container network, they can talk to each other.

Consider the earlier example. We can create a container network using

```
$ docker network create my-network
```

and launch MongoDB as a containerized application in the container network using

```
$ docker run -d --network my-network --name mongodb mongo
```

In the application code, use

```
'mongodb://mongodb:27017/<database name>'
```

as the connecting string, and launch it in the same container network. Docker will automatically resolve `mongodb` as the correct container with that container name when the application sends out the outbound request.

Notice that when using container network for cross container communication, there is no need to do port match using `-p`. Indeed, `-p` is used only when the container needs to talk to the outside world through the host machine. Cross container communication as well as container to host machine communication happens inside the host machine, thus not requiring port mapping.

14.6.2 Docker Compose

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 Portainer and Kubernetes are helpful with managing containers. Many of these tools are able to automatically adjust the number of containers and balance their loads.

Many applications nowadays are multi-service multi-container applications. This trend has made container orchestration very useful in practice.

Consider building and launching a multi-container applications. Multiple `docker build` and `docker run` commands need to be executed in specific order, and each command coming with a long list of arguments. Doing this manually can be tedious and introduce a high chance of human error.

A docker compose file is essentially a configuration file in YAML, usually named `docker-compose.yml` (in later versions of docker compose, it can also be named `compose.yml`). It plays as the user script or configuration file that automate the processes. Docker compose can process that configuration file, then build and launch containers accordingly. From that sense, docker compose can be seen as a container orchestration tool.

This section gives a brief introduction to the preparation of a docker compose file. Examples of docker compose files can be found elsewhere online, such as github.com/docker/awesome-compose.

A typical docker compose file looks like the following.

```
version: "<compose file format version>"  
services:  
  <container name 1>:  
    image: "<image name>"  
    container_name: "<container name>"  
    volumes:  
      - <volume name 1>:<container path> # named volume  
      - <container path> # anonymous volume  
      - <host path>:<container path> # bind mount  
      - ...  
    environment:  
      - <variable name 1>=<value 1>  
      - <variable name 2>=<value 2>
```

```

    - ...
env_files:
    - <path to environmental variable file 1>
    - <path to environmental variable file 2>
    - ...
networks:
    - <network name>
ports:
    - "<host port>:<contianer post>"
<container name 2>:
build:
    context: <path to Dockerfile directory>
    dockerfile: <name of docker file>
volumes:
    - <volume name 2>:<container path> # named volume
    - <container path> # anonymous volume
    - <host path>:<container path> # bind mount
    - ...
environment:
    - <variable name 1>=<value 1>
    - <variable name 2>=<value 2>
    - ...
env_files:
    - <path to environmental variable file 1>
    - <path to environmental variable file 2>
    - ...
networks:
    - <network name>
ports:
    - "<host port>:<contianer post>"
depends_on:
    - <container name>
    - ...
volumes:
    <volume name 1>: # for named volumes only
    <volume name 2>:
    ...

```

Some highlights are as follows.

- **version** tells the the compose file format version. Docker compose will then parse the compose file accordingly.

Different compose file format version may support different fields. If the version field is omitted, the composed file is parsed according to the latest version.

- **services** describes the container(s) to be deployed. Each container is corresponding with a subcomponent under services.

For simple containers whose image is readily available without

`docker build`, the following fields are commonly used in the subcomponent.

- `image` specifies the image.
- `container_name` specifies the container name; if not specified, default names will be used which are still human-readable.
- `volumes` defines volumes and bind mounts.
- `environment` defines environmental variables.
- `env_file` includes a file inside which are environmental variables.
- `networks` adds the service to a docker network. Notice that this is often not necessary as all the services will be automatically added to the same newly created default docker network by default, unless a network is specified.
- `ports` maps host ports to container ports.
- `depends_on` lists containers that need to be launched before the current container.

In the case where a new image needs to be built before launching the container, docker compose file requires the following fields to build the image.

- `build` gives the paths to the docker file.
 - * `context` gives the directory of the docker file. Everything to be built into the image must be in this directory or its subdirectories.
 - * `dockerfile`: the name of the Dockerfile.

Notice that one can simply use `build: <context>` if Dockerfile is named `Dockerfile`.

- `volumes` lists all the named volumes used by the services.

A docker compose file example is given below. This example is taken from `awesome-compose` on GitHub.

```
version: "3.8"
services:
  web:
    image: nginx
    volumes:
      - ./nginx/nginx.conf:/tmp/nginx.conf
    environment:
      - FLASK_SERVER_ADDR=backend:9091
    command: /bin/bash -c "envsubst < /tmp/nginx.conf > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"
```

```

depends_on:
  - backend
backend:
  build:
    context: flask
    target: builder
  # flask requires SIGINT to stop gracefully
  # (default stop signal from Compose is SIGTERM)
  stop_signal: SIGINT
  environment:
    - FLASK_SERVER_PORT=9091
  volumes:
    - ./flask:/src
depends_on:
  - mongo
mongo:
  image: mongo

```

Finally, execute docker compose as follows.

```
$ docker-compose up -d
```

if the docker compose file is named as the default name (usually `docker-compose.yml`), or

```
$ docker-compose -f <docker compose file name> up -d
```

if otherwise. Docker images shall be built and containers be launched accordingly. And to shutdown all the services, simply use

```
$ docker-compose down
```

Docker compose would not re-build the images by default if it thinks that the images it will use have been built before. To enforce rebuild, consider using `--build` flag. Alternatively, use

```
$ docker-compose build
```

to build the images in the docker compose file without launching any container.

14.7 Container Cloud Deployment

So far we have been discussing container development and local testing. One of the most important selling points of containerization is that it is easy to ship applications. This is because the environment that supports an application is packaged together with the application in the image, which ensures that the application can run consistently whatever the host server is.

Nevertheless, when the application is pushed to production environment,

some adjustments need to be made. To name a few, below is a list of probable adjustments when shipping from development to production.

- Bind mounts should not be used. Use `COPY` to encapsulate all the resources needed, and use volume if data persistence is required.
- Some container setups may need to be changed.
- For multi-container applications, the containers may need to be deployed on multiple hosts.

This section discusses the deployment of containers on the cloud in production environment. Notice that for large scale applications deployment in the production environment, container orchestration tool such as Kubernetes is almost certainly necessary. Kubernetes and its cloud deployment are discussed in the later chapter and they are not considered in this section.

14.7.1 Docker Hosting Providers

Assume that the application can run correctly on the host machine. To deploy it on the Internet and allow public access, the first step is to find a docker-supporting hosting provider. There are many such providers, the most popular of which include Amazon Web Services (AWS), Microsoft Azure and Google Cloud. They are not just docker hosting providers but more general cloud services providers.

Details about the cloud services providers and cloud computing are beyond the scope of this notebook. Only the services relevant to container deployment are introduced here. AWS is considered.

14.7.2 VM-Based Approach

The most intuitive way to deploy containers on AWS is to use EC2, the VM as part of AWS's IaaS solution. To do that, the following steps are required.

1. Start EC2, and enable SSH access to the EC2 from the local machine.
2. Install docker on the EC2 instance.
3. Upload the images which have been built in the local machine to the EC2 instance.
4. Start the containers on EC2.
5. Enable public access to the EC2 by publishing its IP address.

Notice that EC2-based deployment is not the best practice as AWS and many other cloud service providers have managed container management services such as Elastic Container Service (ECS) and Elastic Kubernetes Service

(EKS). However, for illustration purpose, we will start from EC2-based deployment.

The key steps to launch and connect to a EC2 instance from a Linux local machine are given below.

1. Create and configure the EC2 instance from AWS dashboard.
2. In the last step of EC2 instance configuration, create or select a key pair. The key pair is used later to SSH to the instance. Download the private key pair. The name of the private key pair is by default `<key pair name>.pem`.
3. Change the ACL of the private key pair to 400 to disable public view using `chmod 400 <key pair name>.pem`.
4. Launch the EC2 instance from AWS dashboard. Wait until the instance state becomes “running”. Locate the public DNS of the instance from the dashboard which usually looks like `<serial number>.<region>.compute.amazonaws.com`.
5. SSH to the EC2 instance using `ssh -i "<key pair name>.pem" <username>@<public dns>`.

To install and start docker on the EC2 instance, use

```
$ sudo yum update -y  
$ sudo yum install docker  
$ sudo systemctl start docker
```

To upload an image from the local machine to the EC2 instance, use Docker Hub as a bridge. The user can then launch containers on the EC2 instance the same way he would do on a local machine. Lastly, to enable public access to the EC2 from HTTP, configure the security group inbound rules attached to the EC2 instance. By default, the only allowed inbound rule is SSH using TCP protocol via port range 22 from all IP addresses. Depending on the applications running on the EC2 instance, the user may want to add more inbound rules, for example, enabling HTTP/HTTPS access via port 80.

For multi-container applications, the user can upload all necessary images to Docker Hub, and on the EC2 instance simply execute a docker compose file. Notice that the docker compose file should use the images on Docker Hub instead of building any on the fly. In production environment, all the images should be pre-built.

The EC2-based approach has some disadvantages compared with ECS and EKS that will be introduced in later sections. There are at least the following disadvantages.

- Lack of robustness. EC2 belongs to the Infrastructure as a Service (IaaS) family where AWS only supports the infrastructure (in this case, the VM) and not anything running on it (in this case, docker and the containerized application). The user needs to deal with all the “accidents” that may

break the system such as someone launching a cybor attack on the VM. This makes the solution less robust.

- Difficult to scale. The total computational power of the application depends on the setup of VM which is pre-determined and cannot be easily scaled up or down on the fly.
- Difficult to manage. The user needs to SSH to the VM to audit and upgrade the applications, which can become annoying.

This makes EC2-based container approach almost always suboptimal compared with ECS and EKS, especially if the user is not an experienced administrator.

14.7.3 Managed-Service-Based Approach

ECS and EKS are the recommended managed services of AWS for containerized applications. EKS will be introduced in the later chapter with Kubernetes. This section discusses ECS. Notice that when using a managed service like ECS, it is no longer the user's concern what container runtime or engine the managed service is using in the backend. The user does not need (and should not) interact with Docker directly. Instead, he should use the interface ECS provides.

As a prerequisite, make sure that the images to be used are already in Docker Hub.

Single-Container Applications

Key steps to deploy containerized applications using ECS are summarized below. The user needs to configure “container”, “task”, “service” and “cluster” to be used for the application.

Container defines how a container should be launched. In container definition, the following items need to be configured.

- Container name.
- Image path. If the image has been uploaded to Docker Hub, use `<username>/<image>:<tag>`.
- Memory limit.
- Port mapping.
- If health check is necessary, health check related configurations such as timeout, number of retires, etc.
- If the default `CMD` needs to be overwritten when starting the container, the command to overwrite `CMD`.

- If the default WORKDIR needs to be overwritten, the revised working directory path.
- The value of environmental variables, if any.
- Network configuration.
- Volume configuration.
- Log configuration.
- ...

One can think of most of the above as the customization of `docker run` command which has already been introduced in earlier sections.

Task defines the server requirements that host the containers. The user needs to define task definition name, network model, compatibilities request, etc. For example, in compatibilities if “FARGATE” is selected, AWS runs the containers in a serverless manner, whereas if “EC2” is selected, AWS creates dedicated VMs to run the containers.

Service plays as the “orchestrator” of tasks. It defines the number of tasks to be used and how load is balanced among the tasks.

Finally, cluster defines the virtual private cloud (VPC) and subnets where the application should be running, as part of the network configuration in AWS architecture. Cluster definition is important if multiple applications need to be run under the same virtual network, or if the application needs to access certain resources inside certain VPC.

Once all the configurations above are done properly, ECS will start the application. From the dashboard, one can get the public IP address of the task where the containers are hosted, and access the containers using that IP.

To update the image, go to “Task Definitions”, click “Actions”, and choose “Update Service”, “Force New Deployment”. AWS will then retrieve the new image and restart the task. Of course, one can always start a new task, in which case EC2 will also retrieve the new image.

Multi-Container Applications

On the local machine or the VM, multi-container applications can be launched easily using docker compose. ECS also supports the launch of multi-container applications.

However, ECS does not use docker compose file to start multi-container applications. One of the reasons for that is that docker compose assumes single physical machine deployment, hence supporting all the notions that the user can use to setup networks, volumes, etc. This is not the case of ECS which does not necessarily deploy everything on one host. This makes the docker compose file less relevant. Nevertheless, the docker compose file, if already exists, can be used as a blue print and the user can refer to the file when deploying containers using ECS.

In ECS, if multiple containers are assigned to the same task, then they are guaranteed to run on the same machine and under the same network. In that case, the application can use `localhost` to communicate with other containers, where `localhost` can be taken as a docker container created for all the containers running on the task. (Recall that to enable container-to-container communication on a local machine, `localhost` in the application code needs to be changed to either the IP address of the container, or the name of the container if they are in the same docker network.)



