Example 4-10 *Using the ? Repetition Character*

```
[NetProg@server1 grep-scripts]$ grep -E 11?23 data-file-2

123

1123

11123

111123

1111123
```

Example 4-11 presents another instance of the usage of the question mark that may seem more intuitive. The regex **2311?32** is used to match any occurrence of the literal 231, followed by *zero or one* occurrence of 1, followed by the literal 32. As you can see, only **23132** and **231132** match this pattern.

Example 4-11 *Using the ? Repetition Character - A More Intuitive Example*

```
[NetProg@server1 grep-scripts]$ grep -E 2311?32 data-file-2

23132

231132
```

Moving on to the plus symbol (**+**), in Example 4-12 **grep** is used to find matches to the pattern **2311+32** in the same data file used in Example 4-11. The regex translates to any sequence of characters that starts with the literal 231, followed by *one or more* occurrences of the literal 1, followed by the literal 32. This results in the matches shown in the example. Notice that **23132** does *not* match the pattern.

Example 4-12 *Using the + Repetition Character*

```
[NetProg@server1 grep-scripts]$ grep -E 2311+32 data-file-2

231132

2311132

23111132

231111132
```

As mentioned earlier, the repetition notation **{*N*}** is used to find any *N* number of occurrences of the character immediately preceding it. The notation **{*N*,*M*}** provides a range for the number of occurrences. Leaving out the upper range is equivalent to infinity. The question mark (**?**) is equivalent to **{0,1}**, the plus symbol (**+**) is equivalent to **{1,}**, and the asterisk symbol (*) is equivalent to **{0,}**. For example, the pattern **1[0-9]{1,2}** will match any number that starts with a 1 followed by either a single digit between 0 and 9, or two digits, each of them between 0 and 9. In other words, the matching criteria **[0-9]** can occur once, or twice. Therefore, this pattern will match any number between 10 and 19 in addition to any number between 100 and 199.

To print out the lines that do *not* contain a particular regex, you use the option **-v** with **grep**. In Example 4-13, all lines that do *not* contain the pattern **Jeff** are printed out. Notice that jeff with a small j does not match this criterion because **grep** is case sensitive. So the line containing the string jeff will *not* be excluded.

Example 4-13 *Using **grep -v** to Select Lines That Do Not Contain a Certain Pattern*

```
[NetProg@server1 grep-scripts]$ grep -E -v Jeff data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton
```

Use of regular expressions is not limited to the **grep** command. As you will see later in this chapter, regular expressions are a global concept that is used extensively in Linux scripting.

The following are a few more options you can use with the **grep** command to extend its functionality:

- **-c**: Returns the number of lines in which matches were found
- **-n**: Returns the lines with matches and prepends each line with its line number from the original input file
- **-A** N: Returns the line(s) with matches along with the *N* trailing lines from the original input file
- **-B** *N*: Returns the line(s) with matches along with the *N* leading lines from the original input file
- **-C** *N*: Return the line(s) with matches along with *N* leading and *N* trailing lines from the original input file

In all the previous examples, a filename was provided to the **grep** command for a file containing the text to search through. When a filename is not provided to **grep**, the command behaves in one of two ways. Either **grep** searches text provided through stdin (from the keyboard) or it searches through the files in the current working directory. The second case applies if either the **-r** or **-R** options are used, indicating recursive searches. Make sure to visit the man page for **grep** for a wealth of other options and use cases.

The awk Programming Language

awk is a full-fledged programming language. **awk** was created by (and named after) Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan, who also wrote a book titled *The AWK Programming Language* in 1988. **awk** is based on a simple paradigm: Find a pattern in the input and then perform an action.

Like any other programming language, **awk** can manipulate variables and has, among other features, flow control constructs. However, the coverage of **awk** in this section focuses on single-line operations that use the **awk** command and can be integrated into Bash scripts. The power of **awk** is in its capabilities to process the contents of spreadsheet-like text files formatted into rows and columns. Much like **grep**, **awk** makes it possible to search text for matching patterns. It also has the capability to take actions such as print out data in a specific column position in the matching text. Moreover, it can execute other operations that are commonly done by spreadsheet processing programs, such as arithmetic operations on values extracted from the text. These are just a few basic examples of the capabilities of **awk**.

Note

On some systems, when you try to access the man pages of the **awk** command, you are redirected to the man pages of **gawk**, which is the GNU version of **awk**. Similarly, if you use the **ls -l** command on the /usr/bin directory, you might see that the /usr/bin/awk file is actually a soft link to /usr/bin/gawk. Do not let this confuse you. It simply means that the version of **awk** on the system (in this case CentOS 8) is actually the GNU version (or implementation) named **gawk**, which is an enhanced version of **awk**. A newer version of awk called **nawk** (for *newawk*) is also available, but it is not available on CentOS 8 by default.

The basic syntax of the command-line version of awk is **awk '/***search-pattern***/ {***action***}'** *filename*. As in **grep**, an **awk** search pattern utilize regular expressions. The command can be used to only match on a pattern or to only take an action, or both. In Example 4-14, the output of **ls -l** for the directory /dev is saved in the file awk-data-ls to be used for testing the functionality of **awk**.

Example 4-14 *Data File to Test **awk***

```
[NetProg@server1 awk-scripts]$ ls -l /dev > awk-data-ls

[NetProg@server1 awk-scripts]$ cat awk-data-ls

total 0

crw-------. 1 root    root     10, 235 Sep 21 18:29 autofs

drwxr-xr-x. 2 root    root         320 Sep 21 18:29 block

drwxr-xr-x. 2 root    root         120 Sep 21 18:29 bsg

crw-------. 1 root    root     10, 234 Sep 21 18:29 btrfs-control

drwxr-xr-x. 3 root    root          60 Sep 21 18:29 bus

lrwxrwxrwx. 1 root    root           3 Sep 21 18:29 cdrom -> sr0

drwxr-xr-x. 2 root    root          80 Sep 21 18:29 centos

drwxr-xr-x. 2 root    root        3060 Sep 21 18:29 char

crw-------. 1 root    root      5,   1 Sep 21 18:30 console

lrwxrwxrwx. 1 root    root          11 Sep 21 18:29 core -> /proc/kcore

drwxr-xr-x. 6 root    root         140 Sep 21 18:29 cpu

crw-------. 1 root    root     10,  61 Sep 21 18:29 cpu_dma_latency

crw-------. 1 root    root     10,  62 Sep 21 18:29 crash


------ output truncated for brevity ------
```

In Example 4-15, **awk** is used to search for and print the line containing the file sdb2 using the regex pattern **sdb2**. Then awk is used to search for and print out a list of all files that are symlinks, identified by the letter l at the start of the line using the regex pattern **^l**.

Example 4-15 *Printing the Line Containing the Pattern **sdb2** and Then a List of Symlinks in the File awk-data-ls*

```
[NetProg@server1 awk-scripts]$ awk '/sdb2/' awk-data-ls

brw-rw----. 1 root    disk      8,  18 Sep 21 18:29 sdb2


[NetProg@server1 awk-scripts]$ awk '/^l/' awk-data-ls

lrwxrwxrwx. 1 root    root           3 Sep 21 18:29 cdrom -> sr0

lrwxrwxrwx. 1 root    root          11 Sep 21 18:29 core -> /proc/kcore

lrwxrwxrwx. 1 root    root          13 Sep 21 18:29 fd -> /proc/self/fd
```

```
lrwxrwxrwx. 1 root    root          25 Sep 21 18:29 initctl -> /run/systemd/initctl/fifo

lrwxrwxrwx. 1 root    root           4 Sep 21 18:29 rtc -> rtc0

lrwxrwxrwx. 1 root    root          15 Sep 21 18:29 stderr -> /proc/self/fd/2

lrwxrwxrwx. 1 root    root          15 Sep 21 18:29 stdin -> /proc/self/fd/0

lrwxrwxrwx. 1 root    root          15 Sep 21 18:29 stdout -> /proc/self/fd/1

[NetProg@server1 awk-scripts]$
```

Recall from the previous section that regular expressions use the caret symbol (**^**) to match the beginning of a line, so that the regex **^l** will match any line that starts with the letter l. Example 4-16 uses the **print** action to print out only the filenames, which is the data in the ninth column in Example 4-15.

Example 4-16 *Printing Only the Ninth Column After Matching on the Pattern* ***^l***

```
[NetProg@server1 awk-scripts]$ awk '/^l/ {print $9}' awk-data-ls

cdrom

core

fd

initctl

rtc

stderr

stdin

stdout

[NetProg@server1 awk-scripts]$
```

The column number is specified using the syntax **$***column_number* inside the action parentheses right after the action (**print** in this case), as highlighted in Example 4-16. The default *field separator* used by **awk** is the whitespace, so one column is distinguished from another by a whitespace. Therefore, in order to print out the filenames as well as the target of the symlink, you need to print out three columns: the filename, the right arrow and the target. These are columns 9, 10 and 11. The command **awk '/^l/ {print $9,$10,$11}' awk-data-ls** is used in Example 4-17 to print out the three required columns.

Example 4-17 *Printing Only the Ninth, Tenth, and Eleventh Columns*

```
[NetProg@server1 awk-scripts]$ awk '/^l/ {print $9,$10,$11}' awk-data-ls

cdrom -> sr0

core -> /proc/kcore

fd -> /proc/self/fd

initctl -> /run/systemd/initctl/fifo

rtc -> rtc0

stderr -> /proc/self/fd/2

stdin -> /proc/self/fd/0

stdout -> /proc/self/fd/1

[NetProg@server1 awk-scripts]$
```

What if the fields in a file are separated by something other than a whitespace? Consider, for example, the popular CSV format, in which the fields are separated by commas rather than whitespaces. In this case, the field separator can be changed by using the **-F** option, and the command is **awk -F , '***/pattern/* **{***action***}'** *filename*.

To process the text in a file based on column position, you need to make sure that the column numbers in a file are uniform. For example, in the file used in the previous examples, some devices in the /dev directory have a major number and a minor number, and some others have only a minor number. Therefore, for example, if you print out column 5, you will get a different piece of information for each row. To print out the number of columns (fields) per line in the file, you use the command **awk '{print NF}'** *filename*. In Example 4-18, the number of columns is printed out for the data file. The output of **awk** is piped to the command **sort -u**, which removes redundant output. As you can see, the number of columns varies between 2, 9, 10, and 11.

Example 4-18 *Using* ***print NF*** *to Print Out the Number of Fields*

```
[NetProg@server1 awk-scripts]$ awk '{print NF}' awk-data-ls | sort -u

10

11
```

```
2

9

[NetProg@server1 awk-scripts]$
```

Finally, say that there is a requirement to print out the minor number (fifth column) of all symlinks and multiply that by a number, such as 512. Example 4-19 illustrates how you can use **awk** to perform arithmetic operations on the data in the file. The minor number is printed first, followed by the string **"----"** and, finally, the result of the multiplication.

Example 4-19 *Using **print NF** to Print Out the Number of Fields*

```
[NetProg@server1 awk-scripts]$ awk '/^l/ {print $5,"----",$5*512}' awk-data-ls

3 ---- 1536

11 ---- 5632

13 ---- 6656

25 ---- 12800

4 ---- 2048

15 ---- 7680

15 ---- 7680

15 ---- 7680

[NetProg@server1 awk-scripts]$
```

Keep in mind that, like any other command in Linux, **awk** can be used with **grep** and **sed** (covered in the next section) through piping. Typically, **grep** is preferred for the search operation, and **sed** can be used for text manipulation. The output of any of the commands can be piped to any other command, as long as the output of one command is valid input for the other.

The sed Utility

sed, which stands for *stream editor*, is a utility that reads text from a file line by line, from stdin, or from a pipe; processes this text in some way; and outputs the edited version. The output of **sed** may be directed to stdout, piped to a new file, or written back to the source file. **sed** is an important utility that you need to be familiar with because it allows you to integrate into your scripts powerful text editing capabilities that are not feasible using visual text editors.

sed allows you to search a file for a match on a particular pattern, expressed using a regex, or to specify a particular line or range of lines. Much like **grep** and **awk**, **sed** processes the whole line containing a match. After the lines to be processed are identified, an action is taken. The following are some common actions with **sed**:

- Printing (**p**)
- Substitution (**s**)
- Deleting (**d**)
- Writing to files (**w**)
- Appending (**a**)
- Changing (**c**)
- Inserting (**i**)

The general syntax for **sed** is **sed** *options* **'***line_identification action***'** *filename*. Example 4-20 provides a simple example of the print action.

Example 4-20 *Data File to Test **sed***

```
! File show-bgp used to test sed

[NetProg@server1 sed-scripts]$ cat show-bgp

*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i

*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i

*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i

*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i

*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Attempting to print the first line of the file (without -n)
```

```
[NetProg@server1 sed-scripts]$ sed '1p' show-bgp
*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Attempting to print the first line of the file (with -n)
[NetProg@server1 sed-scripts]$ sed -n '1p' show-bgp
*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
```

File **show-bgp** contains five text lines that are routes extracted from the Internet routing table. The command **sed '1p' show-bgp** is used to attempt to print the first line of the file. As you can see, the line number, **1**, is followed by the action, **p**, in the command, inside single quotation marks. However, the output is not what you would expect. **sed** prints out the *whole file* and duplicates the first line. If you add the **-n** option to the command, it prints only the first line, as required.

This section dives right into using the **-n** option because understanding the behavior in Example 4-20 is your path to understanding the workflow of **sed**: When **sed** reads a line from a file, that line is saved in memory in a *pattern buffer*. When the processing of this one line is complete, the pattern buffer is emptied and then populated with the next line to be processed.

By default, the **sed** command prints out the contents of the pattern buffer. Because the whole file passes through the pattern buffer line by line, **sed** prints out each of these lines and eventually prints the whole file. In addition, the **1p** in the command in the example instructs **sed** to print out the first line, which means the first line is printed out twice. The **-n** option causes **sed** *not* to print out the content of the pattern buffer, which means only the first line is printed out.

To elaborate on this, in Example 4-21 the command **sed ' ' show-bgp** prints the whole file. The whitespace between the double quotation marks means no selection of lines and no action is to be taken. This, in turn, means that only the pattern buffer—equivalent to the whole file—is printed out. **sed -n ' ' show-bgp** prints out nothing because each line is placed in the pattern buffer, no action is taken on it, the line is discarded, and then the next line is placed in the pattern buffer, and so on.

Example 4-21 *The **-n** Option Instructing **sed** to Not Print the Contents of the Pattern Buffer*

```
! The default behavior of sed is to print out the whole file
[NetProg@server1 sed-scripts]$ sed ' ' show-bgp
*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Using the -n option to not print the contents of the pattern buffer
[NetProg@server1 sed-scripts]$ sed -n ' ' show-bgp
[NetProg@server1 sed-scripts]$
```

Now that the meaning of the **-n** option is clear, Example 4-22 displays how to print the whole file, a single line, and a range of lines. It also illustrates the use of the **$** symbol to indicate the last line of the file and, finally, the use of the *begin,+N* notation to specify a number of lines *N* to process after a specific line number indicated by *begin*.

Example 4-22 *Selecting Specific Lines to Print*

```
! Print out the whole file
[NetProg@server1 sed-scripts]$ sed -n 'p' show-bgp
*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
```

```
[NetProg@server1 sed-scripts]$


! Print out the third line only
[NetProg@server1 sed-scripts]$ sed -n '3p' show-bgp
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Print out the last line of the file using the $ sign
[NetProg@server1 sed-scripts]$ sed -n '$p' show-bgp
*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Print out lines 3 and 4
[NetProg@server1 sed-scripts]$ sed -n '3,4p' show-bgp
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Print out line 3 till the last line
[NetProg@server1 sed-scripts]$ sed -n '3,$p' show-bgp
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Print out line 2 and another 2 line afterwards
[NetProg@server1 sed-scripts]$ sed -n '2,+2p' show-bgp
*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
```

To delete a line instead of printing it out, you use the **d** action. In Example 4-23 lines 2 and 3 are deleted, and the rest of the file is printed out. Notice that in this case the **-n** option is not used because the purpose of the example is to show the whole file and omit the deleted lines. The **-n** option is also used in the example, and the output is empty because the lines have been deleted.

Example 4-23 *Using the **d** Action to Delete Lines from a File*

```
! Omitting the -n option
[NetProg@server1 sed-scripts]$ sed '2,3d' show-bgp
*>i5.41.131.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.134.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
*>i5.41.135.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Adding the -n option
[NetProg@server1 sed-scripts]$ sed -n '2,3d' show-bgp
[NetProg@server1 sed-scripts]$
```

It is important to note that the **sed** command does not write anything to the disk using any of the commands introduced so far. In order to store the results of the processing done by the **sed** command, you need to either pipe the output to a file or use the **w** action as in **sed -n** '*processing* **w** *results_file*' *file_to_be_processed*. Example 4-24 illustrates both methods of writing the results to disk.

Example 4-24 *Writing the Results of **sed** to Disk*

```
! Printing out lines 2 and 3 using the p action with piping
[NetProg@server1 sed-scripts]$ sed -n '2,3p' show-bgp > results-using-pipe
[NetProg@server1 sed-scripts]$ cat results-using-pipe
```

```
*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i

*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i


! Printing out lines 2 and 3 using the w action

[NetProg@server1 sed-scripts]$ sed -n '2,3 w results-using-w' show-bgp

[NetProg@server1 sed-scripts]$ cat results-using-w

*>i5.41.132.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i

*>i5.41.133.0/24    196.201.61.245               100      0 16637 39386 25019 39891 i
```

To overwrite the source file instead of creating a new file containing the results, you simply specify the source file's name in either of the commands in Example 4-24, instead of using a new filename. For example, using piping, instead of using the command **sed -n '2,3p' show-bgp > results-using-piping** as in Example 4-24, you use the command **sed -n '2,3p' show-bgp > show-bgp** to write the results of the processing back to the show-bgp file, which is the source file. A word of caution though: Make sure you don't overwrite the file with the wrong data and always make backups of important files before attempting to write any data to them.

Instead of specifying line numbers, the **sed** command can identify specific lines in a file by searching for a match in a pattern expressed as a regex. The syntax of the command in that case is **sed '***/pattern/ action***'** *filename*. In Example 4-25, **sed** is used to print out all text lines that have either **1** or **2** as the first octet and **.23** as the second octet, from a file containing a list of miscellaneous BGP routes extracted from the Internet routing table

Example 4-25 *Identifying the Lines to Be Processed by Matching on a Regex Pattern*

```
[NetProg@server1 sed-scripts]$ sed -n '/[12]\.23\./ p' bgp-routes-misc

 *   1.23.220.0/24    138.187.128.20      500000    300      0 (65000) 6453 4755 45528 i

 *   1.23.224.0/24    138.187.128.20      500000    300      0 (65000) 6453 4755 45528 i

 *   1.23.225.0/24    138.187.128.20      550000    300      0 (65000) 9498 45528 i

 *   1.23.226.0/24    138.187.128.20      500000    300      0 (65000) 6453 4755 45528 i

 *   1.23.227.0/24    138.187.128.20      500000    300      0 (65000) 6453 9498 45528 i

 *   2.23.144.0/20    138.187.128.38      601000    300      0 (65000) 3320 1299 ?

 *   2.23.160.0/22    138.187.128.79      500000    300      0 (65000) 6453 20940 20940

16625 i

 *   2.23.164.0/23    138.187.128.79      500000    300      0 (65000) 6453 20940 20940

16625 i
```

The regex pattern used should be familiar to you by now. The **[12]** character class matches a string with either **1** or **2**. The characters **\.** are used to escape the dot and match on a literal dot. This is followed by the number **23** and then another **\.** that matches another dot. Therefore, the regex **[12]\.23\.** will match on any IP address starting with **1.23** or **2.23**, which is exactly what is listed in the output in Example 4-25. Note that **sed** can match on a simple literal such as **3356** to print out all routes containing **3356** in their AS paths, as shown in Example 4-26.

Example 4-26 *Identifying the Lines to Be Processed by Matching on the Literal 3356*

```
[NetProg@server1 sed-scripts]$ sed -n '/3356/ p' bgp-routes-misc

 *   4.55.0.0/16      138.187.128.38      500000    300      0 (65000) 3356 i

 *   4.128.0.0/9      138.187.128.4       500000    300      0 (65000) 3356 i

 *   5.24.64.0/19     138.187.128.4       550000    300      0 (65000) 3356 34984 16135

i
```

To perform several operations using the same **sed** command, you use the **-e** option. In Example 4-27, the routes originating in each of the two ASs 3356 and 16625 are identified using two regular expressions, and the results of the match operations are written to different text files.

Example 4-27 *Multiple Operations Using the* **-e** *Option*

```
[NetProg@server1 sed-scripts]$ sed -n -e '/3356 i/ w 3356.txt' -e '/16625 i/ w 16625.txt'

bgp-routes-misc


[NetProg@server1 sed-scripts]$ cat 3356.txt

 *   4.55.0.0/16      138.187.128.38      500000    300      0 (65000) 3356 i

 *   4.128.0.0/9      138.187.128.4       500000    300      0 (65000) 3356 i
```

```
interface GigabitEthernet100/0/0/1 l2transport
 description *** This is the OLD description ***
 encapsulation dot1q 100
 rewrite ingress tag pop 1 symmetric
 ethernet-services access-group INGRESS-POLICY ingress
 ethernet-services access-group EGRESS-POLICY egress
!


! Changing the intf description and encapsulation
[NetProg@server1 sed-scripts]$ sed -e 's/desc.*/description *** NEW description AFTER
sed ***/' -e 's/encap.*/encapsulation dot1q 200/' intf-config-template > intf-config-
RTR-01.conf


! New configuration file ready to be applied to RTR-01
[NetProg@server1 sed-scripts]$ cat intf-config-RTR-01.conf
!
interface GigabitEthernet100/0/0/1 l2transport
 description *** NEW description AFTER sed ***
 encapsulation dot1q 200
 rewrite ingress tag pop 1 symmetric
 ethernet-services access-group INGRESS-POLICY ingress
 ethernet-services access-group EGRESS-POLICY egress
!
```

Later in this chapter you will see how you can use Expect to automate the process of logging in to a device, executing commands in that open session, and then logging out. When you couple this with **sed**'s substitution capabilities and the Bash scripting that you will learn starting in the next section, there is a lot you can accomplish with respect to configuration management automation using only Linux, Bash, and Expect.

Several other actions are possible with **sed**, using actions such as the change action (**c**), where a whole line is replaced by another; the insert action (**i**), where a line is inserted in the file at a specific line number; and the append action (**a**), which adds a line after a specific line number. For a list of all actions that are possible with **sed**, see the **sed** man page.

General Structure of Shell Scripts

A *shell script* is simply a Linux file. Such a file should always start with a *hashbang*, which is the character sequence **#!**. The hashbang is then followed by the absolute path of the shell program that is supposed to run this script. For example, **#!/bin/bash** means that the code in the script should be parsed and interpreted using the Bash shell, whose absolute path is /bin/bash. The character sequence **#!** at the beginning of the script tells the program loader (which is the part of the kernel responsible for loading programs into memory) two important things. First, it indicates that this is a shell script (not a compiled executable). Second, it indicates that the path following the **#!** (in this case /bin/bash) is where the program (the shell that will parse and interpret this script) is located. The script will be passed as an argument to the shell that the hashbang points to, regardless of where the script is run from or what the default shell of the user who runs the script is.

Note that omitting the hashbang in a script may result in unexpected behavior; the behavior depends on what shell it is being run from. If a shell script without a hashbang is run from a Bash shell, the shell spawns a subshell to execute the script.

To write scripts that are portable from one system to another and from one distro to another, it is good practice to place something on that first line that does not need to be changed as the script is run on different systems. The absolute path to the Bash shell binary on different systems might be different. Therefore, it is recommended to use **#!/usr/bin/env bash** instead of the absolute path. This tells the program loader to pass the script to the Bash shell whose exact path is specified in the **PATH** environment variable in the /usr/bin/env file. In order to view the **PATH** environment variable and confirm that the path to the Bash shell is there, you use the command **env | grep "PATH"**. The **env** file is a binary executable, so it cannot be viewed using regular file viewing utilities such as **cat**, **more**, or **less**.

The Linux OS usually does not use file extensions to determine file types. File extensions in Linux are primarily used by humans to quickly identify file types. They are also used for interoperability with other operating systems. Think of a tarball created on a Linux system and de-archived on a Windows machine. File extension are also used by some applications running on top of the Linux OS. A shell script can be saved in a file with any valid Linux filename and extension. This book follows the common practice of using the .bash extension for Bash script files.

Files containing shell scripts need to be executable. Depending on whether a script is run by the file owner, a user who is a member of the file group, or someone who is neither this nor that (other), an **x** (execute permission) has to be reflected in the file mode bits for the corresponding category. Recall that the **chmod** command is used for that, using the syntax **chmod u|g|o|a+x** *filename*, where **u** is for user, **g** is for group, **o** is

for other, and **a** is for all. Note that in case the file has to be executable for all, the **a** option can be omitted, and the command becomes **chmod +x** *filename*.

In case a script already exists, or if you have a template that you frequently use to create new scripts (by copying and renaming the template), you can use the **-p** option with the **cp** command to preserve the original file permissions so that if the template or the original script is already executable, you do not have to issue the **chmod** command to make it so.

When the script is complete and the file is saved and made executable, the next step is to run the script. You run a script by typing its name in the shell. However, if two files on the system happen to have the same name, how does the program loader differentiate between both and then choose one of them? If the full path of the script file (whether relative or absolute) was used for execution, there is no doubt about which file will be executed. If only the name of the file is given, the system consults the **PATH** environment variable and searches for the filename in all the paths listed in the **PATH** variable, in order. The first file it finds is then executed. The current working directory (identified by the **pwd** command) plays no part here. Therefore, in order to execute a shell script, you need to either provide the path to the file or make sure your scripts are in a directory listed in the **PATH** environment variable.

Example 4-30 shows a simple Bash script, which is just the hashbang followed by a sequence of commands. At different points, the script uses the **echo** command to output text enclosed in quotation marks. The script also uses the **pwd** command to print the current working directory before and after issuing the **cd ..** command, which navigates to the parent directory. The command **ls -l | wc -l** uses piping to print the number of lines in the output of **ls -l**. The script also uses the **sleep** command to pause the execution of the next line of code for a number of seconds (provided as a parameter to the **sleep** command). Note the use of the **#** symbol to insert comments in the script. These commented lines are for informational purpose only, and the shell ignores them.

Example 4-30 *Simple Bash Script*

```
[NetProg@server1 Scripts]$ cat BasicScript.bash

#!/usr/bin/env bash

# This is the first section of the script

echo "The current working directory is.."

pwd

echo "The script sleeps for 3 seconds here"

sleep 3

# This is the second section of the script

echo

cd ..

echo "The current working directory now is.."

pwd

echo "The number of lines in the output of the ls -l cmd is.."

ls -l | wc -l

echo "The script sleeps for 5 seconds here"

sleep 5
```

This script was created by the vim text editor, but any text editor—such as **nano**, **gedit**, or **emacs**—could be used instead. The script is then run by using the shorthand notation **./BasicScript.bash**, as shown in Example 4-31. (Remember to make sure the file is executable before running it.)

Example 4-31 *Executing the Bash Script*

```
[NetProg@server1 Scripts]$ ./BasicScript.bash

The current working directory is..

/home/NetProg/Scripts

The script sleeps for 3 seconds here


The current working directory now is..

/home/NetProg

The number of lines in the output of the ls -l cmd is..

10

The script sleeps for 5 seconds here

[NetProg@server1 Scripts]$
```

The interpreter ignores commented lines that are part of the script. Comments are added to a script for better readability and easier maintenance. Comments in Bash are identified by the **#** symbol right before the comment. The **#** can be at the start of a line, which causes the whole line to be ignored, or after a line of code, so that only the text after the **#** is ignored. Example 4-32 provides two examples of comments.

Example 4-32 *Comments at the Beginning of a Line and at the End of a Line of Code*

```
[NetProg@server1 Scripts]$ cat Comments.bash

#!/usr/bin/env bash

# This is a script to illustrate

# common usage of comments in Bash


Random_Num_1=123 # This is the first Random Number

Random_Num_2=321 # This is the second Random Number


echo -e "The 1st random number is $Random_Num_1\n"

#echo -e "The 2nd random number is $Random_Num_2\n"


#This is the end of the script


[NetProg@server1 Scripts]$ ./Comments.bash

The 1st random number is 123


[NetProg@server1 Scripts]$
```

Note that the only line in Example 4-32 that starts with a **#** symbol and is not a comment is the line that starts with the hashbang at the very beginning of the script.

Output and Input

A script runs the same logic every single time it is run. However, the input to the script may change each time the script is executed, possibly producing different results. Therefore, an important part of any scripting language is the capability to output the results of running the script, if any. Equally important is the capability of passing input to the script. In this section you will see how to pass input to a Bash script in a few different ways and how to the display the output to stdout.

Output

As shown earlier in this chapter, you use the **echo** command to print characters to the screen. The very basic syntax of the **echo** command is **echo** *options* **"***text***"**. By default, the **echo** command outputs the text inside the quotation marks, followed by a newline character. Example 4-33 shows a script composed of three **echo** commands. The first **echo** commands prints the text "This is the first line" followed by a newline character, such that the second **echo** command starts its output at the beginning of the following line. The second **echo** command outputs the text "This is the second line", followed by a newline character, such that the third **echo** command outputs its line of text at the beginning of the third line, and so forth.

Example 4-33 *Using the* ***echo*** *Command Without Any Options*

```
[NetProg@server1 ~]$ cat Echo_Vanilla.bash

#! /usr/bin/env bash

echo "This is the first line"

echo "This is the second line"

echo "This is the third line"


[NetProg@server1 ~]$ ./Echo_Vanilla.bash

This is the first line

This is the second line

This is the third line

[NetProg@server1 ~]$
```

The default behavior can be changed by using the **-n** option with the **echo** command. Using this option, the three lines of text are output on the

```
echo "And the current working directory is: $(pwd)"


[NetProg@server1 Scripts]$ ./EchoWithVariable.bash

The server hostname is Server1

And the current working directory is: /home/NetProg/Scripts
```

Note

Single quotation marks preserve the literal value of the characters enclosed within quotation marks. Double quotation marks also preserve the literal value of the characters within the quotation marks, except for the dollar symbol (**$**), the single quotation marks ('), and the backslash (\). You will learn later in this chapter how each of these symbols translates into a special meaning inside the double quotation marks. You can also refer to Section 3.2.1 of the *GNU Bash Manual* for further details on the use of quotation marks in scripts.

POSIX, which stands for Portable Operating System Interface, is a family of standards specified by the IEEE for maintaining compatibility between operating systems. For the **echo** command to be used portably across POSIX-compliant operating systems, the **-n** option and escape characters must not be used. Because of this and the limitations of **echo**, particularly with formatted output, the IEEE encourages the use of the **printf** command instead of **echo**. (See https://pubs.opengroup.org/onlinepubs/009695399/utilities/echo.html for more information.)

The **printf** command is a very common command in the C and C++ programming languages. The general syntax of the **printf** command is **printf "***format***" "***arguments***"**. In Example 4-37 **printf** is used in its most basic form: to print out a string to stdout.

Example 4-37 *Basic Usage of the* ***printf*** *Command to Print Out a String*

```
NetProg@server1 Scripts]$ cat printf.bash

#!/usr/bin/env bash

printf "This is a simple string"


! Executing the script

[NetProg@server1 Scripts]$ ./printf.bash

This is a simple string[NetProg@server1 Scripts]$
```

The most visible difference between **echo** and **printf** is the fact that, unlike **echo**, **printf** does not print a trailing newline. Therefore, as you can see in the output in Example 4-37, the command prompt appears on the same line as the script output. When you add the newline character, as in Example 4-38, the **printf** command emulates the functionality of **echo**.

Example 4-38 *Printing Out a String and a Trailing Newline Character*

```
[NetProg@server1 Scripts]$ cat printf.bash

#!/usr/bin/env bash

printf "This is a simple string\n"


[NetProg@server1 Scripts]$ ./printf.bash

This is a simple string

[NetProg@server1 Scripts]$
```

printf is more commonly used as shown in Example 4-39.

Example 4-39 *Using a Placeholder to Print Out the String Arguments*

```
[NetProg@server1 Scripts]$ cat printf.bash

#!/usr/bin/env bash

printf "%s-%s-%s-%s\n" "Khaled" "Jeff" "Vinit" "Anton"


[NetProg@server1 Scripts]$ ./printf.bash

Khaled-Jeff-Vinit-Anton

[NetProg@server1 Scripts]$
```

The **printf** command is followed by the format field in double quotation marks, containing a series of the character combination **%s** with dashes in between. **%s** is a *placeholder* that is used as a directive to indicate that a string will be placed in this position. The first placeholder

```
echo "This is the third argument: ${3}"

echo "This is the script name: ${0}"

echo "This script received $# arguments"


[NetProg@server1 Scripts]$ ./Arguments.bash Khaled Jeff Vinit

This is the first argument: Khaled

This is the second argument: Jeff

This is the third argument: Vinit

This is the script name: ./Arguments.bash

This script received 3 arguments
```

The second way to pass data to a script is by using the **read** command. This command pauses a script indefinitely until the user inputs a line of data and presses the Enter key. Common syntax of the **read** command is **read -p** "*message*" *variable1 variable2 .. variableN*. Example 4-43 shows how to use the **read** command to read the user's first and last names and then output the user's full name by using the **echo** command.

Example 4-43 *Reading User Input Using the **read** command*

```
[NetProg@server1 Scripts]$ cat InOut.bash

#!/usr/bin/env bash

read -p "Please enter your first name: " firstname

read -p "Please enter your last name: " lastname

echo "$firstname $lastname"


[NetProg@server1 Scripts]$ ./InOut.bash

Please enter your first name: Khaled

Please enter your last name: Abuelenain

Khaled Abuelenain
```

The operation of the **read** command is slightly more complex than Example 4-43 indicates. The **read** command actually accepts a line of text from the user, and this line of text is terminated when the user presses the Enter key. This line of text is then split into words, each word separated from the next by a space, which is called the *inter-field separator* (*IFS*). Then the first word is assigned to the first variable, the second word is assigned to the second variable, and so forth. Rewriting the script in the previous example, Example 4-44 uses a single **read** command to accept the first and last names from the user and still assign them to two different variables.

Example 4-44 *Reading User Input with a Single Read Command*

```
[NetProg@server1 Scripts]$ cat InOut-1.bash

#!/usr/bin/env bash

read -p "Please enter your first and last names separated by a space: " firstname

lastname

echo "$firstname $lastname"


[NetProg@server1 Scripts]$ ./InOut-1.bash

Please enter your first and last names separated by a space: Khaled Abuelenain

Khaled Abuelenain
```

What if the input from the user produces more words than there are variables in the command? In this case, each variable is assigned a word from the user input until the last variable remains, and that last variable is then assigned all remaining words. For example, if the user inputs a line that is composed of five words, and the **read** command has only three variables as arguments, the first and second variables are assigned the first and second words, respectively, and the third variable is assigned the remaining three words. What if the number of variables exceeds the number of words in the user input? In this case, starting with the first variable, each variable is assigned a word, until the words run out, and then the rest of the variables are assigned empty values.

The **read** command has several options, including the following:

• **-s**: Makes user input silent (that is, it does not appear on the screen as it is typed). This is a very handy option for reading passwords into a script.

```
[NetProg@server1 Scripts]$ ./Variables1.bash

The value of testvar is 10 - type:integer


The value of testvar is Cairo - type:string
```

In Bash, you can declare a variable before it is assigned a value by using the **declare** command. The syntax of the **declare** command is **declare** *options variable_name*=*variable_value*. As you can see from the syntax, a variable may also be assigned a value in the **declare** command. Declaring variables is mandatory in some programming languages, where a variable accepts data of only one specific type and not any other. In Bash, you can declare a variable to accept only integer values by using the **-i** option, or you can declare a variable as a constant (that is, read only) by using the **-r** option. You can use the **-l** option to convert any uppercase letters in a variable value to lowercase, and you can use the **-u** option to do just the opposite. Each of these options sets the corresponding property of a variable, called a *variable attribute*. Although it may sound counterintuitive, any attribute set using any of the previously mentioned options may be *unset* by replacing the option with the **+** option in the **declare** command. You can view attributes set for a variable by using the **declare** command with the **-p** option.

Example 4-48 illustrates the use of the **declare** command. Variable **var1** is declared with the **-l** option and is not assigned a value. Variable **var2** is declared as an integer using the **-i** option and is assigned the value **200**. Finally, variable **var3** is declared as a constant using the **-r** option and is given the value **Alexandria**. A string is assigned to **var1** and, as expected, all uppercase letters are converted to lowercase. Then an attempt is made to assign a string to **var2**, and while the assignment does not generate an error, the **echo** command outputs a 0 instead of the correct variable value because var2 was declared as an integer but is being assigned a value of type string. Finally, an attempt is made to assign an integer value to **var3**, which generates an error because **var3** was declared as a constant.

Example 4-48 *Variable Declaration Using the* ***declare*** *Command*

```
[NetProg@server1 Scripts]$ cat Variables2.bash

#!/usr/bin/env bash

declare -l var1

declare -i var2=200

declare -r var3=Alexandria

var1=ThIsIsATeSTSTRInG

echo "The value of var1 is $var1"

var2=TestString

echo "The value of var2 is $var2"

var3=Sinai

echo "The value of var3 is $var3"

echo "The attributes of var1 are: $(declare -p var1)"


[NetProg@server1 Scripts]$ ./Variables2.bash

The value of var1 is thisisateststring

The value of var2 is 0

./Variables2.bash: line 15: var3: readonly variable

The value of var3 is Alexandria

The attributes of var1 are: declare -l var1="thisisateststring"
```

Notice that in order to execute the **declare** command and display its result using **echo**, you use **$(declare -p var1)** inside the double quotation marks of the **echo** command. The same applies to any system command. This is an example of *command substitution*: A system command's output is used in some context, such as in the output of the **echo** command, or assigned to a variable. For instance, to assign the output of the command to the variable **Var1**, you use **Var1=$(declare -p var1)**.

Arithmetic operations in Bash are performed using *arithmetic expansion*. The notation **$((***expression***))** is used to evaluate the arithmetic expression between the double parentheses. The result can then be processed using the **echo** *variable* command. In Example 4-49, the result of dividing nine by two is evaluated using arithmetic expansion and then assigned to the variable **expr**. The value of **expr** is then printed out to stdout.

Example 4-49 *Simple Integer-Based Arithmetic Operations*

```
[NetProg@server1 Scripts]$ cat arithmetic.bash

#!/usr/bin/env bash

expr=$((9/2))

echo "The value of 9/2 is: $expr"
```

```
[NetProg@server1 Scripts]$ ./arithmetic.bash

The value of 9/2 is: 4
```

Notice that although the result of dividing nine by two should be 4.5, the result of the arithmetic expansion in the script output has been truncated to 4. As stated earlier in this section, Bash does not have a floating point data type and does not support floating point operations. However, a workaround involves using the **bc** utility. To assign the precise result of an arithmetic operation to a variable, you use the syntax *variable*=**$echo("scale=***decimal_places* ; *expr*" | **bc)**. The **scale** option sets the number of decimal places required in the output. A whitespace before or after the semicolon (for readability) is optional. Example 4-50 shows the **bc** utility being used to output the value of some arithmetic operations to a precision set by **scale**.

Example 4-50 *The Use of the* ***scale*** *Command with Addition, Subtraction, Multiplication, and Division*

```
[NetProg@server1 Scripts]$ cat bcutiliy.bash

#!/usr/bin/env bash

Add1=$(echo "1.2+3.546789" | bc)

Add2=$(echo "scale=3 ; 1.2+3.546789" | bc)

Subt1=$(echo "10-3.546789" | bc)

Subt2=$(echo "scale=3 ; 10-3.546789" | bc)

Mult1=$(echo "1.234*5.678" | bc)

Mult2=$(echo "scale=10 ; 1.234*5.678" | bc)

Div1=$(echo "10/3" |bc)

Div2=$(echo "scale=7 ; 10/3" | bc)

echo $Add1

echo $Add2

echo $Subt1

echo $Subt2

echo $Mult1

echo $Mult2

echo $Div1

echo $Div2


[NetProg@server1 Scripts]$ ./bcutiliy.bash

4.746789

4.746789

6.453211

6.453211

7.006

7.006652

3

3.3333333
```

Notice that the **bc** utility does not always respect the **scale** setting in the command. With addition and subtraction, regardless of whether the scale is set, it is ignored, and the scale that is used for the result is the same as that of the highest-precision number in the arithmetic operation. For multiplication, the scale that is set is honored, and if the scale is not set, the scale of the highest-precision number in the multiplication is used. Not setting a scale with division results in using the default scale of zero. Otherwise, the scale setting is honored.

Indexed and Associative Arrays

An *array* is a data structure that consists of one or more *elements*. A variable representing the array data structure is typically just called an *array*. Each array element stores one of the array's values. For example, the array **capitals** may hold three values: **Cairo**, **Washington**, and **Minsk**. Each of the array elements is identified either by its position in the array, called the *index*, or by a unique value, called a *key*. An *indexed array* uses indexes to identify its elements. An *associative array* uses keys to identify its elements.

If the array **capitals** were an indexed array, one way the elements would be identified is by **capitals[0]**, **capitals[1]**, and **capitals[2]**, where the

numbers enclosed in brackets are the array element indexes, and **capitals[0]** would hold the value **Cairo**, **capitals[1]** would hold the value **Washington**, and **capitals[2]** would hold the value **Minsk**. Note that the first element has an index of 0. However, the actual elements holding values do not have to be consecutive, and not all elements have to be assigned values. The elements holding the three values could be **capitals[3]**, **capitals[18]**, and **capitals[179]**. The elements not holding values, such as **capitals[0]** or **capitals[100],** have the value **null**. This will become clear after you study the next example.

In the case of an associative array, the elements of the array may be identified by **capitals[Egypt]**, **capitals[USA]**, and **capitals[Belarus]**, where **Egypt**, **USA**, and **Belarus** are the array elements *keys*. You can guess what the value of each element would be.

When a variable is assigned a value using the syntax *variable***[***index***]=***value*, the variable is automatically treated as an indexed array, and you can simply start assigning values to its elements by using this syntax. Alternatively, you can use the **declare** command to declare an array before values are assigned to its elements by using the syntax **declare -a|-A** *array_name*, where the **-a** option is for indexed arrays, and the **-A** option is for associative arrays.

While the benefit of declaring an indexed array is questionable (it *might* speed up subsequent array operations), it is mandatory for associative arrays. You can use the **declare** command to declare *and* initialize the array's elements with values by using the syntax **declare -a** *array_name***=(** *value1 value2 ... valueN* **)** for indexed arrays or **declare -A** *array_name***=(** **[***key1***]=***value1* **[***key2***]=***value2 ...* **[***keyN***]=***valueN* **)** for associative arrays. On the other hand, to access element values, the syntax **${***array_name***[***index***]}** is used for indexed arrays and **${***array_name***[***key***]}** for associative arrays. Indexed arrays and associative arrays are very similar in operation except that the former uses an index to address each of its elements, and the latter uses a key for this. This chapter focuses on indexed arrays and highlights the differences between indexed arrays and associative arrays, where applicable.

Example 4-51 displays some examples of basic array operations.

Example 4-51 *Declaring Arrays, Initializing Them Without Declaration, and Printing Their Values by Using the* ***echo*** *Command*

```
[NetProg@server1 Scripts]$ cat Arrays1.bash

#!/usr/bin/env bash

declare -a authors=(Khaled Jeff Vinit Anton) #Declare and initialize an indexed array

declare -A last_names #Declare an associative array


#Assign values to array capitals (without declaration) using element indices

#Element indices don't have to be consecutive numbers

capitals[2]=Cairo

capitals[10]="Washington DC" #Double quotes used due to whitespace

capitals[50]=Minsk

capitals[53]=999


#Assign values to the elements of last_name using the element keys

last_names[Khaled]=Abuelenain

last_names[Jeff]=Doyle

last_names[Vinit]=Jain

last_names[Anton]=Karneliuk


echo "The value of capitals[1] is ${capitals[1]}"

echo "The value of capitals[2] is ${capitals[2]}"

echo "The value of capitals[10] is ${capitals[10]}"

echo "The value of capitals[50] is ${capitals[50]}"

echo "The value of capitals[53] is ${capitals[53]}"


Sum=$((${capitals[53]}+123))


echo "The value capitals[53]+123 is $Sum"

echo "The number of elements in capitals is ${#capitals[@]}"

echo -e "The index values of capitals are ${!capitals[@]}\n"

echo "The value of the 2nd element of authors is ${authors[1]}"
```

```
echo "The value of the 3rd element of last_names is ${last_names[Vinit]}"

echo "The key values of last_names are ${!last_names[@]}"

echo "All values of last_names are ${last_names[@]}"


[NetProg@server1 Scripts]$ ./Arrays1.bash

The value of capitals[1] is

The value of capitals[2] is Cairo

The value of capitals[10] is Washington DC

The value of capitals[50] is Minsk

The value of capitals[53] is 999

The value capitals[53]+123 is 1122

The number of elements in capitals is 4

The index values of capitals are 2 10 50 53


The value of the 2nd element of authors is Jeff

The value of the 3rd element of last_names is Jain

The key values of last_names are Khaled Jeff Vinit Anton

All values of last_names are Abuelenain Doyle Jain Karneliuk
```

As you can see in Example 4-51, to access all elements of the array, you use the syntax **${***array_name***[@]}**. The command **echo "${!***array_name***[@]}"** prints out the index numbers in the case of an indexed array or the key values in case of an associative array, and **echo "${#***array_name***[@]}"** prints out the number of elements in the array. Alternatively, **echo "${#***array_name***[***N***]}"** prints out the length of the *N*th element. Using *array_name***[*]** is equivalent to using *array_name***[@]** in all the previous examples.

Notice in Example 4-51 that the first three elements of the array **capitals** are strings, and the fourth element is an integer. Elements of the same array may each hold a value of a different type. To prove this further, the number 123 is added to the value of **capitals[53]**, and the result of this arithmetic operation is assigned to the variable **Sum** and printed out.

You can add a new array element by simply making an assignment using the array name and index (or key). Alternatively, for indexed arrays, you can use the syntax *array_name***+=(***new_element_value***)** to add a new element that has an index number that is 1 more than the last index used in the array. For associative arrays, the equivalent syntax is *array_name***+=([***new_element_key***]=***new_element_value***)**. To remove an element from an array, you use the command **unset** *array_name***[***index*|*key***]**. To delete an entire array, you use **@** instead of *index* or *key* in the command. Example 4-52 shows how to add and remove array elements.

Example 4-52 *Adding and Removing Array Elements*

```
[NetProg@server1 Scripts]$ cat Arrays2.bash

#!/usr/bin/env bash

capitals[0]=Cairo

capitals[1]="Washington DC"

capitals[5]=Minsk

echo -e "The array capitals is currently: ${capitals[@]}\n"

capitals+=("Riyadh")

echo -e "The array capitals is now: ${capitals[@]}\n"

echo -e "And the list of indices is ${!capitals[@]}\n"

unset capitals[5]

echo -e "The array elements are now: ${capitals[@]}\n"

echo -e "And the list of indices becomes: ${!capitals[@]}\n"


[NetProg@server1 Scripts]$ ./Arrays2.bash

The array capitals is currently: Cairo Washington DC Minsk
```

```
The array capitals is now: Cairo Washington DC Minsk Riyadh


And the list of indices is 0 1 5 6


The array elements are now: Cairo Washington DC Riyadh


And the list of indices becomes: 0 1 6
```

The command **capitals+=("Riyadh")** adds a new element to the array. The array value in this case is, obviously, **Riyadh**, and the assigned index is the next available index number, **6**. The command **unset capitals[5]** is used to delete the element whose index is **5** and value is **Minsk**.

Arrays can be concatenated and the result assigned to a third array. Example 4-53 concatenates the arrays **capitals1** and **capitals2** and assigns the result to the array **capitals3**.

Example 4-53 *Adding and Removing Array Elements*

```
[NetProg@server1 Scripts]$ cat Arrays4.bash
#!/usr/bin/env bash
capitals1=(Cairo Washington Minsk)
capitals2=(London Dusseldorf)
capitals3=(${capitals1[@]} ${capitals2[@]})
echo -e "Array capitals1 has the elements: ${capitals1[@]}\n"
echo -e "And array capitals2 has the elements: ${capitals2[@]}\n"
echo -e "Concatenate them into capitals3: ${capitals3[@]}\n"

[NetProg@server1 Scripts]$ ./Arrays4.bash
Array capitals1 has the elements: Cairo Washington Minsk


And array capitals2 has the elements: London Dusseldorf


Concatenate them into capitals3: Cairo Washington Minsk London Dusseldorf
```

Arrays in any programming language allow for very complex data manipulation operations. Try to experiment with arrays to familiarize yourself with this fascinating data type.

Conditional Statements

When you need to execute a command or perform an action in programming only if a certain condition exists, you use conditional statements. Conditional statements may be a part of a larger construct that evaluates a condition, and if the condition is true, one or more commands are executed. However, if the condition is not true, further testing may be performed and other alternative commands executed. There are two primary conditional constructs in Bash:

• if-then-elif-then-else-fi (which we refer to as the if-then construct for brevity)

• case-in-esac

The if-then Construct

The flow through an if-then construct is very intuitive: The **if** keyword tests whether a condition is true, and if it is, the **then** keyword executes one or more commands. The **fi** keyword then marks the end of the construct. Example 4-54 shows the syntax of the if-then construct in its simplest form.

Example 4-54 *if-then Construct in Its Simplest Form: if-then-fi*

```
if [[ condition ]]
then
        command-block
fi
```

In Example 4-54, notice the mandatory whitespace between the condition and the double brackets. Example 4-55 shows a simple if-then-fi

construct in which the user is prompted to enter his or her username. The **if** statement tests whether the username is equal to the string **NetProg**. If it is, a message is output to stdout using the **echo** command. Otherwise, nothing happens, and the script exits.

Example 4-55 *Testing User Input and Output Message if Condition Is True*

```
[NetProg@server1 Scripts]$ cat if-then.bash

#!/usr/bin/env bash

read -p "Please enter your username: " username

if [[ $username = "NetProg" ]]

then

  echo "Hello $username"

fi


! This is the execution result when the condition evaluates to true

[NetProg@server1 Scripts]$ ./if-then.bash

Please enter your first name: NetProg

Hello NetProg

[NetProg@server1 Scripts]$


! This the execution result when the condition evaluates to false

[NetProg@server1 Scripts]$ ./if-then.bash

Please enter your first name: OtherUser

[NetProg@server1 Scripts]$
```

To avoid situations similar to the one shown in Example 4-55, where nothing happens if the condition tested is false, the if-then construct is designed to accommodate more complex algorithms than what you have seen so far. Example 4-56 shows the full syntax of the if-then construct.

Example 4-56 *The Full Syntax of the If-Then Construct*

```
if [[ condition ]]

then

        command-block

elif [[ alternative-condition ]]

then

        command-block-2

else

        default-command-block

fi
```

This is how the logic flows:

1. The **if** keyword tests whether *condition* is true. If it is, execute the command(s) in *command-block*.

2. If *condition* tested by the **if** keyword is not true, the **elif** keyword tests whether *alternative-condition* is true or not. If *alternative-condition* is true, execute the command(s) in *command-block-2*.

3. If more than one **elif** block exists, move to an **elif** block if and only if the previous **elif** block's *alternative-condition* evaluates to false.

4. If *alternative-condition* tested by the last **elif** keyword is not true, go to the **else** block and execute the command(s) in *default-command-block*.

5. Exit the construct at the **fi** keyword

The if-then construct is very flexible in that it can test for a single condition and execute one command block using a simple if-then-fi construct, as shown in Example 4-55. Or you can add to that an **else** block *without* an **elif** block so that the construct becomes if-then-else-fi. Alternatively, you may require a construct that uses **if** and **elif** blocks without an **else** block, so that the construct becomes if-then-elif-then-fi. And then you may have a full construct employing **if**, **elif**, and **else** blocks—that is, an if-then-elif-then-else-fi construct. The construct may also have more than one **elif** block testing for further alternative conditions, where each **elif** block is executed if the condition in the previous **elif** is false.

In any case, and regardless of what blocks are included in the code, only one command block is executed in any one run of the program. Either the execution block under the **if** keyword is executed or the block under *one of the* **elif** keywords is executed, or the block under the **else** keyword is executed. But never are two (or more) of these blocks executed in the same script run.

Example 4-57 enhances the script from Example 4-56 so that, if the username is not NetProg, it uses **elif** to test whether the username is NetDev. If the username is neither NetProg nor NetDev, a message is output to stdout by using an **else** statement.

Example 4-57 *Using the Full if-then-elif-then-else-fi Construct to Test User Input*

```
[NetProg@server1 Scripts]$ cat if-then-elif-else.bash

#!/usr/bin/env bash

read -p "Please enter your username: " username

if [[ $username = "NetProg" ]]

then

  echo "Hello $username"

elif [[ $username = "NetDev" ]]

then

  echo "Hello $username - please log out and log in using the NetProg account"

else

  echo "This access is not authorized - please log out immediately !"

fi


! Script execution results for three different user inputs

[NetProg@server1 Scripts]$ ./if-then-elif-else.bash

Please enter your username: NetProg

Hello NetProg


[NetProg@server1 Scripts]$ ./if-then-elif-else.bash

Please enter your username: OtherUser

This access is not authorized - please log out immediately !


[NetProg@server1 Scripts]$ ./if-then-elif-else.bash

Please enter your username: NetDev

Hello NetDev - please log out and log in using the NetProg account

[NetProg@server1 Scripts]$
```

In order for the if-then statement to make a decision, an expression has to be evaluated to be either true or false. Comparison operators are used to compare two values to each other, and the end result is either true or false. Strings, numbers, and files have different sets of comparison or evaluation operators that are used to evaluate a condition to be either true or false. This evaluation of a condition using comparison or evaluation operators may then be used in any programming construct requiring such an evaluation, including the if-then conditional statement. Table 4-1 lists the string comparison operators supported in Bash.

Table 4-1 *String Operators in Bash*

Operator	Description
<	Less than
>	Greater than
=	Equal to
==	Equal to
!=	Not equal to
-z	True if the length of the string is zero
-n	True if the length of the string is nonzero

String values used for comparisons are based on alphabetical order. For example, strings that start with the letter a are less than strings that start with the letter b, and those that start with the letter c are greater than those that start with the letter b, and so forth. Example 4-58 illustrates the use of some string comparison operators.

Example 4-58 *Using String Comparison Operators*

```
[NetProg@server1 Scripts]$ cat stringcompare.bash
#!/usr/bin/env bash
if [[ abcd < e ]]
then
  echo "abcd is less than e"
else
  echo "e is less than abcd"
fi
if [[ X = Y ]]
then
  echo "X is equal to Y"
else
  echo "X is not equal to Y"
fi
if [[ "Cairo" != "Colorado" ]]
then
  echo "Cairo is in Egypt, Colorado is in the US !"
fi
if [[ -n "Cairo" ]]
then
  echo "The string \"Cairo\" has non-zero length!"
fi
if [[ -z "" ]]
then
  echo "The empty quotes are a string with zero length"
fi


[NetProg@server1 Scripts]$ ./stringcompare.bash
abcd is less than e
X is not equal to Y
Cairo is in Egypt, Colorado is in the US !
The string "Cairo" has non-zero length!
```

```
The empty quotes are a string with zero length
```

Both the single equal sign (=) and the double equal sign (==) are used for evaluating equality in conditional statements. In the context of this book, which covers Bash scripting fundamentals, you can assume that they behave in an identical manner. There are, of course, subtle differences between them that are not covered here. Feel free to refer to the *GNU Bash Manual* for further elaboration.

Table 4-2 lists the arithmetic operators supported in Bash.

Table 4-2 *Arithmetic Operators in Bash*

Operator	Description
-lt	Less than
-gt	Greater than
-le	Less than or equal to
-ge	Greater than or equal to
-eq	Equal to
-ne	Not equal to

Integers are compared based on their numeric value. For example, 1 is less than 2, and 2 is less than 12. Example 4-59 illustrates the use of some of the integer comparison operators.

Example 4-59 *Using Integer Comparison Operators*

```
[NetProg@server1 Scripts]$ ./intcompare.bash
#!/usr/bin/env bash
if [[ 1 -lt 2 ]]
then
  echo "1 is less than 2"
else
  echo "2 is less than 1"
fi
if [[ 2 -gt 12 ]]
then
  echo "2 is greater than 12"
else
  echo "2 is less than 12"
fi
if [[ 135 -ne 246 ]]
then
  echo "135 and 246 are not equal !"
fi


[NetProg@server1 Scripts]$ ./intcompare.bash
1 is less than 2
2 is less than 12
135 and 246 are not equal !
```

Table 4-3 lists some of the file evaluation operators supported in Bash.

Table 4-3 *File Comparison Operators in Bash*

Operator	Description
-a or **-e**	True if file exists
-d	True if file is a directory
-f	True if file is a regular file
-r	True if file exists and is readable
-s	True if file is not zero in size
-w	True if file exists and is writable
-x	True if file exists and is executable

Example 4-60 illustrates the use of some of the file evaluation operators.

Example 4-60 *Using File Comparison Operators in Bash*

```
[NetProg@server1 Scripts]$ cat fileeval.bash
#!/usr/bin/env bash
if [[ -f /home/NetProg/Scripts/intcompare.bash ]]
then
  echo -e "The file intcomparison.bash exists and is a regular file !\n"
fi
if [[ -x stringcompare.bash ]]
then
  echo -e "The file stringcomparison.bash exists and is executable !\n"
fi


[NetProg@server1 Scripts]$ ./fileeval.bash
The file intcomparison.bash exists and is a regular file !


The file stringcomparison.bash exists and is executable !
```

The case-in Construct

The case-in conditional construct compares the value of a variable (or a word) to one or more expressions and executes the command block corresponding to the matching value. Example 4-61 shows the general syntax of this construct.

Example 4-61 *General Syntax of the case-in Construct*

```
case $variable in
        first-value)
                command-block-1
                ;;
        second-value)
                command-block-2
                ;;
        Nth-value)
                command-block-N
                ;;
        *)
                default-command-block
```

```
        ;;
esac
```

The case-in construct is very flexible and provides more than what immediately meets the eye. Each section of the construct corresponding to a testing value is called a *clause*. Each clause starts with a testing value, is followed by a command block, and ends with double semicolons (**;;**), a semicolon and an ampersand (**;&**), or double semicolons and an ampersand (**;;&**), each indicating a different course of action.

Each testing value ends in a closing parenthesis. The value may be a constant, a variable, or a regular expression. Following the testing value is a command block that is executed if the variable is equal to the testing value.

If the clause ends in double semicolons (**;;**), the construct exits if a match is found. If the clause ends in double semicolons and an ampersand (**;;&**), then even if a match is found, the testing continues with the testing value in the next clause. If a clause ends in a semicolon and an ampersand (**;&**), the command block of the next clause is executed without testing.

This is how the logic flows:

1. The **case** keyword tests whether *variable* and *first-value* are equal. If they are, the commands in *command-block-1* are executed. If they are not equal, *command-block-1* is skipped and *variable* is compared to *second-value*.

2. In case *command-block-1* is executed and the first clause ends in double semicolons (**;;**), the construct exits, and the execution is complete.

3. In case *command-block-1* is executed and the first clause ends in a semicolon and an ampersand (**;&**), *command-block-2* is also executed without *variable* being compared to *second-value*.

4. In case *command-block-1* is executed and the first clause ends in double semicolons and an ampersand (**;;&**), *second-value* is compared to *variable*, and if they match, the commands in *command-block-2* are executed. If they don't match, the commands are skipped.

5. These steps repeat until the last clause is reached, signified by the ***)**. An asterisk matches any value, so if the logic reaches this point in the construct, the commands in the *default-command-block* are executed.

6. The construct ends with the **esac** keyword.

Example 4-62 shows a simple multiple-choice quiz that tests the input from a user against two possible answers and provides default output in the event that the user's input does not match either testing values.

Example 4-62 *The case-in Construct Used to Conduct a Simple Quiz*

```
[NetProg@server1 Scripts]$ cat case-in.bash
#!/usr/bin/env bash
read -p "What is the Capital of Columbia ?  " Capital
case $Capital in
    Bogota)
    echo -e "Yes you are right - the capital of Columbia is Bogota\n"
    ;;
  Medellin)
    echo -e "Close but not correct\n"
    ;;
        *)
     echo -e "You need a geography lesson !\n"
    ;;
esac


[NetProg@server1 Scripts]$ ./case-in.bash
What is the Capital of Columbia ?  New York
You need a geography lesson !


[NetProg@server1 Scripts]$ ./case-in.bash
What is the Capital of Columbia ?  Medellin
Close but not correct
```

```
[NetProg@server1 Scripts]$ ./case-in.bash

What is the Capital of Columbia ?  Bogota

Yes you are right - the capital of Columbia is Bogota
```

Keep in mind that you may end any of the clauses in double semicolons (**;;**), in double semicolons and an ampersand (**;;&**), or in a semicolon and an ampersand (**;&**). This list of choices is *not* exclusive to the first clause only.

Loops

When you need to execute a command or a sequence of commands multiple times in a script, looping constructs come into play. A looping construct repeats a sequence of actions as long as a particular condition evaluates to true. Three looping constructs are commonly used in Bash:

• for-do-done

• while-do-done

• until-do-done

The for-do Loop

The for-do loop has two general forms, as shown in Example 4-63.

Example 4-63 *The for-do Loop General Syntax*

```
! The first form of the for-do loop

for VAR in list-of-values

do

  command-block

done


! The second form of the for-do loop

for (( expr1 ; expr2 ; expr3 ))

do

  command-block

done
```

The for-do loop starts with the **for** keyword and an argument that takes one of two forms. It also contains a command block that may be one or more commands and starts with the **do** keyword and ends with the **done** keyword.

In the first form, the variable *VAR* is assigned the values that follow the **in** keyword, one by one, and the commands in *command-block* are executed once for each value. Keep in mind that the commands may or may not reference or use the variable *VAR*. For example, the values may be the strings **Khaled** and **Jeff**, and *command-block* would be **echo "Hello World"**. The result of running the script would then be the string **Hello World** printing out two times without any reference to the two values. Example 4-64 shows the for-do loop being used to iterate through a list of values and then print out those values.

Example 4-64 *Printing a Sequence of Values by Using a for-do Loop*

```
[NetProg@server1 Scripts]$ cat for-do.bash

#!/usr/bin/env bash

count=1

for authors in Khaled Jeff Vinit Anton

do

  echo "Author-${count} is $authors"

  (( count++ ))

Done


[NetProg@server1 Scripts]$ ./for-do.bash

Author-1 is Khaled

Author-2 is Jeff
```

```
Author-3 is Vinit

Author-4 is Anton
```

Say that you want to have a user input a number, and then you want to print out all numbers from zero up to that number, in order. You would want to iterate through the values, starting from zero up to the number that the user has entered that will be stored in a variable. Assume that variable is called **maxnumber**. The first form of the for-do loop would not permit iterating through a list of values starting with zero and ending with **$maxnumber**. But the second form of the for-do loop would.

The second form of the for-do loop has a similar format to the for-do loop in the C programming language. The **for** keyword is followed by three expressions in double parentheses (arithmetic expansion), and separated by semicolons. The logic of these expressions is as follows:

1. The first expression is evaluated once. The result of this evaluation does not affect the loop.

2. The second expression is then evaluated.

3. If the result of evaluating the second expression is zero, exit the loop. Otherwise, proceed to the next step.

4. If the result of evaluating the second expression is a nonzero value, the commands in *command-block* are executed, and then the third expression is evaluated.

5. Repeat steps 2 through 4.

The for-do loop in Example 4-65 illustrates this logic.

Example 4-65 *Printing Numbers from 1 to a Maximum Number by Using a for-do Loop*

```
[NetProg@server1 Scripts]$ cat for-do-1.bash

#!/usr/bin/env bash

read -p "Enter maximum number: " max

for ((i=1; i<=$max; i++))

do

  echo "$i"

done


[NetProg@server1 Scripts]$ ./for-do-1.bash

Enter maximum number: 3

1

2

3
```

In this example, the user enters the number **3** when prompted for input by the **read** command. Starting with the for loop, the expression in the first field is evaluated. The variable **i** is set to **1**. This is done only once, regardless of how many times the loop iterates.

Then the expression in the second field is evaluated: Is **i** less than or equal to **max**? Yes, it is, so the commands in the body of the loop are executed. In this case, the **echo** command outputs the value of **i**, which is **1**.

On the loop's second iteration, the third expression is evaluated, which results in incrementing the value of **i** to **2**. Note that the expression *variable***++** increments the value of *variable* by **1**. Then the second expression is evaluated again: Is **i** still less than or equal to **max**? Yes, it is, so the **echo** command executes a second time, printing out the value of **i**, which is now **2**.

On the loop's third iteration, the third expression is evaluated, incrementing the value of **i** to **3**. The second expression is now evaluated: Is **i** still less than or equal to **max**? Yes, it is, so the **echo** command prints out the number **3**.

Finally, the third expression is evaluated, and the value of **i** is incremented to **4**. When the second expression is evaluated, **i**, which is **4** now, is not less than or equal to **max**. Therefore, the second expression evaluates to false, the loop exits, and the script finishes execution.

An interesting example for the use of the for loop is to parse through a file, word by word, and perform some processing on these words, such as a search and replace operation. Example 4-66 combines the second form of the for loop with arrays, covered earlier in this chapter, to do just that.

Example 4-66 *Parsing Through a File Word by Word*

```
! The contents of the file DataFile

[NetProg@server1 Scripts]$ cat DataFile

The for-do loop starts with the keyword for and an argument that takes up one of two

forms.
```

```
! The Script content
[NetProg@server1 Scripts]$ cat for-do-2.bash
#!/usr/bin/env bash
Content=( $(cat ./DataFile) )
for ((i=0 ; i<${#Content[@]} ; i++))
do
  echo "${Content[$i]}"
done


! Script execution result
[NetProg@server1 Scripts]$ ./for-do-2.bash
The
for-do
loop
starts
with
the
keyword
for
and
an
argument
that
takes
up
one
of
two
forms.
```

In Example 4-66, the file DataFile contains a line from this chapter as a sample test. The script uses command substitution to assign the output of the **cat** command to the array **Content**. Then a **for** loop sets **i=0** and iterates as long as **i** is less than the value of **${#Content[@]}**—which is the number of elements in the array **Content**. Through each iteration, the **echo** command prints the value of the element **Content[$i]** and the for loop increments the value of **i**. The value of **Content[1]** is the first word in the file, **Content[2]** is the second, and so forth, effectively printing each word in the file on a separate line. Think of all the processing possibilities, apart from printing out the words to the screen, that you can accomplish by mapping each word in a text file to an element in array.

The while-do Loop

Example 4-67 shows the general syntax of the while-do loop.

Example 4-67 *The while-do Loop General Syntax*

```
while condition
do
  command-block
done
```

The while-do loop is as simple as its syntax: While *condition* is true, execute the command(s) in *command-block*. Example 4-68 shows the same algorithm previously implemented using the for-do loop but this time using the while-do loop. The script requests an integer from a user and outputs to stdout all numbers, in order, from 0 to that number.

Example 4-68 *Printing a Sequence of Numbers by Using a while-do Loop*

```
[NetProg@server1 Scripts]$ cat while-do.bash
#!/usr/bin/env bash
read -p "Please enter any integer: " maxnumber
count=0
while [[ count -le maxnumber ]]
do
  echo $count
  (( count++ ))
done


[NetProg@server1 Scripts]$ ./while-do.bash
Please enter any integer: 11
0
1
2
3
4
5
6
7
8
9
10
11
[NetProg@server1 Scripts]$
```

The until-do Loop

Example 4-69 shows the general syntax of the until-do loop.

Example 4-69 *The until-do Loop General Syntax*

```
until condition
do
  command-block
done
```

Unlike the while-do loop, which executes the commands in *command-block* if and only if *condition* is true, the until-do loop actually breaks when *condition* is true; until this happens, the commands in *command-block* keep executing. Example 4-70 is a rewrite of the algorithm in Example 4-68 that outputs a sequence of numbers using the until-do loop.

Example 4-70 *Using an until-do Loop to Print Numbers from 0 to a Maximum Number*

```
[NetProg@server1 Scripts]$ cat until-do.bash
#!/usr/bin/env bash
read -p "Please enter any integer: " maxnumber
count=0
until [[ count -eq maxnumber ]]
do
  echo $count
  (( count++ ))
done
```

```
[NetProg@server1 Scripts]$ ./until-do.bash

Please enter any integer: 7

0

1

2

3

4

5

6

[NetProg@server1 Scripts]$
```

In the output in Example 4-70, notice that the numbers printed are 0 to 6, which is one less than the number that the user actually entered (**7**). The reason for this is that on the last loop iteration, the variable **count** is equal to 6. Its value is output to stdout by the **echo** command and then incremented to 7. On the following loop iteration, the variable **count** is actually equal to the value in the variable **maxnumber** (that is, 7); therefore, the commands in the command block are not executed, and the loop is terminated. In order to print all numbers up to and including the number entered by the user, the condition has to be changed to **[[count -gt maxnumber]]**, which evaluates to true only if **count** is *greater* than **maxnumber**.

Functions

Programming languages provide a programming construct called a *function*, which is a segment of code inside a script that is identified by a name, has a start and an end, can be passed arguments, and has a return status that is passed back to the calling script.

Why would you want to use functions in a script? Functions are highly efficient when a certain task has to be repeated more than once. When you use a function, the code for accomplishing a task is written once, and the function is called any number of times required, which saves you the effort of writing the same code more than once.

Example 4-71 shows the syntax for writing a function inside a script.

Example 4-71 *General Syntax of Functions*

```
function_name() {

  function_code

}
```

A function typically contains code that otherwise would exist in the body of the main script. Arguments are passed to the function the same way arguments are passed to the main script. The first argument value is then stored in **$1**, the second argument value in **$2**, and so forth. Example 4-72 shows a script that has a function named **print_values()** that prints to stdout all numbers between the first and last arguments passed to it, inclusive.

Example 4-72 *The **print_values()** Function*

```
[NetProg@server1 Scripts]$ cat main-script.bash

#!/usr/bin/env bash


# function code starts here inside the main script

print_values() {

  for (( i=$1 ; i<=$2 ; i++))

  do

    echo -n "$i "

  done

  echo

}

# function code ends here  - still inside the main script


read -p "Please enter the first number and last number: " START END
```

```
print_values $START $END
```

The last two lines of the script request the user to input two values, which are assigned to the variables **START** and **END**. The values of these two variables are then passed to the function **print_values()** as arguments. The function uses the value of the first argument, **$1**, as the starting value of the variable **i** in the **for** loop, and the second argument, **$2**, to evaluate the condition. When the variable **i** is equal to or greater than the second argument, the **for** loop exits. The body of the loop simply prints out the value of the variable **i** at every loop iteration. The loop is evaluated in Example 4-73 using the values 1 and 10.

Example 4-73 *Running a Script Using 1 and 10 as the Two Arguments for the Function* ***print_values()***

```
[NetProg@server1 Scripts]$ ./main-script.bash

Please enter the first number and last number: 1 10

1 2 3 4 5 6 7 8 9 10
```

When a function completes execution, it returns a value, called the *exit status*, to the calling script. By default, if the function executes successfully and without errors, the value of the exit status is zero. If some error occurs, the exit status returned is a nonzero value. The returned value is stored in **$?**. In Example 4-74, the command **echo $?** is added to the script, and the execution result shows that the value of **$?** is **0**.

Example 4-74 *Printing the Exit Status of the Function* ***print_values()*** *by Using* ***$?***

```
[NetProg@server1 Scripts]$ cat main-script-1.bash

#!/usr/bin/env bash

print_values() {

  for (( i=$1 ; i<=$2 ; i++))

  do

    echo -n "$i "

  done

echo

}


read -p "Please enter the first number and last number: " START END


print_values $START $END


echo $? # Printing out the exit status


[NetProg@server1 Scripts]$ ./main-script-1.bash


Please enter the first number and last number: 1 10

1 2 3 4 5 6 7 8 9 10

0
```

You can change the exit status by using the command **return** *status_value*. In more complex scripts, you can use a conditional statement to return one of many possible exit status values to the script calling the function. Each value has a different meaning beyond just the simple success/fail of the function.

In Example 4-75, the script from Example 4-74 is updated with **return 100** so that **echo $?** prints out **100** instead of **0**.

Example 4-75 *Printing the Return Status of the Function* ***print_values()*** *by Using* ***$?***

```
[NetProg@server1 Scripts]$ cat main-script-2.bash

#!/usr/bin/env bash

print_values() {

  for (( i=$1 ; i<=$2 ; i++))

  do

    echo -n "$i "

  done
```

```
echo

return 100

}


read -p "Please enter the first number and last number: " START END


print_values $START $END


echo $? # Returning a custom exit status instead of 0


[NetProg@server1 Scripts]$ ./main-script-2.bash

Please enter the first number and last number: 1 10

1 2 3 4 5 6 7 8 9 10

100
```

Variables inside a function block are, by default, global variables. A *global variable* is a variable that exists throughout the whole life of a script. A global variable retains its value and can be used anywhere in the script. The code block in which a variable is visible is called the *variable scope*. Alternatively, a local variable is visible only inside one specific code block, such as a function. In Example 4-75, the variable **i** used in the **for** loop is, by default, a global variable. In Example 4-76, the value of **i** is printed outside the function block, and as you can see, its value, **11**, is intact.

Example 4-76 *Printing the Value of The Global Variable **i** Outside The Function Where Is It Used*

```
[NetProg@server1 Scripts]$ ./main-script-3.bash

#!/usr/bin/env bash

print_values() {

  for (( i=$1 ; i<=$2 ; i++))

  do

    echo -n "$i "

  done

echo

}


read -p "Please enter the first number and last number: " START END


print_values $START $END


echo $i


[NetProg@server1 Scripts]$ ./main-script-3.bash

Please enter the first number and last number: 1 10

1 2 3 4 5 6 7 8 9 10

11
```

You can change a variable from global to local by using the **local** keyword to declare the variable. In Example 4-77, the variable **i** is declared as a local variable inside the function. When the function exits, **i** becomes an undeclared variable, and **echo $i** returns a null value.

Example 4-77 *Attempting to Print the Value of The Local Variable **i** Outside The Function Where Is It Declared*

```
[NetProg@server1 Scripts]$ cat main-script-4.bash

#!/usr/bin/env bash

print_values() {
```

```
local i
  for (( i=$1 ; i<=$2 ; i++))
  do
    echo -n "$i "
  done
echo
}

read -p "Please enter the first number and last number: " START END

print_values $START $END

echo $i
```

```
[NetProg@server1 Scripts]$ cat main-script-4.bash
Please enter the first number and last number: 1 10
1 2 3 4 5 6 7 8 9 10

[NetProg@server1 Scripts]$
```

Bash functions do not have the capability to return any values to the calling script apart from the exit status. Therefore, in order to transfer values from functions back to the calling script, you need to do one of three things: Either use global variables, which is *not* a recommended practice, or pipe the output produced by the function (such as the output of the **echo** command in the previous few examples) or alternatively, you can use the syntax *variable*=**$(***function_name function_args***)** to assign the output produced by a function to a variable.

Expect

Expect is a programming language that streamlines and automates interactive operations. A very popular example of an interactive operation is providing a device with the required login credentials. This section does not cover the Expect language in detail, but it covers a number of commands that, when integrated with a Bash script, make interactive operations much easier to code. Four commonly used commands that are used to achieve this functionality are covered in this section: **spawn**, **expect**, **send**, and **interact**.

Much like a Bash script, an Expect script is run using an interpreter. By default, the Expect interpreter is not installed on CentOS 8 and can be installed by using the command **yum install expect** in admin mode. To invoke the Expect interpreter, you need to begin an Expect script with **#!usr/bin/env expect** and then provide the body of the script.

You use the **spawn** command to run another script, process, or utility. For example, **spawn ./somescript.bash** simply runs the script named **somescript.bash**. The command **spawn ssh 10.0.0.1** runs the **ssh** command to connect to IP address 10.0.0.1.

Examples 4-78 and 4-79 show how the **expect** and **send** commands work in an Expect script to "speak" to a Bash script. In Example 4-78, the Bash script expect-1.bash is used to read the first and last names of the user using the **read** command and then print out the user's full name by using the **echo** command. Earlier in this chapter, you learned that such input is typically provided by the user through stdin (the keyboard).

Example 4-78 *A Bash Script expect-1.bash Requesting Input from the User*

```
#!/usr/bin/env bash
read -p "Enter your firstname: " firstname
read -p "Enter your lastname: " lastname
echo "Your full name is $firstname $lastname"
```

However, in this case, there is no user. The user functionality will be emulated using an Expect script. In Example 4-79, an Expect script executes the Bash script shown in Example 4-78 by using the **spawn** command, and then *expects* the strings printed out by the Bash script using the **expect** command, and finally, sends the responses back to the Bash script by using the **send** command.

Example 4-79 *An Expect Script "Speaking" with the Bash Script from the Example 4-78*

```
#!/usr/bin/env expect
spawn ./expect-1.bash
expect "Enter your firstname: "
```

```
send "Khaled\r"

expect "Enter your lastname: "

send "Abuelenain\r"

expect eof
```

After it executes the Bash script **./expect-1.bash**, the Expect script waits for the string **"Enter your firstname: "** from the spawned (that is, executed) script, and when it receives it, it sends the string **"Khaled\r"**, which acts as input to the Bash script. Note that the **\r** escape character, which represents a carriage return, is the equivalent of pressing the Enter key. The same goes for the following **expect** and **send** commands. Finally, the last statement, **expect eof**, lets the Bash script run until the end of file (that is, the end of the Bash script) has been encountered, at which point the Expect script is complete, and control is passed back to the shell. Example 4-80 shows the result of running the Expect script.

Example 4-80 *The Result of Running the Expect Script*

```
[NetProg@server1 Scripts]$ ./expect-1.exp

spawn ./expect-1.bash

Enter your firstname: Khaled

Enter your lastname: Abuelenain

[NetProg@server1 Scripts]$
```

The **interact** command, on the other hand, lets the user interact with the spawned script. In Example 4-81, the Expect script from Example 4-80 has been amended to have the user enter his or her last name instead of sending a preset last name saved in the script.

Example 4-81 *Using the **interact** Command to Allow User Input to the Expect Script*

```
#!/usr/bin/env expect

spawn ./expect-1.bash

expect "Enter your firstname: "

send "Khaled\r"

expect "Enter your lastname: "

interact
```

Example 4-82 shows the result of running the script. Note that the last name **SomeOtherName** was entered manually from the keyboard.

Example 4-82 *Running the Script Using the **interact** Command and Entering the Last Name Manually from the Keyboard*

```
[NetProg@server1 Scripts]$ ./expect-1.exp

spawn ./expect-1.bash

Enter your firstname: Khaled

Enter your lastname: SomeOtherName

Your full name is Khaled SomeOtherName

[NetProg@server1 Scripts]$
```

Example 4-83 puts the Expect language to good use and shows a script that logs in to a public route server via Telnet, sends the username and password, and then issues the command **show ip route connected**. The script then hands over control to the user by using the **interact** command. However, if no input is received from the user within 10 seconds, the Expect script sends the command **exit** to the route server, ending the Telnet session.

Example 4-83 *Logging In to a Public Server by Using **expect** and Collecting the Output of the Command **show ip route connected***

```
#!/usr/bin/env expect

log_file ip_route_connected

spawn telnet route-server.opentransit.net

expect "Username: "

send "rview\r"

expect "Password: "

send "Rview\r"

expect "OAKRS1#"

send "show ip route connected\r"

interact timeout 10 { send "exit\r" }
```

Example 4-84 shows the output that results from running the script shown in Example 4-83. The script sends the highlighted sections and the hidden password automatically, without user interaction.

Example 4-84 *The Result of Executing the Expect Script from Example 4-83*

```
[NetProg@server1 Scripts]$ ./expect-rview.exp

spawn telnet route-server.opentransit.net

Trying 204.59.3.38...

Connected to route-server.opentransit.net.

Escape character is '^]'.

C


&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

         route-server.opentransit.net -- & Opentransit

                        IPv4/IPv6 views


This router keeps peering sessions with all the Opentransit Backbone Routers,

throughout the Opentransit IP Backbone as follows:


[IPv4/IPv6 view]


IPv4: 193.251.245.1            IPv6: 2001:688:0:1::158 Dallas

IPv4: 193.251.245.3            IPv6: 2001:688:0:1::1c  Los Angeles

IPv4: 193.251.245.7            IPv6: 2001:688:0:1::55  London

IPV4: 193.251.245.9            IPv6: 2001:688:0:1::41  Palo Alto

IPv4: 193.251.245.10           IPv6: 2001:688:0:1::4e  Paris

IPv4: 193.251.245.16           IPv6: 2001:688:0:1::168 New York

IPv4: 193.251.245.19           IPv6: 2001:688:0:1::4b  Barcelona

IPv4: 193.251.245.28           IPv6: 2001:688:0:1::8   Frankfurt

IPv4: 193.251.245.37           IPv6: 2001:688:0:1::    Frankfurt

IPv4: 193.251.245.49           IPv6: 2001:688:0:1::19  London

IPv4: 193.251.245.53           IPv6: 2001:688:0:1::1e  Chicago

IPv4: 193.251.245.57           IPv6: 2001:688:0:1::45  Miami

IPv4: 193.251.245.66           IPv6: 2001:688:0:1::44  Geneva

IPV4: 193.251.245.69           IPV6: 2001:688:0:1::56  Singapore

IPv4: 193.251.245.76           IPv6: 2001:688:0:1::d   New York

IPv4: 193.251.245.78           IPv6: 2001:688:0:1::22  Madrid

IPv4: 193.251.245.81           IPv6: 2001:688:0:1::4   Barcelona

IPv4: 193.251.245.88           IPv6: 2001:688:0:1::f   London

IPv4: 193.251.254.92           IPv6: 2001:688:0:1::24  Paris

IPv4: 193.251.245.96           IPv6: 2001:688:0:1::3E  Brussels

IPv4: 193.251.245.134          IPv6: 2001:688:0:1::4a  Zurich

IPv4: 193.251.245.147          IPv6: 2001:688:0:1::2A  HongKong

IPv4: 193.251.245.163          IPv6: 2001:688:0:1::18  New York

IPv4: 193.251.245.170          IPv6: 2001:688:0:1::2f  Madrid

IPv4: 193.251.245.181          IPv6: 2001:688:0:1::3C  Singapore

IPv4: 193.251.245.196          IPv6: 2001:688:0:1::57  Frankfurt
```

Chapter 5. Python Fundamentals

Computer programming has gained a lot of popularity in past three decades, but what exactly is programming? Computers were designed to take sets of instructions from humans and perform those tasks. For instance, users can instruct a computer to perform a series of calculations on the numbers that can be defined statically or via user input. A computer is instructed to perform calculations via programming. Thus, *programming* can be defined as the instructing a computer to perform various tasks. The complexities of the tasks can vary based on what a user is trying to achieve. A task may be as simple as performing some basic mathematical operations such as addition or subtraction between two numbers or as complex as rendering and clearing the noise in an image.

If programming is instructing a computer to perform various tasks, what is a program? A computer software *program* is a specific set of ordered instructions for a computer to perform. To conceptualize a software program, think of the process of preparing tea. A person has to go through a set of steps (instructions) in order to prepare tea. The steps may vary from person to person based on the kind of tea being made, but some steps always remain the same. For instance, the following steps can be followed to prepare tea:

1. Boil the water.

2. Add tea leaves and the hot water to a tea pot and wait 3–4 minutes.

3. Pour the brewed tea into cups.

4. Add sugar and milk to each cup, as needed.

Steps 1 to 3 are common to most styles of preparing tea, but only a person who prefers sweetened tea or milk tea needs to follow step 4. As you can see, onto the steps for preparing tea need to be followed in a particular order. The same goes for a computer program. Let us take a simple example of reversing a string, a task that can be done in different ways. Example 5-1 shows a function for calculating the sum of the first *n* integers. In this example, the integer *n* is passed as a parameter to the **sum()** function. This function iterates through all the integers from 1 to *n*, starting from 0 and adding the integer at the *i*th position to the sum of 1 up to *i* – 1 integers. The iteration of integers from 1 to *n* is done via a **for** loop. (**for** loops are explained later in this chapter.)