15

Kubernetes

CONTENTS

15.1	Kubernetes Basics	225
15.1.1	Infrastructure	226
15.1.2	Installation	229
15.1.3	Kubernetes Cluster Manipulation	231
15.2	Basic Kubernetes Objects	237
15.2.1	Pod Object	237
15.2.2	Deployment Object	238
15.2.3	Service Object	241
15.2.4	ConfigMap Object	242
15.2.5	Ephemeral Volume	244
15.2.6	Persistent Volume	247
15.3	Connectivity	250
15.3.1	Pod-Internal Communication	250
15.3.2	Cluster-Internal Communication	251
15.3.3	External Communication	253
15.3.4	Ingress Object	254
15.4	Kubernetes Cloud Deployment	255
15.4.1	Setup and Connect to EKS Cluster	255
15.4.2	Configure Node Group	256
15.4.3	Apply Kubernetes Configuration Files	256
15.4.4	Use Volumes	257
15.5	Kubernetes IDE and Alternatives	257

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 Kubernetes Basics

Kubernetes, also known as **k8s**, is an open-source container orchestration system originally developed by Google. It automates the deployment, health screening and scaling of containers for containerized applications.

As an orchestration tool, Kubernetes mainly focuses on the following tasks:

- Monitor the status of the containers, and restart / replace broken ones.
- Balance the load assigned to the containers.
- Strategically scale up and down the number of containers based on the total load.

From the above, Kubernetes can be seen as a more powerful practice of what docker compose tries to do. It comes with additional features such as the support on multi-server deployment.

Managed services such as ECS can also do container orchestration. However, these managed services are often proprietary and inclusive to a particular cloud service provider. With such proprietary services, it is difficult to migrate the applications across platforms or to local premises. Kubernetes, on the other hand, is open-source and platform-independent, hence would not pose that problem.

Nowadays, many cloud service providers also support Kubernetes-based container deployment solutions such as Elastic Kubernetes Service (EKS) by AWS and Google Kubernetes Engine (GKE) by Google Cloud. They are also managed services to some extend and each platform usually have some unique features, but in general the application can be migrated across platforms without too much difficulty.

Kubernetes and Docker Engine

Kubernetes was built on top of docker engine. It used a special program **dockershim** to talk to the underlying docker engine. Recently, however, docker support has been deprecated in Kubernetes. Consequently, **dockershim** has also been removed. Now Kubernetes directly talks with container runtimes.

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 “master-worker” manner, where the **master node** (also known as **control plane**) in yellow interacts with the user and schedules the Pods to be deployed

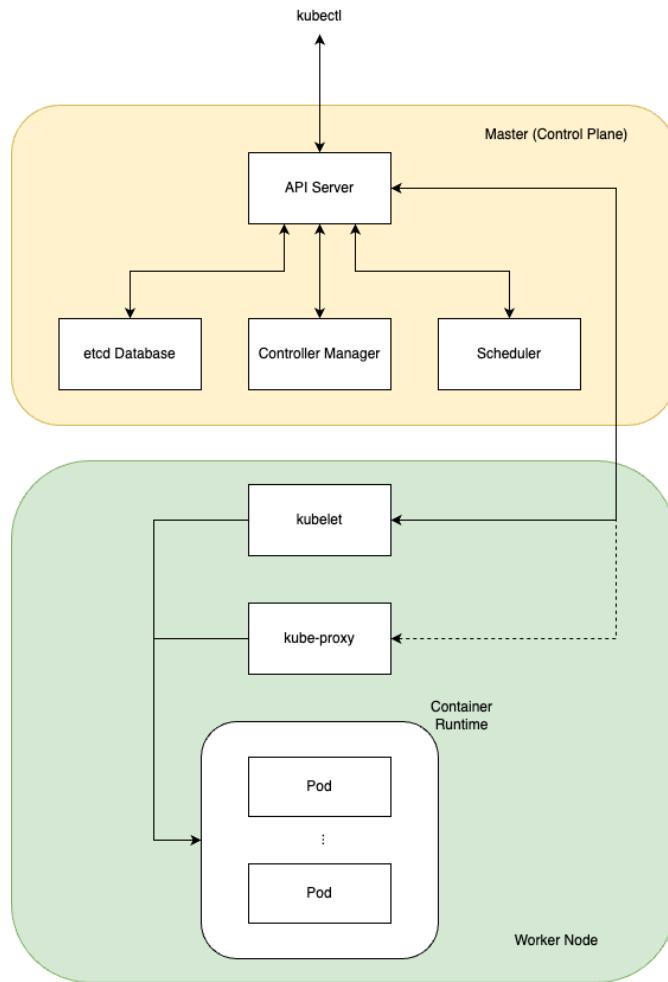


FIGURE 15.1
Kubernetes cluster and its key components.

for each work node. The **work nodes** (also known as **nodes**, for short) in green, on the other hand, process the application data. The arrows in Fig. 15.1 represents the control flow. Notice that the application data flows differently and it does not pass through the master. There can be multiple master nodes (in replica mode) and worker nodes, though only one of each is shown in Fig. 15.1.

Key components in the master and the nodes and their functions are summarized in Table 15.1.

Each node can host multiple Pods. A Pod object is a single container or a

TABLE 15.1

Key components in Kubernetes master and nodes.

Component	Location	Description
<code>etcd</code>	Master	A distributed key-value pair database that stores the status of the Kubernetes cluster. The consistency of the distributed database is achieved based on Raft consensus algorithm.
API server	Master	The gateway to handle all interactions with the Kubernetes cluster. For example, <code>kubectl</code> requests from the user are handled by API server.
Scheduler	Master	The optimizer that calculates the desired Pod distributions among all nodes based on the total load and the hardware limits. Notice that scheduler is not involved into load balance directly. A separate load balancer service can be created to actually direct the traffic.
Controller Manager	Master	A set of controllers or control functions to reconcile the actual Kubernetes cluster status with what has been calculated by the scheduler and stored in <code>etcd</code> .
Kubelet	Node	Local controller for the node. It ensures that all the Pods assigned to the node are running, and it keeps listening to the master for further instructions.
Kube Proxy	Node	The internet service provider in the node.

collection of containers for a single task, and it is also the smallest unit that Kubernetes controls directly. Notice that in Kubernetes, containers never run directly in a node. They are always grouped into Pods. More details about the Pod object are introduced in later Section 15.2.

Both master and nodes can run on multiple servers or VMs. For master, multiple master replicas can run in parallel to take care of user requests and provide database services. The “decision makers”, i.e. the controller manager and scheduler, run only on one of the master nodes which is elected via a leader election process. Should the decision maker master fail, one of the replicas will be promoted. For nodes, multiple nodes can run together to process information in parallel and Kubernetes manages the distribution of load among them.

It is worth mentioning that many services cluster-level services such as

ClusterIP, NodePort, LoadBalancer and Ingress Controllers, though being cluster-level, run in worker nodes but not in the master.

15.1.2 Installation

Kubernetes is not a single piece of software but more an architecture that involves many pieces of software such as:

- `kubectl` the CLI
- `kube-apiserver` the API server
- `etcd` the distributed key-value pair database
- `kube-scheduler` the scheduler
- `kube-controller-manager` the control manager
- Kubelet
- Kube Proxy
- Kubernetes-supported container runtime such as `containerd`
- Container Network Interface (CNI) plugin such as Calico
- ...

The installation guidance of Kubernetes can be found at its official website kubernetes.io. In a production environment, the administrator shall install and configure the above tools on the servers by himself. This gives him the flexibility of customizing the tools. However, the process can be tedious and requires experience.

In this notebook, for demonstration purpose, we will use Minikube to help build a Kubernetes cluster. Minikube is an open-source software developed by the Kubernetes community to start a VM and run a single-node Kubernetes cluster on a local machine. It automates most of the tools installations and configurations which is in favour of the scope of this notebook.

We still need to install Minikube and also `kubectl`. As introduced earlier, `kubectl` is used to interact with Kubernetes clusters deployed either locally or remotely. It is always recommended to have `kubectl` installed on the local machine wherever the Kubernetes cluster is deployed. The installation of `kubectl` can be found at Kubernetes website at

<https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/>

Minikube installation can be found at

<https://minikube.sigs.k8s.io/docs/start/>

Once Minikube is installed, start it using

```
$ minikube start
```

When starting Minikube, it automatically detects which container engine to use, and some configurations might be required subsequently.

When running Minikube, it uses existing VM engines such as VirtualBox, or container engines such as docker and podman installed on the local machine to start a VM or container that hosts Kubernetes. Therefore, it will need to execute the VM tools or the container engines. In naive RHEL, Minikube needs to execute podman. Some of the commands Minikube need to run require sudo privilege, and as a result, it requires NOPASSWD configuration to start the Kubernetes cluster correctly.

This can be done as follows. Open the sudoer configuration file by

```
$ sudo visudo
```

and append

```
<user name> ALL=(ALL) NOPASSWD: /usr/bin/podman
```

to the file.

Once Minikube is started, `kubectl` installed on the local machine should be able to detect the Kubernetes running inside. See the following command to verify that `kubectl` has detected and connected with the Kubernetes runtime.

```
$ 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
```

Minikube's Built-in `kubectl`

Minikube also comes with a built-in `kubectl`. It is possible to use that `kubectl` instead of installing one separately. To verify the existence of the built-in `kubectl`, start Minikube first and then use

```
$ 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
```

where `--` in the command tells the shell that all the followed arguments shall be passed to `kubectl` inside Minikube, but not `minikube` itself. For convenience, one may want to use alias as follows

```
$ alias kubectl="minikube kubectl --"
```

With the above been said, it is still recommended that `kubectl` to be installed on the local machine, not just in the VM.

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 VMs.

The configuration file for `kubectl` is usually located at `~/.kube/config`. It determines the basic setups of the `kubectl` instance, such as which Kubernetes cluster `kubectl` communicates with.

15.1.3 Kubernetes Cluster Manipulation

There are at least two ways for a user to interact with a Kubernetes cluster and deploy or manipulate services in it.

- Imperative approach: use `kubectl` commands to directly start, change or stop Pods and services.
- Declarative approach: prepare Kubernetes configuration files that describes the “desired state” of the Kubernetes cluster, and use `kubectl -f` command to implement the files. Kubernetes decides how to deploy Pods on servers to achieve the desired state.

Both approaches are introduced as follows.

Cluster Deployment with Imperative Approach

The imperative approach, in some sense, destroys the purpose of Kubernetes as a container orchestration tool since the user decides the deployment of Pods and services manually. On the other hand, the declarative approach takes advantage of the full capability of Kubernetes and it is usually considered a better practice. Yet, learning the imperative approach helps the user with getting familiar with `kubectl` commands. In this section, imperative approach and its relevant generally used `kubectl` commands are briefly introduced.

Use the following to view the entire list of `kubectl` supported commands.

```
$ kubectl help
kubectl controls the Kubernetes cluster manager.

Find more information at: https://kubernetes.io/docs/reference/kubectl
/

Basic Commands (Beginner):
  create      Create a resource from a file or from stdin
  expose      Take a replication controller, service, deployment or
              pod and expose it as a new Kubernetes service
  run         Run a particular image on the cluster
  set         Set specific features on objects

Basic Commands (Intermediate):
  explain     Get documentation for a resource
  get         Display one or many resources
```

```

edit           Edit a resource on the server
delete         Delete resources by file names, stdin, resources and
               names, or by resources and label selector

Deploy Commands:
rollout        Manage the rollout of a resource
scale          Set a new size for a deployment, replica set, or
               replication controller
autoscale      Auto-scale a deployment, replica set, stateful set, or
               replication controller

Cluster Management Commands:
certificate    Modify certificate resources
cluster-info   Display cluster information
top            Display resource (CPU/memory) usage
cordon         Mark node as unschedulable
uncordon       Mark node as schedulable
drain          Drain node in preparation for maintenance
taint          Update the taints on one or more nodes

Troubleshooting and Debugging Commands:
describe        Show details of a specific resource or group of
               resources
logs            Print the logs for a container in a pod
attach          Attach to a running container
exec            Execute a command in a container
port-forward   Forward one or more local ports to a pod
proxy          Run a proxy to the Kubernetes API server
cp              Copy files and directories to and from containers
auth            Inspect authorization
debug           Create debugging sessions for troubleshooting workloads
               and nodes
events          List events

Advanced Commands:
diff            Diff the live version against a would-be applied
               version
apply           Apply a configuration to a resource by file name or
               stdin
patch           Update fields of a resource
replace         Replace a resource by file name or stdin
wait            Experimental: Wait for a specific condition on one or
               many resources
kustomize       Build a kustomization target from a directory or URL

Settings Commands:
label           Update the labels on a resource
annotate        Update the annotations on a resource

```

```
completion    Output shell completion code for the specified shell (
               bash, zsh, fish, or powershell)

Subcommands provided by plugins:

Other Commands:
  api-resources Print the supported API resources on the server
  api-versions   Print the supported API versions on the server, in the
                 form of "group/version"
  config        Modify kubeconfig files
  plugin        Provides utilities for interacting with plugins
  version       Print the client and server version information

Usage:
  kubectl [flags] [options]
```

Use

```
$ kubectl create <object type> <object name> --<argument>
```

to create and deploy an object, and use

```
$ kubectl get <object type>
```

to view deployed objects of certain type. Notice that in `kubectl get` is usually followed by the plural form of the object type. An example is given below.

```
$ kubectl create deployment <deployment name> --image=<image name>
$ kubectl get deployments
```

Deployment object is deployed in the above example. More about Deployment object will be given in later Section 15.2. It is one of the most commonly used Kubernetes objects and serves well as an example to illustrate imperative approach. Notice that the image assigned to the deployment, in the above example `hello-world`, needs to be reachable by Kubernetes. In this example, the image is to be deployed by the Kubernetes cluster deployed in the Minikube environment, and hence any image locally built in the host machine will not be reachable. The image has to be stored on an online registry.

The deployed Pods will be default have a cluster-level internal IP. There are the following problems with this internal IP: it is dynamic, and it is not accessible from outside the cluster. To expose the IP and port, a separate Service object needs to be used as follows.

```
$ kubectl expose deployment <deployment name> --type=<service type>
```

where Service object type can be `ClusterIP`, `NodePort`, `LoadBalancer`, etc. More about Service object will be given in later Section 15.2.

To scale up and down the number of Pods in a Deployment object, use

```
$ kubectl scale --replicas=<number> deployment <deployment name>
```

And finally to update the image of the deployment, use

```
$ kubectl set image deployment/<deployment name> <container name>=<
    image name: tag>
```

For example,

```
$ kubectl set image deployment/nginx-deployment my-nginx-container=
    nginx:1.25.1
```

The user can check the status of the roll out by

```
$ kubectl rollout status deployment/<deployment name>
```

When Kubernetes is rolling out a new version, the old version will not be shutdown. This minimizes the blackout time when an application is being updated.

The user can check the history of the roll out using

```
$ kubectl rollout history deployment/<deployment name> [--revision=<
    revision index>]
```

where [--revision] will provides details to the roll out.

To roll back to an earlier version in the history, use

```
$ kubectl rollout undo deployment/<deployment name> [--to-revision=<
    revision index>]
```

where if --to-revision is not used, the command rolls back to the earlier latest version.

Cluster Deployment with Declarative Approach

Kubernetes configuration files describe the desired final status of the Kubernetes cluster. Should these files be passed to Kubernetes, it should be able to determine the processes to deploy relevant Kubernetes objects and eventually realize and maintain the desired state. This is known as the declarative approach and it is the default and recommended way Kubernetes shall be used. The declarative approach has at least the following advantages over the imperative approach:

- It automates the Kubernetes deploying by using configuration files instead of the user running `kubectl` commands manually, which simplifies the process and reduces human error.
- It leverages on the container orchestration capability of Kubernetes.

These advantages are critical especially in large-scale projects when there are multiple Kubernetes objects in the cluster.

The details of Kubernetes objects are introduced in later Section 15.2. In this section, a simple example is used to illustrate the basic commands for the declarative approach. Below is an example from the Kubernetes website. It contains the configuration of two objects.

```
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:1.14.2
          ports:
            - containerPort: 80
```

Some commonly used fields used in the Kubernetes configuration files are briefly summarized below.

- **apiVersion:** The API version of the object. Different object types are supported in different API versions.
- **kind:** The object type.
- **metadata:** The name and labels of the object.
- **spec:** The specifications of the object. Different object types require different specifications.

More details of the configuration file composing of each type of Kubernetes object are introduced in later Section 15.2.

With the image and the configuration files ready, the next step is to deploy the nodes, Pods, and containers. The `kubectl` CLI is used to instruct Kubernetes to deploy the objects as follows.

```
$ kubectl apply -f <configuration file> / <files directory>
```

This essentially asks the master node in the Kubernetes cluster to start taking actions following the Kubernetes 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 node to restart the container.

It is possible to consolidate the configuration files of objects into one 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>
---
<...>
```

The order of the configurations appears in the file usually does not matter. However, it is often considered a better practice to put cluster-level objects such as Service objects in front of work split objects such as Development objects.

With the help of Kubernetes declarative approach, it is possible to 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 revised configuration files, comparing them with existing running objects, and update them if necessary.

It is recommended to check the status of the Pods using `kubectl get pods` once the cluster is updated. After the update, the Pods should be restarted, and hence the “RESTARTS” tag shall increase. To ensure that the update is successful, use `kubectl describe` to check the details of the relevant objects.

Notice that there is a limitation on what can be updated to an existing Kubernetes deployment. For an existing object, only certain fields in the Kubernetes configuration files are allowed to be revised. For example, for a Pod, the image can be changed while the ports cannot. Some objects types are more flexible than others when comes to object updating. The user shall choose wisely what object types to use considering what updates need to be made in future.

To revert what has been applied in a Kubernetes configuration file, use

```
$ kubectl delete -f <configuration file>
```

The user can also delete certain types of objects by labels by

```
$ kubectl delete <object type 1>[, <object type 2>, ...] -l <label>
```

Kubernetes Cluster Status Check

To retrieve 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.

Minikube provides a web-based dashboard where the user can conveniently check the status of the cluster. Use

```
$ minikube dashboard
```

to start the dashboard and follow the instructions on the console to open the dashboard. The dashboard gives the detailed information of running objects in the Kubernetes cluster, such as the healthy status of Pods, the cluster-level internal IP of objects, etc.

Kubernetes treats the above as an update and will action accordingly.

15.2 Basic Kubernetes Objects

Kubernetes supports a long list of objects, and the list is growing along with Kubernetes version. The list can be displayed by

```
$ kubectl api-resources
```

It is hardly possible to cover all the Kubernetes objects in this notebook. This section only introduces the commonly used basic Kubernetes objects summarized in Table 15.2.

15.2.1 Pod Object

Pod object is the smallest and most fundamental unit in a node that Kubernetes interacts with. A Pod is essentially a “wrap” that hosts one or multiple containers that work closely with each other. Pod contains shared resources such as volumes for all the containers it hosts. The containers in a Pod can also communicate each other via `localhost`. A Pod has a cluster-internal IP by default.

Just like a container, Pods are designed to be ephemeral. Kubernetes can start and stop them depending on the load and the traffic coming into the cluster, and when they are removed, their data vanishes. Of course there are many ways to persist the data, for example by directing the data into a volume or a database.

The user can deploy Pods using either `kubectl` command imperatively or in Kubernetes configuration files declaratively. However, the user would not do so in practice, as it destroys the purpose of using Kubernetes.

A Pod is assigned with a cluster-level internal IP by default, and that IP can be used for communications inside the Kubernetes cluster. It has the following problems though, making the IP address difficult to use:

TABLE 15.2

Commonly used Kubernetes object types.

Object Type	Description
Pod	Represents a single instance of a process running in the cluster. A Pod contains one or multiple containers that work closely to deliver a basic function. It is the smallest unit of process in Kubernetes.
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 the node or 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.
Volume	Provides a way to provision and manage persistent storage resources in a cluster.

- The IP address is internal the cluster, hence useless to entities outside the container.
- Even for communications inside the cluster, the IP address is not convenient to use since it is dynamic.

15.2.2 Deployment Object

It is more often that the user would want to deploy Deployment objects instead of Pods. A **Deployment** object serves as the “controller” of one or a group of identical Pods, and it serves as a good example to illustrate declarative approach where the user specifies a desired state and the Deployment object carries out the necessary procedures to realize that state. A Deployment object can do at least the following for the Pods it manages:

- Start and stop the Pods strategically at different word nodes.
- Monitor the healthy status of the Pods and strategically restart them when necessary.
- Scale up and down the number of Pods.
- Balance the loads that go into each Pod.

- Update or roll back the Pods.

As an example, a Kubernetes cluster file that deploys an Deployment object is given below. Notice that Deployment object is supported in API version `apps/v1`.

```
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 under `spec` are introduced below.

- `replicas`: the number of Pods under the Deployment object should maintain. The value can be zero if no Pods need to be deployed initially.
- `selector.matchLabels`: a set of labels, which if a Pod possesses, will be monitored and likely managed by the Deployment object.
- `template`: the template that describes the Pods this Deployment object creates, including:
 - `metadata.labels`: the labels the Pods will inherit.
 - `spec.containers`: the containers details, such as name, image, port, etc.

Notice that a Deployment object can deploy only identical Pods from the same template. If different types of Pods need to be deployed, different Deployment objects are required. However, under one Pod, there can be multiple containers populated from different images. Notice that `template.spec.containers` contains a list of containers, each with its name, image, ports, etc. Append that list if multiple containers are required under the same Pod.

An alternative to `selector.matchLabels` is `selector.matchExpressions`,

the later of which provides a more powerful and flexible way of mapping labels. More details of `selector.matchExpressions` is not introduced in this notebook, as `selector.matchLabels` should serve well in most scenarios.

Labels in selector and template

It may feel redundant that the same labels appear twice under both `selector.matchLabels` and `template.metadata.labels`. This sense of redundancy often stems from the assumption that a Deployment object should watch (and only watch) the Pods it creates.

In reality, multiple controllers can “monitor” the same Pods, potentially for metrics, auditing, or other read-only operations. A Pod has exactly one owner which is the controller that creates the Pod, but it can be monitored by other controllers or tools via label selector.

By splitting `selector.matchLabels` and `template.metadata.labels` instead of automatically inheriting one from the other, Kubernetes provides flexibility in how Pods and Deployment objects are architected. Additional labels can be added in the template to support other purposes—like monitoring, cost allocation or environment tagging without affecting the Deployment’s fundamental ownership of those 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. The following imperative command which has been introduced earlier can be used to update the image.

```
$ kubectl set image deployment/<deployment name> <container name>=<  
    image name: tag>
```

The image in the specified container will be updated. Notice that the new image will be downloaded only if it has a different tag. This encourages the developer to update the tag formally when publishing images.

Some optional yet useful configurations that Deployment object supports are briefly introduced as follows.

The `livenessProbe` under `spec.containers` of Deployment objects allows the user to define how the healthiness of the container shall be monitored. For example, the user can define a watch dog by asking the container to send an HTTP get request to certain port at certain frequency, and so on. Notice that Kubernetes has default liveness probe mechanism. Still, the flexibility where the user can override it with a custom probe is appreciated.

By default, the Deployment object will pull the images when

- The image is not saved in the locally, in which case Kubernetes has to pull the image.

- The `:latest` tag is used with the image, in which case Kubernetes will pull the image each time before launching the containers.

The `spec.containers.imagePullPolicy` of Deployment objects can be used to change the policy of pulling the image. For example, if `imagePullPolicy: Always` is used, the Deployment project will always pull the image before starting the containers.

15.2.3 Service Object

A **Service** object is used to build network services in a Kubernetes cluster. As introduced earlier, though objects such as Pods will be automatically assigned with cluster-level internal IPs, they are close to useless in practice. Service objects can be used to build a more useful and robust network infrastructure for a Kubernetes cluster.

There are 4 subtypes of Service objects, including

- ClusterIP: a ClusterIP Service object is assigned with a static cluster-level internal IP which is useful for communications of services within the cluster.
- NodePort: a NodePort Service object exposes a port on all the work nodes, using which outside entities can communicate with the NodePort Service object and then talk to objects inside the Kubernetes cluster. Notice that NodePort is often used in development environment and not production environment, in which case there are other preferable solutions such as LoadBalancer.
- LoadBalancer: a LoadBalancer Service object is essentially a NodePort Service object integrated with an externally provisioned load balancer. Notice that Kubernetes does not ship with this load balancer, but rather provides an interface that connects to the external load balancer often provided by the cloud service provider.
- ExternalName: an ExternalName Service object instructs Kubernetes to create a DNS alias within the cluster's DNS. This lets Pods in the cluster use a short, consistent service name to reach an external resource.

A Service object configuration file may look like the following.

```
apiVersion: v1
kind: Service
metadata:
  name: client-service
spec:
  selector:
    app: nginx
  ports:
```

```

- protocol: 'TCP'
  port: <port>
  targetPort: <port>
type: ClusterIP

```

Some highlights are introduced below.

- **spec.selector**: a list of labels with which the objects can connect to this Service object.
- **spec.ports.port**: port exposed by the ClusterIP Service object to the cluster.
- **spec.ports.targetPort**: port exposed by the containers to the ClusterIP Service object.

15.2.4 ConfigMap Object

ConfigMap object, as its name suggests, is used to create a configuration map.

Before diving into ConfigMap object, a closely related to concept, Kubernetes environment variable, is introduced as follows. **Kubernetes environment variable** refers a list of key-value pairs defined in the Kubernetes configuration file. The processes started by the Kubernetes configuration file can then access these environment variables.

For example, consider deploying a Deployment object as follows.

```

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>
volumes:
  - name: data-volume
persistentVolumeClaim:
  claimName: my-pvc
```

where under the `spec.template.spec.containers.env` tag, a list is defined containing key-value pairs of the environment variables. The applications can then access these variables. For example, if the source code of the application is in Python, the variables can be accessed by

```
import os
os.environ['MYCUSTOMVAR']
```

The syntax differs per programming language.

In the above example, the environment variables are integrated with the configuration file of the Deployment object, which can sometimes be inconvenient as it makes the file more complicated than it should have been. ConfigMap object tackles this issue by allowing the user to define a dedicated Kubernetes object that stores the configuration parameters.