Example 5-1 *Python Function to Return the Sum of the First* n *Integers*

```
def sum(n: int):

    sum = 0

    for i in range (1, n+1):

        sum = sum + i

    return sum
```

Why do you need to perform this task programmatically when a human can do it by hand quickly and also apply better mathematical formulas to get to the result? Keep in mind that Example 5-1 demonstrates a function that calculates the sum of the first *n* integers. This task is easy when *n* is a smaller integer, but it gets more time-consuming and complex when the value of *n* is higher, such as the integer value 998,786. With a large integer like 998,786, performing the task manually is definitely going to be time-consuming and involve a substantial chance of human error. Performing tasks programmatically reduces or even eliminate the chance of human error, especially with tasks that are repetitive and complex. A program essentially needs only sound logic and an optimized series of steps.

Scripting Languages Versus Programming Languages

Before we dive into network programmability, it is important to understand the difference between scripting languages and programming languages. Engineers are often confused about the difference between scripting languages and programming languages. Simply put, *programming languages* are languages that need a compiler to convert their code into native machine code, whereas *scripting languages* are interpreted instead of being compiled into machine code. Because the code is compiled into machine code, programming languages run faster compared to scripting languages.

Scripting is a subset of programming that is primarily focused on automating the execution of tasks. These tasks can be some machine-level tasks (system tasks) or user-defined tasks that are usually developed through programming logic. A task can be simply defined as a procedure that is nothing but a set of steps; in order for a computer to perform a task, a procedure must be defined. In the earlier example of preparing a cup of tea, the task was to prepare the tea, and the steps laid out the procedure. As stated previously, scripting languages are interpreted, which means they require an interpreter to convert their code into native machine code. Scripting languages can also be used for the purpose of extracting data from a data set. Some common examples of scripting languages are JavaScript, Perl, and Python.

Let's look at an example of web application development. In end-to-end web application development, there are primarily three layers for which development occurs:

• **Front end**: The front end involves everything a user sees in a browser and is primarily a combination of Hypertext Markup Language (HTML), CSS, and JavaScript.

• **Middleware**: The middleware or middle layer is where you define the logic or create APIs to allow the front end to interact with the back end. For instance, a web application to gather registration information from users as input would be sent to the middleware, and the middleware would make a call to the database or the back end to store the data.

• **Back end**: The back end either consists of server-side scripts or databases. The server-side scripts process the data and return a response based on the processed data to the front-end clients.

Figure 5-1 illustrates the layers in a typical web application.

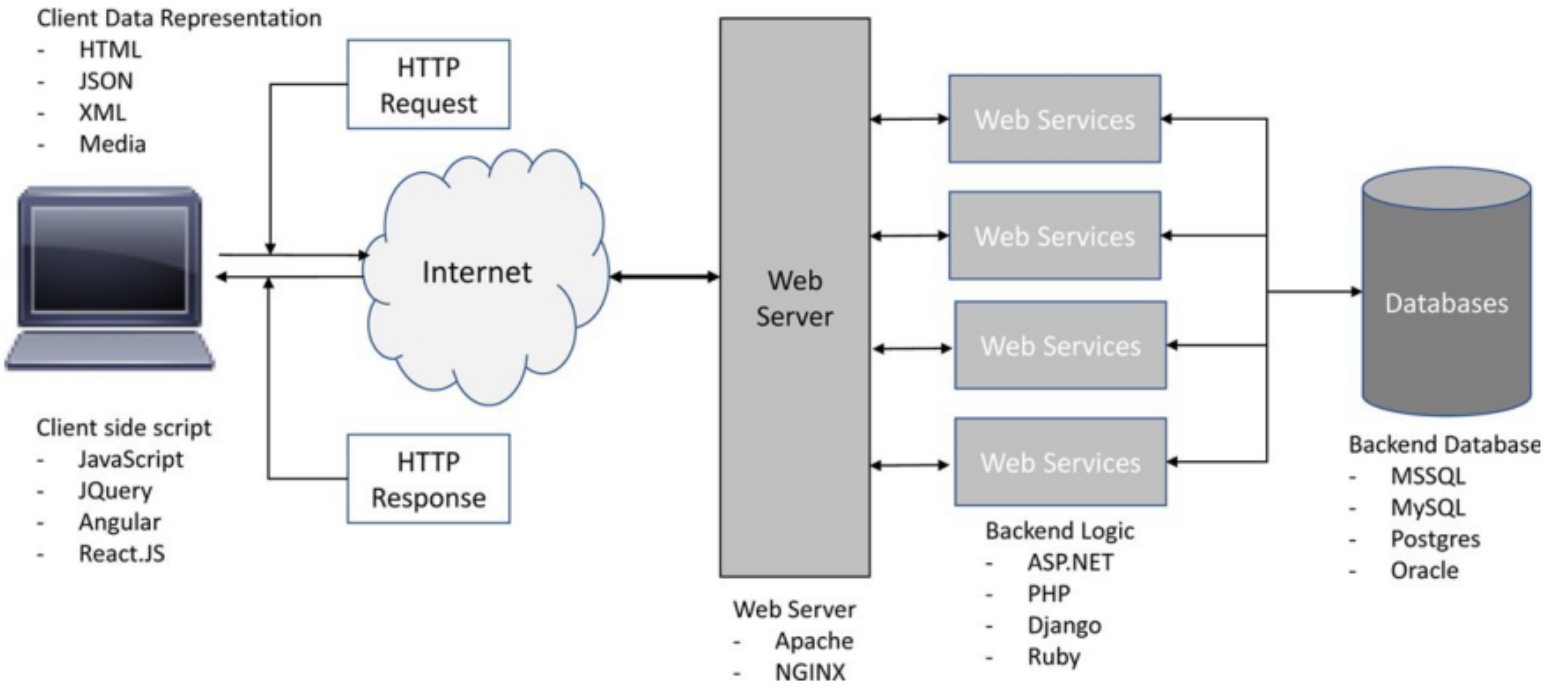


Figure 5-1 *Typical Web Application Layers*

The use of scripting languages such as JavaScript in the front end helps in performing validations on user input, whereas back-end programming logic tells the application what it needs to do with the data it received from the front-end user interface. The validation of front-end input data helps ensure that the back-end logic or database does not return an error due to an invalid data type or data format. The automation of front-end validation through scripting saves a lot of time and resources during data processing. If there were no validation happening in the front end, a software developer would have to write the validator and its logic in the back-end code, which would put more load on the server processing the data. If the data were already validated before being received by the back-end application, the back-end code would just focus on performing the task to store or update the data in the database.

In contrast to scripting languages, programming languages are compiled into a more compact standalone form that is machine consumable. A programming language consists of three primary components:

Pseudocode (that is, human-readable code)

Compiler

Machine-readable compiled code

Pseudocode allows programmers to write their logic in an easily readable and understandable format. The pseudocode is then compiled by the compiler into a machine-readable format. A computer system understands or processes data in binary or assembly code. Most compilers compile the code into assembly code, which can be executed. For example, a typical C program goes through a series of transformations and optimizations before being executed in hardware. The source code file is processed through the C compiler (for instance, gcc) and transforms the pseudocode into an ASCII-based assembly file, which is then converted by an assembler into an object file. The object file is then processed via a linker and converted into an executable file, which loads the program into the memory of the system when executed. In this process, the end result is an executable file (.exe), which is machine consumable. Figure 5-2 illustrates the process of compiling C language pseudocode into a machine-understandable executable file.

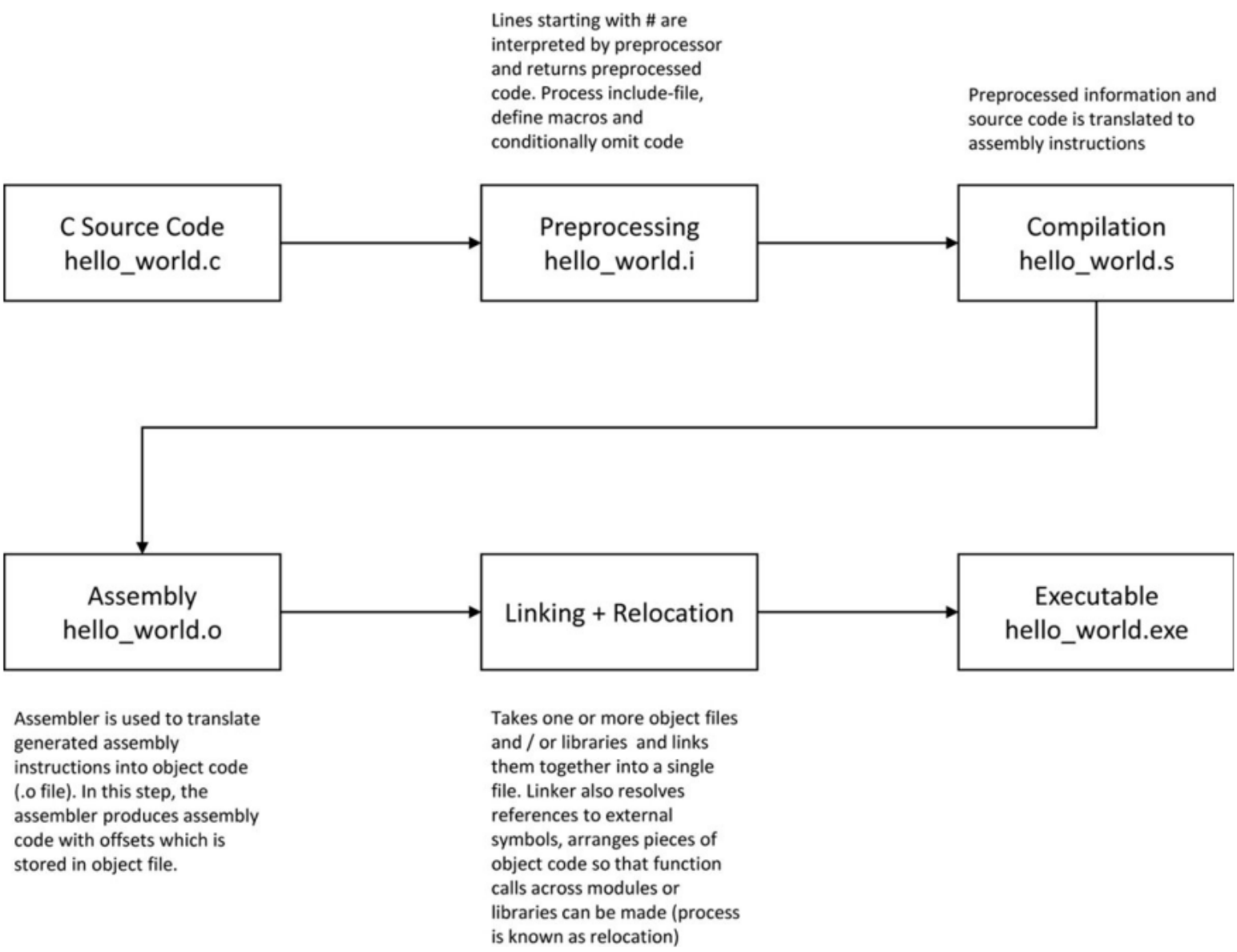


Figure 5-2 *Compiling C into an Executable File*

With programming languages, the complete pseudocode is compiled into native machine code at once. So, when a program is executed, memory allocation occurs for the whole program, regardless of whether a function is being called within the code. This is one of the reasons a scripting language may make more efficient use of system resources during execution, as the memory allocation happens only for the function calls that are made by the user. Some examples of well-known programming languages are C, C++, and Java. Most programming languages can also be used for accessing low-level hardware details, which is not directly possible via scripting languages. The different capabilities of the scripting and programming languages easily help developers identify their respective use cases.

Network Programmability

Now that you understand the need for programming, you might wonder how programming helps with network devices. For many years, network infrastructure components were manually configured, using laborious, monotonous techniques (such as using the CLI) that are prone to human error. Over the years, the demand for network and service availability across the network has changed, as has the infrastructure. Organizations now demand event-driven reactive action to mitigate network outages in order to minimize downtime. Manually performing tasks to maintain high availability of network services is neither scalable nor cost-effective. Automation and programmable network devices can cost-effectively provide the network availability and scalability needed in today's networks.

Note

When we talk about network programmability and its benefits, it's important to understand that we are automating the management plane and not changing the control plane or data plane behavior.

Network programmability refers to the ability to alter the way in which network devices were traditionally managed and control flow via the software that is operated independently from the network hardware. Using network programmability, network engineers can easily reconfigure the network infrastructure at scale. Providing programmable open interfaces allows organizations and service providers to keep pace with the rapid demands of business services.

Network programmability may be used, for example, in scenarios where a network engineer needs to automate the configuration of a large number of devices or validate the software or firmware version for device upgrades or dynamic provisioning of services such as adding a new MPLS VPN customer or enable traffic shaping functionality across the network. Programmable networks have many benefits, including the following:

• **Reduced operational expenses (opex)**: According to industry estimates, the major cost in any given enterprise or data center is the cost of technical personnel required to manage the infrastructure. Hardware expenses have been declining over the years, but the cost to manage the infrastructure has been growing steadily. Automating infrastructure management and service deployment reduces the time an IT professional needs to spend on a day-to-day basis and also the number of IT professionals who need to be engaged. Automation therefore reduces opex and also saves a lot of time and reduces human error.

• **Customization**: Network programmability makes it possible to customize the management plane of network devices based on an

Example 5-2 shows the execution times for the two string-reversing algorithms mentioned previously (written in Python code). The GNU **time** program displays three values:

• **real**: The total execution time taken by the command or program

• **user**: The total execution time taken by the command or program in user mode

• **sys**: The total execution time taken by the command or program in kernel mode

You can also use the **-v** option to display more detailed (verbose) information about the execution.

Example 5-2 *Execution Times of Different Algorithms*

```
root@rnd-srvr:~/python# /usr/bin/time -f "\t%E real,\t%U user,\t%S sys"
./first.py
        0:00.04 real,   0.03 user,      0.00 sys
```

```
root@rnd-srvr:~/python# /usr/bin/time -f "\t%E real,\t%U user,\t%S sys"
./second.py
        0:00.04 real,   0.04 user,      0.00 sys
```

To build algorithms in any programming language, programmers have to define and use various data structures and data types to perform actions on the particular kind of data. Unless a program knows what kind of data it is dealing with, it cannot perform the correct operation on the data, thus making the program either useless or error prone. We discuss data structures and data types further in the next section.

Python Fundamentals

Python is an easy-to-learn, easy-to-use, and powerful programming language that allows easy integration with other programming languages. In fact, the first words of the official Python tutorial were "Python is an easy to learn, powerful programming language." Python, which was created in 1991, was named for the British comedy group Monty Python. Over the years since then, so much development has gone into Python that there are now more than 190,000 Python packages, and many of them are incredibly useful. Python is being used to create web applications, artificial intelligence applications, data science applications, and scientific applications.

The following are some of the core feature that make Python one of the most powerful languages:

• **Easy to learn**: Even if you are a complete beginner and have not learned any programming language before, with Python, you can hit the ground running once you have learned the basics, which are fairly simple to understand. You just need to build the logic, know the basics, and get familiar with Python syntax.

• **Easy to integrate**: Python can easily integrate with third-party applications and also with code written in different programming languages. This integration makes it possible to reuse existing code (even code written in different languages) and helps reduce application development time.

• **Platform independent**: With Python, users don't have to write code based on the platform. Using Python, developers create cross-platform applications that can be seamlessly used without the burden or recompiling the code between platforms.

• **Powerful**: Thanks to the massive repository of available Python packages, you can import the necessary package and perform extremely complex data analysis in minutes—or even seconds. The many available packages create endless possibilities for what you can do with your data; this is one reason Python is popular in data science and machine learning.

There are two well-known versions of Python: Python 2 and Python 3. Python 2 reached end-of-life on January 1, 2020. Multiple branches of Python 3 are available to developers for download, including 3.6, 3.7, 3.8, and 3.9. (At the time of writing, the latest versions of Python 3 are 3.6.12, 3.7.9, 3.8.6, and 3.9.1.) Many Python 2 applications are still running, but because it is end-of-life, this chapter focuses on Python 3. However, most of the content in this chapter is still be relevant to Python 2.

Python Installation

You can download and install Python at www.python.org/downloads. You can download the relevant installer for your operating system and then follow the installation steps in order to install Python. The distribution that you download from the www.python.org website is the standard distribution; that is, it has the standard package that comes with Python by default. To install additional packages, you have to use the Python package manager named **pip**, which is based on Python 2, or **pip3**, which is based on Python 3. The **pip** or **pip3** package manager makes it a lot easier to install or uninstall or even roll back packages in the event of any instability. You can install packages by using the command **pip install** *package_name*. Once this command is executed, the specified package is downloaded, along with its dependencies. Example 5-3 illustrates the installation of a **pip3** package manager on CentOS using the command **yum install -y python3** and the **numpy** package using the **pip3** package manager.

Example 5-3 ***pip3*** *Installation and Package Installation*

```
[root@rnd-srvr ~]# yum install -y python3
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: repos-lax.psychz.net
 * extras: mirror.sjc02.svwh.net
 * updates: mirror.sjc02.svwh.net
base                                                    | 3.6 kB     00:00
docker-ce-stable                                        | 3.5 kB     00:00
extras                                                  | 2.9 kB     00:00
updates                                                 | 2.9 kB     00:00
Resolving Dependencies
--> Running transaction check
```

```
[root@rnd-srvr python]# ls
first.py  __pycache__
[root@rnd-srvr python]# cd __pycache__/
[root@rnd-srvr __pycache__]# ls
first.cpython-36.pyc

[root@rnd-srvr __pycache__]# cat first.cpython-36.pyc
3
Ăb]5,,┐@s
        edĖdS)┐z*Hello, This is my first Python
applicationN) /print©r┐r┐first.py<module>s

[root@rnd-srvr __pycache__]# python3 first.cpython-36.pyc
Hello, This is my first Python application
```

Note

If you receive a .pyc file (a bytecode file) and you want to decode the bytecode to pseudocode, you can install the package named **uncompyle6** by using the command **pip3 install uncompyle6**. Once it is installed, you can use the command **uncompyle6 -o** *your_filename*.**pyc** to create a .py file from the .pyc file.

Each time the Python interpreted program is executed, the interpreter converts the bytecode into machine code and pulls in the runtime libraries. This can be viewed by using the **-v** or **-verbose** option with the Python interpreted program. Example 5-7 illustrates the use of the **-verbose** option when running the first.py Python program. Notice that, initially, multiple system or runtime libraries are imported. When the import process is complete, the Python runtime environment moves to interpreter mode and executes the code. When the code execution is completed, the runtime cleanup process is initiated to dispose of or destroy all the resources from the memory.

Example 5-7 *Python Code Execution*

```
[root@rnd-srvr python]# python3 -v first.py
import _frozen_importlib # frozen
import _imp # builtin
import sys # builtin
import '_warnings' # <class '_frozen_importlib.BuiltinImporter'>
import '_thread' # <class '_frozen_importlib.BuiltinImporter'>
import '_weakref' # <class '_frozen_importlib.BuiltinImporter'>
import '_frozen_importlib_external' # <class '_frozen_importlib.FrozenImporter'>
import '_io' # <class '_frozen_importlib.BuiltinImporter'>
import 'marshal' # <class '_frozen_importlib.BuiltinImporter'>
import 'posix' # <class '_frozen_importlib.BuiltinImporter'>
import _thread # previously loaded ('_thread')
import '_thread' # <class '_frozen_importlib.BuiltinImporter'>
import _weakref # previously loaded ('_weakref')
import '_weakref' # <class '_frozen_importlib.BuiltinImporter'>
# installing zipimport hook
import 'zipimport' # <class '_frozen_importlib.BuiltinImporter'>
# installed zipimport hook
# /usr/lib/python3.6/encodings/__pycache__/__init__.cpython-36.pyc matches
/usr/lib/python3.6/encodings/__init__.py
# code object from '/usr/lib/python3.6/encodings/__pycache__/__init__.cpython-
36.pyc'
# /usr/lib/python3.6/__pycache__/codecs.cpython-36.pyc matches
/usr/lib/python3.6/codecs.py
# code object from '/usr/lib/python3.6/__pycache__/codecs.cpython-36.pyc'
import '_codecs' # <class '_frozen_importlib.BuiltinImporter'>
import 'codecs' # <_frozen_importlib_external.SourceFileLoader object at
0x7f2d4d2b7358>
# /usr/lib/python3.6/encodings/__pycache__/aliases.cpython-36.pyc matches
/usr/lib/python3.6/encodings/aliases.py
# code object from '/usr/lib/python3.6/encodings/  pycache  /aliases.cpython-
```

```
# destroy abc
# destroy errno
# destroy posixpath
# cleanup[3] wiping stat
# cleanup[3] wiping _stat
# destroy _stat
# cleanup[3] wiping genericpath
# cleanup[3] wiping importlib._bootstrap
# cleanup[3] wiping sys
# cleanup[3] wiping builtins
# destroy os
# destroy stat
# destroy genericpath
```

Most Python code testing and verification is done in Python interpreter shell because it is an easy and excellent way to see how Python code works. A user can type the code right into the interpreter shell and get the results right away. Example 5-8 shows some basic examples of dealing with numeric and string data in the Python interpreter shell.

Example 5-8 *Dealing with Numeric and String Data in the Python Shell*

```
[root@rnd-srvr python]# python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 10
10
>>> 10 + 10
20


>>> a = 'Python'
>>> a
Python


>>> print(a)
Python


>>> exit()
root@rnd-srvr:~#
```

Python Data Types

Python data types are the basic code building blocks that are used to build larger blocks of code. Operations can be performed on the data based on the data type. For instance, for an integer data type, a user can perform an operation such as addition or multiplication. Following are the built-in data types for Python along with their classes:

- Numeric data types:
- int
- float
- complex
- fractions
- decimal
- String
- str
- Boolean
- bool
- Lists
- list
- Dictionaries

```
>>> c[3:6]

'lo9'

>>> c[1:6:2]

'el9'
```

Slicing can also be performed using negative indexes. For a slicing operation, the starting index should be 0 or some other number that is lower than the stop index, and the stop index cannot be a number lower than -1. Example 5-18 demonstrates slicing using negative indexes. In this example, the starting index -6 is equal to the positive index 1, and the stop index -1 is equal to the index value 6 and step value 2, which means choosing every alternate character in the string.

Example 5-18 *Slicing Using Negative Indexes*

```
>>> c

'Hello99'

>>> c[3:6]

'lo9'

>>> c[-4:-1]

'lo9'

>>> c[1:6:2]

'el9'

>>> c[-6:-1:2]

'el9'
```

Some predefined functions, including the following, can be called on string literals:

• **str.strip()**: Strips or removes any whitespace at the beginning and at the end of the string.

• **str.lower()**: Converts the whole string to lowercase.

• **str.upper()**: Converts the whole string to uppercase.

• **str.replace()**: Takes two parameters: the substring to replace and the string to be replaced with. The **replace()** function finds the substring specified in the parameter and replaces it with the string specified as the second parameter.

Example 5-19 illustrates the use of these functions on a string assigned to the variable **a**.

Example 5-19 *Using String Functions*

```
>>> a = ' Hello, This is Python '

>>> a.strip()

'Hello, This is Python'

>>> a.lower()

' hello, this is python '

>>> a.upper()

' HELLO, THIS IS PYTHON '

>>> a.replace('Py', 'Hacka')

' Hello, This is Hackathon '
```

Along with some of the functions shown in Example 5-19, the **str** class has a **format()** function that allows multiple substitutions and formatting of the string. The **format()** function takes input variables as parameters, and these variables can be substituted in the string at places where placeholders appear. A placeholder is defined by a pair of curly braces ({}), and curly braces can also be placed with the index of the variables in the **format()** function, as shown in Example 5-20. Example 5-20 demonstrates multiple ways to use the **str.format()** function with the string. Note that the input value in the **format()** function can be an integer, a floating point number, a string, or a character that is either directly passed or passed via a variable.

Example 5-20 *Formatting String Data*

```
>>> age = 33

>>> txt = "My name is Andrew, my age is {}".format(age)

>>> txt

'My name is Andrew, my age is 33'


>>> txt = "My name is Andrew, my age is {}"

>>> print(txt.format(age))

My name is Andrew, my age is 33


>>> name = ' Andrew '


>>> txt = 'My name is {}, my age is {}'.format(name,age)

>>> txt
```

```
'My name is Andrew, my age is 33'


>>> txt = 'My name is {}, my age is {}'
>>> print(txt.format(name,age))
My name is Andrew, my age is 33


>>> txt = 'My name is {1}, my age is {0}'
>>> print(txt.format(age,name))
My name is Andrew, my age is 33
```

Operators

So far in this chapter, various operations have been performed on integers and string literals. Operators make different operations possible, and Python supports several types of operators:

- Arithmetic operators
- Bitwise operators
- Assigning operators
- Logical operators
- Identity operators
- Membership operators

Table 5-1 provides a list of arithmetic and bitwise operators.

Table 5-1 *Arithmetic and Bitwise Operators*

Operator	Description	Example
+	Arithmetic operator for addition	**a + b**
-	Arithmetic operator for subtraction	**a - b**
*	Arithmetic operator for multiplication	**a * b**
/	Arithmetic operator for division	**a / b**
%	Arithmetic operator for modulo	**a % b**
**	Arithmetic operator for exponentiation	**a ** b**
& (AND)	Bitwise operator that sets each bit to 1 if both bits are 1	**a & b**
| (OR)	Bitwise operator that sets each bit to 1 if one of the two bits is 1	**a | b**
^ (XOR)	Bitwise operator that sets each bit to 1 if only one of the bits is 1	**a ^ b**
~ (NOT)	Bitwise operator that inverts all the bits (that is, sets 0 to 1 and 1 to 0)	**a ~ b**
<< (left shift)	Bitwise operator that shifts left by pushing 0s in from the right and letting the leftmost bits fall off	**a << b**
>> (signed right shift)	Bitwise operator that shifts right by pushing copies of the leftmost bit in from the left and letting the rightmost bits fall off	**a >> b**

Understanding the operators shown in Table 5-1 is important for Python programming as these operators help build the logic and are also used for defining data structures. You have seen most of the arithmetic operators listed in Table 5-1 earlier in this chapter. We now examine the exponential operator and other bitwise operators. The exponential operator is a programmatical representation of $(a)^b$ or the **(a * a * a … * a)** mathematical operation, where **a** is multiplied by itself **b** times.

Python bitwise operators treat numbers as strings of bits. In other words, the numbers are represented in binary format—that is, in two's complement binary—and the bitwise operations are performed on those binary numbers. The two's complement binary format is similar to binary representation of positive numbers but slightly different for negative numbers. A negative number, such as **(-x)**, in bit pattern is written as the value **(x - 1)** and then complemented; that is, the bits are switched from 1 to 0 and 0 to 1. Table 5-2 provides some examples of binary representation of both positive and negative numbers.

Table 5-2 *Representing Positive and Negative Numbers in Two's Complement*

assignment operator works in conjunction with other arithmetic or bitwise operators. First, the arithmetic or bitwise operation is applied on the variable, and then the result is assigned to the variable. Table 5-3 shows a variety of combinational assignments.

Table 5-3 *Assignment Operators*

Operator	Example	Same As
=	x = 10	x = 10
+=	x += 10	x = x + 10
-=	x -= 10	x = x - 10
*=	x *= 10	x = x * 10
/=	x /= 10	x = x / 10
%=	x %= 10	x = x % 10
**=	x **= 10	x = x ** 10
&=	x &= 10	x = x & 10
|=	x |= 10	x = x | 10
^=	x ^= 10	x = x ^ 10
>>=	x >>= 3	x = x >> 3
<<=	x <<= 5	x = x << 5

Python and several other programming languages also have comparison operators. A comparison operator is used with conditional statements such as **if-else**. A comparison operator is used to perform a comparison between two values or variables; when used with statements such as **if-else**, it allows you to perform certain steps based on the result.

Comparison operators are not used to compare objects. To compare objects in Python, you use identity operators. The identity operators are **is**, which returns **True** if the objects are the same (that is, if the two objects point to the same address location in memory), and **is not**, which returns **True** if the objects are not same or if they point to different address locations in the memory. Both the comparison and identity operators return Boolean values—that is, either **True** or **False**.

If there are multiple conditions to be validated within a statement or loop, logical operators can be used along with comparison or identity operators. A logical operator returns a Boolean value by comparing the result from the comparison operators or identity operators between two values or objects. There are three logical operators:

• **and**: The **and** operator returns **True** if the result from each of the comparison statements is **True**. The Python interpreter first validates whether the result of the first statement is **True**, and if it is, then it progresses on to validating the second statement.

• **or**: The **or** operator returns **True** if the result from the comparison statements is **True**. If the result of the first statement is **True**, the Python interpreter doesn't even validate the second operator because the comparison has already resulted in **True**.

• **not**: The **not** operator reverses the result of the comparison between values or objects.

Table 5-4 describes the comparison operators, identity operators, and logical operators and provides examples of how to use them.

Table 5-4 *Python Comparison, Identity, and Logical Operators*

Operator	Description	Example
==	Comparison operator: Equal to.	a == b
!=	Comparison operator: Not equal to.	a != b
>	Comparison operator: Greater than.	a > b
<	Comparison operator: Less than.	a < b
>=	Comparison operator: Greater than or equal to.	a >= b
<=	Comparison operator: Less than or equal to.	a <= b
is	Identity operator: Used for object comparison. Returns **True** if both variables are the same object.	a is b
is not	Identity operator: Used for object comparison. Returns **True** if both variables are not the same object.	a is not b
and	Logical operator: Returns **True** if both statements are true.	(a > x) and (b < y)
or	Logical operator: Returns **True** if one of the statements is true.	(a > x) or (b < y)
not	Logical operator: Returns the reverse result. If true, returns **False** and vice versa.	not((a > x) or (b < y))

The membership operators, which are used to validate whether a sequence is present in an object, can be used with conditional statements to return a Boolean value or loops to iterate through the object. There are two membership operators: **in** and **not in**. The **in** membership operator returns **True** if a value is present in the sequence and **False** otherwise; the **not in** operator returns **True** if a sequence was not present in the object and **False** otherwise. Table 5-5 describes the membership operators and provides examples.

Table 5-5 *Membership Operators*

Operator	Description	Example
in	Returns True if the value present in the sequence, False otherwise	a in b
not in	Used for validating if a value exists in a sequence or not. Returns True if the value is not present in the sequence, False otherwise	a not in b

Python Data Structures

Some of the fundamental needs in writing software programs are to manage the data, perform computations, organize the data, and store the data. *Data structures* provides the means to represent related data to be used later within a program. Different types of data structures are used, depending on the kind of data and the manageability of the data. Python has the four built-in data structures:

• List

• Dictionary

• Tuple

• Set

List

A list is an ordered sequence that can hold a variety of object types that are changeable. Lists are defined inside square brackets ([]), with commas separating objects in the list. A list in Python is similar to an array in other programming languages, such as Java or PHP. It allows users to store an enumerated set of items in one place and access them using their position (that is, the index in the list). To illustrate the concept of indexing in a list, the following example shows a list named **colors** that enumerates the colors of the rainbow:

```
colors = ['violet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']
```

Each item in the list is a color name (value) and has a specific position (index). The list named **colors** can simply be expressed in an ordered sequence, as shown in Figure 5-4. Because Python uses zero-based indexing, the first element, **violet**, has the index value 0, and the last element, **red**, has the index value 6. The first element in the list is often called the *head end* of the list, and the last element is called the *tail end*. Programs typically access the elements from the head end comes; however, if a program needs to start with the end of a list or the penultimate element, it starts enumerating elements from the tail end of the list. In such a scenario, negative indexing can be used. In negative indexing, -1 corresponds to the last element of the list, and the programs keeps on decrementing to the first element.

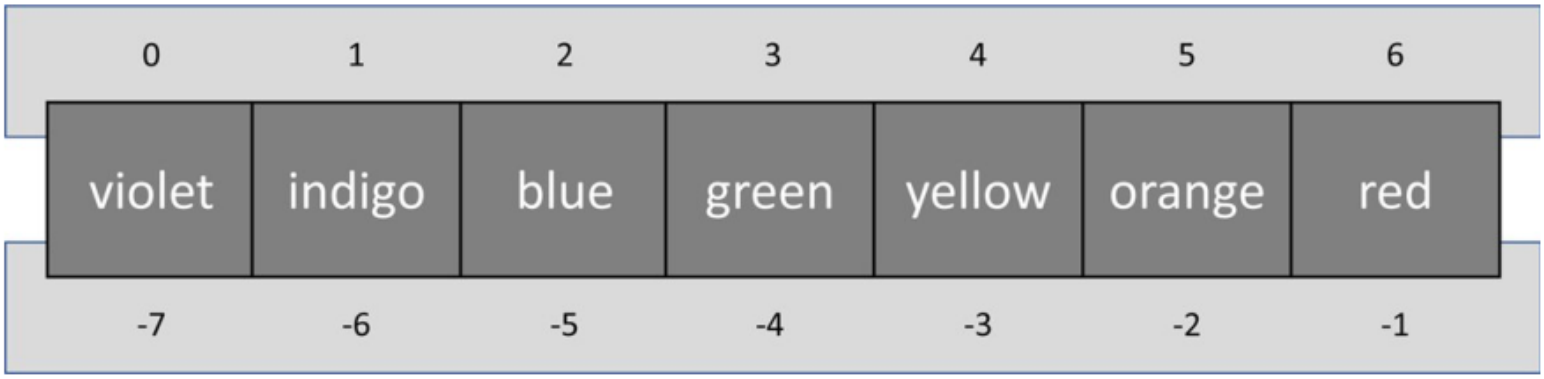


Figure 5-4 *A List with Indexes and Values*

Each member item in a list can individually be accessed using its index number, as shown in Example 5-22. In this example, there are two lists: **list1** and **list2**, each with five different values that can be accessed using the index value of the list. For instance, the fourth index of **list1** returns the value **5**. The **len()** function can also be used to assess the depth of the list. The **+** operator is used to perform concatenation of two lists—and not an arithmetic operation on the members of lists. The arithmetic **+** operation can be performed on members of a list accessed using their index values, as shown at the end of Example 5-22.

Example 5-22 *Accessing List Data and List Operations*

```
>>> list1 = [1,2,3,4,5]
>>> list2 = [6,7,8,9,10]
>>> list1
[1, 2, 3, 4, 5]
>>> list2
[6, 7, 8, 9, 10]
>>> list1[4]
5
>>> list2[3]
9
>>> len(list1)
5
>>> list1 + list2
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1
[1, 2, 3, 4, 5]
>>> list2
[6, 7, 8, 9, 10]
>>> list3 = list1 + list2
>>> list3
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[0] + list1[1] + list2[3] + list2[4]
22
```

Note

Python list members can also be accessed using negative indexing. With Python lists, the indexing begins with -1 and below and goes from right to left. There is no index 0 with negative indexing. The indexing in a list works the same way as indexing in string variables.

Indexes can be used not only to access list members but are also to assign values. When assigning values, only the index part of a list can be used. A value assignment to a new list index is not allowed. Support for indexes also provides the capability of performing list slicing. *List slicing* works the same way as string slicing, but it returns the submembers of the list after slicing. Example 5-23 demonstrates value assignment on different list indexes and list slicing.

Example 5-23 *Value Assignment and Slicing in a Python List*

```
>>> list1[0] = 11
>>> list2[2] = 18
>>> list1
[11, 2, 3, 4, 5]
>>> list2
[6, 7, 18, 9, 10]
>>> list1[1:]
[2, 3, 4, 5]
>>> list1[2:4]
[3, 4]
>>> list1[0::2]
[1, 3, 5]
```

Note

Python list value assignment and list slicing using negative indexes work the same way as for strings.

Python lists have predefined functions: **list.append()** adds a new member to a list, and **list.pop()** removes an existing member from a list. The **list.append()** function takes a parameter as the value and appends the value to the last index of the list. The **list.pop()** function can be used with or without a parameter. If used without a parameter, the **list.pop()** function removes the last member from the list. Otherwise, the **list.pop()** function takes the parameter of the index that needs to be removed or popped from the list. Example 5-24 demonstrates the use of the **list.append()** and **list.pop()** functions.

Example 5-24 *Using List Functions*

```
>>> list3
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list3.append(20)
>>> list3
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20]
>>> list3.pop()
20
>>> list3
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list3.pop(4)
5
>>> list3
[1, 2, 3, 4, 6, 7, 8, 9, 10]
```

One of the benefits that the Python list data structure provides is sorting capabilities, which are provided by the **list.sort()** and **list.reverse()** functions. The **list.sort()** and **list.reverse()** function works for lists holding both numeric values and character/string values. Example 5-25 displays the use of both the **list.sort()** and **list.reverse()** functions.

Example 5-25 *Using the **list.sort()** and **list.reverse()** Functions*

```
>>> list_num = [1,2,5,8,3,4,6]
>>> list_char = ['d','e','g','a','f','b']
>>> list_num.sort()
>>> list_num
[1, 2, 3, 4, 5, 6, 8]
>>> list_char.sort()
```

```
>>> list_char

['a', 'b', 'd', 'e', 'f', 'g']

>>> list_num.reverse()

>>> list_num

[8, 6, 5, 4, 3, 2, 1]

>>> list_char.reverse()

>>> list_char

['g', 'f', 'e', 'd', 'b', 'a']
```

Python list members can be of type **list**, which means that users can create nested list data structures. The operations on the list members depend on the class type of the returned value. If the returned value of an index is also a list, then another set of square brackets with index values can be used to access the members of the inner list. Example 5-26 illustrates how nested lists look and how data within the nested lists can be fetched.

Example 5-26 *Using Nested List*

```
>>> nested_list = [1,2,3,[4,5],[7,8,9],['a','b']]

>>> nested_list

[1, 2, 3, [4, 5], [7, 8, 9], ['a', 'b']]

>>> nested_list[3]

[4, 5]

>>> nested_list[3][1]

5

>>> nested_list[5][0]

'a'

>>> type(nested_list)

<class 'list'>

>>> type(nested_list[3])

<class 'list'>

>>> type(nested_list[3][0])

<class 'int'>

>>> type(nested_list[5][0])

<class 'str'>
```

Dictionaries

A *dictionary* is an unordered collection of objects that are changeable and indexed. Unlike lists, dictionaries use key/value pairs for storing objects. Starting with Version 3.7, Python dictionaries are ordered. These key/value pairs allow users to easily access the data without knowing the index location of an object. In Python, dictionaries are written within curly brackets and use colons to signify the keys and their associated values, and the key/value pairs are separated by commas. One question that usually arises is when to choose a dictionary over a list. On dictionary objects, operations such as indexing, slicing, and sorting cannot be performed. The data can be stored at any location in a dictionary, and the user just needs to know the key to fetch the data. A value in a dictionary can be accessed by specifying the key within square brackets. Example 5-27 shows a dictionary named **my_dict** with three key/value pairs. The value for a referenced key can be modified by using the assignment operator **=**.

Example 5-27 *Using Python Dictionaries*

```
>>> my_dict

{'key1': 0, 'key2': 1, 'key3': 2}

>>> my_dict['key1']

0

>>> my_dict['key3']

2

>>> my_dict.get('key2')

1

>>> my_dict['key1'] = 'Zero'

>>> my_dict

{'key1': 'Zero', 'key2': 1, 'key3': 2}
```

In Python, data insertion on dictionaries can simply be done using the keys and assigning values to those keys. It is not necessary to have a specified key preexist in a dictionary, but it is imperative to remember that keys are immutable. Dictionaries allow for storing values of different kinds, including a Python list or a dictionary itself. Users can use the **type()** function to identify the class type of the returned value and can perform actions or call predefined functions based on the data types. Example 5-28 illustrates how to handle a dictionary with different data types and edit or add new key/value pairs.

Example 5-28 *Dictionaries of Different Data Types*

```
>>> mydict = {1:'Hello' , 2:'Python', 'key3': ['a','b','c'], 'key4':{'n1':'v1',
```

```
'n2':'v2'}}
>>> mydict[1]
'Hello'
>>> mydict[2]
'Python'
>>> mydict['key3']
['a', 'b', 'c']
>>> type(mydict['key3'])
<class 'list'>
>>> mydict['key3'][1]
'b'
>>> mydict['key4']
{'n1': 'v1', 'n2': 'v2'}
>>> type(mydict['key4'])
<class 'dict'>
>>> mydict['key4']['n2']
'v2'
>>> type(mydict['key4']['n2'])
<class 'str'>
>>> mydict['key4']['n2'].upper()
'V2'
>>> mydict['key4']['n2'] = 'New Value'
>>> mydict['key4']['n2']
'New Value'
>>> mydict[5] = 'Value 5'
>>> mydict
{1: 'Hello', 2: 'Python', 'key3': ['a', 'b', 'c'], 'key4': {'n1': 'v1', 'n2':
'New Value'}, 5: 'Value 5'}
```

Python has predefined functions for dictionary data structure. **dict.keys()**, **dict.values()**, and **dict.items()** are the three key functions for the dictionary class. The **dict.keys()** function yields all the keys present in the dictionary. This function can be useful for iterating through a whole dictionary via a loop or validating the presence of a key within a dictionary. The **dict.values()** function returns all the values present in a given dictionary. Similar to the **dict.keys()** function, the **dict.values()** function can be useful for iterating through the values present in a given dictionary. Finally, the **dict.items()** function returns the key/value pairs. Note that all three of these functions return the values in the form of an array or a list. Example 5-29 shows the result from the use of all three of these functions.

Example 5-29 *Using Dictionary Functions*

```
>>> mydict.keys()
dict_keys([1, 2, 'key3', 'key4', 5])
>>> mydict.values()
dict_values(['Hello', 'Python', ['a', 'b', 'c'], {'n1': 'v1', 'n2': 'New Value'},
'Value 5'])
>>> mydict.items()
dict_items([(1, 'Hello'), (2, 'Python'), ('key3', ['a', 'b', 'c']), ('key4',
{'n1': 'v1', 'n2': 'New Value'}), (5, 'Value 5')])
```

The key/value pairs can be deleted by simply deleting the key using the **del** keyword. The **del** keyword takes in the dictionary parameter with the key that is to be deleted. Example 5-30 illustrates how to delete a specific key/value pair in a dictionary. In this example, notice that the key 1 has the value **'Hello'**. After calling the **del** keyword on key 1 of the dictionary, you can no longer see key 1 present in the dictionary.