A ConfigMap object can be defined as follows. The example is taken from [15].

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  <var1>: <value1>
  <var2>: <value2>
```

It can be seen from the example that a ConfigMap is nothing but a collection of key-value pairs. The name of the configuration file, in this example `game-demo`, can be included in the Pod or Deployment object configuration files as follows.

```
apiVersion: v1
kind: Pod
metadata:
  name: env-configmap
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      env:
        - name: CONFIGMAP_USERNAME
          valueFrom:
```

```
configMapKeyRef:
  name: myconfigmap
  key: <var1>
```

Just like in the case of integrating environment variables, `env` is defined. The difference is that instead of

```
env:
  - name: <name1>
    value: <value1>
```

which specifically lists the values,

```
env:
  - name: <name1>
    valueFrom:
      configMapKeyRef:
        name: myconfigmap
        key: <var1>
```

is used where the ConfigMap object is attached under `valueFrom`. By this setup, the value of `<var1>` which has been defined in the ConfigMap, in this example `<value1>`, will be assigned to `<name1>`.

As a special case of configuration information storage, Kubernetes uses **Secrets object** to save sensitive data. It provides different ways for the program to access confidential information without the user hard coding passwords into the configuration files. Details about Secrets object can be found at [15].

15.2.5 Ephemeral Volume

Volumes are handy tools to persist data so that it will not vanish when containers shut down. Container volumes have been introduced in Section 14.4, where they are useful for users deploying containers locally with a container engine like Docker or Podman.

For a similar purpose, Kubernetes also supports volumes. However, there are notable differences between how Kubernetes and local container engines implement volumes. For example, while a local container engine mounts volumes into containers, Kubernetes mounts volumes into Pods. More details are introduced in this section.

There are two types of volumes in Kubernetes. Examples of each type are also given.

- **Ephemeral Volume:** A volume defined inline within a Pod specification. It vanishes when the Pod is removed. The volume can be used to share information among multiple containers in the same Pod. These volumes do not exist as standalone Kubernetes objects, but part of the Pod's `spec.volumes`.

- `emptyDir`: An initially empty directory what can be used for containers to share information. It also persists data when the container

(not Pod) crashes, and help it to regain the lost information when the container is restarted.

- **Persistent Volume (PV):** A standalone Kubernetes object representing a piece of storage. It does not vanish even if the associated Pods are removed, depending on its reclaim policy. A PV can be mounted to Pods as persistent storage and enables data sharing across Pod restarts or among multiple Pods.

A Pod can issue a **Persistent Volume Claim (PVC)**, which is not a volume by itself but rather a request for storage. Kubernetes will then look for existing PVs meeting the PVC's requirements, or if dynamic provisioning is enabled, create a new PV that satisfies the claim.

Ephemeral volumes are introduced in this section, and PVs, in the next section.

Below is an example of deploying emptyDir, a commonly used ephemeral volume [15]. As introduced earlier, an ephemeral volume is claimed as part of the Pod specifications, and it follows the Pod's life cycle.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: registry.k8s.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir:
        sizeLimit: 500Mi
```

Ephemeral volumes can also be used with Deployment object. An example is given below.

```
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
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
  emptyDir: {}

```

Notice that the volume is defined under the Pods specifications the same way as in the earlier Pod object configuration example. The size limit is removed. In this example, 3 replicas Pods will be deployed and each of them will have its own emptyDir ephemeral volume.

These emptyDir ephemeral volumes, though under the same Deployment object, are not shared across Pods. Consequently, if user requests happen to land on different Pods (for example, because the container in the original Pod crashed and needed restarting, making that Pod temporarily unavailable), there is no way for the new Pod to retrieve information stored on the first Pod's local emptyDir. This means user-specific data left on the first Pod will remain inaccessible to the other Pods, especially if there is no mechanism for session persistence or "sticky sessions". This can be an issue with multiple replica Pods applications.

For the pods running on the same node to be able to share information, consider using hostPath instead. The hostPath is essentially a volume mounts from the host node's filesystem into the Pods. The data in hostPath persists even if the Pods are removed or replaced. A path on the node needs to be specified. An example is given below.

```

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
        volumeMounts:
          - mountPath: /cache
            name: cache-volume
    volumes:
      - name: cache-volume
        hostPath:
          path: /data # path in the host machine
          type: DirectoryOrCreate # policy
```

where `DirectoryOrCreate` means the path is a directory, and if it does not exist, create that directory.

It is a bit tricky whether to consider `hostPath` as an ephemeral volume or PV. The reasons are given below.

- `hostPath` is Ephemeral volume because:
 - It can be claimed as Pod or Deployment specs instead of a standalone PV.
 - It has many restrictions compared with other PVs. For instance, it is tied to a single node and cannot be used for cross-node data persistence or sharing. This destroys the purpose of Kubernetes to some extent.
 - When the node is restarted (this can happen in the Kubernetes context because nodes are often VMs), the data disappears.
 - For the above reasons, `hostPath` is rarely used in production environment.
- `hostPath` is PV because:
 - It is Pod-independent and can survive Pod failure.
 - It can also be claimed as standalone Kubernetes object. But of course, this does not clear its restrictions.

In this section, `hostPath` is introduced as an ephemeral volume. In the next Section 15.2.6, `hostPath` will be introduced again as a PV, and it will be claimed as a standalone Kubernetes object and consumed by PVC.

15.2.6 Persistent Volume

Unlike ephemeral volumes defined in the specs of Pods and Deployment objects, PVs are independently defined Kubernetes objects and are Pod-independent. Pods and Deployments can then use PVCs to claim the PVs.

Notice that PVCs can also be used to provision PVs if no existing PVs meet the requirements. Details are introduced in this section.

Kubernetes supports many PV types. For example, hostPath which has been introduced earlier in Section 15.2.5 can also be claimed as a standalone Kubernetes PV object as follows. The example is taken from [15].

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Notice that `storageClassName`

Different PVs support different access modes. A list of commonly used access modes are summarized in Table 15.3. In the case of hostPath, “ReadWriteOnce” can be used.

TABLE 15.3

Commonly used access modes for PVs.

Access Mode	Description
ReadWriteOnce	The volume can be mounted as read-write by multiple Pods on a single node.
ReadOnlyMany	The volume can be mounted as read-only by multiple Pods on multiple nodes.
ReadWriteMany	The volume can be mounted as read-write by multiple Pods on multiple nodes.

An example of a PVC is given below. The hostPath PV introduced in the earlier example meets the requirement of this PVC. Notice that it is possible to tie a PVC with a PV, in which case the PV name can be claimed in the PVC under `spec.volumeName`. However, this is not the case in the example below. Instead of specifying a particular PV, it specifies the required resources and let Kubernetes decide which PV to use.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
```

```
storageClassName: manual
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 3Gi
```

It is worth mentioning that multiple PVCs can land on the same PV, in which case the PVCs share the resources the PV has. For that, the PV should be “powerful enough” to support all the PVCs at the same time.

Finally, a Pod can add a PVC to `spec.volumes` as follows.

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

When implementing the configurations, the user shall apply PV, then PVC, then Pod or Deployment objects that use the PVC.

In addition to the Kubernetes naive PV types, many cloud providers provide storage solutions, for example, AWS’s Elastic Block Store (EBS), Elastic File System (EFS), etc. Kubernetes provides a universal interface known as the **Container Storage Interface** (CSI) which can be used to connect to these storage systems. Obviously, these storage systems are Pod- and node-independent and will persist regardless of the status of the Kubernetes cluster. They are considered third-party PVs.

Examples of configuring AWS’s EBS as Kubernetes PV can be found at [16], where CSI is used to back up the connectivity. One of the examples is pasted below. The prerequisite is to install `aws-ebs-csi-driver` and to have an AWS account with an EBS volume created.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-pv
spec:
```

```

accessModes:
- ReadWriteOnce
capacity:
  storage: 5Gi
csi:
  driver: ebs.csi.aws.com
  fsType: ext4
  volumeHandle: {EBS volume ID}
nodeAffinity:
  required:
    nodeSelectorTerms:
      - matchExpressions:
          - key: topology.kubernetes.io/zone
            operator: In
            values:
              - {availability zone}

```

15.3 Connectivity

This section explores common connectivity challenges in Kubernetes, including Pod-internal, Cluster-internal, and external communication. In the earlier Section 15.2.3, Service object has been introduced as part of the building blocks of Kubernetes connectivity. In this section, Service object is revisited with more details introduced. Other relevant Kubernetes objects such as Ingress object are also introduced.

Notice that in production environment, third-party technologies such as RabbitMQ (a messaging system not covered in this notebook) might be used to enhance the communication between components. This section, however, focuses on the native solutions of Kubernetes.

15.3.1 Pod-Internal Communication

Assume that two applications are running inside the same pod but in two different containers, and each of them exposes a different port. In their source code, they can reach to each other via `localhost` followed by the exposed port of the other container. For example, if the application wants to send an HTTP request to the other application in the other container, it can use

```
'http://localhost:<port>/<content>'
```

as the input argument to the command.

A better practice is to use environmental variables to pass the address to the source code. For example in JavaScript, consider

```
'http://${process.env.TARGET_ADDRESS}:80/<content>'
```

and in the Kubernetes configuration file that launches the Deployment, under `containers`, use

```
containers:
- name: <application>
  image: <application image>:latest
  env:
    - name: TARGET_ADDRESS
      value: localhost
```

15.3.2 Cluster-Internal Communication

Cluster-internal communication or Pod-to-Pod communication cannot be done in the same straight forward manner as Pod-internal communication. This is because Kubernetes by default assumes that the Pods in the cluster are distributed everywhere, not necessarily on the same node. As a result, applications running in another Pod cannot be reached with fixed IP or default domain such as `localhost`.

In the case of Cluster-internal communication, we need to use Service object, in particular, ClusterIP Service object to establish the communication. ClusterIP Service object has been introduced in Section 15.2.3. Here an example is used to demonstrate the association of a ClusterIP Service object to an application, as a recap.

In the Kubernetes configuration file of the application, assign a label to the Pod to be deployed. Later, ClusterIP Service object can refer to this label for the Pod. For example, in the Development object below `app: nginx` is assigned to the

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx # this is the label
  spec:
    containers:
      - name: nginx
        image: nginx
      ports:
```

```
- containerPort: 80
```

In the Kubernetes configuration file of the ClusterIP Service object, use the label in the `selector` as follows.

```
apiVersion: v1
kind: Service
metadata:
  name: client-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: 'TCP'
      port: <port>
      targetPort: <port>
  type: ClusterIP
```

With the above been done, all the request sent to the ClusterIP Service object will be automatically forwarded to the application. Any application in the Kubernetes cluster who wants to reach the application shall now reach the ClusterIP Service object instead.

When a Service object such as the aforementioned ClusterIP Service object is created, an environment variable is automatically created by Kubernetes which can be conveniently used to reach the that object. The name of the object is derived from the Service name. In this example, the ClusterIP Service name is

```
client-service
```

And hence the environment variable name is

```
CLIENT_SERVICE
```

which capitalizes all the characters and replace dash – with underscore `_`. The source code of the source application can then use this environment variable to reach the target application, for example with JavaScript,

```
'http://${process.env.CLIENT_SERVICE}:80/<content>'
```

Notice that the environment variable with Service object is generated automatically. Therefore, there is no need for the user to declare that variable under `env`. One thing to notice is that the environment variable is available only with Kubernetes. Hence when switching back to a local container engine, the changes made in the source code,

```
'http://${process.env.CLIENT_SERVICE}:80/<content>'
```

will raise an error that says `CLIENT_SERVICE` is undefined.

An alternative and more convenient approach is to use a cluster-internal Domain Name System (DNS) server. In the late versions of Kubernetes, CoreDNS [17], a fast and flexible DNS server, is automatically shipped and ready-to-use.

To use the DNS inside the cluster, simply adopt back to the environment variable approach we used in Pod-internal communication

```
'http://${process.env.TARGET_ADDRESS}:80/<content>'
```

and

```
containers:
- name: <application>
  image: <application image>:latest
  env:
    - name: TARGET_ADDRESS
      value: "<service name>.<namespace>"
```

but instead of using `localhost` as the case of Pod-internal communication, use "`<service name>.<namespace>`", where `<namespace>` is by default `default`. So in this example,

```
containers:
- name: <application>
  image: <application image>:latest
  env:
    - name: TARGET_ADDRESS
      value: "client-service.default"
```

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. More about namespace can be found at [15].

15.3.3 External Communication

For the communication between Pods and outside world, it is recommended to use the LoadBalancer Service object as follows. The use of LoadBalancer Service object is similar with that of ClusterIP Service object, with `spec.type` changed to LoadBalancer.