Example 5-30 *Deleting a Specific Dictionary (Key/Value) Entry*

```
>>> mydict
{1: 'Hello', 2: 'Python', 'key3': ['a', 'b', 'c'], 'key4': {'n1': 'v1', 'n2': 'v2'}}
>>> del mydict[1]


>>> mydict
{2: 'Python', 'key3': ['a', 'b', 'c'], 'key4': {'n1': 'v1', 'n2': 'v2'}}
```

Tuples

A Python *tuple* is a collection of data in an ordered and immutable form, so once a tuple is created, the data inside the tuple cannot be modified. This data structure comes in handy in cases where an application needs static or read-only data. Python tuples are represented in

parentheses and can hold data of different data types. If an attempt is made to modify data stored in a tuple, a type error is returned, stating that the object does not support item assignment. Example 5-31 illustrates the creation of a tuple and how data at different indexes can be fetched. Note that based on the data type of the data within the tuple, Python functions can be called. For instance, if data is of type **str**, then functions such as **str.upper()** and **str.reverse()** can be called on the data at a given index of a tuple.

Example 5-31 *Handling Python Tuples*

```
>>> tup1 = (1,2,3,'Hello','This is Python3')
>>> tup1
(1, 2, 3, 'Hello', 'This is Python3')
>>> type(tup1)
<class 'tuple'>
>>> tup1[0]
1
>>> tup1[3]
'Hello'
>>> tup1[0] = 'Hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment


>>> type(tup1[3])
<class 'str'>
>>> tup1[3].upper()
'HELLO'
```

Although the data within a tuple cannot be modified, it is possible to join two different tuples and create a third tuple. This is done by using the **+** operator between the two tuples. You can also delete a tuple but not the data within the tuple. You use the **del** keyword to delete a tuple. Example 5-32 demonstrates both the joining of two tuples and the deletion of a tuple.

Example 5-32 *Joining Tuples and Deleting a Tuple*

```
>>> tup1 = (1,2,3)
>>> tup2 = ('a', 'b', 'c')
>>> tup3 = tup1 + tup2
>>> tup3
(1, 2, 3, 'a', 'b', 'c')
>>> del tup3
>>> tup3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tup3' is not defined
```

Sets

A *set* is an unordered and unindexed collection of unique data elements. Because a set is unindexed, the data items in the set cannot be accessed by any index. The only way to access the elements of a set is in a **for** loop, using membership operators such as **in** or **not in**. The class set has three key functions to perform update, add, and delete operations on the elements of the set: **set.add()**, **set.update()**, and **set.remove()**. To add an element to a set, you use the **set.add()** function. Similarly, you can update the elements in a set, but you cannot update a specific element with a newer value. The **set.update()** function updates a whole set—that is, all the elements in the set. To remove a specific element in a set, you can call the **set.remove()** function. Example 5-33 shows how to create a new set and how different operations can be performed on the set with the **set.add()**, **set.update()**, and **set.remove()** functions.

Example 5-33 *Defining Sets and Performing Operations on Sets*

```
>>> set1 = {'a',2,3,4,'e'}
>>> set2 = {1,2,3,4,5}
>>> set3 = {'a','b','c','d','e'}
>>> set1
{2, 3, 'e', 4, 'a'}
>>> set1.add(6)
>>> set1
{2, 3, 'e', 4, 6, 'a'}
>>> set2.update([1,2,4,6,5,7])
>>> set2
{1, 2, 3, 4, 5, 6, 7}
```

```
>>> set2.remove(7)

>>> set2

{1, 2, 3, 4, 5, 6}
```

Note

You can use the **set.pop()** function to remove the last item from a set. Because a set is unordered in nature, you do not know which item this function will remove.

Much like other data structures, sets can be combined with other sets; the function for this is **set.union()**. This function returns a new set that is a union of all the elements of two sets. Note that with the **set.union()** function, the elements returned from two sets are unique. If there are overlapping elements, the **set.union()** function returns only the common element as a single element. The **set.clear()** function is used to empty all the elements from a set. The **set.difference()** function returns a set showing all the differences between two sets. Example 5-34 illustrates the use of all the three of these functions.

Example 5-34 *Using Set Functions*

```
>>> set4 = set1.union(set2)

>>> set4

{1, 2, 3, 'e', 4, 6, 5, 'a'}


! Empty the elements in the set using clear() function.

>>> set4.clear()

>>> set4

set()


>>> set1.difference(set2)

{'a', 'e'}
```

Control Flow

In a program, *control flow*refers to the order in which a code block is executed. In Python, control flow is driven by conditional statements, loops, and function calls. The control flow syntax includes colons and indentation (whitespace) used to separate the parts of a code block that will be executed when a certain condition is met or when the execution control of the code is with the loop or a function call. Python code is structured using indentation, which sets it apart from other programming languages. Each line of code within a code block is aligned vertically as part of a function or a loop. If a code block has nested code blocks within it, the indentation moves further to the right, and the alignment follows. A block ends on a line that is less indented than the code lines in code block. A colon in the code declares the beginning of an indentation block or code block. Example 5-35 shows a nested code block with conditional statements and loop statements. At this point, you are not expected to know how to use the conditional statements or loops; this example just presents how a nested code blocks can be written. In this example, there is an **if-else** statement at the top level, and there is a nested **if-else** statement within that code block.

Example 5-35 *Nested Code Block with Conditional Statements*

```
>>> a = 10

>>> b = 20

>>> c = 21

>>> if a > b:

...     print('{} is greater than {}'.format(a,b))

...     if a > c:

...         print('{} is greater than {}'.format(a,c))

...     else:

...         print('{} is greater than {}'.format(c,a))

... else:

...     print('{} is greater than {}'.format(b,a))

...     if c > b:

...         print('{} is greater than {}'.format(c,b))

...     else:

...         print('{} is greater than {}'.format(b,c))

...


! When the code executes, below output is printed.


20 is greater than 10

21 is greater than 20
```

Note

There is no limit on the depth of the code blocks that can be written, but when writing code, focus should be given to the performance of the code within a code block.

---

if-else Statements

A Python **if-else** statement provides the functionality of conditional code execution. A Python program with an **if-else** statement evaluates an expression, returns a Boolean value (**True** or **False**), and executes the code block if the expression is **True**. If the expression is **False**, the code block is not executed, and the **else** section of the code block is executed (if present). In Python, the control flow of the **if** statement is indicated by the colon (**:**) and structured using indentation. The first unindented line marks the end of the code block. This unindented line can be the beginning of the **else** statement or a continuation of the parent code block. Figure 5-5 illustrates **if-else** control flow.

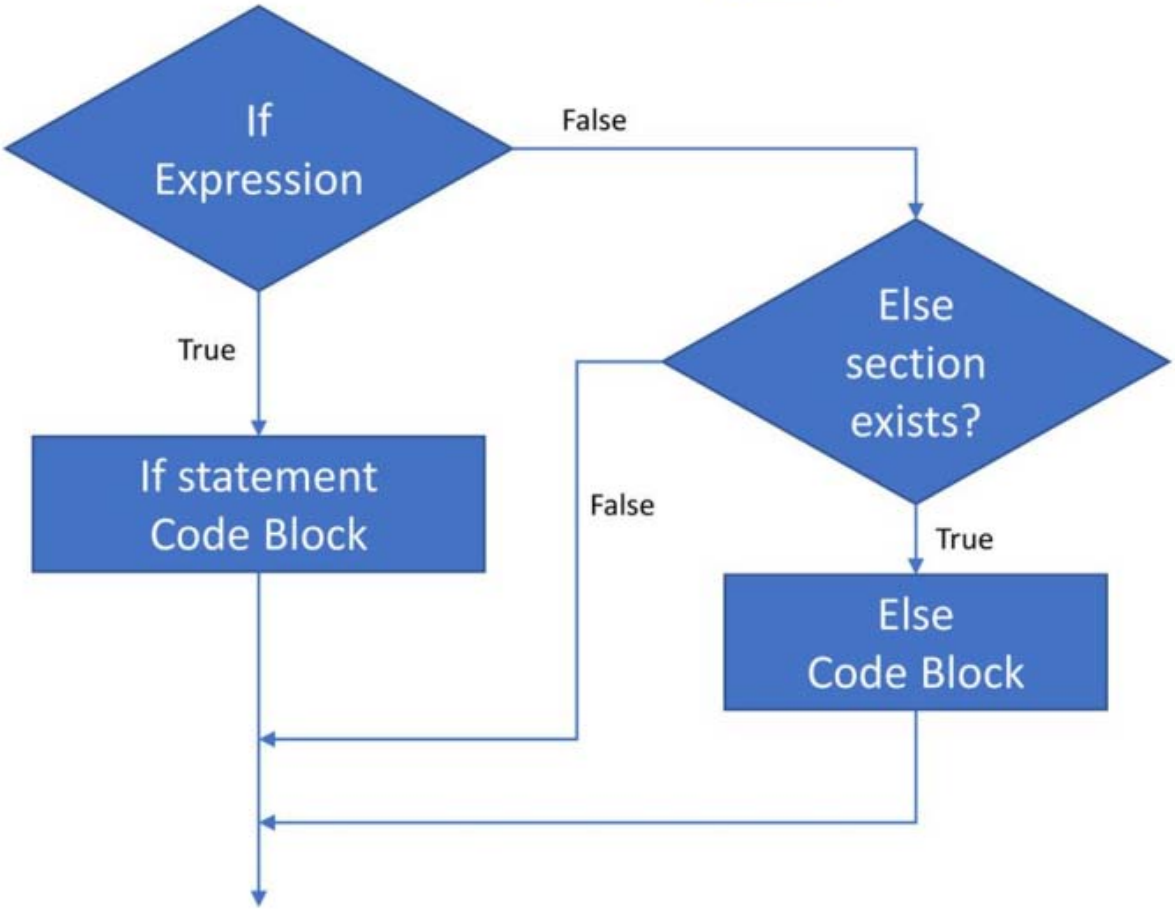


Figure 5-5 *if-else Statement FlowDiagram*

if-else statements can be nested within other **if-else** statements; that is, an **if-else** statement can appear within the code block of an **if** or an **else** statement. Example 5-36 illustrates nested **if-else** statements by comparing the values of three numeric variables **a**, **b**, and **c**. In this example, the first **if** expression validates whether the value of **a** is greater than the value of **b**. If that holds true, further conditions are validated. In this example, because the value of **a** is less than the value of **b**, the control goes to the code block of the **else** statement. Within the **else** statement code block, further evaluation is done to see if the value of **c** is greater than value of **b**; because it is, the code block within the **if** statement is executed.

Example 5-36 *Nested* ***if-else*** *Statements*

```
a = 10
b = 20
c = 21
if a > b:
    print('{} is greater than {}'.format(a,b))
    if a > c:
        print('{} is greater than {}'.format(a,c))
    else:
        print('{} is greater than {}'.format(c,a))
else:
    print('{} is greater than {}'.format(b,a))
    if c > b:
        print('{} is greater than {}'.format(c,b))
    else:
        print('{} is greater than {}'.format(b,c))
```

```
! Output from above code.
```

```
20 is greater than 10

21 is greater than 20
```

In Example 5-36, even though there are nested **if-else** statements, the code execution becomes complex. The same task can be reduced with a smaller number of validations by using logical operators and validating multiple expressions in a single expression and then using the result to execute the respective code block of the **if** or **else** statement. Also, if more than one expression is to be evaluated, an **elif** statement (which is actually an **else-if** statement) can be used. Example 5-37 demonstrates the use of logical operators in evaluating multiple conditions in a single expression and the use of an **elif** statement to evaluate more than one primary expression in an **if-else** statement.

Example 5-37 *Using Logical Operators with an* ***if-else*** *Statement*

```
a = 10

b = 20

c = 21

if a > b and a > c:

    print('{} is greater than {} and {}'.format(a,b,c))

elif b > a and b > c:

    print('{} is greater than {} and {}'.format(b,a,c))

else:

    print('{} is greater than {} and {}'.format(c,a,b))
```

```
! Output from above code.


21 is greater than 10 and 20
```

In all the previous examples, the values of the variables were always preassigned. However, a dynamic program allows users to enter values at runtime and then uses that input to perform validations or take actions. In order to take user input, you use the **input()** function, which takes the parameter as a string (text) to be displayed on the terminal window and waits for the user input. When the user has input a value, it can be stored in another variable for later use or can be consumed right away. To understand how to build different building blocks of a program, examine the Python code shown in Example 5-38. In this example, two lists, named **person** and **age**, are created; the two lists have the same lengths such that so meone at a given index in the **person** list has a matching age at the same index in the **age** list. This program takes user input for a name and validates the input against the **person** list. Based on this input, different **if-elif-else** conditions are evaluated, and the program prints either the person's age or indicates whether the person is a teenager or a senior citizen.

Example 5-38 *An* ***if-else*** *Statement with a List*

```
person = ['Jason','Ray','Chris','Juan']

age = [23,51,19,63]

name = input('Enter your name : ')

if name in person:

    p_age = age[person.index(name)]

    if p_age > 60:

        print('{} is a Senior Citizen'.format(name))

    elif p_age < 20:

        print('{} is a teenager'.format(name))

    else:

        print('Age of {} is {}'.format(name, p_age))

else:

    print('{} is not on our list'.format(name))
```

```
! First Execution

Enter your name : Ray

Age of Ray is 51
```

```
! Second Execution

Enter your name : Andrew

Andrew is not on our list
```

```
! Third Execution

Enter your name : Juan

Juan is a Senior Citizen
```

```
! Fourth Execution

Enter your name : Chris

Chris is a teenager
```

Managing data in Python lists is easy, but it may not be the right choice in Example 5-38. For instance, if the length of the **person** list differed

from the length of the **age** list, the application would not function properly. This can easily happen due to human error in a scenario where the application is also allowing the user to add a new person to the list along with the person's age. When developing applications, it is important to use the right type of data structure to reduce the chances of human error.

To expand on the application that validates a person's name and age, say that new user input can be saved into the data structure and made available for later validation. In Example 5-39, a data structure of type dictionary is created with the person's name as the key and age as the value. The program ensures that the age is correctly mapped to a person. In Example 5-39, if the input name does not exist in the keys, a new key is created, and the user input is used to map the name to the person's age. This example also illustrates the use of membership operators to validate whether a name exists in keys. Notice how different code blocks with **if-else** statements are being used to further enhance the functionality of the application.

Example 5-39 *An **if-else** with Dictionaries*

```
person = {

    'Jason': 23,

    'Ray': 51,

    'Chris': 19,

    'Juan': 63,

}


name = input('Enter your name : ')

if name not in person.keys():

    response = input('Do you want to enter your name on our list? yes or no : ')

    if response.lower() == 'yes':

        new_age = input('Enter your Age : ')

        person[name] = new_age

        print("You have been added to our list.\n ")

        print("Current List: \n")

        print(person)

    else:

        print("You have not been added to our list")

        print("Current List: \n")

        print(person)

else:

    p_age = person[name]

    if p_age > 60:

        print('{} is a Senior Citizen'.format(name))

    elif p_age < 20:

        print('{} is a teenager'.format(name))

    else:

        print('Age of {} is {}'.format(name, p_age))
```

```
! First Execution

Enter your name : Andrew

Do you want to enter your name on our list? Yes or No : Yes

Enter your Age : 33

You have been added to our list.


Current List:


{'Jason': 23, 'Ray': 51, 'Chris': 19, 'Juan': 63, 'Andrew': '33'}
```

```
! Second Execution

Enter your name : Juan

Juan is a Senior Citizen
```

```
! Fourth Execution

Enter your name : Vincent

Do you want to enter your name on our list? Yes or No : No

You have not been added to our list

Current List:
```

Chapter 6. Python Applications

When Python was initially developed in 1989, it was built on the fundamental of Don't Repeat Yourself (DRY) principle. Over the years, Python has evolved as one of the most popular programming languages and can be used to develop a host of applications such as web applications, text and image processing applications, machine learning, or even Enterprise-level applications. But before digging into the various applications of Python, it is important as a developer to setup the development environment to simplify and organize the process of application development.

Organizing the Development Environment

Software programmers often face challenges such as understanding and writing code, building logic, and linking different pieces of code together. In addition to these challenges, there are a few other challenges programmers and software developers often face, including the following:

• **Maintaining version control**: Developing a software/web application usually involves multiple team members, and during the course of the development, it becomes immensely difficult to track the changes made every day by different team members. Version control helps restore code to the last known good state so that if are any defects are introduced in a new version of the code, the last known good code can be called up to restore the services. In addition, version control can help track feature enhancements across releases.

• **Different environments**: Every project that comes across a developer's desk has different requirements, such as different software versions and different third-party packages. Modifying the existing environment to match different project requirements is not an efficient way of settings up the environment. Rather, the packages, software versions, and so on should be isolated to respective project requirements.

• **Replicating the production environment**: There will always be instances (at different stages of the development process or during production) when software applications may run into defects. Defects can either be corner cases that may result from how an application was deployed or the way the production environment was set up. In such scenarios, the first step would be to identify the difference between the testing/development environment and the production environment and then to replicate the exact problem or defect that occurred during production. Mimicking the production environment or the environment where a problem occurred can be a challenging task as it requires matching the exact versions of other applications, such as third-party tools and databases. This becomes more challenging when a programmer wants to replicate a problem without making changes to the existing development environment.

• **Reusable code**: The key to being a good and efficient programmer is to write reusable code in the form of modules that can later be integrated or used in other modules.

The following subsections cover some of the tools and technologies that can help overcome these challenges.

Git

Git is an open-source tool that is used for distributed version control and to efficiently manage everything from small-scale projects to large-scale projects. Git also helps with collaboration between multiple team members. Git was originally developed in 2005 by Linus Torvalds, and today, an astounding number of projects rely on it. Apart from version control, Git provides several other benefits:

• **Performance**: The Git tool is optimized for high performance when committing new changes, branching, merging, and comparing past versions of a project. The distributed nature of Git also provides significant performance benefits. The built-in algorithms of Git use the common attributes of real source code file trees and their modifications over time. The Git tool focuses on the content of a file rather than the name of the file. Git uses a combination of delta encoding, compression, and storage of directory contents and version metadata objects.

• **Security**: Git provides and maintains data integrity by securing all the data in the repository using a secure hashing algorithm called SHA1 that protects the code and the change history against any malicious or accidental changes and makes it fully traceable.

• **Flexibility**: Git supports multiple nonlinear development workflows and is compatible with various existing systems and protocols. Git not only maintains history of the code files and changes but also history based on branches as well as tags.

Git is one of the main tools that developers use for maintaining version control for ongoing projects, especially when working in a team. Git's version control relies on repositories. A *repository* can be thought of as a primary folder where everything associated with a specific project is maintained. A repository can have subfolders or files within it. By using Git, you can maintain both a local copy as well as an online copy of all the files in the repository. Multiple online Git servers, including GitHub, BitBucket, and GitLab, can be used for repositories and are generally used by individuals to share or maintain code. You can create public repositories that are accessible to everyone, repositories for selected users, and private repositories on a project-by-project basis. When it comes to managing code within an organization, it typically makes sense to have a local Git server where all the code commits for the team can be done.

Setting up a Git server requires a few simple steps:

1. Generate an SSH key on the client machine.

2. Install Git by using the command **yum install git**.

3. Add a user and create an authorized_keys file under the .ssh directory of the user.

4. Copy the public key on the client machine to the Git server.

Example 6-1 shows these steps being used to set up a Git server on CentOS.

Example 6-1 *Setting Up a Git Server on CentOS*

```
! Generating SSH Key on client

[root@node2 ~]# ssh-keygen -C "dev@gmail.com"

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id_rsa): /root/.ssh/id_dev_rsa

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /root/.ssh/id_dev_rsa.

Your public key has been saved in /root/.ssh/id_dev_rsa.pub.

The key fingerprint is:
```

```
SHA256:oXUbsjEhDw8syKeJ/MG2MYaAwtzf/psmSyYrIyl4bbI dev@gmail.com
The key's randomart image is:
+---[RSA 2048]----+
|+... .+ .        |
|+oo.o .* .       |
|+.o+... O o      |
|.ooB . + B o     |
|  + = o S .      |
|   o   .         |
|. .. . o.        |
|ooooo =. o.      |
|..E+o. .+o.      |
+----[SHA256]-----+
```

```
! Installing Git on CentOS Server
 [root@master-node ~]# yum install -y git
```

```
! Add user git on Server
[root@master-node ~]#useradd git



! Change password for user git
[root@master-node ~]# passwd git
Changing password for user git.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.


! Create authorized_keys file under .ssh directory


[root@master-node ~]# su git
Password:
[git@master-node ~]# mkdir ~/.ssh && touch ~/.ssh/authorized_keys
```

```
! Copy Public key On Client machine to remote server
[root@node2 ~]# cat .ssh/id_dev_rsa.pub | ssh root@172.16.102.131 "cat >>
/home/git/.ssh/authorized_keys"
The authenticity of host '172.16.102.131 (172.16.102.131)' can't be established.
ECDSA key fingerprint is SHA256:sTsJgHuf+SLBIePL+LLGtA9eG50xZsva18aFvzxjLBQ.
ECDSA key fingerprint is MD5:4d:eb:46:c2:00:26:e5:c0:04:ed:80:e1:e1:e7:72:06.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.16.102.131' (ECDSA) to the list of known hosts.
root@172.16.102.131's password:
[root@node2 ~]#
```

Once the Git server is set up, you can start using it. To use a Git server, you need to first create an empty repository on the Git server by using the command **git init -bare** *project-name.git*. A set of commands can be used to perform the initial setup on the server. Table 6-1 lists some of the commands that are required during Git setup and initialization.

Table 6-1 *Git Setup and Initialization Commands*

Command	Description
git config --global user.name "*firstname lastname*"	Sets up a name that appears in the version history
git config --global user.email "*email-address*"	Sets up an email address that will be associated with each history marker
git init	Initializes an existing directory as a Git repository

A number of commands can be used to add repositories and copy files to a Git server from a client machine. Table 6-2 lists and describes these command.

Table 6-2 *Commands for Updating Local Repos and Fetching Updates from Remote Repos*

Command	Description
git pull	Fetches and merges any commits from the tracked remote branch
git remote add *alias url*	Adds a Git URL as an alias for the remote Git repository
git push *alias branch*	Transmits local branch commits to a branch in a remote repository
git merge *alias/branch*	Merges a remote branch into the current branch to bring it up to date
git fetch *alias*	Fetches all branches from the remote Git repository
git add *file*	Adds a file in the current state to the next commit
git commit -m "*commit-description*"	Commits staged content to a new commit snapshot

Note

These are not the only Git-related commands. More commands are available to modify history, track changes, inspect and compare changes, handle staging and snapshots, and so on. For more details on the Git commands, see http://education.github.com and other Git portals.

Example 6-2 illustrates the use of Git commands on a server as well as on a client machine to stage a commit from the client machine to the remote server. To set up a bare or empty repository on the server, you use the command **git init** with the **-bare** option. A bare Git repository has no working directory, whereas a non-bare Git repository is initialized with a working directory. Bare repositories are useful when you are working as part of a team, and you want a repository to act as a central repository, to which all team members can move their work. (For more information on the commands used in this example, refer to Table 6-1 and Table 6-2.)

Example 6-2 *Setting Up a Git Repository and Using Git on a Client Machine*

```
! Setup a Bare (empty) repository on Server
[git@prime-node ~]$ mkdir project && cd project
[git@prime-node project]$ git init --bare project.git
Initialized empty Git repository in /home/git/project/project.git/
[git@master-node project]$
```

```
[root@node2 ~]# mkdir project
[root@node2 ~]# cd project/
[root@node2 project]# git init
Initialized empty Git repository in /root/project/.git/
[root@node2 project]# git add .
[root@node2 project]# git commit -m 'initial commit'
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
[root@node2 project]# git remote add origin
ssh://git@172.16.102.131:/home/git/project/project.git

[root@node2 project]# touch initial
[root@node2 project]# git add initial
[root@node2 project]# git commit -m 'initial commit'
[master (root-commit) aad844f] initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 initial
[root@node2 project]# git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 203 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@master-node:/home/git/project/project.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Once the repository has been updated on the server, it can be cloned to another directory on the same client or on another client. The **git clone** command allows you to clone the Git repository, which consists of the last updated files on the repository, as shown in Example 6-3.

Example 6-3 *Cloning a Git Repository*

```
[root@node2 ~]# mkdir new_proj
```

```
[root@node2 ~]# cd new_proj/

[root@node2 new_proj]# git clone git@master-node:/home/git/project/project.git

Cloning into 'project'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (3/3), done.

[root@node2 new_proj]# ls -al

total 0

drwxr-xr-x. 3 root root  21 Dec 28 22:39 .

dr-xr-x---. 6 root root 231 Dec 28 22:38 ..

drwxr-xr-x. 3 root root  33 Dec 28 22:39 project

[root@node2 new_proj]# cd project/

[root@node2 project]# ls

initial

[root@node2 project]# pwd

/root/new_proj/project
```

Git workflow can be summarized in three simple phases, which are demonstrated in Example 6-2 and Example 6-3:

• **Commit**: After saving the files, you are required to commit the changes to the repository. The committed work is saved as a version of the repository, which can now be saved on the online repository.

• **Pull**: Before saving the changes on a remote or online repository, a pull of the existing repository is required to ensure that the files are completely up to date with the online repository.

• **Push**: When the local copy of the repository is up to date, you can push the changes to the online repository.

Docker

There have been many advancements over the years in programming languages, but even with modern programming languages, programmers still face steep challenges in setting up their development environments. It is imperative for a development team to use the same development environment so that integration of the application isn't affected due to the differences. When it comes to setting up the development environment, there are many variables involved, such as different operating systems, different versions of Python, virtual environments, and third-party modules; the problem is magnified for users working as a team. Similar challenges arise when replicating a problem that is occurring in a production environment.

All the challenges just mentioned can be resolved easily with the help of virtualization. A cloud provider such as Amazon Web Services (AWS) provides virtual instances of servers that can be dynamically added and removed as required. With virtualization, it is largely the software behind the scenes that is changed rather than the actual hardware. Virtual instances of servers are provided to the end users using virtual machines. The downside of using virtual machines is that they lack speed and large amounts of CPU, memory, and storage resources. A typical virtual machine with a standard operating system requires at least a single CPU core, 1 to 2 GB of storage, and 2 to 4 GB of random-access memory (RAM). Thus, a machine running five virtual machines would require at least 5 CPU cores, 5 to 10 GB of storage, and 10 to 20 GB of RAM. Even though CPU, storage, and RAM are not as expensive as they once were, these resources can still result in high costs.

Docker provides a lightweight virtualization solution that isolates an entire operating system via Linux containers. With containers, you can virtualize the environment on the upper layers of Linux, which speeds up deployment and also segregates the upper layers from the rest of the environment. Figure 6-1 illustrates container virtualization with Docker.

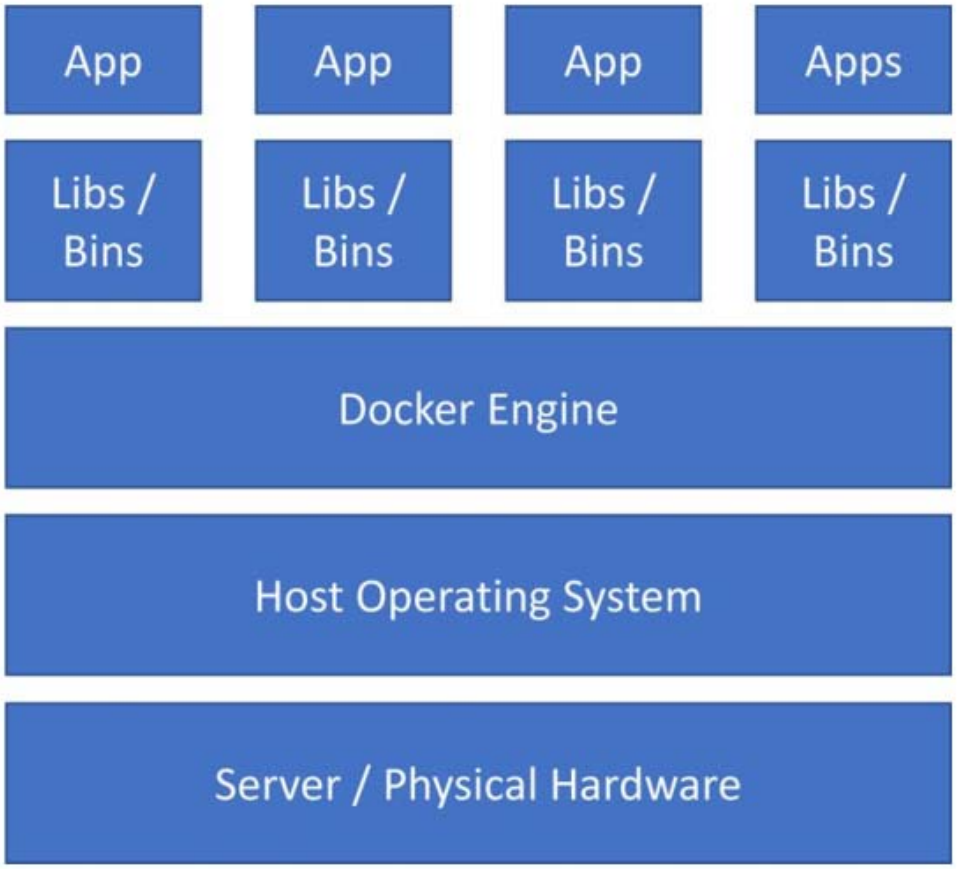


Figure 6-1 *Container Virtualization*

At the time of writing, the CentOS software repository does not maintain a default repository for the Docker application. To install the Docker application, the default repository should be updated to the location of the Docker repository. To begin the installation of Docker on CentOS, you need to install a few prerequisite packages. The **yum-utils** package provides the **yum-config-manager** utility, which allows user to add a repository. Docker uses the **devicemapper** storage driver, and the **device-mapper-persistent-data** and **lvm2** packages are required by **devicemapper**. Once these packages are installed using the **yum** package manager, you can add the **docker-ce** repository by using the **yum-config-manager** utility as shown in Example 6-4. You can then install the **docker-ce** package by using the **yum** package manager. The **docker-ce** package comes with other dependencies, such as the **docker-cli**, **container-selinux**, and **containered.io** packages. The **container-selinux** package provides support for SELinux security for containers, and the **containered.io** package provides a standard runtime environment for containers.

Example 6-4 *Installing Docker on CentOS*

```
[root@rnd-srvr ~]# yum install -y yum-utils device-mapper-persistent-data lvm2
[root@rnd-srvr ~]# yum-config-manager \
>     --add-repo \
>     https://download.docker.com/linux/centos/docker-ce.repo
Loaded plugins: fastestmirror
adding repo from: https://download.docker.com/linux/centos/docker-ce.repo
grabbing file https://download.docker.com/lin


[root@rnd-srvr ~]# yum install -y docker-ce
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: repos-lax.psychz.net
 * extras: mirror.sjc02.svwh.net
 * updates: mirror.sjc02.svwh.net
docker-ce-stable                                         | 3.5 kB     00:00
(1/2): docker-ce-stable/x86_64/updateinfo                  |   55 B   00:05
(2/2): docker-ce-stable/x86_64/primary_db                  |  37 kB   00:05
Resolving Dependencies
--> Running transaction check
---> Package docker-ce.x86_64 3:19.03.5-3.el7 will be installed
--> Processing Dependency: container-selinux >= 2:2.74 for package: 3:docker-ce-
19.03.5-3.el7.x86_64
--> Processing Dependency: containerd.io >= 1.2.2-3 for package: 3:docker-ce-
19.03.5-3.el7.x86_64
```

```
--> Processing Dependency: docker-ce-cli for package: 3:docker-ce-19.03.5-
3.el7.x86_64
--> Running transaction check
---> Package container-selinux.noarch 2:2.107-3.el7 will be installed
---> Package containerd.io.x86_64 0:1.2.10-3.2.el7 will be installed
---> Package docker-ce-cli.x86_64 1:19.03.5-3.el7 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package              Arch      Version               Repository           Size
================================================================================
Installing:
 docker-ce            x86_64    3:19.03.5-3.el7       docker-ce-stable     24 M
Installing for dependencies:
 container-selinux    noarch    2:2.107-3.el7         extras               39 k
 containerd.io        x86_64    1.2.10-3.2.el7        docker-ce-stable     23 M
 docker-ce-cli        x86_64    1:19.03.5-3.el7       docker-ce-stable     39 M

Transaction Summary
================================================================================
Install  1 Package (+3 Dependent packages)

Total download size: 87 M
Installed size: 362 M
Downloading packages:
(1/4): container-selinux-2.107-3.el7.noarch.rpm            |  39 kB   00:05
warning: /var/cache/yum/x86_64/7/docker-ce-stable/packages/containerd.io-1.2.10-
3.2.el7.x86_64.rpm: Header V4 RSA/SHA512 Signature, key ID 621e9f35: NOKEY
Public key for containerd.io-1.2.10-3.2.el7.x86_64.rpm is not installed
(2/4): containerd.io-1.2.10-3.2.el7.x86_64.rpm             |  23 MB   00:07
(3/4): docker-ce-19.03.5-3.el7.x86_64.rpm                  |  24 MB   00:07
(4/4): docker-ce-cli-19.03.5-3.el7.x86_64.rpm              |  39 MB   00:02
--------------------------------------------------------------------------------
Total                                          9.3 MB/s |  87 MB  00:09
Retrieving key from https://download.docker.com/linux/centos/gpg
Importing GPG key 0x621E9F35:
 Userid     : "Docker Release (CE rpm) <docker@docker.com>"
 Fingerprint: 060a 61c5 1b55 8a7f 742b 77aa c52f eb6b 621e 9f35
 From       : https://download.docker.com/linux/centos/gpg
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : 2:container-selinux-2.107-3.el7.noarch                       1/4
  Installing : containerd.io-1.2.10-3.2.el7.x86_64                          2/4
  Installing : 1:docker-ce-cli-19.03.5-3.el7.x86_64                         3/4
  Installing : 3:docker-ce-19.03.5-3.el7.x86_64                             4/4
  Verifying  : containerd.io-1.2.10-3.2.el7.x86_64                          1/4
  Verifying  : 1:docker-ce-cli-19.03.5-3.el7.x86_64                         2/4
  Verifying  : 2:container-selinux-2.107-3.el7.noarch                       3/4
  Verifying  : 3:docker-ce-19.03.5-3.el7.x86_64                             4/4
```

```
Installed:
  docker-ce.x86_64 3:19.03.5-3.el7

Dependency Installed:
  container-selinux.noarch 2:2.107-3.el7  containerd.io.x86_64 0:1.2.10-3.2.el7
  docker-ce-cli.x86_64 1:19.03.5-3.el7

Complete!
[root@rnd-srvr ~]#
```

Note

Along with Docker, the tool Docker Compose can be used to help automate commands. To install this tool, you use the command **pip3 install docker-compose**. When you install Docker Compose, other tools are installed along with it, such as PyYAML, Cryptography, jsonschema, and importlib-metadata.

After Docker is installed, you can verify the Docker version by using the command **docker --version**. Similarly, you can use the command **docker-compose --version** to validate the version of the Docker Compose tool, as shown in Example 6-5. Along with verifying the version, you can use the command **docker info** to get detailed information about the Docker setup on the system. The **docker info** command provides a summary of all the containers in different states, including runtime information, storage information, security information, information on registries, and any proxy information used by the Docker daemon.

Example 6-5 *Verifying the Docker and Docker Compose Versions*

```
[root@rnd-srvr ~]# docker --version
Docker version 19.03.5, build 633a0ea
[root@rnd-srvr ~]# docker-compose --version
docker-compose version 1.25.0, build b42d419
[root@rnd-srvr ~]# docker info
Containers: 8
 Running: 0
 Paused: 0
 Stopped: 8
Images: 9
Server Version: 1.13.1
Storage Driver: overlay2
 Backing Filesystem: xfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: journald
Cgroup Driver: systemd
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: docker-runc runc
Default Runtime: docker-runc
Init Binary: /usr/libexec/docker/docker-init-current
containerd version:  (expected: aa8187dbd3b7ad67d8e5e3a15115d3eef43a7ed1)
runc version: 9c3c5f853ebf0ffac0d087e94daef462133b69c7 (expected:
9df8b306d01f59d3a8029be411de015b7304dd8f)
init version: fec3683b971d9c3ef73f284f176672c44b448662 (expected:
949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
 seccomp
  WARNING: You're not using the default seccomp profile
  Profile: /etc/docker/seccomp.json
 selinux
Kernel Version: 3.10.0-1062.9.1.el7.x86_64
```

```
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
Number of Docker Hooks: 3
CPUs: 1
Total Memory: 487 MiB
Name: rnd-srvr
ID: IEFM:3N2G:HYZT:PDBE:DOBK:NR7I:OEMB:MDIH:IC2Q:VEQ3:LEUM:ADAF
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Http Proxy: http://proxy.cisco.com:80/
Https Proxy: http://proxy.cisco.com:80/
Registry: https://index.docker.io/v1/
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
Registries: docker.io (secure)
```

The Docker engine is a client/server application with three main components:

• **Server**: Docker is a long-running program represented by a daemon process. It listens to API requests and manages objects such as images, containers, networks, and volumes.

• **REST API**: An API interface allows applications to interact with daemon process and provide them with instructions.

• **Client**: The client is represented by the command-line interface (CLI) command **docker**. When you execute the **docker run** command, the client sends the request to **dockerd** (the Docker daemon process), which executes the request.

To begin using Docker and get an idea of how it works, you can run a hello world container by using the command **docker run hello-world**. Successful execution of this container helps you verify the successful installation of the Docker software. When you execute the **docker run** command, the Docker client and Docker daemon go through the following steps:

1. The Docker client contacts the Docker daemon.

2. The Docker daemon pulls the hello world image from Docker Hub (http://hub.docker.com).

3. The Docker daemon creates a new container from that image to run the executable.

4. The Docker daemon streams the output to the Docker client, which is then sent to the terminal.

The public Docker repository at Docker Hub contains various images of different applications and operating systems. Example 6-6 illustrates the execution of the **hello-world** Docker image and the CentOS image that is pulled from Docker Hub. Note that the images are not pulled from the repository every time they are executed. The images are pulled the first time and stored on the disk. The next time the Docker daemon tries to spin up a container, it first checks the local storage to see if it has the image, and if it finds the image, it spins up the container using that image. The **docker** command can be executed with various options. One of the most commonly used options is **-it**, which is a combination of the two options **-i** and **-t**; it tells the Docker daemon that you want an interactive session with a TTY attached. The command option **bash** enables Bash shell access to the container. When the shell is exited, the container stops and releases the resources.

Example 6-6 *Executing **hello-world** and **centos** Containers*

```
[root@rnd-srvr ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
Trying to pull repository docker.io/library/hello-world ...
latest: Pulling from docker.io/library/hello-world
1b930d010525: Pull complete
Digest: sha256:4fe721ccc2e8dc7362278a29dc660d833570ec2682f4e4194f4ee23e415e1064
Status: Downloaded newer image for docker.io/hello-world:latest
[86886.763550] docker0: port 1(vethf57043d) entered blocking state
[86886.764829] docker0: port 1(vethf57043d) entered disabled state
[86886.766195] device vethf57043d entered promiscuous mode
[86886.767555] IPv6: ADDRCONF(NETDEV_UP): vethf57043d: link is not ready
[86886.768973] docker0: port 1(vethf57043d) entered blocking state
[86886.770252] docker0: port 1(vethf57043d) entered forwarding state
[86886.773996] docker0: port 1(vethf57043d) entered disabled state
```

```
[86886.936805] SELinux: mount invalid.  Same superblock, different security
settings for (dev mqueue, type mqueue)
[86887.042984] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[86887.045667] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[86887.047194] IPv6: ADDRCONF(NETDEV_CHANGE): vethf57043d: link becomes ready
[86887.048679] docker0: port 1(vethf57043d) entered blocking state
[86887.049948] docker0: port 1(vethf57043d) entered forwarding state
[86887.051307] IPv6: ADDRCONF(NETDEV_CHANGE): docker0: link becomes ready

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

[root@rnd-srvr ~]#
```

```
[root@rnd-srvr ~]# docker run -it centos bash
Unable to find image 'centos:latest' locally
Trying to pull repository docker.io/library/centos ...
latest: Pulling from docker.io/library/centos
729ec3a6ada3: Pull complete
Digest: sha256:f94c1d992c193b3dc09e297ffd54d8a4f1dc946c37cbeceb26d35ce1647f88d9
Status: Downloaded newer image for docker.io/centos:latest
[87022.416608] docker0: port 1(veth1a10bb9d) entered blocking state
[87022.417940] docker0: port 1(veth1a10bb9d) entered disabled state
[87022.421141] device veth1a10bb9d entered promiscuous mode
[87022.422546] IPv6: ADDRCONF(NETDEV_UP): veth1a10bb9d: link is not ready
[87022.423952] docker0: port 1(veth1a10bb9d) entered blocking state
[87022.425243] docker0: port 1(veth1a10bb9d) entered forwarding state
[87022.427503] docker0: port 1(veth1a10bb9d) entered disabled state
[87022.589013] SELinux: mount invalid.  Same superblock, different security
settings for (dev mqueue, type mqueue)
[87022.644691] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[87022.646575] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[87022.648045] IPv6: ADDRCONF(NETDEV_CHANGE): veth1a10bb9d: link becomes ready
[87022.649560] docker0: port 1(veth1a10bb9d) entered blocking state
[87022.650827] docker0: port 1(veth1a10bb9d) entered forwarding state
[root@fb6a5b14bcdb /]#
```

```
[root@fb6a5b14bcdb /]# ls -al
total 0
drwxr-xr-x.   1 root root  17 Dec 31 09:22 .
drwxr-xr-x.   1 root root  17 Dec 31 09:22 ..
-rwxr-xr-x.   1 root root   0 Dec 31 09:22 .dockerenv
lrwxrwxrwx.   1 root root   7 May 11  2019 bin -> usr/bin
drwxr-xr-x.   5 root root 360 Dec 31 09:22 dev
drwxr-xr-x.   1 root root  66 Dec 31 09:22 etc
drwxr-xr-x.   2 root root   6 May 11  2019 home
lrwxrwxrwx.   1 root root   7 May 11  2019 lib -> usr/lib
lrwxrwxrwx.   1 root root   9 May 11  2019 lib64 -> usr/lib64
drwx------.   2 root root   6 Sep 27 17:13 lost+found
drwxr-xr-x.   2 root root   6 May 11  2019 media
drwxr-xr-x.   2 root root   6 May 11  2019 mnt
drwxr-xr-x.   2 root root   6 May 11  2019 opt
dr-xr-xr-x. 104 root root   0 Dec 31 09:22 proc
dr-xr-x---.   2 root root 162 Sep 27 17:13 root
drwxr-xr-x.   1 root root  21 Dec 31 09:22 run
lrwxrwxrwx.   1 root root   8 May 11  2019 sbin -> usr/sbin
drwxr-xr-x.   2 root root   6 May 11  2019 srv
dr-xr-xr-x.  13 root root   0 Dec 30 09:12 sys
drwxrwxrwt.   7 root root 145 Sep 27 17:13 tmp
drwxr-xr-x.  12 root root 144 Sep 27 17:13 usr
drwxr-xr-x.  20 root root 262 Sep 27 17:13 var
[root@fb6a5b14bcdb /]# cat /etc/centos-release
CentOS Linux release 8.0.1905 (Core)
[root@fb6a5b14bcdb /]# exit
exit
[root@rnd-srvr ~]#
```

You can also pull Docker images by using the command **docker pull** *image-name*. This command pulls an image from the registry and stores it locally but does not execute the image. Example 6-7 demonstrates the process of pulling the **alpine** image from the Docker Hub registry and listing downloaded images by using the **docker images** command.

Example 6-7 ***docker pull*** *and* ***docker images*** *Command Output*

```
[root@rnd-srvr ~]# docker pull alpine:latest
Trying to pull repository docker.io/library/alpine ...
latest: Pulling from docker.io/library/alpine
c9b1b535fdd9: Pull complete
Digest: sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367d
Status: Downloaded newer image for docker.io/alpine:latest
```

```
[root@rnd-srvr ~]# docker images
REPOSITORY             TAG             IMAGE ID            CREATED             SIZE
docker.io/alpine       latest          e7d92cdc71fe        2 days ago          5.59 MB
docker.io/python       3               038a832804a0        3 weeks ago         932 MB
docker.io/centos       latest          0f3e07c0138f        2 days ago          220 MB
docker.io/hello-world  latest          fce289e99eb9        12 months ago       1.84 kB
```

A few other commands are useful with Docker. Table 6-3 lists some of the commands that can be useful with Docker clients.

Table 6-3 *Docker Commands*

Command	Description
docker image ls	Lists all the images locally stored with Docker Engine. (It is similar to the **docker images** command.)
docker image rm *image*	Deletes an image from the local image store.
docker push *image*	Pushes an image to a registry.
docker container run --name *name* **-p** *port-ext:port-int image*	Runs a container with the specified image. The **--name** option allows you to specify the name of the running container. The **-p** option exposes the external port (*port-ext*) that is mapped to the internal port inside the container (*port-int*).
docker container ls	Lists the running containers. Use this command with the **-all** option to include the stopped containers in the output.
docker container stop *name*	Stops the running container that has the specified name. When using this command, the container is stopped using **SIGTERM**.
docker container kill *name*	Stops the running container that has the specified name. When using this command, the container is stopped using **SIGKILL**.
docker container logs [*options*] *name*	Fetches the logs of a container. Use the **--tail** *number* option to print the last specified number of lines.
docker attach *name*	Enters the shell of a specified running container.
docker rm *name*	Deletes the container that has the specified name.

Now that you have seen how to use the Docker client and the Docker daemon, you need to learn how to dockerize an application (in this case, a Python application). *Dockerizing* an application means converting the application to run within a Docker container. The first step in dockerizing an application is to create a Dockerfile. A Docker image consists of stacked read-only layers, each of which represents a Dockerfile instruction. The Dockerfile instructions are laid out using layers, as described in Table 6-4.

Table 6-4 *Dockerfile Instructions*

Instruction	Usage	Description
FROM	**FROM** *image* **FROM** *image*:*tag* **FROM** *image*@*digest*	This is the first non-comment instruction in the Dockerfile. Can appear multiple times within a single Dockerfile in order to create multiple images. The *tag* and *digest* values are optional. If they are omitted, the Docker builder assumes the latest values by default.
LABEL	**LABEL** *key*=*value* [*key*=*value*] . . .	The **LABEL** instruction adds metadata to an image.
RUN	**RUN** *command* **RUN** "*executable*", "*param1*", "*param2*"	**RUN** allows you to run commands using a base image that does not contain the specified shell executable. The default shell can be modified using the **SHELL** command
CMD	**CMD** "*executable*", "*param1*","*param2*" **CMD** *command param1 param2*	There can be only one **CMD** instruction in a Dockerfile. The **CMD** instruction provides defaults for an executing container. These defaults can include an executable or can commit the executable. The defaults specified with **CMD** can be overridden by a user specifying arguments to the Docker **RUN** command.
ADD	**ADD** *src src . . . dest* **ADD** "*src*","*src*" . . . "*dest*" (used in cases where paths include whitespace)	**ADD** copies new files, directories, or remote file URLs from *src* and adds them to the file system of the image at the path *dest*. If *src* is a file or directory, then it must be relative to the source directory that is being used while building the Docker image. *dest* is an absolute path or a path relative to **WORKDIR**.
COPY	**COPY** *src src . . . dest* **COPY** "*src*","*src*" . . . "*dest*" (used in cases where paths include whitespace)	**COPY** copies new files, directories, or remote file URLs from *src* and adds them to the filesystem of the image at the path *dest*. If *src* is a file or directory, then it must be relative to the source directory that is being used while building the Docker image. *dest* is an absolute path or a path relative to **WORKDIR**. **COPY** works the same as **ADD** but without the **tar** and remote URL handling.
ENTRYPOINT	**ENTRYPOINT** "*executable*", "*param1*","*param2*" **ENTRYPOINT** *command param1 param2*	**ENTRYPOINT** allows a user to configure a container that will run as an executable. It overrides all elements specified under the **CMD** instruction.
VOLUME	**VOLUME** "*path*", . . . **VOLUME** *path* [*path* . . .]	**VOLUME** is used to expose any database storage area, configuration storage, or files/folders created by the Docker container.
USER	**USER** [*username* | *UID*]	**USER** allows you to set the *username* or *UID* to use when running an image and for any **RUN**, **CMD**, and **ENTRYPOINT** instructions that follow in the Dockerfile.
WORKDIR	**WORKDIR** *path-to-workdir*	**WORKDIR** sets the working directory for any **RUN**, **CMD**, **COPY**, **ADD**, and **ENTRYPOINT** instructions specified in the Dockerfile. It can be used multiple times in one Dockerfile.
ONBUILD	**ONBUILD** *Dockerfile INSTRUCTION*	**ONBUILD** is executed only after the current Dockerfile build is complete. Any *INSTRUCTION* specified under **ONBUILD** executes in any child image derived from the current image.
ENV	**ENV** *key*=*value* [*key*=*value* ...] **ENV** *key value*	The **ENV** instruction sets the environment variable *key* to the value *value*. Environment variables defined using **ENV** always override an **ARG** instruction with the same key value.
ARG	**ARG** *name*=*default-value*	**ARG** defines a variable that you can pass at build time, such as when using the **docker build** command with the **--build-arg** *varname*=*value* flag.

Example 6-8 illustrates the process of dockerizing a Python application that uses the **numpy** library. In this example, the **FROM** instruction set specifies the use of Python 3 for the Dockerfile, adds the hist.py file to the Dockerfile by using the **ADD** instruction, runs the command **pip3 install numpy** by using **RUN** instruction, and executes the Python code by using the **python3** command specified under the **CMD** instruction. Once the Dockerfile is built, the command **docker build** is used to create a Dockerfile named **plotter**. When the docker image is built, the command **docker run plotter**, where **plotter** is the name of the container, is used to run the container.

Example 6-8 *Dockerizing a Python Application*

```
! Creating a Dockerfile

[root@rnd-srvr plotter]# touch Dockerfile


! Dockerfile is edited with the following contents

FROM python:3

ADD hist.py /

RUN pip3 install numpy

CMD [ "python3", "./hist.py" ]
```

```
! hist.py File
```

```
import numpy as np


greyhounds = 10


grey_height = 28 + 4 * np.random.randn(greyhounds)


print(grey_height)
```

```
[root@rnd-srvr plotter]# docker build -t plotter .
Sending build context to Docker daemon 3.072 kB
Step 1/4 : FROM python:3
Trying to pull repository docker.io/library/python ...
3: Pulling from docker.io/library/python
8f0fdd3eaac0: Pull complete
d918eaefd9de: Pull complete
43bf3e3107f5: Pull complete
27622921edb2: Pull complete
dcfa0aalae2c: Pull complete
bf6840af9e70: Pull complete
21f900120cf5: Pull complete
644b4ceca849: Pull complete
50f0ac11639a: Pull complete
Digest: sha256:58666f6a49048d737eb24478e8dabce32774730e2f2d0803911a2c1f61c1b805
Status: Downloaded newer image for docker.io/python:3
 ---> 038a832804a0
Step 2/4 : ADD hist.py /
 ---> alaa56c4a003
Removing intermediate container 8a0ffc6ae9e7
Step 3/4 : RUN pip3 install numpy
[171502.757410] docker0: port 1(vethe9cc10a) entered blocking state
[171502.761254] docker0: port 1(vethe9cc10a) entered disabled state
 ---> Running in 2d2c56d17289
[171502.778704] device vethe9cc10a entered promiscuous mode
[171502.780834] IPv6: ADDRCONF(NETDEV_UP): vethe9cc10a: link is not ready
[171502.782820] docker0: port 1(vethe9cc10a) entered blocking state
[171502.784738] docker0: port 1(vethe9cc10a) entered forwarding state
[171502.796246] docker0: port 1(vethe9cc10a) entered disabled state


[171502.934059] SELinux: mount invalid.  Same superblock, different security
settings for (dev mqueue, type mqueue)
[171503.001974] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[171503.005056] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[171503.007321] IPv6: ADDRCONF(NETDEV_CHANGE): vethe9cc10a: link becomes ready
[171503.009437] docker0: port 1(vethe9cc10a) entered blocking state
[171503.011278] docker0: port 1(vethe9cc10a) entered forwarding state
Collecting numpy
  Downloading
https://files.pythonhosted.org/packages/f5/4d/cbea29c189e2a9c5d3e2d76307be15f7f86
4a073cdb6c1abbc8e4311afbc/numpy-1.18.0-cp38-cp38-manylinux1_x86_64.whl (20.6MB)
Installing collected packages: numpy
Successfully installed numpy-1.18.0
[171534.064076] docker0: port 1(vethe9cc10a) entered disabled state
[171534.079041] docker0: port 1(vethe9cc10a) entered disabled state
[171534.082044] device vethe9cc10a left promiscuous mode
```

```
[171534.083874] docker0: port 1(vethe9cc10a) entered disabled state
 ---> f36b3a2a503a
Removing intermediate container 2d2c56d17289
Step 4/4 : CMD python3 ./hist.py
 ---> Running in cd9f08d4e70c
 ---> d90c530f9c87
Removing intermediate container cd9f08d4e70c
Successfully built d90c530f9c87
```

```
[root@rnd-srvr plotter]# docker run plotter
[172184.450904] docker0: port 1(vethc2445d5) entered blocking state
[172184.453092] docker0: port 1(vethc2445d5) entered disabled state
[172184.457887] device vethc2445d5 entered promiscuous mode
[172184.459767] IPv6: ADDRCONF(NETDEV_UP): vethc2445d5: link is not ready
[172184.461788] docker0: port 1(vethc2445d5) entered blocking state
[172184.463616] docker0: port 1(vethc2445d5) entered forwarding state
[172184.475302] docker0: port 1(vethc2445d5) entered disabled state
[172184.597108] SELinux: mount invalid.  Same superblock, different security
settings for (dev mqueue, type mqueue)
[172184.643800] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[172184.645876] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[172184.647860] IPv6: ADDRCONF(NETDEV_CHANGE): vethc2445d5: link becomes ready
[172184.649851] docker0: port 1(vethc2445d5) entered blocking state
[172184.651668] docker0: port 1(vethc2445d5) entered forwarding state
[30.24019652 25.82537121 29.06987139 23.12124747 26.15472775 24.82905669
 24.32525392 26.372184   24.0266377  25.66512881]
[root@rnd-srvr plotter]# [172185.041128] docker0: port 1(vethc2445d5) entered
disabled state
[172185.049379] docker0: port 1(vethc2445d5) entered disabled state
[172185.051577] device vethc2445d5 left promiscuous mode
[172185.053236] docker0: port 1(vethc2445d5) entered disabled state


[root@rnd-srvr plotter]#
```

Note

In the event that the repository that holds the packages you want to download resides behind a proxy server, other system and environment variables have to be updated. You use the **ENV** command in the Dockerfile as shown here to update the system variables:

```
ENV http_proxy http://proxy-srvr.xyz.com:8080/
ENV https_proxy http://proxy-srvr.xyz.com:8080/
```

The virtualenv Tool

Python, like most other languages, has a unique way of storing and managing packages. Even though it's all if everything is stored and maintained properly, it becomes challenging to maintain packages that are solely used on a project-by-project basis. Once an application is developed and ready for deployment, it is hard to remember which packages and versions are required. Also, it is challenging to keep the development environment separate from the production environment. To overcome these challenges, you can use **virtualenv**.

virtualenv is a tool for creating isolated Python virtual environments to manage Python packages for different projects. This means that each project can have its own dependencies, regardless of the dependencies of the system or other projects running in different virtual environments. In addition, these virtual environments allow programmers to maintain separate Python versions for individual projects. Example 6-9 demonstrates how to install **virtualenv** by using **pip3** package manager.

Example 6-9 *Installing* ***virtualenv***

```
[root@node2 ~]# pip3 install virtualenv
WARNING: Running pip install with root privileges is generally not a good idea.
Try `pip3 install --user` instead.
Collecting virtualenv
  Downloading
https://files.pythonhosted.org/packages/05/f1/2e07e8ca50e047b9cc9ad56cf4291f4e041
fa73207d000a095fe478abf84/virtualenv-16.7.9-py2.py3-none-any.whl (3.4MB)
```

```
Installing collected packages: virtualenv

Successfully installed virtualenv-16.7.9

[root@rnd-srvr project]#
```

To create a virtual environment with **virtualenv**, you use the command **virtualenv** *name*. This command creates a directory with the specified name and installs Python packages into that directory. If you specify a dot (.) instead of a name, the current directory is used to set up the virtual environment. Inside the directory, four subdirectories are created:

- **bin**: Contains the files that interact with the virtual environment.
- **include**: Contains the C headers that compile the Python packages.
- **lib**: Holds a copy of the Python version along with a site packages folder where each dependency is installed.
- **lib64**: Holds a copy of the Python version along with a site packages folder where each dependency is installed.

Note

lib64 site packages are preferred over **lib** site packages on 64-bit systems.

Another way to create a virtual environment is by using the **python3** command. You can use the command **python3 -m venv** *name* to create a virtual environment with the specified name. Example 6-11 shows how to create a virtual environment using both the **virtualenv** command and the **python3 -m venv** command. To use the virtual environment, you need to call the **activate** script, which is located in the **bin** directory of the virtual environment. You can do this by using the command **source** *venv*/**bin/activate**, where *venv* is the name of the virtual environment. In Example 6-11, notice that the changes made to the site packages within the virtual environment are not reflected to the global packages, so you can maintain separate environments for different projects.

Example 6-11 *Creating Virtual Environments*

```
[root@rnd-srvr project]# virtualenv env

Using base prefix '/usr'

  No LICENSE.txt/LICENSE found in source

New python executable in /root/project/env/bin/python3

Also creating executable in /root/project/env/bin/python

Installing setuptools, pip, wheel...


done.

[root@rnd-srvr project]#

[root@rnd-srvr project]# source env/bin/activate

(env) [root@rnd-srvr project]# python --version

Python 3.6.8

(env) [root@rnd-srvr project]# python3 --version

Python 3.6.8

(env) [root@rnd-srvr project]# deactivate

[root@rnd-srvr project]# python --version

Python 2.7.5

[root@rnd-srvr project]# python3 --version

Python 3.6.8
```

```
[root@rnd-srvr project]# python3 -m venv newenv

[root@rnd-srvr project]# ls

env  newenv  test.py

[root@rnd-srvr project]# source newenv/bin/activate

(newenv) [root@rnd-srvr project]# python --version

Python 3.6.8

(newenv) [root@rnd-srvr project]# pip3 install --upgrade pip

Cache entry deserialization failed, entry ignored

Collecting pip

  Downloading

https://files.pythonhosted.org/packages/00/b6/9cfa56b4081ad13874b0c6f96af8ce16cfb

c1cb06bedf8e9164ce5551ec1/pip-19.3.1-py2.py3-none-any.whl (1.4MB)


Installing collected packages: pip
```

```
  Found existing installation: pip 9.0.3
    Uninstalling pip-9.0.3:
      Successfully uninstalled pip-9.0.3
Successfully installed pip-19.3.1
(newenv) [root@rnd-srvr project]# pip3 --version
pip 19.3.1 from /root/project/newenv/lib64/python3.6/site-packages/pip (python
3.6)
(newenv) [root@rnd-srvr project]# deactivate
[root@rnd-srvr project]# pip3 --version
pip 9.0.3 from /usr/lib/python3.6/site-packages (python 3.6)
[root@rnd-srvr project]#
```

Note

It is possible to use **pipenv** instead of **virtualenv**. **pipenv** automatically creates and manages a virtual environment for projects and allows you to manage packages from a pipfile (that is, add or remove packages as they are installed or uninstalled). With **pipenv**, you are not required to use **pip**/**pip3** and **virtualenv** separately.

Python Modules

A Python *module* is a file that contains Python code with definitions and statements. Python modules allow you to logically organize code by grouping classes, functions, and variables; modules also help you write reusable code. A Python module can be used and called by using the **import** statement, which takes a list of modules as parameters, as shown here:

```
import module1 [, module2 [, . . . , module]
```

When Python code is being executed and the interpreter encounters an **import** statement, it imports the module into the current code so that the definitions and the statements part of the module become accessible to the current code. Example 6-12 shows an example of using the module **mymod**, which is defined in the file mymod.py. The **mymod** module defines the function **say_hello()**, which takes a name (as a string) as the parameter. When the **mymod** module is imported into the test.py file, the **say_hello()**function becomes accessible in test.py, and you can call it by referencing the module name as shown in Example 6-12. Note that the module is loaded only once, regardless of the number of times it has been imported in the Python code. Once it is loaded, the definitions and statements within the module are accessible throughout the program.

Example 6-12 *Using Modules with **import** Statements*

```
! mymod.py in /root/python directory
[root@rnd-srvr python]# cat mymod.py


def say_hello(name):
        print('Hello ' + name)
```

```
! test.py in /root/python directory
[root@rnd-srvr python]# cat test.py
import mymody


mymody.say_hello('John')
```

```
[root@rnd-srvr python]# python3 test.py
Hello John
```

The **import** statements import all the methods and classes into the Python interpreter. In Python, a module is always fully imported into the **sys.modules** mapping. Using a **from-import** statement in Python is another way of importing a module. The main difference when using a **from-import** statement is that it binds a name pointing directly to the attribute contained in the module. The complete module still gets imported using this **import** statement format. The **from-import** statement takes the list of functions/definitions as parameters, as shown here:

```
from module-name import name1 [, name2 [, . . . , nameN]
```

Note

To import all non-private names without having to specify them using commas (,), you can use the wildcard character * after the **import** keyword, as shown here:

```
from module-name import *
```

Example 6-13 demonstrates the use of a **from-import** statement. Note that in this example, the module **mymod** is present in the directory named **hello**, and the module has functions named **say_hello()** and **say_bye()**. The **from-import** statement in this example directly binds a reference to the **say_hello()** function, but the rest of the function is still accessible to a programmer who wants to access it.

Example 6-13 *Using a **from-import** Statement*

```
! mymod.py in /root/python/hello directory
[root@rnd-srvr python]# cat hello/mymod.py
```

```
def say_hello(name):
        print('Hello ' + name)


def say_bye(name):
        print('Bye ' + name)
```

```
! test.py in /root/python directory
[root@rnd-srvr python]# cat test.py
from hello.mymod import say_hello


say_hello('John')
```

```
[root@rnd-srvr python]# python3 test.py
Hello John
```

Another way to implement modules is by using classes. Example 6-14 illustrates the use of a class defined in a module. **__init__** is a reserved method in Python classes that is used as a constructor (which is a concept in object-oriented programming). The method is called when an instance of the class is created that allows the class to initialize its attributes. The **__init__** method can be called with just **self** as the argument or along with extra arguments. The **self** keyword represents an instance of the class and helps in accessing the attributes and methods of the class in Python. In Example 6-14, a constructor of the **Hello** class is created with the string parameter **name**. When the constructor is called, it sets the name attribute of the **Hello** class to the value specified in the parameter. Thus, when **mymodule.Hello()** is called with **'John'** as the parameter, the **name** attribute of class **Hello** is set to the value **'John'**. Since an object is created here by calling the constructor named **test** for the class **Hello**, you can now access the method **say_hello()** by using the same object.

Example 6-14 *A Python Module with a Class and a Constructor*

```
! mymodule.py in /root/python/hello directory
[root@rnd-srvr python]# cat hello/mymodule.py


class Hello:

    def __init__(self, name):
        self.name = name


    def say_hello(self):
        print('Hello ' + self.name)
```

```
! test.py in /root/python directory
[root@rnd-srvr python]# cat test.py
from hello import mymodule


test = mymodule.Hello('John')
test.say_hello()
```

```
[root@rnd-srvr python]# python3 test.py
Hello John
```

From the examples in this section, it is clear that modules make it possible to reuse the code within a project. The next section dives into various applications of the Python programming language.

Python Applications

The use of Python has increased greatly in the past several decades. Due to its emphasis on the DRY (Don't Repeat Yourself) principle and ease of readability, Python has been adopted by developers across multiple application domains. Because of Python's wide acceptance, the Python Package Index lists thousands of third-party modules that can be used to develop robust, highly scalable, and secure applications. Today Python is being widely used in multiple application domains, including the following:

- Web application development
- Desktop applications
- Business applications (ERP, CRM, and so on)
- Machine learning and artificial intelligence
- Data science and data visualization
- Automation and orchestration
- Audio and video applications

The following sections discuss some of these domains.

Web/API Development

A web application is a client/server program in which the client runs on the browser. The application is hosted on a remote server and delivered over the Internet or across the network through a browser interface. A web application has two major components:

• **Front end**: Refers to the user interface, where the information or data is displayed to the user.

• **Back end**: Refers to the application and database where the logic is implemented for the actions that need to be performed on the data and how the data will be sent to the front-end application.

Some web applications are also developed with three-layer architecture, which has a front end for user interaction, a back end for databases, and a middle layer for logic and validation. Multiple web development frameworks, including these, make it possible to develop web application using Python:

• Django

• Pyramid

• Flask

• Bottle

• FastAPI

The following sections discuss how to install and use Django and Flask for web/API development.

Django

Django is a high-level web framework based on Python that makes it possible to rapidly develop scalable and secure web applications. Django follows a pragmatic design that allows developers to incrementally add functionality to a web application without having to impact other components or other sections of the code. It takes care of most of the web development hassle and allows developers to focus on writing modular apps. In addition, Django has support for multiple databases, such as MySQL and Postgres, which makes it easier to build a back-end databases and logic without having users to directly interact with the database itself. At the time of writing, some high-profile and highly scalable websites—such as Instagram, National Geographic, and OpenStack—are using Django in their back ends to some extent.

Currently, Django supports its users with three different trains. The latest versions of Django are 3.0.3, 2.2.10, and 1.11.28. The Django web framework can be installed using the Python package manager, **pip** or **pip3**. Once Django is installed, you can start creating projects by issuing the command **django-admin startproject** *project-name*. Example 6-15 shows the installation of Django Version 2.2.10 in a virtual environment and the creation of a new project.

Example 6-15 *Installing Django and Starting a NewDjango Project*

```
[root@rnd-srvr opt]# virtualenv web

[root@rnd-srvr opt]# source web/bin/activate

(web) [root@rnd-srvr opt]# pip3 install django==2.2.10

Collecting django==2.1.1

  Downloading

https://files.pythonhosted.org/packages/ca/7e/fc068d164b32552ae3a8f8d5d0280c083f2

e8d553e71ecacc21927564561/Django-2.2.10-py3-none-any.whl (7.3MB)

     |████████████████████████████████| 7.3MB 3.8MB/s

Requirement already satisfied: pytz in ./web/lib/python3.6/site-packages (from

django==2.2.10) (2019.3)

Installing collected packages: django

Successfully installed django-2.2.10


! Creating a project in Django

(web) [root@rnd-srvr opt]# django-admin startproject Demo1
```

Every Django project starts with a manage.py file in its project root directory. The **manage.py** script allows you to run administrative tasks such as Django's **django-admin**. As mentioned earlier, Django applications allow you to interact with the databases. The database tables are defined using models. Updating models created in Django on the database requires migrations. Django migrations allows for propagation of changes made to the models into the database schema. These migrations are automatic, but it is important for developers to understand and know when to make these migrations. The following commands can be used with manage.py as parameters to perform migration operations:

• **migrate**: Applies migrations.

• **makemigrations**: Makes new migrations, based on changes made to the models.

• **sqlmigrate**: Displays SQL statements for migrations.

• **showmigrations**: Lists projects' migrations and their status.

When migrations are performed for an application, the next step is to test the application. For testing web applications or web APIs, Django comes with its own web server. You can invoke Django's web server by using the **runserver** command to manage.py. The **runserver** command can be executed with options such as *IP-address:port*, where *IP-address* can be the IP address of one of the physical network interface cards (NICs) or 0.0.0.0, which allows client machines to send requests on any IP address configured on the server.

Example 6-16 illustrates how to perform migrations and test a web application by enabling Django's web server on port 8080.

Example 6-16 *Performing Django Migrations and Running a Django Application Web Server*

```
(web) [root@rnd-srvr Demo1]# pwd
```

```
/opt/Demo1

(web) [root@rnd-srvr Demo1]# ls

db.sqlite3  Demo1  manage.py

(web) [root@rnd-srvr Demo1]# python manage.py migrate

Operations to perform:

  Apply all migrations: admin, auth, contenttypes, sessions

Running migrations:

  Applying contenttypes.0001_initial... OK

  Applying auth.0001_initial... OK

  Applying admin.0001_initial... OK

  Applying admin.0002_logentry_remove_auto_add... OK

  Applying admin.0003_logentry_add_action_flag_choices... OK

  Applying contenttypes.0002_remove_content_type_name... OK

  Applying auth.0002_alter_permission_name_max_length... OK

  Applying auth.0003_alter_user_email_max_length... OK

  Applying auth.0004_alter_user_username_opts... OK

  Applying auth.0005_alter_user_last_login_null... OK

  Applying auth.0006_require_contenttypes_0002... OK

  Applying auth.0007_alter_validators_add_error_messages... OK

  Applying auth.0008_alter_user_username_max_length... OK

  Applying auth.0009_alter_user_last_name_max_length... OK

  Applying sessions.0001_initial... OK

(web) [root@rnd-srvr Demo1]# python manage.py runserver 172.16.102.134:8080

Performing system checks...


System check identified no issues (0 silenced).

January 03, 2020 - 08:55:04

Django version 2.1.1, using settings 'Demo1.settings'

Starting development server at http://172.16.102.134:8080/

Quit the server with CONTROL-C.

Invalid HTTP_HOST header: '172.16.102.134:8080'. You may need to add

'172.16.102.134' to ALLOWED_HOSTS.

Bad Request: /

[03/Jan/2020 08:55:13] "GET/HTTP/1.1" 400 59543
```

Note

Multiple command options are available for use with manage.py. To get a list of all the available commands, use the command **python3 manage.py –help**.

Notice that in Example 6-16, even though the web application is running on the server at port 8080, any request made to the server is treated as a bad request. This is due to the permissions and settings on the Django application. To change the settings of the Django project, you need to edit the settings.py file that is generated for each project. The settings.py file resides in the project directory and contains all the configurations for a Django installation. The settings.py file includes the following sections:

• **Core settings**: Contains settings such as allowed hosts, admin settings, and absolute URL overrides.

• **Auth**: Contains settings for Django authentication models and back ends.

• **Messages**: Contains settings for Django messages.

• **Sessions**: Contains settings for managing sessions in a Django application.

• **Static files**: Contains settings for static URLs, static file directories, and so on.

The bad request shown in Example 6-16 can be resolved by changing the **ALLOWED_HOSTS** settings in the settings.py file as shown in Example 6-17. This setting takes in a list of strings representing the host or domain names that the Django site can serve. The default setting for this field is an empty list ([]), or the field can be set to a particular IP address or a domain name, such as *example.com* or even * (which means the application will match any value). Example 6-17 shows that changing the settings.py file so that it matches any host allows the application to respond to the **GET** request coming from the HTTP client.

Example 6-17 *Editing Changes in the Settings.py File and Running a Django Application*

```
 (web) [root@rnd-srvr Demo1]# cd Demo1/

(web) [root@rnd-srvr Demo1]# ls

__init__.py  settings.py  urls.py  wsgi.py
```

! Edit the settings.py file to allow All hosts using '*' or specific hosts by specifying the IP address range

```
(web) [root@rnd-srvr Demo1]# cat settings.py
"""
Django settings for Demo1 project.

Generated by 'django-admin startproject' using Django 2.1.1.

For more information on this file, see
https://docs.djangoproject.com/en/2.1/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/2.1/ref/settings/
"""

import os

# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))



# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'i69x735220ef*qz^qr&ix0bw8-bwp@^))5u1x134ztd)&*t@_5'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = ['*']

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

! Output omitted for brevity
```

```
(web) [root@rnd-srvr Demo1]# python manage.py runserver 172.16.102.134:8080
Performing system checks...

System check identified no issues (0 silenced).
January 03, 2020 - 08:59:34
Django version 2.1.1, using settings 'Demo1.settings'
Starting development server at http://172.16.102.134:8080/
```

```
Quit the server with CONTROL-C.

[03/Jan/2020 08:59:41] "GET/HTTP/1.1" 200 16348

[03/Jan/2020 08:59:41] "GET /static/admin/css/fonts.css HTTP/1.1" 200 423

Not Found: /favicon.ico

[03/Jan/2020 08:59:41] "GET /favicon.ico HTTP/1.1" 404 1976

[03/Jan/2020 08:59:41] "GET /static/admin/fonts/Roboto-Bold-webfont.woff

HTTP/1.1" 200 82564

[03/Jan/2020 08:59:41] "GET /static/admin/fonts/Roboto-Regular-webfont.woff

HTTP/1.1" 200 80304

[03/Jan/2020 08:59:41] "GET /static/admin/fonts/Roboto-Light-webfont.woff

HTTP/1.1" 200 81348
```

When you access a Django application in a browser, if you see a web page like the one shown in Figure 6-2, you know that the Django framework was successfully installed.

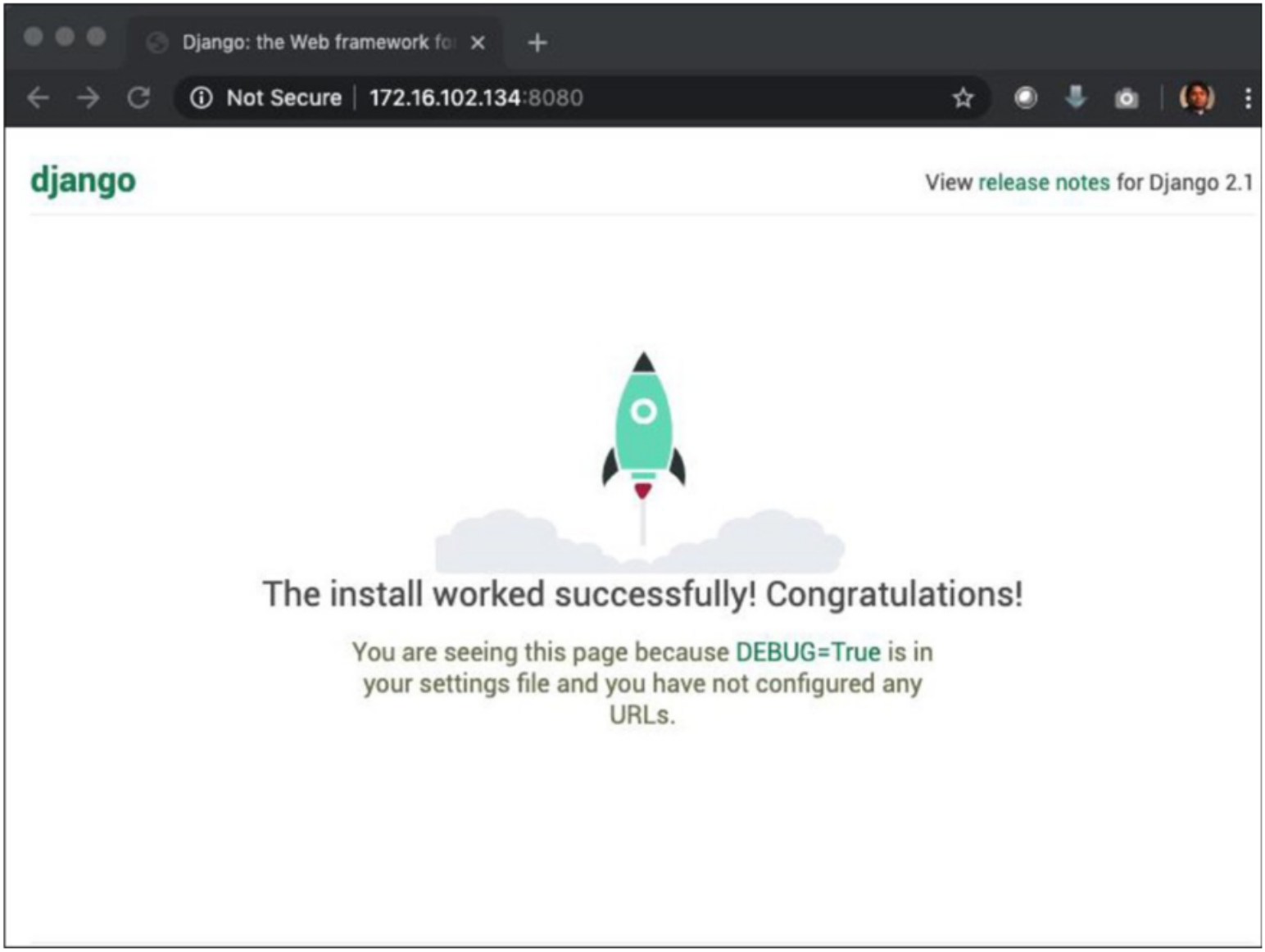


Figure 6-2 *Accessing a Django Application in a Web Browser*

The Django framework can be used to build a web application, but Django Rest Framework (DRF) is very handy for creating APIs. It is possible to build APIs using the Django framework, but DRF provides several useful functions, such as serializers, filtering, and OAuth support. DRF allows developers to create RESTful APIs and use serializers to convert HTTP requests into valid Django objects and vice versa when a response is received from a data source. Example 6-18 shows the installation of DRF and also illustrates how to create an app inside a Django project by using the command **django-admin startapp** *app-name*. After an app is created, in order to use DRF, both *app-name* and *rest_framework* should be added to the **INSTALLED_APPS** list in the settings.py file.

Example 6-18 *Installing DRF and Creating a Django App*

```
(web) [root@rnd-srvr Demo1]# pip3 install djangorestframework

Collecting djangorestframework

  Downloading

https://files.pythonhosted.org/packages/be/5b/9bbde4395a1074d528d6d9e0cc161d3b99b

d9d0b2b558ca919ffaa2e0068/djangorestframework-3.11.0-py3-none-any.whl (911kB)

Requirement already satisfied: django>=1.11 in /opt/web/lib/python3.6/site-
```

```
packages (from djangorestframework) (2.1.1)
Requirement already satisfied: pytz in /opt/web/lib/python3.6/site-packages (from
django>=1.11->djangorestframework) (2019.3)
Installing collected packages: djangorestframework
Successfully installed djangorestframework-3.11.0
(web) [root@rnd-srvr Demo1]# django-admin startapp demoapp
(web) [root@rnd-srvr Demo1]# ls
db.sqlite3  Demo1  demoapp  manage.py
```

```
! Edit the INSTALLED_APPS sections in settings.py file inside Demo1 Directory.
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'demoapp',
    'rest_framework',
]
```

The next step is to build the back end and logic for **demoapp**. Each app within Django contains the following files:

• **models.py**: This file defines the structure of the user data.

• **serializer.py**: This file uses Serializer to allow complex data such as query sets and model instances to be converted into native Python data types. This data can be then rendered into XML or JSON formats.

• **views.py**: This file allows you to define views or view sets. Views are Python functions that take in web requests and return web responses.

• **urls.py**: This file, which is present in both the project directory and the app directory, allows you to define a URL route for the app as well as the project.

Example 6-19 illustrates a basic employee model that takes in first name, last name, and data creation time as input and saves that in the back-end database. In this example, notice the use of the dot (**.**) in the **import** statement; this dot indicates a relative import, starting with the current package.

Example 6-19 *Building a Demo App*

```
! models.py


from django.db import models
from django.utils import timezone


class Employee(models.Model):
    firstName = models.CharField(max_length=400)
    lastName = models.CharField(max_length=400)
    created_at = models.DateTimeField(default=timezone.now)
```

```
! serializer.py


from rest_framework import serializers
from .models import Employee


class EmpSerializer(serializers.HyperlinkedModelSerializer):


    class Meta:
        model = Employee
        fields = ('id', 'firstName', 'lastName', 'created_at')
```

```
! views.py


from django.shortcuts import render
```

```
from rest_framework import viewsets

from .models import Employee

from .serializer import EmpSerializer


class EmpViewSet(viewsets.ModelViewSet):


    queryset = Employee.objects.all()

    serializer_class = EmpSerializer
```

```
! urls.py under demoapp


from rest_framework import routers

from . import views


router = routers.DefaultRouter()

router.register(r'Employees', views.EmpViewSet)


# Wire up our API using automatic URL routing.

# Additionally, we include login URLs for the browsable API.

urlpatterns = [

    path('', include(router.urls))
```

```
! urls.py under Demo1 project folder


from django.contrib import admin

from django.urls import path, include


urlpatterns = [

    path('admin/', admin.site.urls),

    path('api/', include('demoapp.urls')),
```

Note

The details of how to write code in each file are beyond the scope of this book. The goal of this example is to illustrate how a web application/web API can be developed using Django and DRF. To read more about Django, see www.djangoproject.com.

When the demo app is complete, you execute the **runserver** command again and test the API either in a browser or by using a tool such as Postman.

Note

Postman is a collaboration platform for API development. It allows you to test the APIs without having to write the code for the front end and visualize what the data response is going to look like from the server.

Flask

Flask is a Python-based microframework for building web applications and APIs. It is called a *microframework* because it does not provide a database abstraction layer or validation layer as Django does. Like most other Python libraries, the Flask package can be installed from the Python Package Index (PyPI). You may also want to install **flask-sqlalchemy** in order to allow a Flask application to interact with the SQL database. Example 6-20 shows the installation of the Flask package in a virtual environment. Note that there are a few other packages that get installed as part of the Flask installation.

Example 6-20 *Installing Flask in a Virtual Environment*

```
[root@rnd-srvr opt]# virtualenv flask

Using base prefix '/usr'

  No LICENSE.txt/LICENSE found in source

New python executable in /opt/flask/bin/python3

Also creating executable in /opt/flask/bin/python

Installing setuptools, pip, wheel...

done.