```
apiVersion: v1
kind: Service
metadata:
  name: client-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: 'TCP'
      port: <port>
      targetPort: <port>
  type: LoadBalancer
```

Notice that Kubernetes does not ship with a default load balancer and the

LoadBalancer Service object merely provides an interface for third-party load balancer often issued by the cloud service provider. Despite not providing the load balancer implementation, this approach effectively bridges the boundary of the Kubernetes cluster for external access.

An external client will connect to the public IP or DNS name provided by the external load balancer on <port>, and Kubernetes forwards that traffic to <targetPort> on the Pod. The external IP is not determined by the Service configuration file but assigned by the environment that provisions the load balancer. For example, if the Kubernetes is deployed by Minikube, then

```
$ minikube service <service name>
```

where `service name` being the name of the LoadBalancer Service object should return its public IP address. When deploying the service on a cloud service provider, the IP can be made static and a global domain name can be assigned for that IP address.

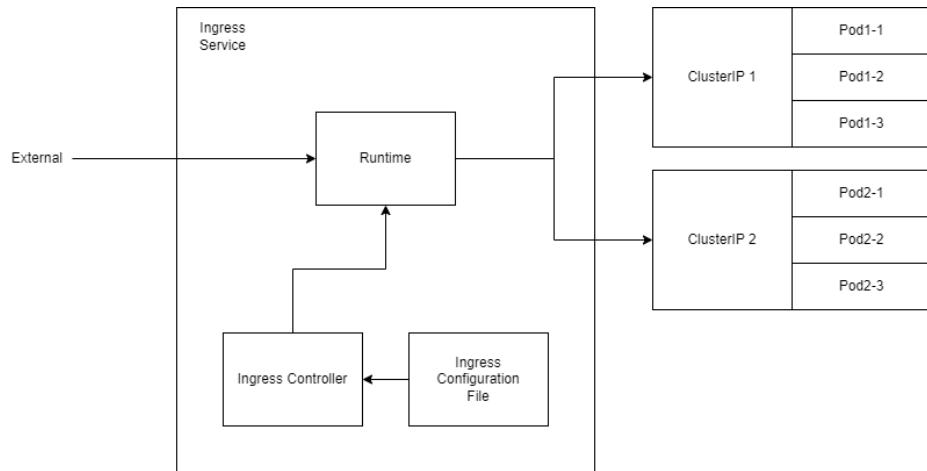
15.3.4 Ingress Object

Kubernetes Ingress object manages layer 7 (HTTP/HTTPS) routing. It is protocol-aware and allows the user to setup routing rules based on URIs, hostnames, paths, etc. It works with ingress controllers such as NGINX Ingress Controller (github.com/kubernetes/ingress-nginx) to route the load to the nodes.

A demonstrative example of ingress service realization is given in Fig. 15.2. In this implementation framework, the configuration file, mainly consists of routing rules, is used to define an the ingress controller which manages the runtime that controls inbound traffic. In some applications such as NGINX Ingress Controller, the ingress controller and the runtime are integrated together.

Below is an example of a Ingress object configuration file from [15].

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
```

**FIGURE 15.2**

An example of ingress service framework.

```
port:  
number: 80
```

15.4 Kubernetes Cloud Deployment

Many cloud service providers provide support for Kubernetes deployment. Recall from earlier that AWS supports ECS as the default managed container service. While being very powerful and flexible, the problem with ECS is that it is difficult to migrate across platforms because ECS is proprietary. Once ECS is used, the application is tied to AWS unless the user wishes to setup the configurations all over again on other platforms.

AWS EKS, on the other hand, allows the user to deploy a Kubernetes-based cluster using AWS resources. It allows more flexible migration of the cluster across different platforms. The Kubernetes cluster created and managed by EKS is known as a EKS cluster.

15.4.1 Setup and Connect to EKS Cluster

The following has to be configured when setting up EKS.

- Select Kubernetes version.

- Select cluster service role which tells AWS resources the EKS cluster can access, such as database, EC2, Container Registry, etc.
- Configure AWS Virtual Private Cloud (VPC), the AWS network service.
- Configure logging.

Make sure that AWS CLI is installed on the local machine. Login to the AWS account from the AWS CLI. Once the EKS cluster is setup and active, the user can modify the `kubectl` configuration file which can be found at `/.kube/config` so that it points to the EKS cluster. This can be done using AWS CLI as follows.

```
$ aws eks update-kubeconfig --region <region name> --name <eks cluster name>
```

With the above, `kubectl` will talk to the EKS cluster but not the local Kubernetes clusters such as the one in Minikube.

15.4.2 Configure Node Group

In the AWS dashboard under the EKS cluster just created, the user needs to configure nodes by adding node groups. This includes at least the following setups.

- Attach roles to the node group, for example, to allow nodes to use AWS Elastic Compute Cloud (EC2) and Container Registry.
- Select OS for the nodes.
- Select EC2 instance type and disk size which will be configured for the nodes.
- Select minimum and maximum number of nodes.
- Configure AWS VPC for the nodes.

It can be seen from the above that node configuration is essentially EC2 configuration. Once these EC2 instances are started, the user can view their status from the AWS dashboard under EC2. However, these EC2 instances shall be managed by EKS entirely and the user should not login to them. EKS should install Kubernetes components upon starting these EC2 instances.

15.4.3 Apply Kubernetes Configuration Files

The Kubernetes configuration files does not need to be uploaded to the cloud. The user can use `kubectl` to apply the configurations to the EKS cluster. Notice that `kubectl` should link to the EKS cluster but not a local Kubernetes cluster such as Minikube. The configuration of `kubectl` to link to the EKS cluster has been introduced earlier in Section 15.4.1.

At this stage, AWS container registry has not been enabled, so make sure that all the images used in the EKS cluster are from Docker Hub.

15.4.4 Use Volumes

Volumes such as emptyDir and hostPath have been introduced in earlier Sections 15.2.5 and 15.2.6. They are not useful in the context of EKS cluster. Volume type emptyDir is ephemeral and tied with a Pod, and it is not much useful with the EKS cluster in the production environment where Pods can get started and removed frequently. Volume hostPath can only apply to single node applications which is often not the case with EKS cluster.

With EKS cluster, it is more common to use AWS storage such as EBS, EFS to persist data. CSI is used to link EKS cluster with those storages. PVC is used in the Kubernetes configuration files to claim storage for the applications.

Details of attaching the third-party storage systems such as EFS to Kubernetes services such as EKS are not introduced in details here mainly because the procedures differ from platform to platform. The user should check the user manual for the specific service he wants to use. In general, the user needs to do the following:

- Create the storage system with the cloud service provider.
- Configure firewall settings and security groups so that the Kubernetes cluster can reach the storage system.
- Install necessary drives locally so that `kubectl` supports the storage system related functions.
- Create a PV object with a Kubernetes configuration file. Include the storage system specifications such as IP address and drivers in the configuration file, usually under `spec.csi`. More details are given in [15].
- Create a PVC object corresponding with the PV.

With the above been done, the EKS cluster objects can use the PVC to persist data.

15.5 Kubernetes IDE and Alternatives

While powerful, Kubernetes and `kubectl` CLI are not beginner friendly. The followings are possible to tackle this issue.

- Use Kubernetes IDE, which provides a more user-friendly Kubernetes control graphical interface. Examples include LENS [18].

- User alternative container orchestration tools which are often less flexible and complicated but more user-friendly.

Portainer is an open-source container orchestrator that has a user-friendly interface and is relatively easier to use than the more powerful and famous container orchestrator Kubernetes which is introduced in the next chapter. In this section, it is used as an example, just to give the reader an idea what container orchestrators in the market may look like. The section does not go into details about Portainer.

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 from the business edition) is `portainer/portainer-ce`.

Use `https://localhost:9443` to login to the container. The following page in Fig. 15.3 should pop up in the first-time login, asking the user to create an administration 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. 15.4, 15.5 and 15.6. Notice that in Fig. 15.6, 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`.

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.

▼ New Portainer installation

Please create the initial administrator user.

Username

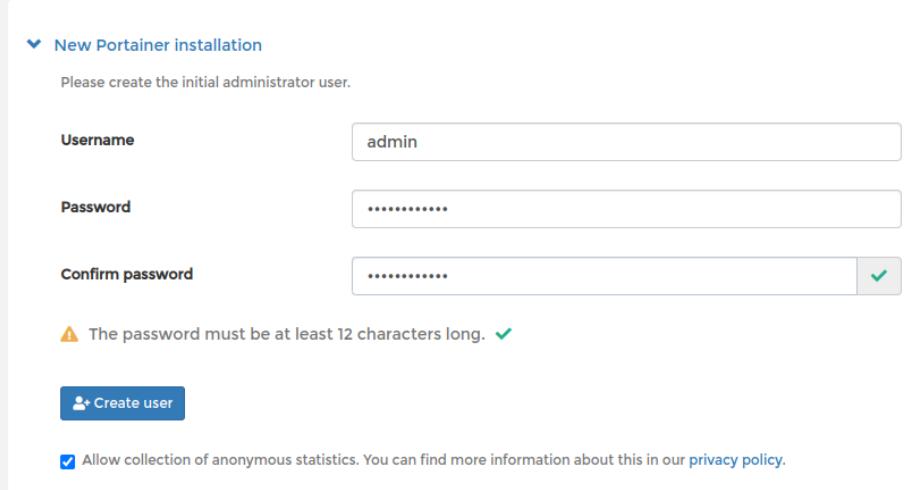
Password

Confirm password

⚠ The password must be at least 12 characters long.

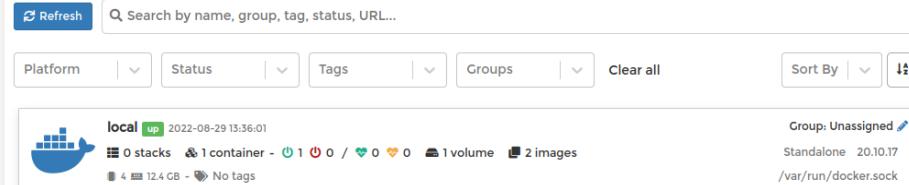
[Create user](#)

Allow collection of anonymous statistics. You can find more information about this in our [privacy policy](#).



This screenshot shows the 'New Portainer installation' setup page. It has fields for 'Username' (admin), 'Password' (a masked string), and 'Confirm password' (also a masked string with a checkmark). A warning message says 'The password must be at least 12 characters long.' Below the fields is a 'Create user' button and a checkbox for anonymous statistics collection, which is checked.

FIGURE 15.3
Portainer login page to create admin user.



This screenshot shows the Portainer dashboard. At the top, there's a search bar and filters for Platform, Status, Tags, Groups, and Sort By. Below that is a summary card for the 'local' host: it shows 1 container up, 0 stacks, 1 container - 1 healthy, 0 unhealthy, 1 volume, 2 images, 4 volumes (12.4 GB), and 0 tags. To the right, it shows the group 'Unassigned' (Standalone version 20.10.17) and the path /var/run/docker.sock.

FIGURE 15.4
Portainer dashboard overview of docker servers.

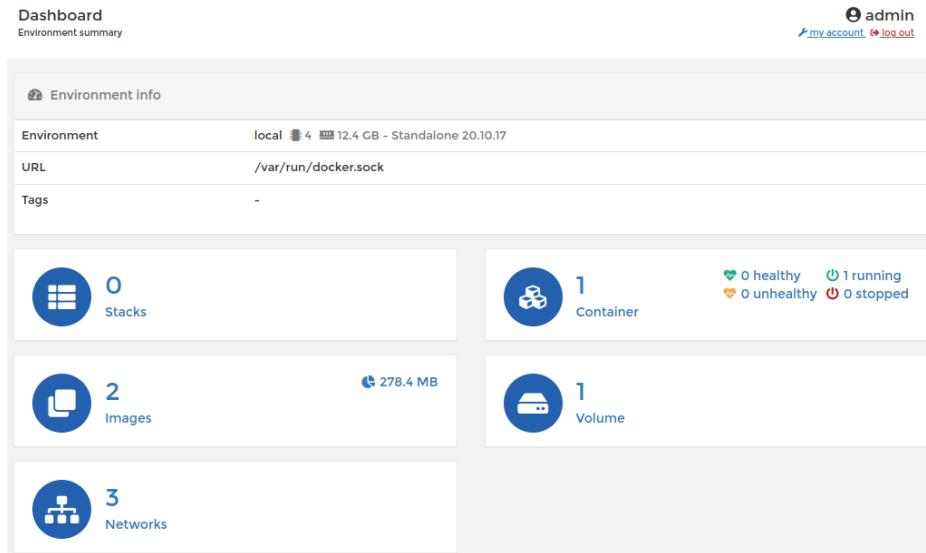


FIGURE 15.5
Portainer dashboard overview in a docker server.

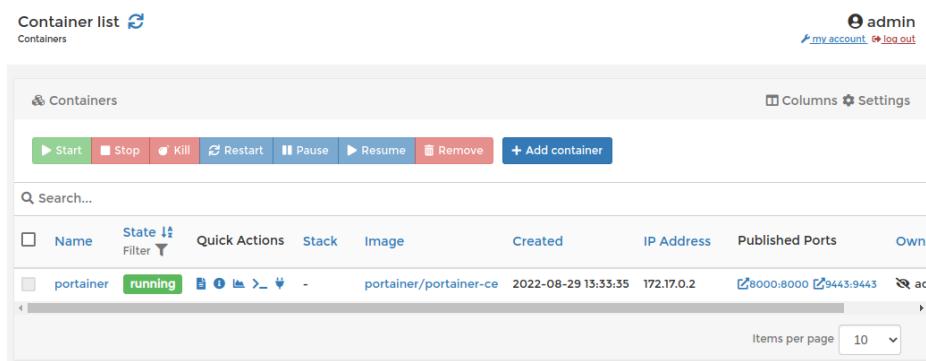


FIGURE 15.6
Portainer dashboard list down of all running containers.