[root@rnd-srvr opt]# source flask/bin/activate

(flask) [root@rnd-srvr opt]#
```

```
(flask) [root@rnd-srvr opt]# pip3 install flask
Collecting flask
  Downloading
https://files.pythonhosted.org/packages/9b/93/628509b8d5dc749656a9641f4caf13540e2
cdec85276964ff8f43bbb1d3b/Flask-1.1.1-py2.py3-none-any.whl (94kB)

Collecting itsdangerous>=0.24
  Downloading
https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d100b3b35c2
239807046a4c953c7b89fa49e/itsdangerous-1.1.0-py2.py3-none-any.whl
Collecting Jinja2>=2.10.1
  Downloading
https://files.pythonhosted.org/packages/65/e0/eb35e762802015cab1ccee04e8a277b03f1
d8e53da3ec3106882ec42558b/Jinja2-2.10.3-py2.py3-none-any.whl (125kB)

Collecting click>=5.1
  Downloading
https://files.pythonhosted.org/packages/fa/37/45185cb5abbc30d7257104c434fe0b07e5a
195a6847506c074527aa599ec/Click-7.0-py2.py3-none-any.whl (81kB)

Collecting Werkzeug>=0.15
  Downloading
https://files.pythonhosted.org/packages/ce/42/3aeda98f96e85fd26180534d36570e4d181
08d62ae36f87694b476b83d6f/Werkzeug-0.16.0-py2.py3-none-any.whl (327kB)

Collecting MarkupSafe>=0.23
  Downloading
https://files.pythonhosted.org/packages/b2/5f/23e0023be6bb885d00ffbefad2942bc51a6
20328ee910f64abe5a8d18dd1/MarkupSafe-1.1.1-cp36-cp36m-manylinux1_x86_64.whl
Installing collected packages: itsdangerous, MarkupSafe, Jinja2, click, Werkzeug,
flask
Successfully installed Jinja2-2.10.3 MarkupSafe-1.1.1 Werkzeug-0.16.0 click-7.0
flask-1.1.1 itsdangerous-1.1.0
(flask) [root@rnd-srvr opt]#
```

Unlike with Django, developers can build APIs quickly on Flask. All you need to begin developing an API in Flask is an entry point and a function with the URL route. The entry point of an application is at **__main__**, which indicates a top-level script. **__name__** is a variable that defines whether the script is being run as the main module or as an imported module. In the **__main__** section, you can define the host IP address and port number to access the web application or API and also enable Debugs if required. Once an entry point is defined, the app routing can be done using the **route** directive, which takes in the parameter as the path to access the API; a function is defined beneath this directive. Example 6-21 shows a simple Flask application that runs on port 5000 and prints "My First Flask Application." When the application is executed, you can use the **curl** command to test the response from the web server running on port 5000.

Example 6-21 *A Simple Flask Application*

```
! App.py

#!/opt/flask/bin/python

from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "My First Flask Application"



! Define the entry point of the App
```

```
if __name__ == '__main__':
    app.debug = True
    app.run(host = '0.0.0.0',port=5000)
```

```
(flask) [root@rnd-srvr Demo2]# chmod a+x app.py
(flask) [root@rnd-srvr Demo2]# ./app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 162-511-044
10.24.70.54 - - [05/Jan/2020 07:30:15] "GET/HTTP/1.1" 200 -
10.24.70.54 - - [05/Jan/2020 07:30:15] "GET/HTTP/1.1" 200 -
10.24.70.54 - - [05/Jan/2020 07:30:15] "GET/HTTP/1.1" 200 -
10.24.70.54 - - [05/Jan/2020 07:30:16] "GET /favicon.ico HTTP/1.1" 404 -
```

```
! Testing the API using CURL

[root@node2 ~]# curl -i http://172.16.102.134:5000/
HTTP/1.1 200 OK
Server: Werkzeug/0.16.0 Python/3.6.8
Date: Sun, 05 Jan 2020 07:37:01 GMT
Content-Length: 26
Content-Type: text/html; charset=utf-8
Via: 1.1 sjc5-dmz-wsa-3.cisco.com:80 (Cisco-WSA/X)
Connection: keep-alive
Proxy-Connection: keep-alive

My First Flask Application
[root@node2 ~]#
```

Every application is built around a single purpose: handling data (that is, performing actions such as adding, deleting, or updating data). Because Flask is based on Python, developers can leverage other libraries and packages to perform actions on data. Most modern applications represent data in JSON format because it is easy to manage data in JSON. By using Flask, you can work on either data that is stored in a database table or in-memory data. Example 6-22 shows an example of handling in-memory data and a simple API to get all the JSON data stored in a variable. In this example, the variable **employees** is defined and contains the ID, first name, last name, and title of the employee. Then an API with the path api/v1.0/employees is created; it can be accessed using the HTTP **GET** method, which returns a JSON representation of the data via the **jsonify()** function that is part of the Flask package. Once the app is executed, you can access the employee data by using **curl** and specifying the path of the API.

Example 6-22 *Accessing In-Memory Employee Data*

```
#!/opt/flask/bin/python

from flask import Flask, jsonify

app = Flask(__name__)

employees = [
    {
        'id': 1,
        'firstname': u'John',
        'lastname': u'Doe',
        'title': u'CEO'
```

```
    },
    {
        'id': 2,
        'firstname': u'Jason',
        'lastname': u'Bruch',
        'title': u'CFO'
    }
]


@app.route('/api/v1.0/employees', methods=['GET'])
def get_employees():
    return jsonify({'employees': employees})


if __name__ == '__main__':
    app.debug = True
    app.run(host = '0.0.0.0',port=5000)
```

```
[root@node2 ~]# curl -i http://172.16.102.134:5000/api/v1.0/employees
HTTP/1.1 200 OK
Server: Werkzeug/0.16.0 Python/3.6.8
Date: Sun, 05 Jan 2020 08:35:28 GMT
Content-Length: 242
Content-Type: application/json
Via: 1.1 wsa.xyz.com:80
Connection: keep-alive
Proxy-Connection: keep-alive

{
  "employees": [
    {
      "firstname": "John",
      "id": 1,
      "lastname": "Doe",
      "title": "CEO"
    },
    {
      "firstname": "Jason",
      "id": 2,
      "lastname": "Bruch",
      "title": "CFO"
    }
  ]
}
```

Fetching all the data from a variable is easy, but web applications are usually required to fetch specific data and perform actions on that data. Example 6-23 shows how to get the ID of the employee in the URL and translate it in the *emp_id* argument in the function. With the **get_employee(***emp_id***)** function, a search is performed on the **employee** array. If the ID that is received as the argument does not exist, an HTTP 404 error is received, indicating that the resource is not found. If an entry is found, it is returned and printed in JSON format.

Example 6-23 *Retrieving Data Based on ID*

```
from flask import abort


app = Flask(__name__)


! Output omitted for brevity


@app.route('/api/v1.0/employees/<int:emp_id>', methods=['GET'])
```

```
def get_employee(emp_id):
    emp = [emp for emp in employees if emp['id'] == emp_id]
    if len(emp) == 0:
        abort(404)
    return jsonify({'Employee': emp[0]})
```

```
[root@node2 ~]# curl -i http://172.16.102.134:5000/api/v1.0/employees/1
HTTP/1.1 200 OK
Server: Werkzeug/0.16.0 Python/3.6.8
Date: Sun, 05 Jan 2020 09:19:02 GMT
Content-Length: 110
Content-Type: application/json
Via: 1.1 sjc12-dmz-wsa-5.cisco.com:80 (Cisco-WSA/X)
Connection: keep-alive
Proxy-Connection: keep-alive

{
  "Employee": {
    "firstname": "John",
    "id": 1,
    "lastname": "Doe",
    "title": "CEO"
  }
}
```

```
[root@node2 ~]# curl -i http://172.16.102.134:5000/api/v1.0/employees/3
HTTP/1.1 404 Not Found
Server: Werkzeug/0.16.0 Python/3.6.8
Date: Sun, 05 Jan 2020 09:20:20 GMT
Content-Length: 232
Content-Type: text/html
Via: 1.1 sjc5-dmz-wsa-3.cisco.com:80 (Cisco-WSA/X)
Connection: keep-alive
Proxy-Connection: keep-alive

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually
please check your spelling and try again.</p>
[root@node2 ~]#
```

The previous examples demonstrate the use of HTTP **GET** methods to fetch data from the web server. To add or update existing data on the server, HTTP **POST** or **PUT** methods are required. Example 6-24 illustrates the creation of an API to perform HTTP **POST** operation on the in-memory employee data. In this example, the **route()** method sets the HTTP method as **POST**. As the name suggests, the role of the **add_employee()** function in Example 6-24 is to add a new employee to the employees list. **request.json** has the request data only if that data came marked as JSON. The code checks whether the request data is present, and if it is present, it checks to ensure that the **'firstName'** field is not missing. If all the necessary fields are present, the function appends the data present in **request.data** to the **employees** list.

Note

When testing **POST** methods in API calls using **curl**, it is important to mention the content type as application/JSON; otherwise, the code will be unable to find any data in **request.json**.

Example 6-24 *Inserting Data by Using the HTTP **POST** Method*

```
from flask import request


@app.route('/api/v1.0/employees', methods=['POST'])
def add_employee():
```

```
    if not request.json or not 'firstName' in request.json:
        abort(400)
    employee = {
        'id': employees[-1]['id'] + 1,
        'firstName': request.json['firstName'],
        'lastName': request.json['lastName'],
        'title': request.json['title']
    }
    employees.append(employee)
    return jsonify({'employee': employee}), 201
```

```
[root@node2 ~]# curl -i -H "Content-Type: application/json" -X POST -d
'{"firstName":"Billy", "lastName":"Mathews", "title":"Software Engineer"}'
http://172.16.102.134:5000/api/v1.0/employees

HTTP/1.1 201 Created
Server: Werkzeug/0.16.0 Python/3.6.8
Date: Sun, 05 Jan 2020 09:35:53 GMT
Content-Length: 126
Content-Type: application/json
Via: 1.1 sjcl2-dmz-wsa-2.cisco.com:80 (Cisco-WSA/X)
Connection: keep-alive
Proxy-Connection: keep-alive

{
  "employee": {
    "firstName": "Billy",
    "id": 3,
    "lastName": "Mathews",
    "title": "Software Engineer"
  }
}
```

```
[root@node2 ~]# curl -i http://172.16.102.134:5000/api/v1.0/employees
HTTP/1.1 200 OK
Server: Werkzeug/0.16.0 Python/3.6.8
Date: Sun, 05 Jan 2020 09:38:05 GMT
Content-Length: 360
Content-Type: application/json
Via: 1.1 sjcl2-dmz-wsa-1.cisco.com:80 (Cisco-WSA/X)
Connection: keep-alive
Proxy-Connection: keep-alive

{
  "employees": [
    {
      "firstname": "John",
      "id": 1,
      "lastname": "Doe",
      "title": "CEO"
    },
    {
      "firstname": "Jason",
      "id": 2,
      "lastname": "Bruch",
```

```
      "title": "CFO"
    },
    {
      "firstName": "Billy",
      "id": 3,
      "lastName": "Mathews",
      "title": "Software Engineer"
    }
  ]
}
```

Other web development frameworks are available as well, but Django and Flask are the most commonly used web development frameworks. You can begin using Flask very quickly and with very few lines of code. With Django, you need to do some homework even for a simple "Hello world" program. However, as an application grows, managing a project and its flow is much simpler in Django than in Flask. Basically, you can use Django to develop large-scale applications, and you can use Flask for lightweight applications or applications that have very low turnaround time.

Network Automation

Another field in which Python has gained a lot of interest in the past few years is network automation. Most companies have huge open budgeted for managing and maintaining network infrastructure. Most network outages occur due to human error when deploying new services or while making changes in the existing environment. In addition, a network operating system may run into software defects that cause massive network outages; it is tricky to identify the problem in such situations. Even if the root cause of an outage is known, the network operations team has to manually check on the network devices from time to time in order to ensure that the network is stable and that known issues are not occurring. Another potentially problematic area of network administration is manual application of patches to devices throughout a network. Manually applying patches can take up to several weeks or even months. All these challenges can be solved with network automation tools, and these tools can also help save time and minimize operational costs.

To build a foundation for network automation, it is important to understand that there are primarily two key players around which the architecture needs to be built:

• Human interface

• Network infrastructure

Automation tasks are created to reduce manual work that requires human interaction. However, automation tasks often require that some inputs as well as validations be performed through human interaction. Most automation tasks involve configuration and monitoring. A human interface may be needed to perform data management, and the data is then provided as an input to the configuration management module, which in turn interacts with the network devices to perform the configuration tasks. Data management can be performed by maintaining data in various formats, such as YAML, JSON, and XML, but the data exchange between the modules or components is usually in JSON or XML format.

A network device collects and maintains live as well as historical data for various features and components. This data can be used by collection tools to provide insights into the device. For instance, a configuration change might change the data flow on the device, and it is important that network operators and administrators be able to see the change and its reflection on the network. Network devices are capable of providing more visibility into the network through **show** commands, historical data, and telemetry data that can be used for various purposes. Data retrieved from telemetry or by using **show** commands and in combination with historical data can be used for further verification and testing and can then be displayed on the user interface, which makes it easier to manage the network and know its state. The data can also be sent to the data management module so that improvements can be made if the data shows signs of problems. This closed-loop architecture ensures that the visibility and data from the network devices can be used to make further changes to devices and improve the state of the network. Figure 6-3 provides a flow diagram that illustrates the closed-loop network architecture.

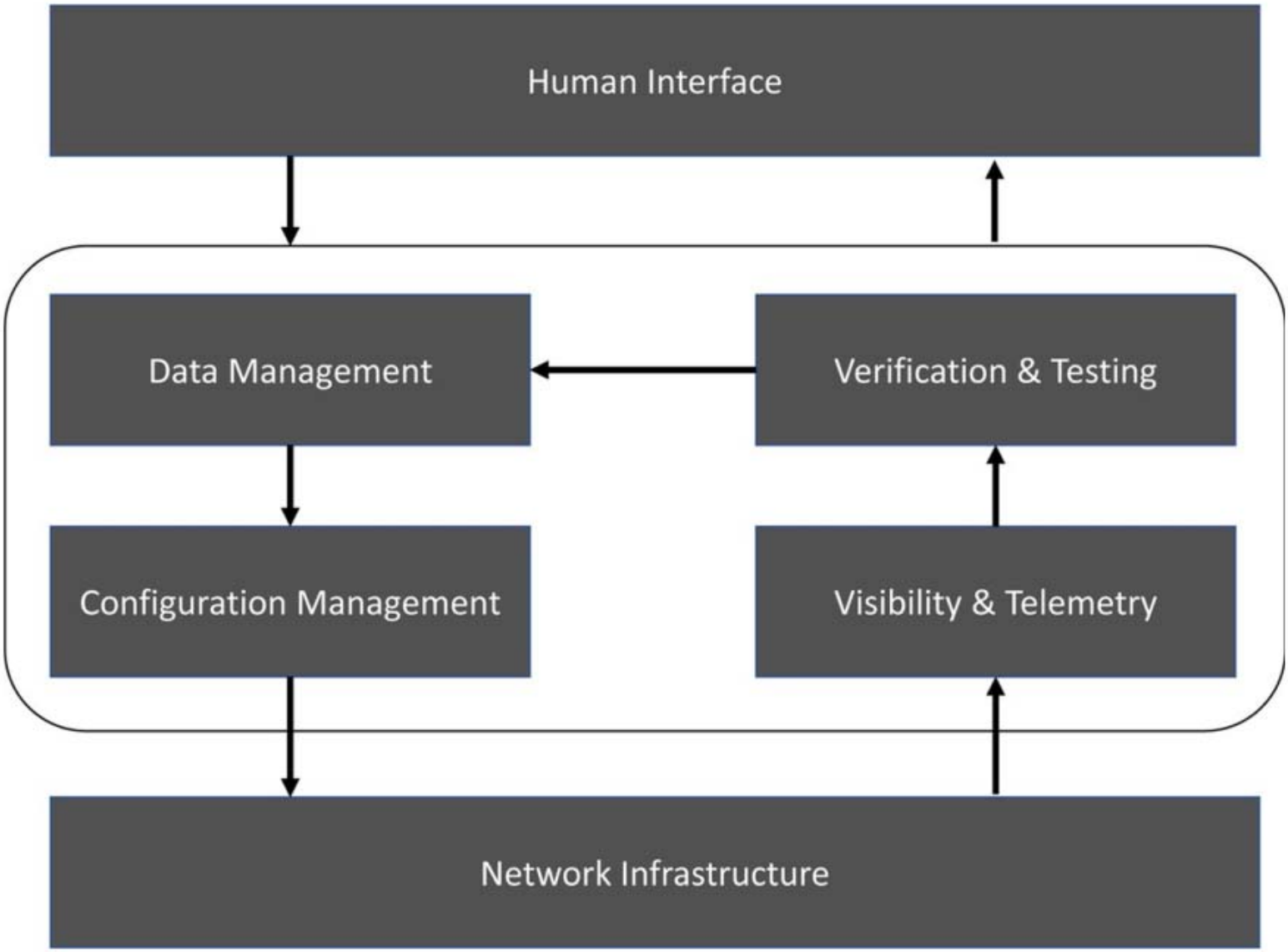


Figure 6-3 *Network Automation Architecture*

Multiple open-source libraries and automation frameworks available on PyPI, including NAPALM and Nornir, contain Python functions that allow you to interact with various network operating systems through a unified API. The next few sections cover various automation frameworks and demonstrate how and where these frameworks can be useful.

NAPALM

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor Support) provides functions that allow you to perform configuration operations such as commit or rollback operations and also to retrieve state data from network devices, regardless of the network operating system. The NAPALM library contains methods to establish connections to network devices and can work in conjunction with automation tools such as Ansible to manage a group of devices in a network at once. The NAPALM library provides support for various network operating systems, including Cisco IOS, IOS XR, and NX-OS; Juniper Junos OS; and Arista EOS. You can install the NAPALM library by using the Python package manager **pip**, using the command **pip install napalm**. Once the library is installed, you can use the **get_network_driver()** function from the NAPALM library to access the various network operating system drivers. Example 6-25 illustrates the use of the NAPALM library to access various network drivers.

Example 6-25 *Accessing Various Network Drivers by Using the NAPALM Library*

```
>>> from napalm import get_network_driver
>>> get_network_driver('ios-xr')
<class 'napalm.iosxr.iosxr.IOSXRDriver'>
>>> get_network_driver('ios')
<class 'napalm.ios.ios.IOSDriver'>
>>> get_network_driver('junos')
<class 'napalm.junos.junos.JunOSDriver'>
```

Note that the NAPALM library requires some preliminary configuration, such as management IP address and SSH configuration, in order to access the devices. Before we can dive in to the fundamentals of building a network automation tool, it is imperative to understand the importance of a networking devices inventory database, which usually contains the following information:

- Device name (hostname)
- Management IP address
- Network operating system
- Username/password

This information can be leveraged to establish a connection to a network device using a matching network driver and authentication

credentials. Network databases can be created in different formats, including JSON and YAML. To illustrate the use of the NAPALM library, this section shows how to build a database in JSON format. Examine the hosts.json file shown in Example 6-26. In this JSON file, two hosts are defined, along with their management IP addresses, the network operating system type, and the username and password. A Python program can read in this JSON-based host database file to determine whether the specified host is a valid host.

Example 6-26 *Building a Host Database in JSON Format*

```
! hosts.json

{

  "xe-1": {

    "IP": "172.16.102.136",

    "type": "ios",

    "user": "cisco",

    "password": "cisco"

    },

    "nx-2": {

    "IP": "172.16.102.137",

    "type": "nxos",

    "user": "admin",

    "password": "admin"

    }

}
```

After defining the host database file, you can fetch the configuration from the network devices. The NAPALM library provides a **get_config()** function that fetches the configuration of a specified network device. Example 6-27 illustrates the use of the NAPALM library to fetch the network configuration. In this example, the Python program takes the parameters **get_config** and **hostname** for the configuration that is to be returned. If the hostname does not match with the host database file, an error is returned. When the hostname specified in the parameter matches the one specified in the host database JSON file, the relevant driver is loaded as per the host database, and a connection to the device is established. Then the program uses the device object to call the **get_config()** method to fetch the configuration from the device.

Example 6-27 *Fetching the Device Configuration by Using the NAPALM **get_config()** Function*

```
! napalm_example.py


import sys

import json

from napalm import get_network_driver




def err_report(*err_list):

    error_msg = ' '.join(str(x) for x in err_list)

    sys.exit(error_msg.rstrip("\n\r"))




if len(sys.argv) != 3:

    err_report("Usage: get_config hostname")


hostname = sys.argv[2]


try:

    with open('hosts.json', 'r') as f:

        device_db = json.load(f)

except (ValueError, IOError, OSError) as err:

    err_report('Could not read the host file: ', err)


try:

    device_info = device_db[hostname.lower()]

except KeyError:

    err_report("Unknown Device '{}'".format(hostname))


driver = get_network_driver(device_info['type'])
```

```
with driver(device_info['IP'], device_info['user'], device_info['password']) as device:
    config = device.get_config()
    print(config['running'])
```

```
(venv) [root@rnd-srvr napalm]# python napalm_example.py get_config xe-1
Building configuration...


Current configuration : 7221 bytes
!
! Last configuration change at 06:57:17 UTC Sun Mar 22 2020
!
version 16.11
service timestamps debug datetime msec
service timestamps log datetime msec
service call-home
platform qfp utilization monitor load 80
no platform punt-keepalive disable-kernel-core
platform console serial
!
hostname XE-1
!
! Output omitted for brevity
```

Note

Errors in the program are handled through **try** and **except** blocks. A **try** block contains the code that may possibly return an error, and the **except** block holds the code to handle the error that is returned.

The NAPALM library can also be used to perform configuration-related operations on the network devices. The following operations can be performed:

- **Replace**: Allows you to replace the existing running configuration with an entirely new configuration.
- **Merge**: Allows you to merge configuration changes from a file to the running configuration on the device.
- **Compare**: Compares the newly proposed configuration with the existing one. Only applies to replace operation and not to merge operation.
- **Discard**: Resets the merge configuration file to an empty file, thus not allowing the new configuration to be applied on the device.
- **Commit**: Commits the proposed configuration to the network device. In other words, used to deploy a staged configuration.
- **Rollback**: Rolls back (reverts) the running configuration to the saved configured prior to the last commit.

All these operations can be achieved through predefined functions in the NAPALM library. Some of these functions are illustrated in Example 6-28. In this example, a new loopback is proposed to be configured from the file config.txt. The newly proposed configuration is a merge candidate for the existing running configuration and can be loaded by using the **load_merge_candidate()** function. This function takes the filename or the configuration itself as the parameter. Once the merged config is loaded, the **compare_config()** function can be used to compare the newly proposed configuration with the existing configuration. (Note that only the delta configuration can be committed to the device.) The configuration can be committed by using the **commit_config()** function and can be discarded by using the **discard_config()** function.

Example 6-28 *Fetching Device Configuration by Using the NAPALM **get_config()** Function*

```
! config.txt
interface loopback100
  ip address 100.1.1.1 255.255.255.255
exit
```

```
! napalm_example.py


! Output omitted for brevity
driver = get_network_driver(device_info['type'])
with driver(device_info['IP'], device_info['user'], device_info['password']) as
device:
    # config = device.get_config()
    # print(config['running'])
    device.load_merge_candidate(filename='config.txt')
    diffs = device.compare_config()
```

```
    if diffs != "":
        print(diffs)
        yesno = input('\nDo you wish to apply the changes? [y/N] ').lower()
        if yesno == 'y' or yesno == 'yes':
            print("Applying changes...")
            device.commit_config()
        else:
            print("Discarding changes...")
            device.discard_config()
    else:
        print("Configuration already present on the device")
        device.discard_config()
```

```
(venv) [root@rnd-srvr napalm]# python napalm_example.py merge_config xe-1


+ interface loopback100
+  ip address 100.1.1.1 255.255.255.255


Do you wish to apply the changes? [y/N] y
Applying changes...
```

By using the NAPALM library, you can easily manage the network devices: You can fetch information and apply changes to the network devices.

Note

Refer to the NAPALM library documentation at https://napalm.readthedocs.io to explore all the available functions in the library.

Nornir

Nornir is a pure Python-based automation framework that takes care of managing the network and host inventory and provides a common framework to write plug-ins for network devices and hosts. The Nornir library requires a minimum Python version of 3.6.2 and can be installed using the Python package manager. An important and interesting fact about Nornir is that it is multithreaded. You can execution tasks simultaneously and in parallel on multiple hosts. Parallelization is triggered by running a task via **nornir.core.Nornir.run** and setting the global variable **num_workers**, which defaults to 20 worker threads. If **num_workers** is set to **1**, the tasks are handled by a single worker thread and are executed one after another, in a simple loop. This can be time-consuming, but it can be very useful during debugging.

Note

The complete Nornir documentation can be found at https://nornir.readthedocs.io.

Now, before we dig into the Nornir framework, it is important to understand the files required to initialize Nornir. Nornir can be initialized by using a configuration file or by using code or by using a combination of both. Along with the configuration file, two other files are referenced within the configuration file:

• Host inventory file

• Group file

Both of these files are YAML based. The host inventory file is similar to the host database, which consists of the hostname, management IP address, port number, username and password, platform, and other information. The group file is used to assign common characteristics to the hosts. The hosts are part of a defined group, and additional characteristics that are common to multiple hosts are assigned to those groups. For instance, there might be 10 hosts in the host inventory file, which may be part of ASN 100, and there might be 5 other hosts that are part of ASN 200. In such a case, rather than define repeated information to 10 hosts, you can group the hosts together, and the ASN value can be assigned within the group. Example 6-29 illustrates how to configure the host.yaml, group.yaml, and config.yaml files that are used to initialize Nornir.

Example 6-29 *Inventory and Configuration Files for Nornir*

```
! hosts.yaml
---
XE-1:
    hostname: 172.16.102.136
    port: 22
    username: cisco
    password: cisco
    platform: ios
    groups:
        - xe-routers
```

```
XR-3:
    hostname: 172.16.102.138
    port: 22
    username: genie
    password: sonpari
    platform: iosxr
    groups:
        - xr-routers


NX-2:
    hostname: 172.16.102.137
    port: 443
    username: admin
    password: admin
    platform: nxos
    groups:
        - nx-routers
```

```
! groups.yaml
---
global:
    data:
        domain: domain.local
        asn: 65000


xe-routers:
    groups:
        - global


nx-routers:
    groups:
        - global


xr-routers:
    data:
        asn: 100
```

```
! config.yaml
---
core:
    num_workers: 20
inventory:
    plugin: nornir.plugins.inventory.simple.SimpleInventory
    options:
        host_file: "hosts.yaml"
        group_file: "groups.yaml"
```

Once the inventory and configuration files are defined, you can initialize Nornir by using the **InitNornir()** function defined under the **nornir** package. You then have the flexibility to use either the NAPALM library part of the Nornir framework or the Netmiko library, which can be used to send commands to the network devices. Example 6-30 demonstrates how to initialize Nornir by using the configuration file. Once Nornir is initialized, you can use the **run()** command to assign tasks for the worker threads. One of the tasks that can be used to gather information from network devices is **napalm_get**, which is defined under **nornir.plugins.tasks.networking**. It has options for following parameters:

• **getters**: Getters are the calls that are made to the devices to fetch information from them.

• **getters_options**: This parameter is used when passing multiple getters.

• ****kwargs**: This parameter specifies any additional arguments required by the getters. ****kwargs** is used to pass a variable-length argument list. The ** before **kwargs** allows you to pass any number of keyword arguments.

The Example 6-30 illustrates the use of the **get_facts** getter, which fetches from a network device information much as **show** commands do, including the hostname, fully qualified domain name (FQDN), and interface information.

Example 6-30 *Initializing Nornir*

```
! run_nornir.py

from nornir import InitNornir

from nornir.plugins.tasks.networking import napalm_get

from nornir.plugins.functions.text import print_result



nr = InitNornir(config_file="config.yaml")


result = nr.run(

             napalm_get,

             getters=['get_facts'])



print_result(result)
```

```
(nornir) [root@rnd-srvr nornir]# python3 run-nornir.py

napalm_get*************************************************************************

* NX-2 ** changed : False ************************************************

vvvv napalm_get ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv INFO

{ 'facts': { 'fqdn': 'NX-2',

             'hostname': 'NX-2',

             'interface_list': [ 'mgmt0',

                                 'Ethernet1/1',

                                 'Ethernet1/2',

                                 'Ethernet1/3',

                                 'Ethernet1/4',

                                 'Ethernet1/5',

                                 'Ethernet1/6',

                                 'Ethernet1/7',

                                 'Ethernet1/8',

],

             'model': 'Nexus9000 9000v Chassis',

             'os_version': '',

             'serial_number': '9IWWC65KRZ5',

             'uptime': 47743,

             'vendor': 'Cisco'}}

^^^^ END napalm_get ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

* XE-1 ** changed : False ************************************************

vvvv napalm_get ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

INFO

{ 'facts': { 'fqdn': 'XE-1.not set',

             'hostname': 'XE-1',

             'interface_list': [ 'GigabitEthernet1',

                                 'GigabitEthernet2',

                                 'GigabitEthernet3',

                                 'GigabitEthernet4',

                                 'GigabitEthernet5',

                                 'GigabitEthernet6',

                                 'GigabitEthernet7',

                                 'GigabitEthernet8'],

             'model': 'CSR1000V',
```

```
            'os_version': 'Virtual XE Software '
                          '(X86_64_LINUX_IOSD-UNIVERSALK9-M), Version '
                          '16.11.1b, RELEASE SOFTWARE (fc2)',
            'serial_number': '9OJVU3ML6ZM',
            'uptime': 47700,
            'vendor': 'Cisco'})
^^^^ END napalm_get ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* XR-3 ** changed : False ************************************************
vvvv napalm_get ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
INFO
{ 'facts': { 'fqdn': 'XR-3',
            'hostname': 'XR-3',
            'interface_list': [ 'GigabitEthernet0/0/0/0',
                                'GigabitEthernet0/0/0/1',
                                'GigabitEthernet0/0/0/2',
                                'GigabitEthernet0/0/0/3',
                                'GigabitEthernet0/0/0/4',
                                'GigabitEthernet0/0/0/5',
                                'GigabitEthernet0/0/0/6',
                                'GigabitEthernet0/0/0/7',
                                'MgmtEth0/RP0/CPU0/0',
                                'Null0'],
            'model': 'R-IOSXRV9000-CC',
            'os_version': '6.6.2',
            'serial_number': '18C5B2EF3A6',
            'uptime': 47500,
            'vendor': 'Cisco'})
^^^^ END napalm_get ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
(nornir) [root@rnd-srvr nornir]#
```

Other tasks are available in Nornir, such as **napalm_configure** (which is also defined under **nornir.plugins.takss.networking**). **napalm_configure** can be used in conjunction with Jinja templates, which are covered in the following section.

Templating with Jinja2

Jinja is a modern templating language for Python that is modeled after Django. Jinja templates can be used to dynamically generate content/snippets. The name Jinja2 identifies the latest version of Jinja. Jinja2 templates allow you to easily interact with the Python program and use the data received from functions to quickly generate dynamic content. Jinja provides several benefits, including the following:

• Template inheritance

• Optimal just-in-time compilation

• Easy debuggability

• Configurable syntax

Jinja2 is commonly used with web frameworks such as Flask, and it is also used as a template language by configuration management tools such as Ansible and network automation frameworks such as Nornir. This section refers to Jinja2 templates as simply Jinja templates.

Jinja can generate any text-based format, such as HTML, XML, or CSV. A Jinja template is a simple text file that can be stored in a file with any extension. A Jinja template may contain variables, statements (such as **if-else** statements), expressions, and comments. A few delimiters can be used in Jinja:

• **{% . . . %}**: For statements

• **{{ . . . }}**: For expressions (used to print template output)

• **{# . . . #}**: For comments (which are not included in template output)

• **# . . . ##**: For line statements

Example 6-31 illustrates how to use statements and expressions in a Jinja template. In this example, a Jinja template is created to generate a configuration for a Nexus switch based on a config.yaml file. Within the config.yaml file, multiple VLANs are defined, along with ports that are acting as either trunk ports or access ports. For each VLAN, an SVI needs to be created. In the Jinja template, a **for** statement is used to generate the configurations of multiple VLANs, and then another **for** statement is used to configure the interfaces. Within this **for** statement, another **if-else** statement is used to validate whether the interface should be an access port or a trunk port. Finally, another **for** statement creates the SVIs. In this example, the Jinja template is used with the Netmiko library to connect to the remote Nexus switch and configure it.

Example 6-31 *Using a Jinja2 Template with the Netmiko Library*

```
! switchport-template.j2
```

```
hostname {{ name }}


feature interface-vlan


# For loop to iterate through vlan list and use it to create multiple vlans
on the device
{% for vlan, name in vlans.items() %}
vlan {{ vlan }}
name {{ name }}
{% endfor %}


{% for interface in interfaces %}


interface {{ interface.id }}
description Link to {{ interface.remote_server }} port {{ interface.port }}
{% if interface.mode == "trunk" -%}
  switchport mode trunk
  {% else -%}
  switchport mode access
  switchport access vlan {{ interface.vlan }}
  {% endif -%}
  no shutdown
{% endfor %}


{% for vlan, name in vlans.items() %}
interface vlan {{ vlan }}
ip address 10.1.{{ vlan }}.1/24
no shutdown
{% endfor %}
```

```
! config.yaml
name: NX-1
vlans:
  10: Management
  100: Data
  200: Voice
interfaces:
  - id: Eth1/1
    mode: trunk
    remote_server: NX-2
    port: Eth1/1
  - id: Eth1/2
    mode: access
    remote_server: Srvr2
    vlan: 100
    port: 1
```

```
! app.py
import yaml
from jinja2 import Environment, FileSystemLoader
from netmiko import ConnectHandler


configs = yaml.load(open('./config.yaml'), Loader=yaml.FullLoader)


env = Environment(loader = FileSystemLoader('./'), trim_blocks=True,
```

```
lstrip_blocks=True)
template = env.get_template('switchport-template.j2')

cfg = template.render(configs)

with open("configs.txt", "w") as f:
    f.write(cfg)

with ConnectHandler(ip = "172.16.102.137",
                    port = "22",
                    username = "admin",
                    password = "admin",
                    device_type = "cisco_nxos") as ch:

                    config_set = cfg.split("\n")
                    output = ch.send_config_set(config_set)
                    print(output)
```

```
(jinja) [root@rnd-srvr jinja]# python3 app.py
config term
Enter configuration commands, one per line. End with CNTL/Z.

NX-2(config)# hostname NX-1
NX-1(config)# feature interface-vlan
NX-1(config)#
NX-1(config)# vlan 10
NX-1(config-vlan)# name Management
NX-1(config-vlan)# vlan 100
NX-1(config-vlan)# name Data
NX-1(config-vlan)# vlan 200
NX-1(config-vlan)# name Voice
NX-1(config-vlan)# interface Eth1/3
NX-1(config-if)# description Link to NX-2 port Eth1/1
NX-1(config-if)# switchport mode trunk
NX-1(config-if)# no shutdown
NX-1(config-if)# interface Eth1/4
NX-1(config-if)# description Link to Srvr2 port 1
NX-1(config-if)# switchport mode access
NX-1(config-if)#   switchport access vlan 100
NX-1(config-if)# no shutdown
NX-1(config-if)# interface vlan 10
NX-1(config-if)# ip address 10.1.10.1/24
NX-1(config-if)# no shutdown
NX-1(config-if)# interface vlan 100
NX-1(config-if)# ip address 10.1.100.1/24
NX-1(config-if)# no shutdown
NX-1(config-if)# interface vlan 200
NX-1(config-if)# ip address 10.1.200.1/24
NX-1(config-if)# no shutdown
NX-1(config-if)# end
(jinja) [root@rnd-srvr jinja]# cat configs.txt
hostname NX-1

feature interface-vlan
```

```
vlan 10
name Management
vlan 100
name Data
vlan 200
name Voice

interface Eth1/3
description Link to NX-2 port Eth1/1
switchport mode trunk
no shutdown

interface Eth1/4
description Link to Srvr2 port 1
switchport mode access
  switchport access vlan 100
no shutdown

interface vlan 10
ip address 10.1.10.1/24
no shutdown
interface vlan 100
ip address 10.1.100.1/24
no shutdown
interface vlan 200
ip address 10.1.200.1/24
no shutdown
```

As mentioned earlier, Jinja templates can also be used with Nornir for network automation purposes. The next few examples illustrate the use of Jinja templates with Nornir. Example 6-32 examines the changes in the groups.yaml file where the groups **xe-routers**, **nx-routers**, and **xr-routers** are assigned site names, ASNs, loopback interface IP addresses, and BGP-related information that will be used by Jinja templates to generate the configurations. Note that the hosts.yaml file remains the same as shown in Example 6-29.

Example 6-32 *A groups.yaml File for Nornir*

```
---
global:
    data:
        domain: domain.local

xe-routers:
    data:
      site: SJC
      asn: 65001
      nos: xe
      loopback0: 1.1.1.1
      networks:
        - net: 1.1.1.1
          mask: 255.255.255.255
        - net: 192.168.1.0
          mask: 255.255.255.0
      neighbors:
        - ip: 10.1.2.2
          remote_asn: 65002
          peering_type: ebgp
        - ip: 10.10.10.10
```

```
          remote_asn: 65001
          peering_type: ibgp
    groups:
        - global



nx-routers:
    data:
      site: RTP
      asn: 65002
      nos: nxos
      loopback0: 2.2.2.2
      networks:
        - net: 2.2.2.2
          mask: 255.255.255.255
        - net: 192.168.2.0
          mask: 255.255.255.0
      neighbors:
        - ip: 10.1.2.1
          remote_asn: 65001
          peering_type: ebgp
        - ip: 10.2.3.3
          remote_asn: 65003
          peering_type: ebgp
        - ip: 20.20.20.20
          remote_asn: 65002
          peering_type: ibgp
    groups:
        - global


xr-routers:
    data:
      site: RCDN
      asn: 65003
      nos: xr
      loopback0: 3.3.3.3
      networks:
        - net: 3.3.3.3
          mask: 255.255.255.255
        - net: 192.168.3.0
          mask: 255.255.255.0
      neighbors:
        - ip: 10.2.3.2
          remote_asn: 65002
          peering_type: ebgp
        - ip: 30.30.30.30
          remote_asn: 65003
          peering_type: ibgp
    groups:
        - global
```

After the group file is updated, the next step is to create the Jinja templates. Example 6-33 shows three different templates for different network operating systems: Cisco IOS XE, NX-OS and IOS XR. Another way to do this would be to use an **if-else** statement within the Jinja templates to differentiate between the configuration snippets for different network operating systems. In Example 6-33, each template creates a loopback0 interface and assigns an IP address to it and then configures BGP with the specified ASN and configures both the network and neighbor statements. Note that BGP configuration is somewhat similar in IOS XR and NX-OS but is quite different in IOS XE. These Jinja templates are a perfect example of how you can use automation tools to apply configurations at scale.

Example 6-33 *Jinja Templates for Cisco IOS XE, NX-OS, and IOS XR*

```
interface loopback0
ip address {{ host.loopback0 }} 255.255.255.255

router bgp {{ host.asn }}
bgp router-id {{ host.loopback0 }}

{% for nei in host.neighbors %}
  neighbor {{ nei.ip }} remote-as {{ nei.remote_asn }}
   {% if nei.peering_type == "ibgp" -%}
     neighbor {{ nei.ip }} update-source loopback0
   {% endif -%}
   address-family ipv4 unicast
   neighbor {{ nei.ip }} activate
   exit
{% endfor %}
address-family ipv4 unicast
{% for net in host.networks %}
  network {{ net.net }} mask {{ net.mask }}
{% endfor %}
End
```

```
interface loopback0
ip address {{ host.loopback0 }}/32

feature bgp
router bgp {{ host.asn }}
router-id {{ host.loopback0 }}

 address-family ipv4 unicast
{% for net in host.networks %}
  network {{ net.net }} mask {{ net.mask }}
{% endfor %}

{% for nei in host.neighbors %}
  neighbor {{ nei.ip }}
   remote-as {{ nei.remote_asn }}
   {% if nei.peering_type == "ibgp" -%}
     update-source loopback0
   {% endif -%}
   address-family ipv4 unicast
{% endfor %}
```

```
interface loopback0
ip address {{ host.loopback0 }}/32

router bgp {{ host.asn }}
bgp router-id {{ host.loopback0 }}

 address-family ipv4 unicast
{% for net in host.networks %}
  network {{ net.net }} {{ net.mask }}
{% endfor %}
```

```
{% for nei in host.neighbors %}
  neighbor {{ nei.ip }}
   remote-as {{ nei.remote_asn }}
   {% if nei.peering_type == "ibgp" -%}
     update-source loopback0
   {% endif -%}
   address-family ipv4 unicast
{% endfor %}
```

After creating the Jinja templates, you can follow these steps to easily configure the network devices:

1. Create a task that uses the Jinja template.

2. Invoke **napalm_configure** within the task.

3. Initialize Nornir by using the **InitNornir()** function.

4. Run the task by using the **run()** method.

Example 6-34 shows a function named **load_config()** being created with the parameter **task**. Within this function, you can start multiple tasks by using the **task.run()** method. The first task transforms inventory data (data from hosts and group files) to configuration data via Jinja templates and then saves the compiled configuration into a host variable. The next task deploys the compiled configuration to the device by using NAPALM. In this example, notice that the **path** value begins with an **f** string. The benefit of using an **f** string is that it evaluates at program runtime.

Example 6-34 *Using Jinja Templates with NAPALM and Nornir*

```
from nornir import InitNornir
from nornir.plugins.tasks.data import load_yaml
from nornir.plugins.tasks import text
from nornir.plugins.tasks.networking import napalm_configure
from nornir.plugins.functions.text import print_result


def load_config(task):
    r = task.run(task=text.template_file, template='bgp_config.j2',
path=f'templates/{task.host["site"]}')
    task.host["template_config"] = r.result
    task.run(task=napalm_configure, name="Loading configuration for the device",
configuration=task.host["template_config"])



nr = InitNornir(config_file="config.yaml")
sjc = nr.filter(site="SJC")
result = sjc.run(load_config)
print_result(result)
```

Once the Python file is executed, all the devices that are part of the site SJC are configured based on their respective network operating systems. Example 6-35 displays the output of the Python program and the changes that will be committed to the devices in the SJC site.

Example 6-35 *Output from a Python Program with Jinja, Nomir, and NAPALM*

```
load_config*********************************************************************
* NX-2 ** changed : True ***************************************************
vvvv load_config ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
INFO
---- template_file ** changed : False ----------------------------------------
INFO
interface loopback0
ip address 2.2.2.2/32

feature bgp
router bgp 65002
router-id 2.2.2.2

 address-family ipv4 unicast
  network 2.2.2.2 mask 255.255.255.255
```

```
  network 192.168.2.0 mask 255.255.255.0

  neighbor 10.1.2.1
   remote-as 65001
   address-family ipv4 unicast
  neighbor 10.2.3.3
   remote-as 65003
   address-family ipv4 unicast
  neighbor 20.20.20.20
   remote-as 65002
   update-source loopback0
   address-family ipv4 unicast

---- Loading configuration for the device ** changed : True --------------------
INFO
interface loopback0
ip address 2.2.2.2/32
feature bgp
router bgp 65002
router-id 2.2.2.2
 address-family ipv4 unicast
  network 2.2.2.2 mask 255.255.255.255
  network 192.168.2.0 mask 255.255.255.0
  neighbor 10.1.2.1
   remote-as 65001
   address-family ipv4 unicast
  neighbor 10.2.3.3
   remote-as 65003
   address-family ipv4 unicast
  neighbor 20.20.20.20
   remote-as 65002
   update-source loopback0
   address-family ipv4 unicast
^^^^ END load_config ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* XE-1 ** changed : True ************************************************
vvvv load_config ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
INFO
---- template_file ** changed : False ----------------------------------------
INFO

interface loopback0
ip address 1.1.1.1 255.255.255.255

router bgp 65001
bgp router-id 1.1.1.1

  neighbor 10.1.2.2 remote-as 65002
   address-family ipv4 unicast
   neighbor 10.1.2.2 activate
   exit
  neighbor 10.10.10.10 remote-as 65001
   neighbor 10.10.10.10 update-source loopback0
   address-family ipv4 unicast
```

```
   neighbor 10.10.10.10 activate
   exit
address-family ipv4 unicast
  network 1.1.1.1 mask 255.255.255.255
  network 192.168.1.0 mask 255.255.255.0
end

---- Loading configuration for the device ** changed : True --------------------
INFO
+interface loopback0
+ip address 1.1.1.1 255.255.255.255
+router bgp 65001
+bgp router-id 1.1.1.1
+  neighbor 10.1.2.2 remote-as 65002
+address-family ipv4 unicast
+  network 1.1.1.1 mask 255.255.255.255
^^^^ END load_config ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
load_config*************************************************************
* XR-3 ** changed : True ************************************************
vvvv load_config ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
INFO
---- template_file ** changed : False ----------------------------------------
INFO
interface loopback0
ip address 3.3.3.3/32

router bgp 65003
bgp router-id 3.3.3.3

 address-family ipv4 unicast
  network 3.3.3.3 255.255.255.255
  network 192.168.3.0 255.255.255.0

  neighbor 10.2.3.2
   remote-as 65002
   address-family ipv4 unicast
  neighbor 30.30.30.30
   remote-as 65003
   update-source loopback0
   address-family ipv4 unicast

---- Loading configuration for the device ** changed : True --------------------
INFO
---
+++
@@ -13,6 +13,9 @@
   active
   destination transport-method http
  !
+!
+interface Loopback0
+ ipv4 address 3.3.3.3 255.255.255.255
 !
 interface MgmtEth0/RP0/CPU0/0
```

```
  shutdown
@@ -46,6 +49,24 @@
   0.0.0.0/0 172.16.102.1
  !
 !
+router bgp 65003
+ bgp router-id 3.3.3.3
+ address-family ipv4 unicast
+  network 3.3.3.3/32
+  network 192.168.3.0/24
+ !
+ neighbor 10.2.3.2
+  remote-as 65002
+  address-family ipv4 unicast
+  !
+ !
+ neighbor 30.30.30.30
+  remote-as 65003
+  update-source Loopback0
+  address-family ipv4 unicast
+  !
+ !
+!
 xml agent tty
  iteration off
 !
^^^^ END load_config ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

You can use NAPALM and Nornir along with Ansible to perform configuration and device management.