16

HTTP Server

CONTENTS

16.1	Brief Introduction to Apache HTTP Server	261
16.2	Installation of Apache HTTP Server	262
16.2.1	Apache HTTP Server Installation on Host Machine	262
16.2.2	Apache HTTP Server Execution in Container	262
16.3	Apache HTTP Server Configuration and Deployment	263
16.3.1	Virtual Host Configuration	264
16.3.2	Kerberos Authentication Configuration	265
16.4	A Brief Introduction to Website Development	265
16.5	Other Web Servers	265

A commonly seen Linux application is to host a web service. “LAMP”, an acronym that stands for Linux, Apache, MySQL, and PHP, is a classic architecture for that application. In this structure, Linux serves as the host OS for the server or for the container orchestration, Apache the web service provider, MySQL (or its alternatives) the backend database, and finally PHP the programming language for web development.

Database services, both RDB and NoSQL, have been introduced in earlier chapters. Apache HTTP server is introduced in this chapter. Programming languages for web application such as PHP, Java, .NET framework, c#, JavaScript and CSS are not covered in this notebook.

Notice that Apache is not the only web service provider in the market. Alternative choices include Nginx, Node.js and many more. They are briefly introduced in the end of the chapter.

16.1 Brief Introduction to Apache HTTP Server

Web server is a network service that serves contents to a client over the internet. The client can be a web browser or any software that supports the pre-agreed communication protocol, one of the most popular ones being the hypertext transport protocol (HTTP). Details about HTTP protocol is not introduced in this notebook.

Apache HTTP server, also known as *httpd* in RHEL, is one of the web service providers in the market. It is an open-source web server project developed by the Apache Software Foundation. A full list of projects managed by the Apache Software Foundation can be found at [19]. Some of the widely appreciated applications in the list include Apache Spark, Apache Iceberg, and many more.

This chapter introduces the basic configuration and usage of Apache HTTP server on RHEL. For more details or Apache HTTP server on other platforms, visit the Apache HTTP Server website at <https://httpd.apache.org/>.

16.2 Installation of Apache HTTP Server

Apache HTTP server can be installed and executed on the host machine or deployed in a container.

16.2.1 Apache HTTP Server Installation on Host Machine

The package for Apache HTTP server on RHEL repositories is *httpd*. To install Apache HTTP server on RHEL, use

```
$ sudo dnf install httpd
```

To start and enable *httpd*, use

```
$ sudo systemctl start httpd  
$ sudo systemctl enable httpd
```

A testing page is provided. Upon installation, retrieve the testing page by

```
$ curl localhost
```

and the test page HTML will show up in the console. Alternatively, use a web browser to visit the host server at port 80 to see the web page visually that looks like Fig. 16.1. Notice that firewall may need to be configured to allow HTTP TCP access on port 80, which can be done by

```
$ sudo firewall-cmd --permanent --add-port=80/tcp  
$ sudo firewall-cmd --reload
```

Firewall configurations will be introduced in more details in later part of the notebook under Linux security.

16.2.2 Apache HTTP Server Execution in Container

Apache HTTP server also has a Docker image, and it can run in a container. More information can be found at [20]. In short, one can build the image using the following Dockerfile

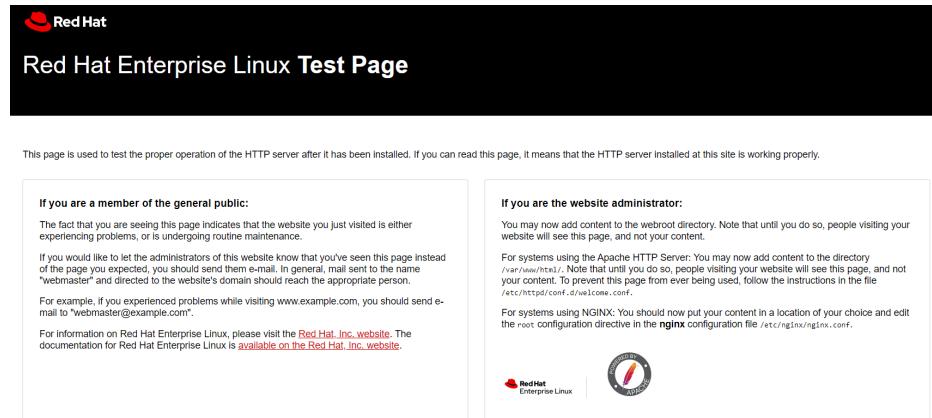


FIGURE 16.1
Apache HTTP server test page on RHEL.

```
FROM httpd:latest
COPY ./public-html/ /usr/local/apache2/htdocs/
```

followed by

```
$ docker build -t <image name> .
$ docker run -dit --name <container name> -p 8080:80 <image name>
```

where `./public-html/` is a directory on the host machine that contains all the HTML files. Alternatively, download and run the image from Docker Hub by

```
$ docker run -dit --name <container name> -p 8080:80 -v "$PWD":/usr/
local/apache2/htdocs/ httpd:latest
```

In the remaining of this chapter, we assume that Apache HTTP server is installed on the host machine. With that being said, all the configurations and functions should work similarly if it were run in a container.

16.3 Apache HTTP Server Configuration and Deployment

Configuration files setup the environmental variables (such as the location of the web pages, the port number, etc.) and control the behavior of the web server. Apache HTTP server configuration files include

- `/etc/httpd/conf/httpd.conf`: the main configuration file

- `/etc/httpd/conf.d/`: an auxiliary directory of configurations files included in the main configuration file
- `/etc/httpd/conf.modules.d/`: an auxiliary directory that contains configuration files for modules

Commonly used configurations are introduced in this section.

16.3.1 Virtual Host Configuration

Default Host VS Virtual Host

By default, an HTTP server deploys one website and all the website contents such as HTML files, JavaScript files and CSS files shall be stored in `/var/www/html/`. The idea of virtual host is to enable one HTTP server to deploy multiple websites, each stored in their associated subdirectory `/var/www/html/<virtual host name>/`. The virtual host information such as the port number, document root, etc., needs to be configured in `httpd.conf` the main configuration file.

Even if deploying only one website, it is still rather a good practice to use a virtual host than to use the default host for consistency and ease of management.

To deploy a virtual host, the main configuration file `httpd.conf` shall be appended with the following content.

```
<VirtualHost *:80>
    ServerAdmin <username>@example.com
    ServerName example.com
    DocumentRoot /var/www/html/example
    ErrorLog /var/log/httpd/example_error.log
    CustomLog /var/log/httpd/example_access.log combined
</VirtualHost>
```

where `example` and `example.com` can be replaced with the website name. Note that this is only a simple configuration with only the document root location, server name, and log locations. A practical virtual host configuration is usually more complex and filled with more details such as security policies, etc.

It is possible to add a configuration file for the particular virtual host at `/etc/httpd/conf.d/`. In this example

```
/etc/httpd/conf.d/example.conf
```

can be created and included as part of the main configuration file. This is optional and can become useful when the configuration is complicated.

With above setup, use

```
http://<server name>
```

to browse the virtual host, where

- The <server name> matches the `ServerName` or `ServerAlias` in the configuration file
- A DNS needs to be setup as a prerequisite, otherwise <server name> cannot be resolved.

16.3.2 Kerberos Authentication Configuration

Kerberos is a network authentication protocol. User registration and authentication have become a widely adopted feature in modern websites so that they can distinguish and provide different contents for different users. Kerberos can be used to back up this feature. Details of Kerberos is not given in this notebook.

RHEL uses GSS-Proxy module to provide support for Apache HTTP server running on it to perform Kerberos authentication. For that, make sure that `gssproxy` package is installed.

16.4 A Brief Introduction to Website Development

“nobreak

16.5 Other Web Servers



Part IV

Linux Security



17

Introduction to OS Security

CONTENTS

17.1	Basics	269
17.1.1	Risks and Attacks	269
17.1.2	General Security Architecture	270
17.1.3	Standards and Requirements	271
17.2	Elements of Security	272
17.2.1	Security Policy	273
17.2.2	Security Mechanism	273
17.2.3	Security Assurance	274
17.2.4	Trusted Computing Base	275
17.3	Access Control	277
17.3.1	Discretionary Access Control	278
17.3.2	Mandatory Access Control	279

Linux, as well as other OSs, uses a 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.

17.1 Basics

The background, motivation, and basic concepts of computer security are introduced in this section.

17.1.1 Risks and Attacks

No computer or OS is 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 initiates an attack deliberately taking advantage of the risk, it may cause trouble.

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.

17.1.2 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. 17.1. 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.

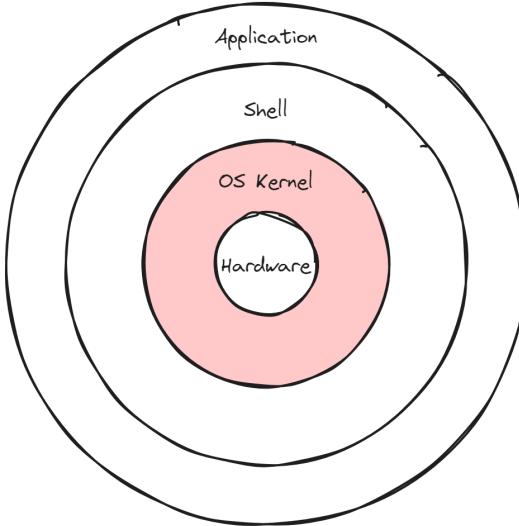


FIGURE 17.1
Layer structure of an operating system.

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 performs 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.

17.1.3 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.

17.2 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.

17.2.1 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.

17.2.2 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. 17.2.

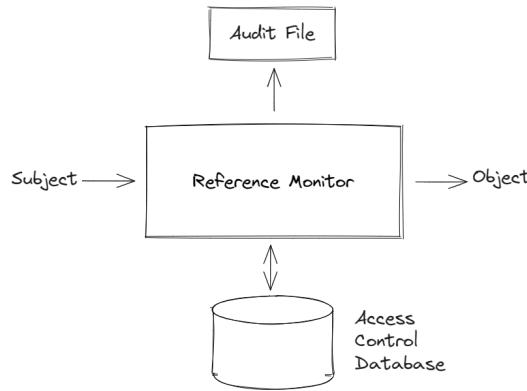


FIGURE 17.2
Reference Monitor Architecture.

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.)

17.2.3 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.

17.2.4 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. 17.3.

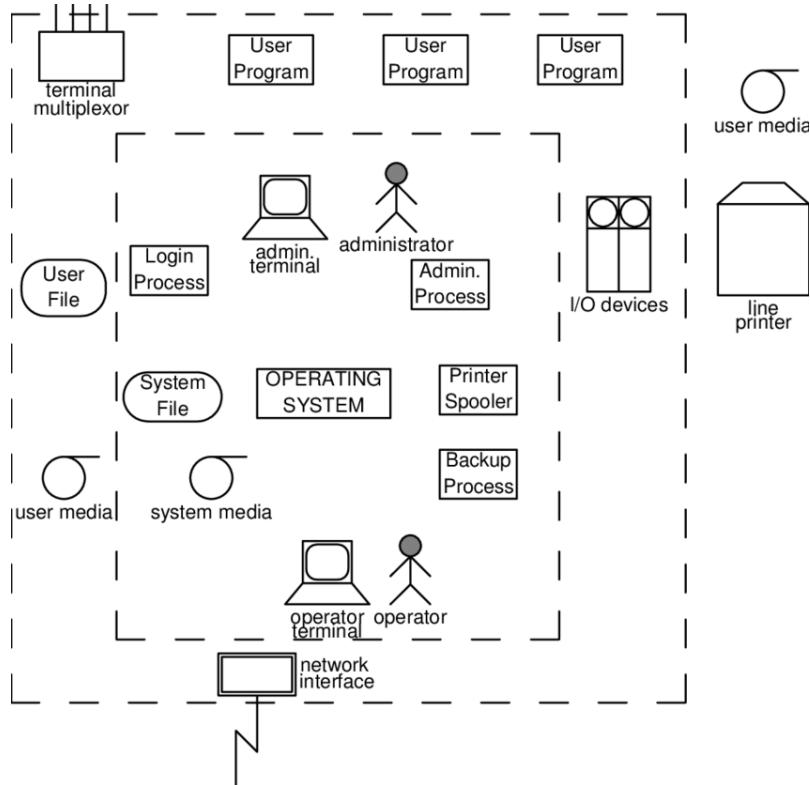


FIGURE 17.3
Reference Monitor Architecture [1].

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.