Orchestration

So far in this chapter, we have looked at how open-source Python libraries can be leveraged for network automation tasks. But there is more to what Python can do. You can also leverage Python for orchestration. Today, most cloud-based solutions use Docker containers, and most of the applications that are deployed in the cloud use orchestration tools such as Kubernetes (also known as K8s) that allows cloud architects to quickly deploy container applications with just few clicks. Keeping track of different resources to manage these orchestration tools and validating container applications for testing can be tedious. Moreover, it can be difficult at times to remember the Docker/containers or Kubernetes commands and use them properly during the development phase. To ease these tasks, both Docker and Kubernetes come with Python libraries that can be used within a Python program to get access to these tools and their respective CLIs as library functions and use them to deploy and test cloud applications.

Docker

For Docker, you can install a Docker package by using the Python package manager. Once it is installed, you can use the **import** keyword to import the Docker library. On a system with a Docker package installed, a Docker daemon is already running. In order to connect to the docker daemon, you are first required to initiate a client. You can use the **from_env()** function to connect to the Docker daemon using the default environment settings. You can then use the client instance to access the list of images or containers, pull a Docker image, or even initiate a container from the pulled image. Following are some of the Docker library functions that you can use:

• **client.images.list()**: Lists all the Docker images on the local system.

• **client.images.pull(***image-name***)**: Pulls a Docker image from the global repository.

• **client.containers.list()**: Lists all the containers.

• **client.containers.get(***container-name* | *container-id***)**: Gets a container by name or ID.

• **client.containers.run(***image*, *command=None*, ***kwargs***)**: Runs a container.

• **client.containers.start(***kwargs***)**: Starts a container. Doesn't support **attach** options.

• **client.containers.stop**(***kwargs*): Stops a container.

Example 6-36 illustrates how to use some of the functions available in the Docker library. In this example, the latest **alpine** image is pulled from the repository, a container is run by calling the **client.containers.run()** function, and the command **ifconfig** is passed in the argument. Notice that when the **client.containers.run()** function is called, all the logs are printed on the terminal, along with the **ifconfig** output from the container. When the job is complete—that is, when the **ifconfig** output is printed to the terminal—the container is disposed from memory. Also notice the use of the **dir()** function in Example 6-36. **dir()** is a powerful built-in function in Python 3 that returns a list of the attributes and methods available for any object. This function can be very useful when you are not familiar with all the functions available or accessible using an object.

Example 6-36 *Pulling a Docker Image and Running a Container Using the Docker Library*

```
(docker) [root@rnd-srvr docker]# python
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import docker
>>> dir(docker)
['APIClient', 'DockerClient', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__',
'__title__', '__version__', 'api', 'auth', 'client', 'constants', 'credentials',
'errors', 'from_env', 'models', 'tls', 'transport', 'types', 'utils', 'version',
'version_info']
>>> client = docker.from_env()
>>> client.images.list()
[<Image: 'plotter:latest'>, <Image: 'docker.io/python:3'>, <Image:
'docker.io/centos:latest'>, <Image: 'docker.io/hello-world:latest'>]
>>> dir(client.images)
['__call__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'build', 'client',
'create', 'get', 'get_registry_data', 'list', 'load', 'model', 'prepare_model',
'prune', 'prune_builds', 'pull', 'push', 'remove', 'search']
>>> client.images.pull('alpine:latest')
<Image: 'docker.io/alpine:latest'>
>>> client.images.list()
[<Image: 'plotter:latest'>, <Image: 'docker.io/python:3'>, <Image:
'docker.io/alpine:latest'>, <Image: 'docker.io/centos:latest'>, <Image:
'docker.io/hello-world:latest'>]
>>> dir(client.containers)
['__call__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'client', 'create',
'get', 'list', 'model', 'prepare_model', 'prune', 'run']
>>> client.containers.run('alpine:latest', 'ifconfig')
[1524800.223807] docker0: port 1(vethc88c6ef) entered blocking state
[1524800.231275] docker0: port 1(vethc88c6ef) entered disabled state
[1524800.251101] device vethc88c6ef entered promiscuous mode
[1524800.256202] IPv6: ADDRCONF(NETDEV_UP): vethc88c6ef: link is not ready
[1524800.258182] docker0: port 1(vethc88c6ef) entered blocking state
[1524800.260075] docker0: port 1(vethc88c6ef) entered forwarding state
[1524800.269984] docker0: port 1(vethc88c6ef) entered disabled state
[1524800.522753] SELinux: mount invalid.  Same superblock, different security
settings for (dev mqueue, type mqueue)
[1524800.629330] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[1524800.632217] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[1524800.634336] IPv6: ADDRCONF(NETDEV_CHANGE): vethc88c6ef: link becomes ready
[1524800.636374] docker0: port 1(vethc88c6ef) entered blocking state
[1524800.638209] docker0: port 1(vethc88c6ef) entered forwarding state
[1524800.816280] docker0: port 1(vethc88c6ef) entered disabled state
[1524800.825025] docker0: port 1(vethc88c6ef) entered disabled state
[1524800.827276] device vethc88c6ef left promiscuous mode
```

```
[1524800.828927] docker0: port 1(vethc88c6ef) entered disabled state



b'eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02  \n          inet

addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0\n          inet6 addr:

fe80::42:acff:fe11:2/64 Scope:Link\n          UP BROADCAST RUNNING MULTICAST

MTU:1500  Metric:1\n          RX packets:2 errors:0 dropped:0 overruns:0

frame:0\n          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0\n

collisions:0 txqueuelen:0 \n          RX bytes:176 (176.0 B)  TX bytes:90 (90.0

B)\nlo        Link encap:Local Loopback  \n          inet addr:127.0.0.1

Mask:255.0.0.0\n          inet6 addr: ::1/128 Scope:Host\n          UP LOOPBACK

RUNNING  MTU:65536  Metric:1\n          RX packets:0 errors:0 dropped:0

overruns:0 frame:0\n          TX packets:0 errors:0 dropped:0 overruns:0

carrier:0\n          collisions:0 txqueuelen:1000 \n          RX bytes:0 (0.0 B)

TX bytes:0 (0.0 B)\n'

>>>
```

Note

For more details and information about the different methods and attributes available in the Docker library, refer to the documentation at https://docker-py.readthedocs.io/en/stable/index.html.

Kubernetes

Fundamentally, Kubernetes is a system for running multiple instances of containerized applications across a cluster of machines, providing redundancy to the applications. Kubernetes manages the complete lifecycle of the containerized applications and services, providing scalability and high availability. Kubernetes brings together individual physical or virtual machines into a cluster, using a shared network to establish communication between the nodes. All the Kubernetes components, capabilities, and workloads are configured on this cluster. One of the nodes is given the role of the primary server. This primary server acts as the brain for the cluster and exposes APIs for users and clients. The other machines are designated as nodes, and they are responsible for accepting and running workloads using local and external resources.

Kubernetes provides JSON REST APIs in order to control a Kubernetes cluster. APIs are available in multiple languages, including Python. Once the Kubernetes package is installed, you can easily import the Kubernetes library and use the client. Within the Kubernetes library, you can use both the client and configuration-related APIs. You use the **config.load_kube_config()** method to load authentication and cluster-related information from the **kube-config** and store it in **kubernetes.client.configuration**. You can then use the **client.CoreV1API()** method to access the client APIs. There are various methods available as part of the client APIs. One of the methods, **list_pod_all_namespace()**, returns a list of pods available in each namespace on the cluster. Example 6-37 demonstrates how to access the Kubernetes client APIs and use the **list_pod_all_namespaces()** method to iterate through and list the pods in all namespaces.

Example 6-37 *Using the Kubernetes Client to Invoke the **list_pod_all_namespace()** Method*

```
(kubernetes) root@node1:~# python

Python 3.6.9 (default, Nov  7 2019, 10:44:02)

[GCC 8.3.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> from kubernetes import client, config

>>>

>>> config.load_kube_config()

>>> api = client.CoreV1Api()

>>> pods = api.list_pod_for_all_namespaces()

>>> for pod in pods.items:

...   print("%s: %s" % (pod.metadata.namespace,pod.metadata.name))

...

default: wordpress-9f7965d6-6fzhl

default: wordpress-mysql-746dd7c4db-p5clw

kube-system: calico-kube-controllers-648f4868b8-zvj85

kube-system: calico-node-fnmqf

kube-system: calico-node-hjtb9

kube-system: calico-node-tb4fs

kube-system: coredns-6955765f44-4dvxk

kube-system: coredns-6955765f44-74178

kube-system: etcd-kubernetes-node1
```

```
clf.fit(fruit_features, labels)

print(clf.predict([[130, 0]]))
```

```
(venv) [root@rnd-srvr ml]# python simple-classification.py

[0]
```

This is a very basic example with a small data set. Real-world applications typically involve massive amounts of data, and it is critical to choose an appropriate algorithm for the kind of data.

Note

Covering all algorithms as well as all the machine learning methods is beyond the scope of this book.

Summary

This chapter covers various applications of Python, including the following:

• How to organize development environments using tools such as Git, Docker, and virtual environments

• Various applications of Python, including web/API development, network automation, orchestration, and machine learning

• Python frameworks such as Django and Flask

• The use of open-source libraries such as NAPALM and Nornir to perform network automation and management

• How to use the Docker and Kubernetes libraries to orchestrate container applications dynamically using Python code

• How machine learning works.

• Code-on-Demand (optional)

An API that adheres to these constraints is said to be a *RESTful API*.

Note

Fielding's dissertation is publicly available at https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

At this point, it is important to clarify the relationship between REST and HTTP. REST is an architectural style for programming software interfaces (APIs). HTTP, on the other hand, is an application layer protocol (in the OSI model) that is leveraged to implement APIs that adhere to the constraints dictated by the REST framework. HTTP is the only protocol designed specifically for transferring resource representations. Instead of explaining the REST framework independently, the following few paragraphs explain how HTTP implements the REST constraints.

As you already know by now, HTTP is a *client/server* protocol; it therefore adheres to the first constraint.

When a server sends back a response to a client request, this particular transaction is completed, and no state information pertaining to this transaction is maintained on the server. Any one client request contains all information required to fully understand and process this request, independent of any previous requests. Therefore, HTTP is a *stateless* protocol and hence adheres to the second constraint on the list.

HTTP resources *may* be cached at intermediate cache servers along the path between the client and server. Resources should be labeled as cacheable or not. HTTP defines a number of header fields to support this functionality. Therefore, *cacheable* resources is a feature of HTTP, so it satisfies the third constraint.

As stated at the beginning of this section, the interface provided by HTTP to resources does not differ, regardless of the resource type. Therefore, HTTP adheres to the fourth constraint by providing a *uniform* interface for clients to address resources on servers.

The fifth constraint dictates that a system leveraging RESTful APIs should be able to support a layered architecture. A layered architecture segregates the functional components into a number of hierarchical layers, where each layer is only aware of the existence of the adjacent layers and communicates only with those adjacent layers. For example, a client may be interacting with a proxy server, not the actual HTTP server, while not being aware of this fact. On the other end of the connection, a server processing and responding to client requests in the front end may rely on a database server in the back end to store the resources.

The final constraint, which is an optional constraint, is support for Code-on-Demand (CoD)—the capability of downloading software from the server to the client to be executed by the client, such as Java applets of JavaScript code downloaded from a website and run by the client web browser.

To appreciate why the REST framework and its application to RESTful APIs using HTTP are very popular, look at its biggest implementation: the Internet. While REST APIs may not be the best fit for some use cases, they have proven to be a massive success in terms of reliability, simplicity, scalability, and performance for Internet-scale applications. Today, the vast majority of services provided over the Internet leverage REST APIs, such as services from Google and social media companies such as Twitter and Facebook.

RFC 3986 formally defines a representation as "a sequence of octets, along with representation metadata describing those octets that constitutes a record of the state of the resource at the time when the representation is generated." Therefore, a representation in HTTP is composed of not only the message body (or the entity body) but also of metadata used to describe the representation in the body of that message to help interpret this representation. For example, a client would not know if a certain representation returned by a server is in JSON or XML unless some metadata stated that information. This representation metadata is stored in header fields in HTTP. (These header fields are covered in detail in the section "The HTTP Entity Header Fields," later in this chapter.)

The HTTP Connection

This section discusses the first building block of HTTP: the client/server connection that leverages TCP to provide a reliable transport for HTTP request and response messages. It also covers the enhancements introduced in HTTP/1.1 related to that connection.

Client/Server Communication

HTTP is a connectionless application layer protocol that uses TCP at the transport layer. Before clients and servers can engage in an HTTP transaction, a TCP connection must be set up. Figure 7-2 illustrates the process of TCP connection establishment, followed by the HTTP transactions, and finally the TCP connection teardown.

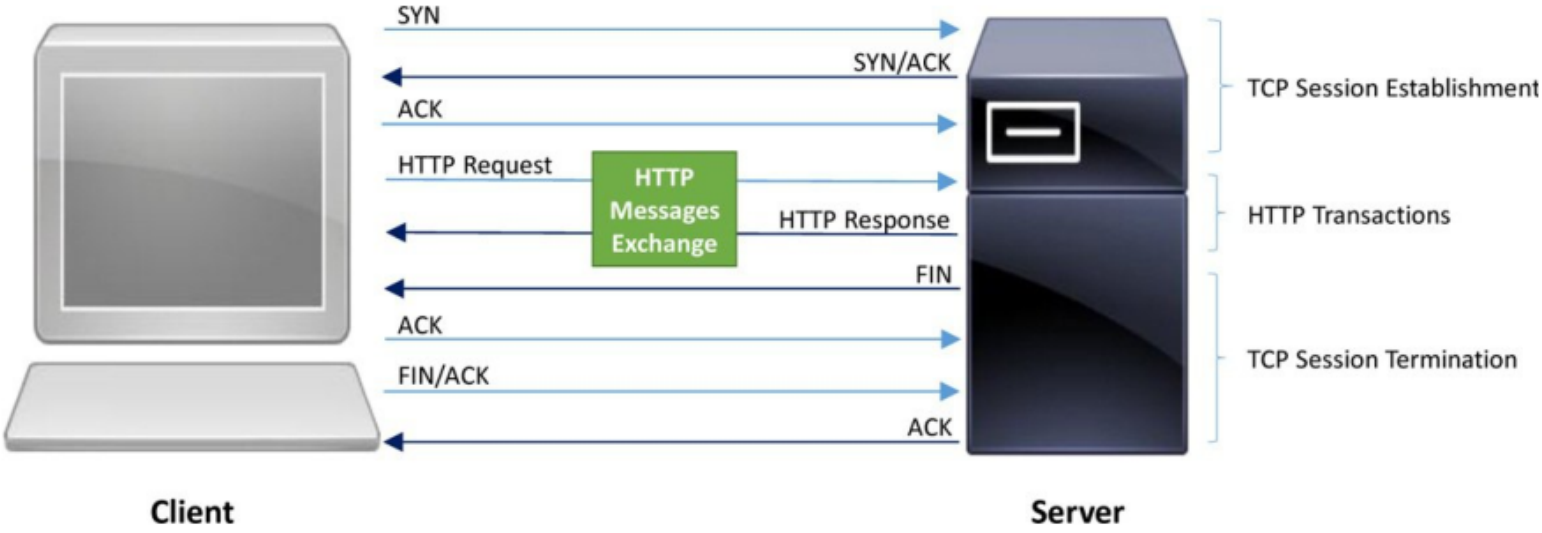


Figure 7-2 *TCP Connection Establishment and Termination with HTTP Message Exchange In Between*

A TCP connection is established through a three-way handshake. Before a client can attempt to connect to a server, the server must first bind to and listen on a port to open it up for connections. It starts with the client sending a SYN message (a TCP segment with the SYN flag set) to the server on the port that the server is listening on. By default, an HTTP daemon on the server binds and listens on port 80. If the server is ready to set up the TCP connection, it responds with a TCP segment with the SYN and ACK flags set. As the final step, before the connection gets established, the client acknowledges the server's segment via an ACK message.

After the TCP connection is established, HTTP transactions can take place. To recap, the client sends one or more messages, each called a

HTTP Transactions

An HTTP transaction involves a client sending an HTTP request message and, ideally, the server responding with a server HTTP response message. A request message contains an HTTP method that defines the action that the client wishes to take on the resource identified by the request URI. The server, in its response message, uses a status code to indicate the result of processing the client request. A request/response pair constitutes an HTTP transaction. This section discusses, in some detail, client request methods and server response status codes, and the next section covers the format and the rest of the content of HTTP messages.

Client Requests

RFC 7231 defines the following client requests:

- GET
- HEAD
- POST
- PUT
- DELETE
- CONNECT
- OPTIONS
- TRACE

An HTTP method is said to be *safe* if using this method means that no state change will happen on the server. In other words, a safe method is a read-only method. GET, HEAD, OPTIONS, and TRACE are safe methods.

When sending the same request multiple times has the same effect as sending it only once, the method used in the request is said to be *idempotent*. For example, when a resource is created using the PUT method, sending the same request just re-creates the resource. The net effect of sending the request once or hundreds of times is the same. The same applies to deleting a resource. All the safe methods as well as PUT and DELETE are idempotent methods.

Cacheable methods are methods whose responses may be saved and re-sent by intermediate cache servers. GET, HEAD, and POST are cacheable methods.

GET

The most basic use of HTTP is to retrieve one or more objects from an HTTP server. These objects may constitute components of a web page or may be a part of a configuration on a network device that exposes a REST API. In order to retrieve information from a server, the client uses the GET method. In REST terminology, the GET method is used to request the current representation of the resource, identified by the URI in the start line of the HTTP request. Because a GET request is used to fetch data, a GET request typically has no payload body; it has only headers.

The response to a GET request may be cached by a cache server. This is controlled by the Cache-Control general header field. If the GET request is processed by the server and the representation requested is returned in the response, the status code in the response is "200 OK." If a current representation of the requested resource does not exist, the server uses status code "404 Not Found" in its response. Alternatively, the server may use status code "410 Gone" to indicate that a current representation has not been found and probably never will; that is, it indicates that the condition is likely to be permanent.

Two different GET requests are discussed in the "HTTP Overview" section, earlier in this chapter: one for a full web page (refer to Example 7-1) and one for a specific image on the same web page (refer to Example 7-2).

HEAD

A client request using the HEAD method returns exactly the same response from the server as a request using the GET method, minus the actual payload body. The server returns only headers. The headers returned in the response constitute the metadata of the representation of the resource that was targeted in the request. These returned headers are typically used for testing and validation. Like a GET request, a HEAD request typically has no payload.

The response to a HEAD request may be cached, and the caching is controlled by the Cache-Control header field.

POST

The POST method is used when a client intends to send information to a server for processing. Although that would not be the best use of HTTP, a client may send two numbers to a script running on a server by using the POST method. The script would then add these two numbers and return the sum back to the client in a response message. Or it might create a new resource at a "subordinate" URI at which the result is saved, and return the new resource URI in the response message with the response code "201 Created."

The resource that processes the information is specified in the URI of the request message. Subsequently, the POST method may amend an existing resource or create a new resource, or it may not result in any resource-specific changes.

POST is the primary method used to send information to Cisco switches through their APIs, such as for login authentication or configuration updates. Example 7-4 provides an example of using the POST method to configure a static route for destination 192.168.1.0/24 with the next hop 10.10.20.254.

Example 7-4 *Using a POST Message to Configure a Static Route on a Nexus Switch*

```
POST /api/mo/sys/ipv4/inst/dom-default.json HTTP/1.1

Content-Type: text/plain

User-Agent: PostmanRuntime/7.26.8

Accept: */*

Host: sbx-nxos-mgmt.cisco.com

Accept-Encoding: gzip, deflate, br
```

```
Connection: keep-alive

Content-Length: 462

Cookie: <cookie-value>


{
  "ipv4Dom": {
    "attributes": {
      "name": "default"
    },
    "children": [
      {
        "ipv4Route": {
          "attributes": {
            "prefix": "192.168.1.0/24"
          },
          "children": [
            {
              "ipv4Nexthop": {
                "attributes": {
                  "nhAddr": "10.10.20.254",
                  "nhIf": "unspecified",
                  "nhVrf": "management"
                  "tag": "1000"
}}}}}}}}}
```

The URI in Example 7-4 targets the default routing domain identified by the object dom-default, which is the resource in this example. Each VRF instance on the switch has its own routing domain. The default routing domain is the global routing table. Chapter 11 discusses the JSON encoding in detail. Chapter 14, "NETCONF and RESTCONF," discusses how to target resources and construct URIs using the RESTCONF protocol. Chapter 17, "Programming Cisco Platforms," describes how to identify and construct URIs to target the different resources that map to objects, specifically on a (programmable) Nexus switch, using a RESTful API named NX-API REST. Therefore, do not worry yourself with the specifics of the message in the example. The purpose of this example is to display the high-level anatomy of an HTTP request that uses the POST method to perform configuration changes on a network device.

The representation in the request body is the static route configuration, formatted in JSON. This representation starts right after the empty line following the Cookie header field. The POST request updates the resource with the newly added route. In other words, the POST request adds the new route to the default routing table. Once the update is accepted and the route is added, a "200 OK" response is sent back to the client, as shown in Example 7-5.

Example 7-5 *The "200 OK" Response Indicating That the Route Was Successfully Added*

```
HTTP/1.1 200 OK

Server: nginx/1.13.12

Date: Sat, 07 Nov 2020 20:11:12 GMT

Content-Type: application/json

Content-Length: 13

Connection: keep-alive

Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept, devcookie

Access-Control-Allow-Origin: http://127.0.0.1:8000

Access-Control-Allow-Methods: POST,GET,OPTIONS

Strict-Transport-Security: max-age=31536000; includeSubDomains

X-Frame-Options: SAMEORIGIN

X-XSS-Protection: 1; mode=block

X-Content-Type-Options: nosniff

Content-Security-Policy: block-all-mixed-content; base-uri 'self'; default-src 'self';

script-src 'self' 'unsafe-inline'; style-src 'self' 'unsafe-inline'; img-src 'self'

data:; connect-src 'self'; font-src 'self'; object-src 'none'; media-src 'self'; form-

action 'self'; frame-ancestors 'self';


{"imdata":[]}
```

Example 7-6 shows the list of static routes before and after the HTTP request. The new route to 192.168.1.0/24 is added to the static routes' configuration and reflected in the routing table.

Example 7-6 *Static Routes on the Switch Before and After the POST Request*

```
! Before sending the POST request
sbx-ao# show ip route
IP Route Table for VRF "default"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

10.98.98.0/24, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:10:50, direct
10.98.98.1/32, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:10:50, local

! After sending the POST request
sbx-n9kv-ao# show ip route
IP Route Table for VRF "default"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

10.98.98.0/24, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:13:06, direct
10.98.98.1/32, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:13:06, local
192.168.1.0/24, ubest/mbest: 1/0
    *via 10.10.20.254%management, [1/0], 00:00:05, static, tag 1000
```

Another POST request is an authentication request to a Nexus switch. But unlike the previous request, this one does not create or change any resources. Example 7-7 shows a request sent to a Nexus switch for authenticating with the switch and receiving a cookie that is then used to authenticate all subsequent requests to the switch.

Example 7-7 *Authentication Request Message with No Resources Created or Amended*

```
POST /api/aaaLogin.json HTTP/1.1
Content-Type: application/json
Cache-Control: no-cache
User-Agent: PostmanRuntime/7.26.8
Accept: */*
Host: sbx-nxos-mgmt.cisco.com
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 116

{
     "aaaUser": {
       "attributes": {
         "name": "admin",
         "pwd": "Admin_1234!"
        }
    }
}
```

The resource URI in the start line of the request message is /api/aaaLogin.json, and the message body contains the information (representation)—in this case, the login credentials. No resources are created or changed in this case. Note that the URI is a relative URI reference that becomes an absolute URI when appended with the scheme (HTTPS) and authority (sbx-nxos-mgmt.cisco.com) to become https://sbx-nxos-mgmt.cisco.com/api/aaaLogin.json. (Relative and absolute URI references are covered later in this chapter.)

Note

This example and the next few examples present the functionality of the different request methods. You are not yet expected to understand much of the content in the HTTP messages presented, so do not worry. The meaning of each of the lines will become clear as you progress through the chapter and the rest of the book.

PUT

The PUT method will replace the state of a resource with the representation in the request payload. If the resource did not exist in the first place, the resource is created. If the method in the static route example in the previous section were a PUT instead of a POST, all static routes would be replaced by the static route configuration represented in the payload body of the request. The resource in this case is, again, the default routing domain (table). Example 7-8 shows a PUT request for the destination 192.168.2.0/24 and next hop .10.10.20.254.

Example 7-8 *Using a PUT Request to Configure a Static Route on a Nexus Switch*

```
PUT /api/mo/sys/ipv4/inst/dom-default.json HTTP/1.1

Content-Type: text/plain

User-Agent: PostmanRuntime/7.26.8

Accept: */*

Host: sbx-nxos-mgmt.cisco.com

Accept-Encoding: gzip, deflate, br

Connection: keep-alive

Content-Length: 462

Cookie: <cookie-value>


{
  "ipv4Dom": {
    "attributes": {
      "name": "default"
    },
    "children": [
      {
        "ipv4Route": {
          "attributes": {
            "prefix": "192.168.2.0/24"
          },
          "children": [
            {
              "ipv4Nexthop": {
                "attributes": {
                  "nhAddr": "10.10.20.254",
                  "nhIf": "unspecified",
                  "nhVrf": "management"
                  "tag": "2000"
}}}}}}}}}
```

The response shown in Example 7-9 is again "200 OK," meaning that the request has been accepted and processed. If the target resource did not exist, the PUT request would create the resource, and in that case, the status code in the server response might be "201 Created." That does not apply in this case, however, because the resource—the default routing table—is a data structure that will always exist on the switch and cannot be deleted and re-created through any interface on the router (CLI or API).

Example 7-9 *The "200 OK" Response to the PUT Request*

```
HTTP/1.1 200 OK

Server: nginx/1.13.12

Date: Sat, 07 Nov 2020 20:18:51 GMT

Content-Type: application/json

Content-Length: 13

Connection: keep-alive

Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept, devcookie

Access-Control-Allow-Origin: http://127.0.0.1:8000

Access-Control-Allow-Methods: POST,GET,OPTIONS

Strict-Transport-Security: max-age=31536000; includeSubDomains
```

```
X-Frame-Options: SAMEORIGIN

X-XSS-Protection: 1; mode=block

X-Content-Type-Options: nosniff

Content-Security-Policy: block-all-mixed-content; base-uri 'self'; default-src 'self';

script-src 'self' 'unsafe-inline'; style-src 'self' 'unsafe-inline'; img-src 'self'

data:; connect-src 'self'; font-src 'self'; object-src 'none'; media-src 'self'; form-

action 'self'; frame-ancestors 'self';


{"imdata":[]}
```

Example 7-10 shows that the PUT method replacing all static routes with the representation in the request, which is the single static route for 192.168.2.0/24.

Example 7-10 *Static Routes on a Switch Before and After a PUT Request*

```
! Before sending the PUT request

sbx-n9kv-ao# show ip route

IP Route Table for VRF "default"

'*' denotes best ucast next-hop

'**' denotes best mcast next-hop

'[x/y]' denotes [preference/metric]

'%<string>' in via output denotes VRF <string>


10.98.98.0/24, ubest/mbest: 1/0, attached

    *via 10.98.98.1, Lo98, [0/0], 00:13:06, direct

10.98.98.1/32, ubest/mbest: 1/0, attached

    *via 10.98.98.1, Lo98, [0/0], 00:13:06, local

192.168.1.0/24, ubest/mbest: 1/0

    *via 10.10.20.254%management, [1/0], 00:00:05, static, tag 1000


! After sending the PUT request

sbx-n9kv-ao# show ip route

IP Route Table for VRF "default"

'*' denotes best ucast next-hop

'**' denotes best mcast next-hop

'[x/y]' denotes [preference/metric]

'%<string>' in via output denotes VRF <string>


192.168.2.0/24, ubest/mbest: 1/0

    *via 10.10.20.254%management, [1/0], 00:00:05, static, tag 2000
```

Instead of targeting the resource dom-default that represents the entire default routing domain, you could instead target just the specific route that you want to operate on. In the case of the static route for 192.168.2.0/24, the resource URI would be https://sbx-nxos-mgmt.cisco.com/api/mo/sys/ipv4/inst/dom-default/rt-[192.168.2.0/24].json.

DELETE

The DELETE method deletes the resource targeted by the URI. Example 7-11 deletes the static route for destination 192.168.1.0/24 by using the URI for that specific route (rather than by targeting the default routing domain, as in the previous examples). The request has no body; it has only headers.

Example 7-11 *Using a DELETE Request to Delete the Static Route to 192.168.1.0/24*

```
DELETE /api/mo/sys/ipv4/inst/dom-default/rt-[192.168.1.0/24].json HTTP/1.1

User-Agent: PostmanRuntime/7.26.8

Accept: */*

Host: sbx-nxos-mgmt.cisco.com

Accept-Encoding: gzip, deflate, br

Connection: keep-alive

Cookie: <cookie-value>
```

The response is, again, "200 OK," as shown in Example 7-12.

Example 7-12 *The "200 OK" Response to the DELETE Request*

```
HTTP/1.1 200 OK
Server: nginx/1.13.12
Date: Sat, 07 Nov 2020 20:48:41 GMT
Content-Type: application/json
Content-Length: 13
Connection: keep-alive
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept, devcookie
Access-Control-Allow-Origin: http://127.0.0.1:8000
Access-Control-Allow-Methods: POST,GET,OPTIONS
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Security-Policy: block-all-mixed-content; base-uri 'self'; default-src 'self';
script-src 'self' 'unsafe-inline'; style-src 'self' 'unsafe-inline'; img-src 'self'
data:; connect-src 'self'; font-src 'self'; object-src 'none'; media-src 'self'; form-
action 'self'; frame-ancestors 'self';

{"imdata":[]}
```

As you can see in Example 7-13, the static route for 192.168.1.0/24 was deleted from the switch.

Example 7-13 *Static Routes on the Switch Before and After the DELETE Request*

```
! Before sending the DELETE request
sbx-n9kv-ao# show ip route
IP Route Table for VRF "default"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

10.98.98.0/24, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:20:50, direct
10.98.98.1/32, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:20:50, local
192.168.1.0/24, ubest/mbest: 1/0
    *via 10.10.20.254%management, [1/0], 00:00:08, static, tag 1000

! After sending the DELETE request
sbx-n9kv-ao# show ip route
IP Route Table for VRF "default"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

10.98.98.0/24, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:22:09, direct
10.98.98.1/32, ubest/mbest: 1/0, attached
    *via 10.98.98.1, Lo98, [0/0], 00:22:09, local
```

A DELETE method deletes the resource altogether. A subsequent GET request to the deleted resource should return the response code "404 Not Found."

CONNECT

The CONNECT method is used to create a tunnel from a client to a server. It is specifically used in situations where a proxy server is used to connect a client to the outside world. The server that the client is attempting to connect to is the *request target*.

Say that Client-1 needs to establish a TCP connection to Server-1 on port 80, and Client-1 is configured to forward all its HTTP requests to

Proxy-1. In this case, the CONNECT method is used as follows:

Step 1. Client-1 sends the CONNECT request to Proxy-1, indicating that the request target is Server-1 and that the port is 80.

Step 2. The proxy server attempts to set up a TCP session to Server-1 on port 80.

Step 3. When and if a successful connection from Proxy-1 to Server-1 is established, Proxy-1 sends a "200 OK" message back to the client, indicating that the tunnel to the request target is up.

Step 4. From this point onward, for messages between Client-1 and Server-1, Proxy-1 functions strictly as a two-way relay.

Step 5. Proxy-1 closes its connection when it detects a closed connection from either side. When this happens, the proxy server forwards any pending data that came from the side that closed the connection and closes its own connections.

But why would a client need to establish a tunnel to a remote server? This tunnel could eventually be secured using TLS in order to implement HTTPS, as you will see in Chapter 8. Moreover, the client could request that the tunnel use some TCP port other than 80, such that the messages from the client to the proxy are wrapped in HTTP, unwrapped at the proxy, and forwarded to the request target as a protocol other than HTTP, as determined by the TCP port used.

Cisco network devices running a version of IOS XR, NX-OS, or IOS XE do *not* support the CONNECT method.

OPTIONS

The OPTIONS method is used in a request when the client needs to know the options and/or requirements for communicating with a resource. For example, a read-only resource might only accept/support the GET and HEAD methods, such as operational data from a router. On the other hand, the resources constituting the router configuration might additionally support the POST, PUT and DELETE methods. In each case, the server would respond with the supported methods (for that particular resource) listed in the Allow header field.

The OPTIONS method can also be used to determine the capabilities of a server (for example, whether the server supports HTTP/1.1). This is similar to testing reachability using a ping.

Example 7-14 shows an OPTIONS request where the target is a JPEG image. The response in the same example shows the allowed methods listed in the value of the Allow header field.

Example 7-14 *Using the OPTIONS Method to Query the Allowed Methods for a Specific Resource*

```
! Client request using the OPTIONS method

OPTIONS /home/netdev/Downloads/Postman/app/resources/app/html/thenetdev.com/NetDev.jpg

Cache-Control: no-cache

Postman-Token: e937c593-ebb2-4179-9d35-6bfc2e20661f

User-Agent: PostmanRuntime/7.4.0

Accept: */*

Host: thenetdev.com

Accept-Encoding: gzip, deflate

Content-Length:


! Server Response indicating the allowed methods

HTTP/1.1 200 OK

Date: Sat, 22 Dec 2018 17:08:58 GMT

Server: Apache/2.4.6 (CentOS)

Allow: GET,HEAD,POST,OPTIONS,TRACE

Content-Length: 0

Keep-Alive: timeout=5, max=100

Connection: Keep-Alive

Content-Type: image/jpeg
```

TRACE

A client request using the TRACE method is looped by the final recipient of the request back to the client after excluding some header fields. The loopback functionality allows the client to know exactly what information the target server receives when it sends a request to it. The response (that is, the looped-back message) is a "200 OK" message with the client's original request included in the response body and the Content-Type header set to the value message/http.

Server Status Codes

Servers use status codes to indicate the status of attempts to understand and process client requests. A client may be requesting a web page or the running configuration of a router, content that the server would ideally return in a response message back to the client. In this case, the status code would indicate a successful request, typically using a 2xx code. The content that the client requested might not be available, or the method in the client request might not be allowed for the specific resource targeted in the URI, in which case the server would return a 4xx status code, such as the notorious "403 Forbidden" code that you would receive when requesting a resource that you are not authorized to access.

RFC 7231 defines the following status code categories for HTTP/1.1:

- 1xx: Informational
- 2xx: Successful
- 3xx: Redirection

• 4xx: Client error

• 5xx: Sever error

Table 7-1 lists all server codes defined in RFCs 7230 to 7235, along with the meaning of each one.

Table 7-1 *Server Response Codes Listed in RFC 7231*

Code	Meaning
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
426	Upgrade Required
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

A detailed description of each status code and its associated triggers is beyond the scope of this book and is not a prerequisite for a comprehensive understanding of network programmability. Therefore, this section discusses the most common codes briefly. For further information, see Section 6 of RFC 7231, Section 4 of RFC 7232, Section 4 of RFC 7233, and Section 3 of RFC 7235.

Note

The list shown in Table 7-1 is not exhaustive. A full list of server response codes that is maintained by IANA is available at https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.

1xx: Informational Status Codes

1xx status codes indicate informational server responses. The purpose of these messages is to convey the current (interim) status of the connection or transaction before the final response is sent to the client.

A client may send a request that should include a message body, and instead of sending the (potentially big) message body, it sends only the start line and headers. For example, with a POST request to a router that contains lengthy configuration, the client may use the Expect header field with the value 100-continue to indicate to the server that there is a message body to follow in the next message. If the received portion is okay, the server sends back a response with the status code "100 Continue" to inform the client that the status line and headers are okay and that it is expecting the body in the next message. If the client were to send several hundred lines of configuration in a POST request without being authenticated, the message body would be sent in vain. In the event that the server has a problem with the start line and/or headers, it reverts to the client with one of the 4xx or 5xx error codes.

The client or server may indicate to the other side a preference to switch the application layer protocol used in a connection to another protocol. The other protocol may be another version of HTTP or an entirely different protocol (that understands the HTTP semantics of the current message). This is accomplished using the Upgrade request header field. When a client lists one or more protocols that it wishes to switch to, the server may respond with a "101 Switching Protocols" status code to indicate that it agrees to switch to one of the protocols in the Upgrade header field of the request, and it includes an Upgrade header field of its own in the 101 response to specify which protocol it will switch to immediately after the empty line that terminates the 101 response.

2xx: Successful Status Codes

2xx status codes indicate that client requests have been successfully received and processed.

Status code "200 OK" means that the request was successfully received and processed and that a payload body is present in this response message, containing a representation that depends on the request method. For example, a 200 response to a GET request contains a payload that is a representation of the resource targeted by the URI in the request. Every 200 response has a payload, except for responses to CONNECT requests.

In the event that the processing of a request results in creating a resource, status code "201 Created" is used. The newly created resource may be referenced by adding a Location header field in the response with a value that is the URI of the new resource. If no Location header field is added to the response, the new resource's URI is the same as the target URI in the request.

Status code "202 Accepted" is used when the server successfully accepts a request but has not processed it yet. A server may, for example, do batch processing at preset times of day, and there is no point in keeping an active connection to each and every client that sends input to this batch process.

Status code "203 Non-Authoritative Information" is used in successful responses when the response is coming from a proxy server, not the origin server, and the content in the payload has been modified by that proxy.

The status codes "204 No Content" and "205 Reset Content" both indicate successful responses that have no payload. The difference between them is that 204 does not imply a change of "document view," as is the case when a user saves a document. A 204 response to a request to save indicates that the document has been saved successfully, but the view does not change. However, with a 205 response, the view is reset; for example, when a user fills the fields of a form and submits the data in a request, the fields of the form are reset to empty fields for the next record to be entered. A 205 response resets the form.

3xx: Redirection Status Codes

3xx status codes indicate that the server is expecting further action from the client before the request can be processed successfully. This further action is mostly likely to be another request initiated by the client, using another URI provided by the server (hence the Redirection label). Redirection heavily utilizes the Location header field, as you will see shortly.

When a server can provide more than one choice of representation for a resource targeted by the client in the request URI, status code "300 Multiple Choices" is used in the response. Each of these different representations has a more specific URI than that in the request URI. These different choices are provided back to the client in the payload body of the response, and the Location header field contains the server's preferred choice, if one exists.

Status codes "301 Moved Permanently," "302 Found," and "307 Temporary Redirect" are used in the server response to indicate to the client that the resource referenced by the client in the request URI is available at a different URI. This different URI is provided in the Location header field.

Status code "303 See Other" is used by the server to redirect the client to a totally different resource. The URI for this other resource is provided in the Location header field.

In the event that the request from the client is received by a cache server, and the cache server needs to redirect the client to another URI representing a previously cached version of the resource, that URI is provided in the Location header field, in a response that uses a "304 Not Modified" status code.

A server may send a 3xx redirect response to a request that uses any HTTP method. When a client receives a 3xx response, it may automatically re-send the original request by using the new URI communicated by the server in its response, or it may ignore the redirection advice from the server.

4xx: Server Error Status Codes

Servers send 4xx status codes in responses to tell clients that they believe there is something wrong with the received requests. The representation in the response includes further elaboration on this error condition.

Status code "400 Bad Request" is a general status code indicating that there is something wrong with the received request, and hence this request will not be processed.

Status code "402 Payment Required" is reserved and not currently used.

Status code "403 Forbidden" is used in a response when there is nothing wrong with the request syntax or semantics and the server has understood the request, but the client is not authorized to access the resource targeted in the request URI. Due to the obvious security vulnerability presented by this status code, the server may respond with status code "404 Not Found" in order to hide the fact that the resource exists in the first place. Using a PUT, POST, or DELETE method on a resource, for example, with read-only authorization credentials would typically trigger a 403 response code.

Status code "404 Not Found" means that the server did not find or is not allowed to disclose a current representation of the resource targeted by the request URI. If this status is permanent—that is, if the resource is not available and will not be available in the foreseeable future—status code "410 Gone" may be used instead.

Status code "405 Method Not Allowed" is the typical response when a client uses a method that is not supported by the target resource, such as using a PUT or POST request method with a read-only resource. A response with this status code must also include the Allow entity header field, which tells the client which methods are supported by this resource.

When a client sends a request to a server, it may include proactive negotiation header fields, such as the Accept header field. These header fields, which are discussed further later in this chapter, provide information on the preferred representation that the client wishes to receive from the server for the resource targeted in its request URI. When the server is not able to provide a representation that meets the restrictions or constraints in those header fields, it responds with a status code "406 Not Acceptable."

When a client takes too long to complete a request, the server closes the connection with the client by sending a "408 Request Timeout" response that has a Connection header field with the value close.

When a client does not include the Content-Length header field in its request but the server requires it, the server responds with a "411 Length Required" status code.

Status codes "413 Payload Too Large," "414 URI Too Long," and "415 Unsupported Media Type" are self-explanatory.

As discussed earlier in this chapter, in the section "1xx: Informational Status Codes," the Expect header field has one and only one valid value, which is 100-continue. If the client uses any other values for this field in its request, the server should respond with the status code "417 Expectation Failed."

Finally, a server uses status code "426 Upgrade Required" to inform the client that it needs to upgrade the protocol used for this transaction. The protocol that the client should upgrade to is indicated in the Upgrade header field of the response. Recall that status code 101 is used by the *server* in its response when the *client* has a requirement for upgrading the protocol and the client uses the Upgrade header field in its request.