17.3 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. 17.4. In this demonstration, like introduced in Section 4.4, “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 17.4

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).

17.3.1 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 4.4 that records which users or subjects have access. This latter method of assigning ACLs to objects is more widely adopted.

As mentioned in Section 4.4, 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.

17.3.2 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.



18

Security of Services and Applications

CONTENTS

18.1	Database Security	281
18.1.1	Database Security Risks Categories	282
18.1.2	Database Access Control	283
18.1.3	Security-Enhanced DBMS Solutions in a Glance	285
18.1.4	Outsourced Database Security	286
18.1.5	Big Data Security	287
18.2	Virtualization Security	287
18.2.1	Security Concerns	288
18.2.2	VMM Security	288

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.

18.1 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 18.1

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.

18.1.1 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 18.1. 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 da-

ta. 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 18.1. Details are given in later sections.

18.1.2 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 grant 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 sense. 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.

18.1.3 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 (but not vice versa). This architecture design is robust to covert channel, but the data consistency and availability becomes a challenge.

18.1.4 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 watermark 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.

18.1.5 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.

18.2 Virtualization Security

Virtualization has been introduced in Chapter 14. While having many beneficial features, the use of virtualization brings new challenges to system security.

18.2.1 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 handing 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.

18.2.2 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

A.1	Vim User Profile	291
A.2	NeoVim User Profile in Lua	292

This appendix chapter collects the example scripts used in this notebook.

A.1 Vim User Profile

An example of a Vim user profile is given below. Such Vim user profile is often stored at `~/.vimrc`.

```
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 User Profile in Lua

Neovim is a 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.

An example of `init.lua` file is given below. Such file can be saved at `~/.config/nvim/`

```

-- 0) Leader (not used here, but good to set early)
vim.g.mapleader = " "
vim.g.maplocalleader = " "
vim.g.loaded_netrw = 1
vim.g.loaded_netrPlugin = 1

-- 1) Bootstrap lazy.nvim (plugin manager)
local lazypath = vim.fn.stdpath("data") .. "/lazy/lazy.nvim"
if not vim.loop.fs_stat(lazypath) then
vim.fn.system({
    "git", "clone", "--filter=blob:none",
    "https://github.com/folke/lazy.nvim.git",

```

```
--branch=stable", lazypath
})
end
vim.opt.rtp:prepend(lazypath)

require("lazy").setup({
    -- Plugins from your .vimrc
    { "vim-airline/vim-airline" },
    { "joshdick/onedark.vim" }, -- colorscheme onedark
    { "nvim-tree/nvim-tree.lua" },
    { "nvim-tree/nvim-web-devicons" },
    { "github/copilot.vim" },

    -- (Optional but harmless QoL: allows :Lazy command UI)
}, {
    ui = { border = "rounded" },
})

require("nvim-tree").setup({
    sort_by = "case_insensitive",
    view = { width = 32, signcolumn = "yes" },
    renderer = { group_empty = false },
    filters = { dotfiles = false },
    git = { enable = true, ignore = false },
    actions = { open_file = { quit_on_open = false } },
})

-- 2) Options (translated 1:1 where applicable)
vim.opt.number = true      -- set number
vim.opt.cursorline = true   -- set cursorline
vim.opt.wrap = true         -- set wrap
vim.opt.wildmenu = true     -- set wildmenu

vim.opt.hlsearch = true     -- set hlsearch
vim.cmd("nohlsearch")      -- exec "nohlsearch" (clear any highlight
                           at startup)
vim.opt.incsearch = true    -- set incsearch
vim.opt.ignorecase = true   -- set ignorecase

vim.opt.scrolloff = 3        -- set scrolloff=3

-- 3) Colorscheme (after plugins are loaded)
vim.cmd.colorscheme("onedark")

-- 4) Keymaps (noremap/inoremap equivalents)
local map = vim.keymap.set
local ns = { noremap = true, silent = true }

-- inoremap jj <Esc>
```

```
map("i", "jj", "<Esc>", ns)

-- noremap s <nop>
map({ "n", "x", "o" }, "s", "<Nop>", ns)

-- noremap S :w<CR>
map({ "n", "x", "o" }, "S", ":w<CR>", ns)

-- noremap Q :q<CR>
map({ "n", "x", "o" }, "Q", ":q<CR>", ns)

-- noremap <Space> :nohlsearch<CR>
map({ "n", "x", "o" }, "<Space>", ":nohlsearch<CR>", ns)

-- Search navigation w/ recenter (Nzz / nzz)
map({ "n", "x", "o" }, "-", "Nzz", ns) -- noremap - Nzz
map({ "n", "x", "o" }, "=", "nzz", ns) -- noremap = nzz

-- Split helpers (match your :set split{right,below} + split/vsplit)
map("n", "sh", ":set nosplitright<CR>:vsplit<CR>", ns)
map("n", "sl", ":set splitright<CR>:vsplit<CR>", ns)
map("n", "sk", ":set nosplitbelow<CR>:split<CR>", ns)
map("n", "sj", ":set splitbelow<CR>:split<CR>", ns)

-- Window navigation (Ctrl-h/j/k/l)
map("n", "<C-h>", "<C-w>h", ns)
map("n", "<C-l>", "<C-w>l", ns)
map("n", "<C-k>", "<C-w>k", ns)
map("n", "<C-j>", "<C-w>j", ns)

-- Resize windows
map("n", "H", ":vertical resize-2<CR>", ns)
map("n", "L", ":vertical resize+2<CR>", ns)
map("n", "K", ":resize +2<CR>", ns)
map("n", "J", ":resize -2<CR>", ns)

-- Toggle spell with sc
map("n", "sc", ":set spell!<CR>", ns)

-- Toggle tree with <leader>e
map("n", "<leader>e", ":NvimTreeToggle<CR>", { noremap = true, silent = true, desc = "Toggle file tree" })
```

B

A Brief Introduction to YAML

CONTENTS

B.1	Overview	295
B.2	Syntax	296
B.2.1	Key-Value Pair	296
B.2.2	Object and Nested Object	296
B.2.3	List	297
B.2.4	Multi-line String	298
B.3	Commonly Seen YAML Use Cases	299

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.

B.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, later on 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 to 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 the objects easily. Therefore, they are widely used in configuration files and for 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.

B.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.

B.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.

B.2.2 Object and Nested Object

Use indentation 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.

B.2.3 List

In addition to object and nested object, the value can also be list. 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 a list of 3 variables. 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 nested 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, each list `<list1>` to `<list4>` has 1 item. 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.

B.2.4 Multi-line String

To save multi-line strings as the value of a key, use `|` as follows.

```
<key>: |
  this is the first line
  this is the second line
  this is the third line
```

This allows file-like saving, i.e., saving the content of a file as it is in the YAML file. When using file-like saving with `|`, a line break is added automatically at the end of each line.

Do not confuse multi-line strings with a long single-line string that is wrapped in YAML for interpretation convenience. To express the wrap of a single string, use `>` instead of `|` as follows.

```
<key>: >
  abc
  def
```

which is equivalent with

```
<key>: abc def
```

Notice that when > is used, a space instead of a line breaker is added automatically at the end of each line, making the value a long single-line string instead of a multi-line string.

In the case where the space is not needed in the long single-line string, use \ at the end of each line as follows. In this way, YAML interpret it as a simple line wrap.

```
<key>: abc\
      def
```

which is equivalent to

```
<key>: abcdef
```

Notice that both | and > will generate a line breaker in the end of the entire text. If no such line breaker is designed, use |- and >- instead.

B.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.



C

Continuous Integration and Continuous Delivery

CONTENTS

C.1	Models	301
C.1.1	Waterfall	301
C.1.2	Agile	303
C.1.3	Roles in Agile-based Development	303
C.2	CI/CD Workflow	304
C.2.1	Pipeline	304
C.2.2	Continuous Integration	306
C.2.3	Continuous Delivery	306

This notebook is mainly about Linux. In this appendix chapter, the scope of the notebook is slightly extended to software development, which is very often what Linux is used for in practice. Continuous Integration and Continuous Delivery (CI/CD) has become a widely adopted software development framework and philosophy, and it is introduced in this chapter.

Some contents of this chapter come from [21].

C.1 Models

Agile and waterfall are both project management models. They are introduced here, starting with the more conventional waterfall model and then followed by the agile model.

C.1.1 Waterfall

Speaking of project proposal, development, testing and delivery pipeline, 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. C.1.

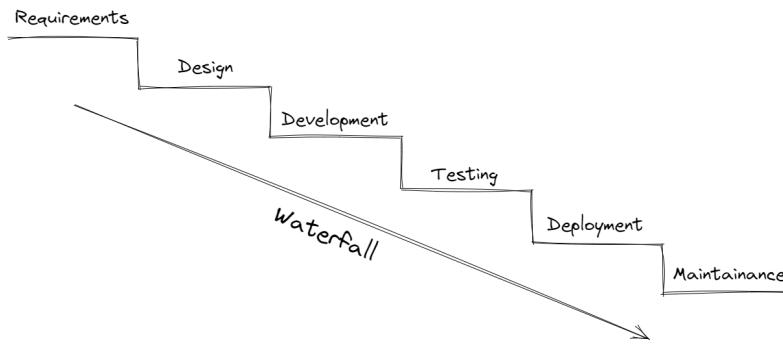


FIGURE C.1
Waterfall model.

With waterfall, a project can be designed, developed and deployed in a relatively efficient manner. However, there is one major 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 users and developers to determine all the requirements and designs in the beginning stage of the project. More likely, the requirements of the user have to change to adapt to the market trend, and so does the technologies used in the solution.

Assume that some changes have to be made to the system. If the project development is still in an early stage, it is possible to start over. The resources already been spent are wasted. If the project development is in a late stage, it might be preferable to deploy the system with its current capability, and later add the changes to the system as new features upgrades. However, the integration of the new features often introduces a blackout period of the system.

If the system is already in use, the customer experience would be affected by the blackout.

C.1.2 Agile

Agile is the counterpart of waterfall. It is proposed to tackle the aforementioned issues: rapid change of requirements and technologies during the development. It allows continuous integration and delivery of new features into the system in a convenient and consistent manner without introducing blackout.

In agile architecture, the development is modularized by features. Each feature, before deploying and integrating into the 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 in Fig. C.2.

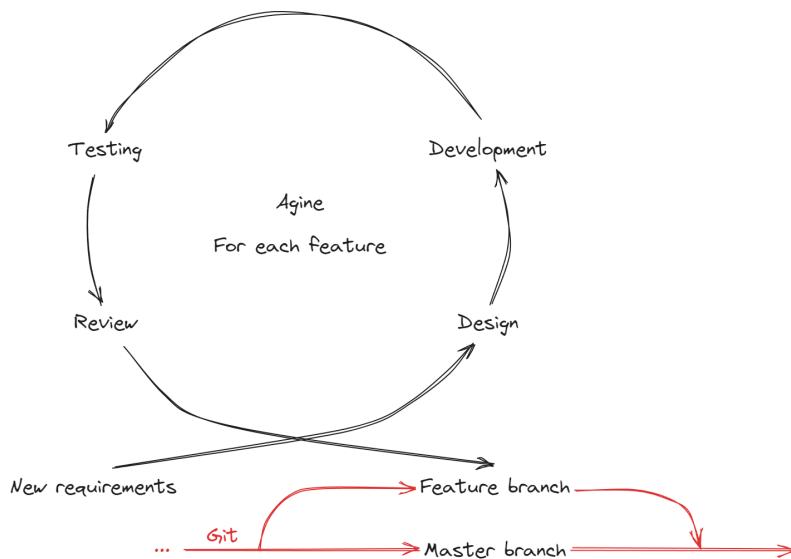


FIGURE C.2

Agile model.

Agile allows rapid changes to be made to the requirements and realizations of a project. Should there be any change made to the system, just keep cycling in Fig. C.2 until everyone is happy before it is pushed 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 respond to the customers immediate requirements.

As will be introduced later, agile is widely used in CI/CD.

C.1.3 Roles in Agile-based Development

Many roles are defined in the agile framework. The list of roles may slightly differ for different projects. The most commonly seen roles known as **scrum** is introduced in Table. C.1.

TABLE C.1
Roles in Agile model.

Role	Description
Product owner	Manage the entire program. They understand all the user requirements and tracks the progresses of all the features under development. They sign off features before they are deployed.
Scrum master	Lead the developer team as team manager or chief developer. They know the challenges and manpower required by each feature. They set priorities and assigns tasks to the developers.
Developer	Based on requirements, program the features.
Tester	Design test cases to verify the developed feature.
Operator/Supporter	Maintain the software after it is delivered.

In this role assignment, the product owner and scrum master come up with the product backlogs, which clarify the tasks and their priorities. A task is also known as the **sprint** in this context. The team then knows which sprints they shall work on first.

In the case where the features concern a professional domain (such as economics, medical, etc.) that the developers do not understand, business analysts are involved. A **business analyst** knows both the user use cases and requirements as well as the developers' skill sets and technologies, and they bridge the user with the developers.

For each sprint, **sprint planning** and **sprint backlog** are proposed that describe the schedule of the sprint. The team works on the sprint and host daily scrum meetings until the sprint is completed. Upon finishing of a sprint, sprint review is hosted for auditon.

C.2 CI/CD Workflow

Continuous Integration and Continuous Delivery (CI/CD) is both a philosophical concept and a bunch of technologies that speeds up the development, testing and deployment cycle of a software. It has become a common and beneficial practice for collaborative projects with rapid updates.

As introduced in Section C.1.2, Agile is widely used in CI/CD.

C.2.1 Pipeline

A **CI/CD pipeline** refers to the procedure a task must go through before it is delivered to the production environment. It often involves the following steps:

1. Integration

- (a) Develop the new feature.
- (b) Integrate the new feature into the entire system and compile the system.

2. Delivery

- (a) Test the compiled new system in pre-production environment.
- (b) Release the new system.
- (c) Deploy the new system in production environment.

In the development of a sprint, new codes are rapidly developed, and they are rapidly built, merged and tested. Conventionally, the integration of a new feature requires involvement from multiple parties. An example is given in Fig. C.3. 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.

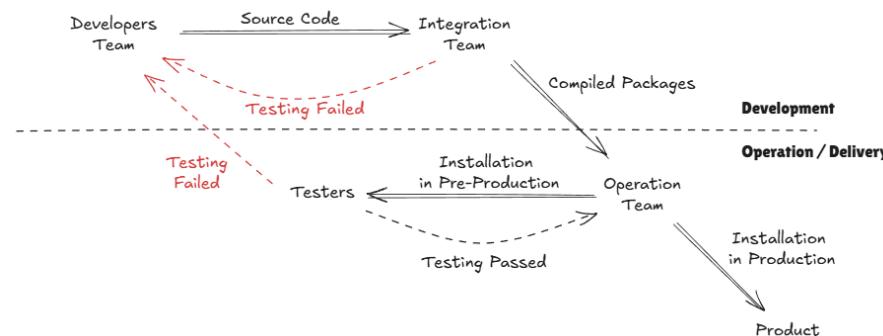


FIGURE C.3

Integration and delivery of a new feature. The integration corresponds with the development and delivery, operation respectively.

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.

In practice, each cycle in Fig. C.3 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.

C.2.2 Continuous Integration

Continuous Integration (CI) (together with Continuous Delivery (CD) which is introduced in the later section) tries to solve the above problems. CI automates the “development” portion in Fig. C.3, 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 combines the roles of developer team and integration team. The integration requirements are prepared by the developers and executed by automation tools such as Git and build server. During CI, the developer not only generates the codes, but also supervises Git and build server to integrate, compile and test the code. Should there be any error, the developer is notified immediately by the automation tools.

C.2.3 Continuous Delivery

The package received from the development side contains the latest version of the system where a new feature is integrated. Basic tests have been carried out to ensure that the system can run and the features seem to be working. It is, however, not certain at this point whether the entire system works properly in a robust manner. 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. 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. C.4. In this CI/CD implementation, the developer is playing a more comprehensive role than what is shown in Fig. C.3.

Under the framework of CI/CD in Fig. C.4, 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. C.4 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.

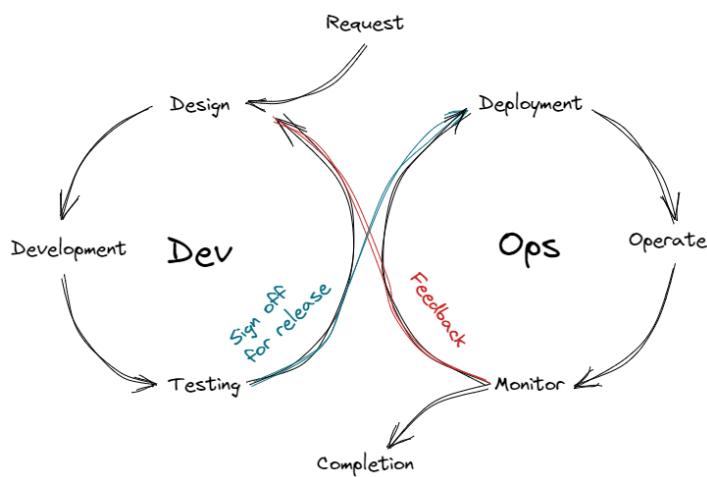


FIGURE C.4
CI and CD.

D

Commonly Used Tools and APIs for Application Development

CONTENTS

D.1	Web Query	309
D.1.1	Google Custom Search	310
D.1.2	Serper Query	310
D.1.3	Playwright	311
D.2	Database Query	311
D.3	Code Execution	311
D.4	Push Notification	311
D.5	Email	311
D.6	User Interface	311

Linux is widely used for application development. In this chapter, the scope of the notebook is extended to cover commonly used tools and APIs that support a wide range of applications. Note that the tools and APIs introduced here may also be used in applications hosted on Windows and macOS. Many APIs are chargeable based on usage.

Unless otherwise specified, examples are given in Python.

D.1 Web Query

Web query tools and APIs allow applications to retrieve up-to-date information from the Internet. Queries generally fall into two broad categories:

- Keyword-based query. The tool or API accepts keywords from the application, performs a web search (similar to Google search), and returns the most relevant results.
- Browser-based query. The tool or API accepts a URL, loads the page in a browser environment, and returns the page contents to the application.

Examples from each category are introduced in the remainder of this section.

D.1.1 Google Custom Search

With **Google Custom Search JSON API**, an application is able to retrieve searching result from Google Programmable Search Engines programmatically. The API can be used in conventional applications or agentic AI systems.

Note that Google Custom Search is part of Google Cloud Platform. To use the API, the user needs to register an account with Google Cloud Platform, and setup a billing account and a project. The user needs to create an Google Custom Search API key associated with the project. The API key is used for the application to connect to Google.

Cost may apply and it varies based on the number of queries carried out. As of this writing, each Google Custom Search JSON API has 100 queries free-of-charge per day, and USD \$5 per 1000 queries afterwards.

More details about Google Custom Search JSON API are given in [22]. A brief introduction is given below.

D.1.2 Serper Query

Serper is claimed to be the cheapest and fastest Google search API, and it is very easy to use. Notice that Serper is not a client of Google and they are not official collaborators. It is up to the user whether and how to user Serper API.

To use Serper API, register an account with Serper and purchase some credits. As of this writing, free credits come with the first registration, and it should be enough for quick testing. Generate an API key with Serper.

Many Python libraries provides tools to link to Serper. An example is given below.

```
import json
import requests

SERPER_API_KEY = "<KEY>"

url = "https://google.serper.dev/search"
payload = json.dumps({
    "q": "<query>",
    "num": <number of returns>
})
headers = {
    "X-API-KEY": SERPER_API_KEY,
    "Content-Type": 'application/json'
}
response = requests.request("POST", url, headers=headers, data=payload)
if response.status_code == 200:
```

```
print(response.json())
```

D.1.3 Playwright

Playwright is primarily a web automation and testing framework developed by Microsoft. Its main purpose is to let developers write end-to-end tests that simulate how a real user would interact with a web application. However, since Playwright controls a real browser, it can also be used for programmatic browsing tasks.

D.2 Database Query

“nobreak

D.3 Code Execution

An application may want to execute a small piece of user script in a dedicated environment with an interpreter.

D.4 Push Notification

“nobreak

D.5 Email

“nobreak

D.6 User Interface



E

Cloud Computing

CONTENTS

E.1	Introduction to Cloud Services	313
-----	--------------------------------	-------	-----

Amazon Web Service, Microsoft Azure, Google Cloud Platform, etc., are examples of services provided by famous cloud service providers. While the cloud services provided are not necessarily Linux, they are closely tied to applications development and delivery, and hence are introduced here.

In this appendix chapter, a general introduction to cloud services is given. AWS services are then introduced as examples. Notice that similar services are very likely provided by other cloud service providers as well.

E.1 Introduction to Cloud Services

Cloud services refer to a collection of managed cloud-based platforms and services including computing, networking, data storing, and many more. The most important advantage of cloud-based solutions, comparing with the traditional data-center-based solutions, is that it can be easily scaled up and down with little or no blackout time, allowing customers to pay only for operating expenses but not capital expenses and redundancies. Cloud solutions are famous for high availability, robustness, accessibility and efficiency.

When developing applications on the cloud, the developer and the cloud service provider share the responsibilities to provide and sustain the applications. There are at least the following different modes as summarized in Table E.1.

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Function as a Service (FaaS), also known as serverless.
- Software as a Service (SaaS)

TABLE E.1

Levels of cloud services and the corresponding responsibilities shared between the cloud service provider and the user.

Model	Hardware	App Runtime	APP Coding	APP Features	Examples
IaaS	C	D	D	D	Virtual machine
PaaS	C	C	D	D	Cloud-based website
FaaS	C	C	C	D	AWS Lambda function
SaaS	C	C	C	C	Microsoft 365

C: prepared by cloud service provider;

D: prepared by developer.

When we as individual users talk about cloud, we usually refer to the **public cloud services** provided by Amazon, Microsoft, Google, IBM, etc. Do notice that public cloud services do not make up the entire cloud market. Many enterprises also have **private cloud services** on premises and their employees can connect to the cloud from the internal network. There are also **hybrid cloud services** that integrate private and public clouds, allowing a user to use services provided by the public clouds while storing sensitive data in the private clouds.

Main public cloud providers include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and many more. While there are overlaps among the services they provide, each of them usually has some unique services and features as selling points, such as Amazon DynamoDB and Azure CosmosDB. The impression is that AWS focuses more on IaaS, hence more flexible, while Azure focuses more on PaaS and SaaS, hence more available and easier to use. The user should choose the cloud service provider based on the problem and there is not a globally best cloud service provider for all problems.

Bibliography

- [1] M. Gasser, *Building a secure computer system*. Citeseer, 1988.
- [2] Ubuntu, “Ubuntu.” <https://ubuntu.com/>, 2025. Accessed: 2025-06-26.
- [3] Fedora, “Fedora.” <https://fedoraproject.org/>, 2025. Accessed: 2025-06-26.
- [4] R. H. E. Linux, “Red hat enterprise linux.” <https://www.redhat.com/>, 2025. Accessed: 2025-06-26.
- [5] S. Kenlon, “10 ways to use the linux find command.” <https://www.redhat.com/sysadmin/linux-find-command>, 2022. Accessed: 2024-07-02.
- [6] R. S. Management, “Dnf, the next-generation replacement for yum.” https://dnf.readthedocs.io/en/latest/command_ref.html#dnf-command-reference, 2021. Accessed: 2024-07-12.
- [7] Debian, “Basics of the debian package management system.” <https://www.debian.org/doc/manuals/debian-faq/pkg-basics.en.html>, 2024. Accessed: 2024-07-12.
- [8] R. Hat, “Red hat enterprise linux 9: Managing, monitoring, and updating the kernel.” https://docs.redhat.com/en-us/documentation/red_hat_enterprise_linux/9/pdf/managing_monitoring_and_updating_the_kernel/Red_Hat_Enterprise_Linux-9-Managing_monitoring_and_updating_the_kernel-en-US.pdf, 2024. Accessed: 2024-07-08.
- [9] Git SCM, “Git.” <https://git-scm.com/>. Accessed: 2025-09-19.
- [10] Git, “Git references.” <https://git-scm.com/docs/>, 2025. Accessed: 2025-1-23.
- [11] Git, “Actions.” <https://github.com/marketplace?type=actions>, 2025. Accessed: 2025-02-24.
- [12] Mongo, “Start developing with mongodb.” <https://www.mongodb.com/docs/drivers/>, 2025. Accessed: 2025-06-12.
- [13] MongoDB, “Aggregation stages.” <https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/#aggregation-stages>, 2024. Accessed: 2024-06-26.

- [14] MongoDB, “Aggregation operators.” <https://www.mongodb.com/docs/manual/reference/operator/aggregation/#aggregation-operators>, 2024. Accessed: 2024-06-26.
- [15] Kubernetes, “Kubernetes volumes.” <https://kubernetes.io/docs/>, 2024. Accessed: 2024-12-27.
- [16] AWS, “Ebs csi examples.” <https://github.com/kubernetes-sigs/aws-ebs-csi-driver/tree/master/examples/kubernetes/>, 2024. Accessed: 2024-12-28.
- [17] CoreDNS, “Coredns: Dns and service discovery.” <https://coredns.io/>, 2025. Accessed: 2025-1-3.
- [18] LENS, “Lens: The way the world runs kubernetes.” <https://k8slens.dev/>, 2025. Accessed: 2025-1-15.
- [19] A. S. Foundation, “Projects directory.” <https://projects.apache.org/projects.html>, 2024. Accessed: 2024-07-17.
- [20] A. S. Foundation, “Docker image for httpd.” https://hub.docker.com/_/httpd, 2024. Accessed: 2024-07-17.
- [21] J. Honai, “CI/CD for beginners.” <https://nlbsg.udemy.com/course/ci-cd-devops/>, 2023. Accessed: 2023-10-10.
- [22] Google Developers, “Custom search json api overview.” <https://developers.google.com/custom-search/v1/overview>. Accessed: 2025-09-29.