5xx: Client Error Status Codes

Servers use 5xx status codes in responses to tell clients that there is a problem from the server's side, and due to this problem, the client request will not be processed. The representation in the response includes further elaboration on the error condition.

Status code "500 Internal Server Error" is a general status code indicating that the server encountered an unexpected error condition, and hence this request will not be processed.

Status code "501 Not Implemented" is the response a server uses when a client requests an operation on a resource, and while there may be nothing wrong with the operation requested, the server is not configured to perform that operation. While this error code may seem identical to error code "405 Method Not Allowed," there is a subtle difference between them. A server uses error code 405 when the method used in the client request *is known* by the origin server *but not supported by the target resource*. A server uses error code 501, on the other hand, when it *does not recognize the request method* and is not capable of supporting it for *any resource*.

Proxy servers having issues with upstream servers use status codes "502 Bad Gateway" and "504 Gateway Timeout" in their responses to the clients to inform the clients of those problems.

When a transient issue is stopping a server, temporarily, from processing new requests (such as being already overloaded with requests or having high resource utilization), the server responds to client requests with a "503 Service Unavailable" status code.

A server responding with status code "505 HTTP Version Not Supported" due to the fact that the client is using an HTTP version that is not supported by the server includes a representation in the response that elaborates on the reasons and alternatives.

Server Status Codes on Cisco Devices

Not all clients are required to understand all status codes defined in the HTTP RFCs; they only need to understand the classes, such as 1xx, 2xx, and so on. Moreover, status codes are extensible. This means that a vendor may only leverage the status codes that are required to implement its own ecosystem of programmability and that it may add to the already defined list of codes, provided that it follows the proper registration procedure for those status codes it creates, as defined in Section 8.2 of RFC 7231.

Cisco network devices do not use the full range of server status codes. Chapter 14 shows status codes used by the RESTCONF protocol on Cisco devices, and Chapter 17 demonstrates a native RESTful API exposed by programmable Nexus switches and the use of status codes by the API.

HTTP Messages

As you already know, an HTTP message is either a client request or a server response. In either case, the message is composed of a start line followed by zero or more header fields and, optionally, a message body containing a resource representation. The general format of an HTTP message, whether a request or response, is shown in Figure 7-3.

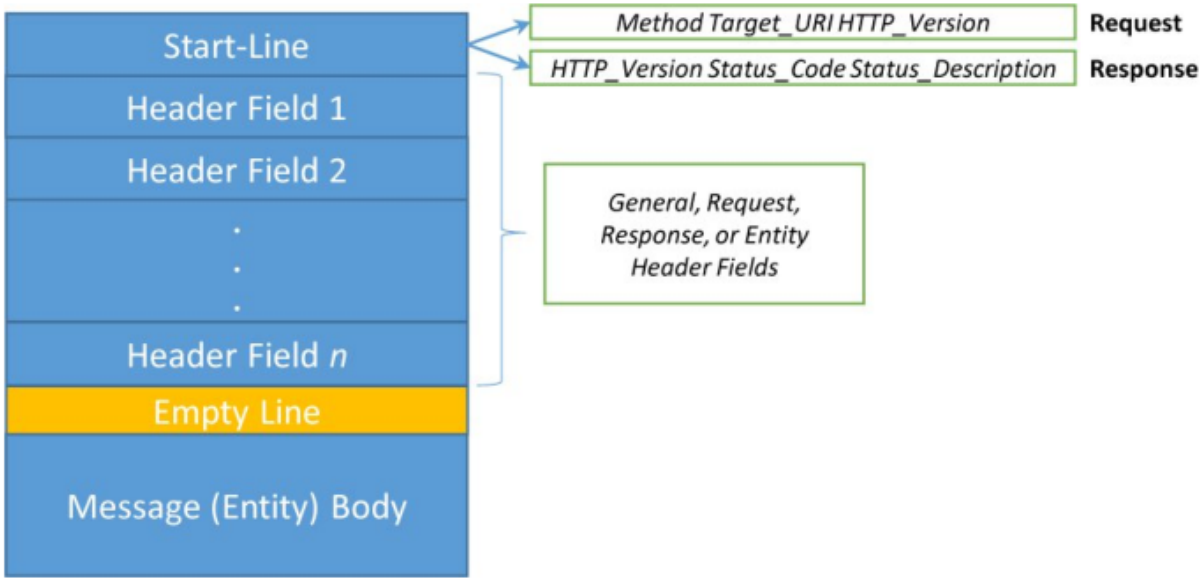


Figure 7-3 *The HTTP Message Format*

The start line in a request message is formatted as follows:

```
{Method} {Target_URI} {HTTP_Version}
```

Here is an example:

```
GET /images/SupportApache-small.png HTTP/1.1
```

The start line in a response message is formatted as follows:

```
{HTTP_Version} {Status_Code} {Status_Description}
```

Here is an example:

```
HTTP/1.1 200 OK
```

The start line is followed by a carriage return and line feed (CRLF), which basically moves the cursor to the beginning of a new line. It is called a CRLF for historic reasons, from the time when moving to the next line (line feed) was different from moving to the start of the line (carriage return).

Following the start line and trailing CRLF is the headers section, which is composed of a number of header fields, each formatted as {*name*}:{*value*}. The header field names are not case sensitive, but the values may be case sensitive, depending on the semantics of the specific header field.

Whitespaces are not allowed between the field name and the colon due to the security risks that this poses. An optional whitespace may be inserted before and/or after the field value for readability.

A header field with the same name may appear more than once on separate lines in a message, with different values. These different lines may be combined into one line, with comma-separated values. For example, the header field Accept-Encoding may appear like this:

```
Accept-Encoding: gzip

Accept-Encoding: deflate
```

Or it might be combined into a single line, like this:

```
Accept-Encoding: gzip, deflate
```

Header fields provide detailed information about the client, server, connection, payload body, or message itself. RFC 2616 classifies HTTP message header fields into four types:

• **General header fields**: These are generic fields that describe values not specific to the message type (request or response), such as the system date or the connection parameters.

• **Request header fields**: These fields are found only in request messages and are request specific, such as the Host (server) or the User-Agent (client) details.

• **Response header fields**: These fields are only found in response messages and are response specific, such as the Last-Modified or Set-Cookie header fields.

• **Entity header fields**: These fields define the properties of the representation in the message body. Both request and response messages may contain entity header fields.

Figure 7-4 illustrates some of the header fields covered in this chapter, along with their types.

Figure 7-4 *HTTP Header Fields and Their Types*

Note

RFC 2616 has been made obsolete by RFCs 7230 through 7235, and in those new RFCs, the distinction by header field type is not very clear. For example, there is no mention of a general header category, and some of the previously identified General header fields are classified now as request header fields, such as the Connection and Cache-Control headers. This book sticks with the RFC 2616 classification due to its good logic. We do not mean that the current HTTP RFCs follow "bad logic"; however, we think RFC 2616 provides a better entry point to understanding HTTP for a reader just getting started with the protocol.

An HTTP message typically has a number of general header fields and either request header fields or response header fields, but not both, depending on whether the message is a request or a response. If the message has a payload, it also has a number of entity header fields.

Note that while RFC 7230 provides some guidance on good-practice ordering of header fields, it describes the ordering of the fields as "not significant." Therefore, you typically see HTTP messages with fields belonging to the different header types in no particular order. Consider the headers sections in the GET request to www.apache.org discussed earlier in this chapter, repeated in Example 7-15 for convenience.

Example 7-15 *The Content of the GET Request Message to www.apache.org*

```
GET / HTTP/1.1
Host: www.apache.org
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Fedora; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/70.0.3538.77 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,ar;q=0.8,es;q=0.7,ms;q=0.6
```

The headers section in the message starts with the Host header field, which is a request header field, followed by the Connection field, which is a general header field. Then the next five fields are all request header fields again. And because there is no entity in the message, there are no entity header fields.

Each header field in the message ends in a CRLF. The header section is separated from the message body by an empty line, which is basically two consecutive CRLFs.

There are permanent message header field names, and there are provisional message header field names. The latter may be defined by third parties and registered with the IETF. An updated list of all header fields can be found at https://www.iana.org/assignments/message-headers/message-headers.xhtml. RFC 3864 defines the procedure to register a new provisional header field name. As of this writing, there are approximately 340 permanent header fields, and while knowing what each and every one of these fields is used for may come in handy, it is not necessary for the scope and purpose of network programmability. The following sections briefly cover the most common header fields for each header type.

HTTP General Header Fields

The following are some of the commonly used general header fields:

- Cache-Control
- Pragma
- Expires
- Connection
- Date
- Upgrade
- Via
- Transfer-Encoding
- Trailer

Cache Servers

The first three header fields, Cache-Control, Pragma, and Expires, provide directives and information related to cache servers. A *cache server* is a server that stores responses from an *origin server* and, upon receiving a request from a client, responds to the client on behalf of the origin server (caching is covered in RFC 7234.) These three header fields provide metadata related to cache server operation in case any of these servers exist along the path between the client and the server.

Both clients and servers can use the Cache-Control header field to provide *directives* to all cache servers along the communications path. Directives are classified into either request directives or response directives.

Examples of request directive are no-cache, only-if-cached, no-store, and no-transform. The no-cache directive indicates that the client does not want to receive a cached response. On the other hand, the only-if-cached directive indicates that the client only wishes to receive a cached response. The client uses the no-store directive to request that the cache server delete the client request as soon as the request is satisfied and not cache any responses to that request unless that response was already cached before the client request was received. The no-transform directive indicates that the cache server should not transform the payload received from the origin server in any way (such as by changing an image format to save space on the cache).

Unlike the previous directives, the other request directives, max-age, max-stale, and min-refresh, are given values in the header field since they specify client requirements related to the *freshness* of the response. Using these directives, the Cache-Control header field is formatted as follows:

```
Cache-Control: [{directive} = {value}]
```

Response directives also include no-cache, no-store, and no-transform, which have the same meanings as when used as request directives. However, the no-cache directive may be specified for one or more fields only instead of for the whole message. In this case, the Cache-Control header field is formatted as follows:

```
Cache-Control: no-cache = "{field_name}"
```

The directives public and private in a response specify whether the response is intended for a single user or for the public. They also provide restrictions on caching the response by public and private caches. Much like the no-cache directive, the private directive may be specified for only one field or more than one field, and not necessarily for the whole message.

The other response directives are must-revalidate and proxy-revalidate, which specify that the response may not be reused without being revalidated with the origin server first, and max-age and s-maxage, which are used to specify how long before a response becomes *stale* (that is, expires and cannot be used).

The Pragma header field has a single possible value, no-cache, which means the same thing as the no-cache directive in the Cache-Control header field. The Pragma field provides backward compatibility with HTTP/1.0, since Cache-Control was implemented only starting with HTTP/1.1.

The Pragma field is used primarily in client requests. When both the Pragma and Cache-Control header fields are included in a request, both having the value no-cache, HTTP/1.1 caches just ignore the Pragma field.

Because the Cache-Control header field allows for directives other than the no-cache directive, it is common for clients to use the Pragma field for the no-cache directive and then use the Cache-Control field for other directives that target only HTTP/1.1 caches.

The Expires header field specifies the date/time after which the response becomes stale. An invalid value in this field, especially the value 0, indicates that the response already expired. Much like the Pragma header field, the Expires header field is used when a recipient does not implement the Cache-Control header field, and it is therefore ignored in the event that the message has a Cache-Control header field with the max-age or the s-maxage directives.

Connection

The Connection header field is used for HTTP connection management, and may have the value close or the value keep-alive. The Connection header field is discussed in the section "HTTP/1.1 Connection Enhancements," earlier in this chapter. The Connection header field may also have the value upgrade as explained in the upcoming section discussing the Upgrade header field.

Date

The Date header field contains the message origination date. The date format preferred by RFC 7231 is a fixed-length, single-zone (UTC) subset of the date and time specification used by the Internet Message Format, as per RFC 5322. It is case sensitive and formatted as follows:

```
Date: {day_name}, {day_number} {month_name} {year} {hour}:{minute}:{second} GMT
```

These are the fields in the Date header:

• ***day_name***: This field should be one of the following three-letter day names: Sun, Mon, Tue, Wed, Thu, Fri, or Sat.

• ***day_number***: This field is the two-digit day of the month.

• ***month***: This field is one of the following three-letter month names: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec.

• ***year***: This is the four-digit year number.

• ***hour***: This is the two-digit hour, in 24-hour format.

• ***minute***: This is the two-digit minute.

• **second**: This is the two-digit second.

The following is an example of a Date field from an earlier example in this chapter:

```
Date: Sat, 22 Dec 2018 17:08:58 GMT
```

Upgrade

The Upgrade header field is discussed earlier in this chapter, in the section "1xx: Informational Status Codes." A client or a server can use the Upgrade header field to indicate a wish to switch to an application layer protocol other that HTTP/1.1. Note that when the Upgrade header field is used, the Connection header field in the same message should have the value upgrade. The Upgrade header cannot be used to switch the transport protocol used (TCP in this case).

Via

The Via header field indicates that there are intermediate recipients of the HTTP message between the client and the server, in either direction, such as a proxy server. Each recipient receives the message, adds an entry to the Via header field, and then forwards it upstream toward its destination, whether the destination is the origin server (in the case of a request) or the client (in the case of a response).

Each recipient adds to the Via field the protocol name and protocol version it uses, its host URI, and the TCP port used. The protocol name may be dropped if it is HTTP, and the port may be dropped if it is the default TCP port 80. The host name may also be replaced with a *pseudonym* if the intermediary wishes to keep its host URI confidential. The general format of the Via field is as follows:

```
Via: [Protocol_Name]/{Protocol_Version} {Host_URI}:[Port]
```

For example, if a client sends a request to a server, passing through a proxy with host name proxy.example.com that uses protocol HTTP/1.1 and TCP port 80, the proxy server can add this Via header field before forwarding the request upstream toward the origin server:

```
Via: 1.1 proxy.example.com
```

Transfer-Encoding

The Transfer-Encoding header field is a comma-separated list of the coding schemes that are applied to the message body (not the headers section). The following coding schemes are currently supported by HTTP/1.1:

• chunked

• compress

• deflate

• gzip

RFC 7230 states that transfer encoding is applied to the payload body to ensure "safe transport" through a network. The primary purpose of transfer encoding is two-fold: to decrease the size of the message for more efficient transport through the network and to properly delimit HTTP messages, keeping in mind that there are no defined limits for an HTTP message size.

The chunked coding scheme breaks up the message payload body into smaller pieces, or chunks. Each chunk consists of a chunk size field indicating the chunk size, in bytes, followed by one or more optional chunk extensions that provide per-chunk metadata, and finally the chunk data. The last chunk has a chunk size of zero. An optional trailer containing header fields may follow the last chunk. A recipient knows that the chunked message transmission is complete when it receives a chunk with a chunk size of zero followed by an empty line, with an optional trailer in between. A client that is willing to receive the optional trailer following the last chunk in a response from the server should include the TE request header field in its request with the value trailers (the TE request header field is covered in the next section).

The compress, deflate, and gzip coding schemes are compression coding schemes that decrease the overall size of the message payload body.

A typical Transfer-Encoding header field would be as follows:

```
Transfer-Encoding: deflate, chunked
```

Any recipient along the path of the message can amend the message body coding and change the Transfer-Encoding header field to match these changes.

Trailer

When using the chunked transfer encoding, the Trailer header field lists the header fields that are included in the optional trailer after the last chunk, in the event that the sender opts to include that trailer. Note that some header fields are not allowed in the trailer due to security and other considerations.

Client Request Header Fields

The following are some of the commonly used request header fields:

• Accept

• Accept-Charset

• Accept-Encoding

- Accept-Language
- Authorization
- Proxy-Authorization
- Cookie
- Host
- Expect
- Max-Forwards
- From
- Referer
- User-Agent
- TE

Content Negotiation Header Fields

In cases where a server may provide the same information to a client using different representations, *content negotiation* allows the client to express its preferences or limitations with respect to which representation the server provides in its response.

The Accept, Accept-Charset, Accept-Encoding, and Accept-Language request header fields (referred to as the Accept-X header fields) enable a client to engage in content negotiation with the server.

Several algorithms are defined for content negotiation, including proactive and reactive negotiation. In reactive negotiation, the server provides the client with a list of options to choose from. Proactive negotiation, on the other hands, requires the client to make its preferences known to the server in its request.

The Accept-X header fields allow the client to express its preferences to the server when engaged in proactive content negotiation. If the server cannot honor the client preferences, the server may respond with the next best match, or it may respond to the client with a "406 Not Acceptable" response.

Each Accept-X header field is formatted as a comma-separated list of all the preferred values for that particular header field. Then an optional weight may be assigned to each value by using a qvalue that indicates the client's preference for that value. The qvalue is a number ranging from 0.001 to 1, where 0.001 is the least preferred and 1 is the most preferred. If the qvalue is omitted, the default value of 1 is assumed. A qvalue of 0 means that this value is explicitly not acceptable. Here is an example of an Accept-X header field:

```
Accept-X: Value1; q=0.7, Value2; q=0.4, Value3, Value4; q=0
```

This means the server should send the representation that uses Value3, and if such a representation does not exist, it should send the representation that uses Value1, and finally, it should send the representation that uses Value2. Value4 is explicitly not acceptable.

The Accept header field is used to indicate the media types that the client is willing to accept. The Accept-Charset header field is used to indicate the charsets that the client is willing to accept. The Accept-Encoding header field is used to indicate the content codings that the client is willing to accept. The Accept-Language header field is used to indicate the set of natural languages (not computer languages) that the client is willing to accept. Media types, content codings, and language tags are explained in the section "Representation Metadata Header Fields," later in this chapter.

Client Authentication Credentials

When a client attempts to apply a method to a resource requiring authentication, the server challenges the client in a response using status code "403 Forbidden" and its own response header fields (explained in the next section). At that point, the client uses the Authorization or Proxy-Authorization header fields to send its authentication credentials back to the server. The difference between the two header fields is that the Authorization header is used to respond to a challenge from the origin server, whereas the Proxy-Authorization header is used to respond to a challenge from a proxy server.

A client uses the Cookie header field to send a cookie to the server that was probably received earlier from the server in order to maintain the state of the HTTP session.

HTTP authentication is covered in RFC 7235, and HTTP state management is covered in RFC 6265. Both HTTP authentication and HTTP state management are discussed in detail in Chapter 8.

Host

The Host header field, which is explained in the section "Client/server Communication," earlier in this chapter, is used to specify which virtual host on a particular IP address the request pertains to.

Expect

The Expect header field, which is discussed in the section "1xx: Informational Status Codes," earlier in this chapter, has one possible value: 100-continue. A client uses this field to indicate to the server that the request body will follow in another message, and in the meantime, the client is expecting a response using the 100 status code.

Max-Forwards

The Max-Forwards header field indicates the maximum number of forwards allowed by proxies for a particular request. This field is used with the TRACE and OPTIONS request methods and acts much like the TTL value. Every proxy receiving the request is required to decrement the value of the Max-Forwards header field and forward it toward its final recipient. If the proxy finds that the Max-Forwards field is equal to zero, it must not forward the request any further but must respond to the client in place of the origin server (the final recipient of the request). This behavior is similar to the functioning of the **traceroute** utility.

Request Context

The From, Referer, and User-Agent header fields provide further information related to the request context.

Although it is a little counterintuitive, the URI mailto:netdev@example.com has no Authority component. The string netdev@example.com is actually the path component. How would you draw this conclusion? Remember that the authority is always preceded by a double slash (//), regardless of the scheme. In the URI https://netdev@example.com:1234/anypath, the authority is netdev@example.com:1234.

The *userinfo* subcomponent is not frequently used in HTTP, but when it is, it provides a username that is used for authorization purposes.

The *host* subcomponent is either an IPv4 or IPv6 address or a registered host name. If *host* matches the IPv4 dotted-decimal format, it is parsed as an IPv4 address. If it is enclosed in square brackets, then it is considered an IPv6 address. If neither of these two conditions apply, it is parsed as a registered host name.

The *generic* URI syntax does not mandate an authority component to be present in a URI. However, it is mandatory to have an authority component in an HTTP URI.

The *port* component specifies the protocol port used and is defined per scheme. In HTTP, it is the port used for the TCP connection, and it defaults to port 80 when not explicitly stated.

Path

The path component is composed of different segments, each separated from the next by a slash (/). The path has a hierarchical format, where the path segments decrease in significance and increase in specificity from left to right, similar to file locations on a computer system.

The format of a generic path component is */[segment_1]/[segment_2]/.../[segment_n]*. The path component begins with a slash (/) and is terminated by a question mark (?) when followed by a query, a pound symbol (#) when followed by a fragment, or nothing when the end of the path component marks the end of the URI.

Along with the query component, the path identifies a unique resource within the scope of the scheme and authority of the URI.

Query

The query component starts with a question mark (?) and is either terminated by a pound symbol (#) when followed by the fragment component or nothing when it is the last component in the URI. The query component provides non-hierarchical data that is required (in addition to the hierarchical data provided by the path) in order to uniquely identify a resource. An example of a query would be a specific parameter/value pair that identifies a specific record in a database, such as ?Name=NetDev. Some schemes, such as HTTP, allow the use of a query component, and other schemes, such as FTP, do not.

Fragment

The fragment component, if present in a URI, starts with the pound symbol (#) and is the last component in the URI. A fragment provides information that allows for the identification of a secondary resource that relates to the primary resource identified by the path and query.

A very common example of the use of fragments is when the resource is an HTML page, where the fragment component is used to identify a specific element on the page to which the browser scrolls when the page is first displayed. For example, the URI https://developer.cisco.com/docs/nx-os/#!getting-started/introduction opens the Open NX-OS documentation and scrolls to the Introduction under the Getting Started section, identified by the fragment #!getting-started/introduction.

As another example, when the resource is a comma-separated values document, the fragment identifies specific rows, columns, or cells. For example, the URI http://example.com/sheet.csv#col=10 identifies the tenth column in the resource sheet.csv on the server (authority) example.com.

Unlike the rest of the URI components, a fragment is dereferenced by the client, not the server, and the fragments are not sent to the server in the first place. When a resource is an HTML page, the client browser dereferences the fragment to know how to display the page to the user. As such, fragments are not scheme-specific and only depend on the media type of the resource referenced by the URI.

Characters

A URI is basically a set of characters used to identify a resource. Some characters, such as the slash (/), have special meanings when used in a URI and are referred to as *reserved* characters. Other characters, such the letters of the alphabet, just represent themselves and are referred to as *unreserved* characters. Together, the reserved and unreserved character sets are referred to as the *allowed* character set.

URIs frequently use the percent-encoding scheme. In this scheme, a character is encoded into its U.S. ASCII hexadecimal equivalent and used in the URI preceded by a percent site (%). For example, the **+** sign is encoded into %2B. The next few sections explain when and why percent encoding of characters is utilized in URIs.

Reserved Characters

Some characters are used in a URI to separate or delimit the different components of the URI discussed in the previous sections. The characters used to delimit a component (such as the scheme or the authority), which are referred to as gen-delims, are :, /, ?, #, [,], and @. For example, the Query component is delimited by the ? character. The characters used inside a component and used to delimit subcomponents, which are referred to as sub-delims, are !, $, &, ', (,), *, +,;, =, and ,. For example, the equal sign (=) is often used to delimit a parameter and its value inside a component.

When a reserved character must be used in a URI and not interpreted as a delimiting character, its percent-encoded value must be used. For example, an ampersand (&) is used in a URI to delimit the pairs in a sequence of parameter/value pairs. Therefore, if a URI contains an ampersand that should *not* be interpreted as a delimiter, the string %26 should be used instead.

Unreserved Characters

All characters that you are allowed to use in a URI and that have no special meaning are called *unreserved characters*. The set of unreserved characters is composed of all letters, decimal digits, the hyphen (-), period (.), underscore (_), and tilde (~).

Non-allowed Characters

Character that are neither in the reserved nor unreserved character sets, such as non-English characters, are encoded into UTF-8 and then inserted into the URI by using the percent-encoded format. For example, if a URI contains the character "LATIN SMALL LETTER E WITH ACUTE" (é), which is frequently used in the French language, the string %c3a9 is used in the URI to represent it.

Absolute and Relative References

A URI is a sequence of characters used to uniquely identify, or refer to, a resource. The URI is not the resource itself, and it also does not

specify how the URI is used to act on the resource that it refers to. Therefore, this sequence of characters is more formally referred to as a *URI reference*.

URI references exist in two different formats: absolute and relative.

An *absolute URI reference* is what you have seen so far throughout this chapter. For example, http://apache.org/img/support-apach.jpg is an absolute URI composed of a scheme, an authority, and a path. An absolute reference such as this has a distinctive hierarchy and is unique from any other absolute reference. An absolute reference has all the information required to resolve and eventually dereference the URI.

Say that the image support-apach.jpg under the directory img is moved from the server apache.org to another server, such as example.com. In this case, the absolute path becomes http://example.com/img/support-apach.jpg. Now imagine if this and tens or hundreds of other URI references are used in HTML code. In such a case, all these absolute references need to be updated with the new server address. That would be a mundane task and is one of the use cases for utilizing relative URI references.

A relative reference refers to a resource in relation to a specific context or a base URI. The base URI, at a minimum, is the scheme, but it might be the scheme and authority, or even a scheme, an authority, and a partial path. For our sample URI reference, if the context is http://apache.org, the relative reference would be /img/support-apach.jpg. Another relative reference on same web page is /img/the-apache-way.jpg. If the web page is moved to example.com, the relative references in the HTML code does not need to be updated since these references are relative to the server on which the HTML code is running.

Before a relative URI reference can be resolved, a reference resolution algorithm is applied to obtain the target URI. This algorithm is explained in a lot of detail in Section 5 of RFC 3986.

Postman

Several programs are available to assist with developing, testing, and sharing APIs. One such program, which is widely used, is Postman. The program's developers refer to Postman as an *API development environment* (*ADE*). You can use Postman to design, test, debug, and fine-tune the performance of APIs. You can use Postman to test an API that you have coded yourself or to test an API provided by someone else. For example, you can use Postman to test APIs provided by Google, Facebook, or Twitter to retrieve data. For example, you can use an API provided by Google Maps to retrieve the latitude and longitude of an address provided to the Maps API as a parameter. Since the majority of network device vendors expose APIs on their devices to be consumed by third parties, Postman can be used to make API calls to a network device to either retrieve configuration or operational data or to actually configure the device.

Postman enables you to execute individual API calls (requests) or to group API calls into *collections* and run all the APIs in a collection sequentially, using a facility called *Collection Runner*. An individual API call to retrieve the routing table on a device may be executed individually. Alternatively, a group of API calls to retrieve the routing table, CDP neighbor information, and interface configurations may be executed one after the other, automatically, as one collection.

Running a collection is not the only action that can be taken on a group of saved requests. A collection can be mocked and monitored as well. Mocking a collection means providing the expected presaved results for an API call. Developers of an API can use these presaved results to observe the response to a request, even if the HTTP server is not ready yet, and cannot provide actual responses. Monitoring, on the other hand, involves running a collection periodically, at a set frequency, and sending alerts to the user when specific triggers occur.

It is possible to run Java scripts from inside Postman before and after API calls. Environment variables with different scopes can also be used to store values from one API call to be used in another API call. Postman also provides advanced facilities such as automated API documentation and automatic API testing for continuous integration and continuous deployment (CI/CD) scenarios. It can also be used to publish APIs to other developers so they can collaborate on API development. These are only a few of the features provided by Postman.

Despite Postman's advanced features, this section only shows how to use Postman to construct HTTP request messages, send those messages to a server, and then examine the responses received. In other words, in this section, Postman is used as an HTTP client, and it provides the advantage of granular visibility into the contents of requests and responses.

The HTTP server in this case may be anything from a Windows or Linux machine to a Cisco switch or router. The important thing is that this server or network device must be running an HTTP server, such as the Apache web server. For the purpose of this chapter, an Open NX-OS sandbox from Cisco DevNet is used for testing. NX-OS is based on a Linux distro called Wind River, customized under a project called the Yocto Project. NX-OS also runs an NGINX web server that handles the HTTP transport for the APIs exposed by the box. NX-OS provides several APIs through which configuration and operational data can be sent to and from the switch—some RESTful and some not. NX-OS programmability is covered in more detail in Chapter 17.

Downloading and Installing Postman

Postman was originally developed as an extension to Google Chrome. Now the Chrome extension has been deprecated, and two version of Postman are being actively developed. One is a native app that is available for the Windows, Linux, and Mac OS X operating systems. The other version is a web app that you may run from your browser.

To download and install the native Postman app, go to the URI https://www.postman.com/downloads/. The web page will automatically detect your operating system and provide a big orange button labeled "Download the App". Click the Download button and a window pops up, asking for the location to save the file. Choose a location and click OK. The downloaded file is a .gzip tar archive. Keep in mind that the download web page may change with time.

After the file has been downloaded to the location of your choice, extract the files using the command **tar -xvf Postman-linux-x64-**{*version*}**.tar.gz**. A new subdirectory named Postman is created. The executable file that runs Postman is a file inside the Postman directory, also called Postman. To run Postman, use the command **./Postman**. (Review Chapter 2, "Linux Fundamentals," for a refresher on these commands.)

At this point, you can start using the program. However, to make Postman accessible from the desktop, on a CentOS 8 system, via the Applications/Programs menu or by pressing on the Activities menu and searching for Postman through the Search box (or the vertical shortcuts bar at the left of the screen), you need to create a file and name it **Postman.desktop** under the directory ~/.local/share/applications. Add the lines shown in Example 7-16 to the file, replacing [*YOUR_INSTALL_DIR*] with the path to the directory in which the downloaded file was extracted.

Example 7-16 *The Content of the Postman.desktop File*

```
[Desktop Entry]

Encoding=UTF-8

Name=Postman

Exec=[YOUR_INSTALL_DIR]/Postman/app/Postman %U

Icon=[YOUR_INSTALL_DIR]/Postman/app/resources/app/assets/icon.png
```

```
Terminal=false

Type=Application

Categories=Development;
```

The Postman Interface

When you run Postman, a launch window appears. Uncheck the Show This Window on Launch box in the bottom-left corner and close the window. The normal Postman interface appears, as shown in Figure 7-6.

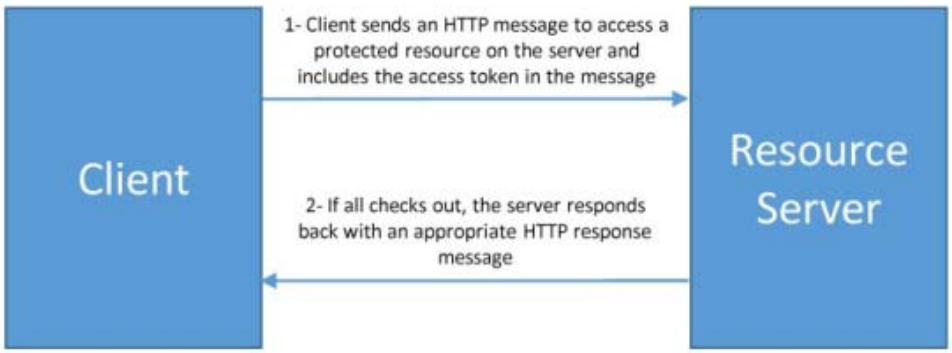


Figure 7-6 *The Postman Interface*

As indicated in the figure, the workspace is composed of a system menu and header bar at the top of the workspace, a sidebar to the left of the workspace, and the main operations area, called the Builder, in the center. In addition, there is a footer bar at the bottom of the workspace.

Note: Before proceeding with the description of the Postman interface, keep in mind that Postman is a very dynamic program and the interface and features seem to change on a *daily* basis. So please take the description here with a grain of salt. The discussion here is based on version 7.24.0. If at the time of reading this a newer interface is released, you may download the version on which this description is based from https://dl.pstmn.io/download/version/7.24.0/lin64. Once you are confident with what you have learnt here, go ahead and upgrade to the latest version. Porting your knowledge to a newer user interface will not be challenging at all.

The header bar is a toolbar with a number of buttons and menus that perform different functions. From left to right, these are the buttons and menus:

• **New button**: Used to create different Postman objects, such as requests and collections.

• **Import button**: Used to import objects, such as collections, into the current workspace. Several formats can be used to import objects, such as a JSON file or a direct import from a URI.

• **Runner button**: Used to start the Collection Runner, which is used to run all requests in a collection automatically.

• **Open New button**: Used to open a new request tab, a new Postman window, or a new Collection Runner window.

• **Workspaces menu button**: Used to manage workspaces. A new workspace has no collections or history entries. A workspace may be a personal or team workspace. With a team workspace, more than one person can access the workspace and can have default read-only access to the collections in that workspace. The access can be elevated for any workspace team member to be able to edit a collection.

• **Invite button**: Used to invite someone to collaborate on a workspace.

• **Sync icon**: Used to sync a user's objects, including workspaces and the collections inside them, between all the computers on which the user is using Postman. When the sync process is in progress, the icon is blue. When it turns orange, all computers are up to date.

• **Interceptor/Proxy icon**: Used to manage proxy or interceptor settings when Postman is used to intercept HTTP messages by acting as a proxy server to the source of the requests.

• **Settings icon**: Used to open a menu for settings and support resources.

• **Notifications icon**: Used to open the list of application notifications.

• **Heart icon**: Used to go to an external web page from which you can share Postman posts on social media.

• **Manage accounts icon**: Used to access user and profile options.

The sidebar has three tabs:

• **History tab**: The History tab shows a list of HTTP requests that have previously been executed.

• **Collections tab**: Collections are used to group saved requests together and, optionally, to act on a whole collection as one entity, such as to run all requests in a collection automatically, using Collection Runner, share the collection with another user or team workspace, and mock or monitor the collection.

• **APIs tab**: The APIs tab is used for end-to-end API design, implementation, and testing.

The footer bar is the thin bar at the bottom of the workspace that has a number of buttons. From left to right, these are the buttons:

• **Hide Side Bar**: Used to toggle between hiding and showing the sidebar.

• **Find**: Used to search for text strings in the current workspace.

• **Postman Console**: Used to open the Postman Console, which provides a "raw" view of requests and responses.

• **Learning Center (Bootcamp)**: Used to open a list of tasks that you may want to learn. When you choose a task from the list, a group of balloon windows walk you through the task on the Postman interface.

• **Build/Browse**: Used to toggle between two different views. The Build button is oriented toward constructing and sending requests and receiving responses, and the Browse button is oriented toward browsing and managing the different objects in the workspace.

```
}
```

When authentication is successful, the switch sends back a cookie that will be used to authenticate all subsequent HTTP requests. This is the value of the token in the output in Example 7-18, and it is also the value of the Set-Cookie header field that can be viewed under the Headers tab in the Response section of the Postman window. Cookies are covered in detail in Chapter 8.

To retrieve any configuration from the switch, a GET request is used with the appropriate URL. As you read through this book, you will learn how to construct URIs for performing particular tasks on a number of different platforms. For now, the BGP configuration on the switch is represented by the URI https://sbx-nxos-mgmt.cisco.com/api/mo/sys/bgp/inst.json. Therefore, open a new Request tab, choose the GET method if it is not already chosen, enter the URI into the corresponding text box, and click the Send button.

In the Response section of the Builder, you should see output similar to the output shown in Example 7-19. This is a JSON dictionary that contains all the information pertaining to the BGP instance running on the switch.

Example 7-19 *The Response to the GET Request for the Switch BGP Configuration*

```
{
    "totalCount": "1",
    "imdata": [
        {
            "bgpInst": {
                "attributes": {
                    "activateTs": "2019-05-09T21:13:28.527+00:00",
                    "adminSt": "enabled",
                    "affGrpActv": "0",
                    "asPathDbSz": "0",
                    "asn": "65535",
                    "attribDbSz": "120",
                    "childAction": "",
                    "createTs": "2019-05-09T21:12:56.487+00:00",
                    "ctrl": "fastExtFallover",
                    "disPolBatch": "disabled",
                    "disPolBatchv4PfxLst": "",
                    "disPolBatchv6PfxLst": "",
                    "dn": "sys/bgp/inst",
                    "epeActivePeers": "0",
                    "epeConfiguredPeers": "0",
                    "fabricSoo": "unknown:unknown:0:0",
                    "flushRoutes": "disabled",
                    "isolate": "disabled",
                    "lnkStClnt": "inactive",
                    "lnkStSrvr": "inactive",
                    "medDampIntvl": "0",
                    "memAlert": "normal",
                    "modTs": "2018-09-18T18:42:44.190+00:00",
                    "name": "bgp",
                    "numAsPath": "0",
                    "numRtAttrib": "1",
                    "operErr": "",
                    "persistentOnReload": "true",
                    "srgbMaxLbl": "none",
                    "srgbMinLbl": "none",
                    "status": "",
                    "waitDoneTs": "2019-05-09T21:13:28.527+00:00"
                }
            }
        }
    ]
}
```

A similar JSON dictionary in a request body used with the POST method is used to configure the switch. Chapter 17 covers this in more detail.

Consider these two very useful hints when using Postman:

HTTP and Python

The previous sections discuss generating HTTP requests and sending them to a server by using Postman or directly from a Bash shell on a Linux machine and then manually parsing through the responses. While these methods are sufficient for testing purposes, they are neither efficient nor scalable, and therefore they are not the best solutions for real-life scenarios. Say, for example, that you are writing a third-party application or need to integrate with one, and you need to generate HTTP requests or parse through HTTP responses as part of a bigger program that performs other functions? This section covers how to work with HTTP using Python and some commonly used libraries.

TCP Over Python: The socket Module

As discussed earlier in this chapter, HTTP servers listen to HTTP requests on TCP port 80, and a client, sending an HTTP request to an HTTP server, opens a TCP connection to that HTTP server on port 80 and then sends the HTTP request. In Python, this can be accomplished by using the socket module.

In the context of networks, a *socket* is a software implementation of an internal network port on the local system. A program that wishes to send data over a network requests that the operating system create a network socket. At the other end of this socket is the remote system to which data is to be transmitted. When the program wishes to send data over the network to the remote system, it sends this data to the socket that the OS created. The OS then takes care of receiving this data on the socket, processing and encoding it accordingly, and placing it on the wire to be sent to the remote system. A similar network socket is created on the remote system. The program requests a socket from the OS by talking to the OS's *socket API*.

A network socket is identified by the IP address and port on the local system. When a program requests that the OS create a socket, the OS creates the socket and assigns to it a socket descriptor, also called a *handle*. This is usually a number that uniquely identifies this socket. The program can then use this descriptor or handle whenever it wishes to send data to the socket.

These are the steps a client needs to perform to send an HTTP request to a server using the socket module:

Step 1. Using the **socket.socket()** method, the client uses the socket API to request that the OS create a socket and assign a socket handle to it.

Step 2. The client connects to the destination server by using the {*socket_handle*}**.connect()** method.

Step 3. The client sends the HTTP request contents to the server by using the {*socket_handle*}**.send()** method.

Step 4. The client receives the server response by using the {*socket_handle*}**.recv()** method.

Step 5. The client closes the connection with the server by using the {*socket_handle*}**.close()** method.

Example 7-28 shows a Python code sample that opens a network socket to www.example.com on port 80 and sends the same GET request as in the previous section.

Example 7-28 *A GET Request to www.example.com Using the Python socket Library*

```
#!/usr/bin/env python

import socket

my_socket_handle = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

my_socket_handle.connect(('www.example.com',80))

request = 'GET http://www.example.com/ HTTP/1.1\n\n'.encode()

my_socket_handle.send(request)

data = my_socket_handle.recv(10000)

print(data.decode())

my_socket_handle.close()
```

In Example 7-28, two arguments are passed to the **socket.socket()** method in order to request a socket from the OS. The first, **socket.AF_INET**, defines the address family of the socket to be **AF_INET**. This is the family of IPv4 network sockets (as opposed to IPv6 network sockets of type **AF_INET6** or interprocess communications sockets of type **AF_UNIX**). The second argument, **socket.SOCK_STREAM**, defines the type of socket. A socket of type **SOCK_STREAM** is a TCP socket. A UDP socket would be of type **SOCK_DGRAM**. The socket handle value is assigned to the variable **my_socket_handle**.

In the second step, the client uses the my_**socket_handle.connect()** method to connect to www.example.com on port 80. This method accepts one argument, which is why double parentheses are used in the example. This argument is the URI or IP address of the server followed by a comma and then the port number.

The request line, which is currently a string, is encoded into a UTF-8 byte object using the {*string*}**.encode()** method and assigned to the variable **request**, which is used as the argument to the **my_socket_handle.send(request)** method to send the request line to the server. Notice in Example 7-28 that only the request line is sent to the server (and no headers), followed by two CRLFs, indicating the end of the header section.

The next line of code uses the method **my_socket_handle.recv(10000)** to receive the response back from the server. The number **10000**, known as the *bufsize*, is the number of bytes to be received from the server. The program in this example has been written this way to keep the code simple, but a more sophisticated form can use a loop, with a smaller bufsize, and a condition to test when no more data is being received from the server.

The data is decoded back to a string with the **string.decode()** method, and then the **print()** function is used to print the data received.

Finally, the **my_socket_handle.close()** method requests the OS to close this specific socket.

Example 7-29 shows the result of running the Example 7-28 code.

Example 7-29 *The Result of Running the Example 7-28 Python Code*

```
[NetDev@localhost HTTP]$ python http_socket.py

HTTP/1.1 200 OK

Accept-Ranges: bytes
```

```
Cache-Control: max-age=604800

Content-Type: text/html; charset=UTF-8

Date: Fri, 01 Feb 2019 23:27:58 GMT

Etag: "1541025663+gzip"

Expires: Fri, 08 Feb 2019 23:27:58 GMT

Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT

Server: ECS (dca/2469)

Vary: Accept-Encoding

X-Cache: HIT

Content-Length: 1270


<!doctype html>

<html>

<head>

    <title>Example Domain</title>


    <meta charset="utf-8" />

    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <style type="text/css">

    body {


--------- output truncated for brevity ---------


[NetDev@localhost HTTP]$
```

Note

The socket module is covered in more detail in the Python documentation at https://docs.python.org/3/library/socket.html.

The urllib Package

The urllib package is composed of the urllib.request, urllib.parse, urllib.error, and urllib.robotparser modules. In this section you will see how to use the urllib.request module to send GET and POST HTTP requests. Note that urllib supports other schemes, not just HTTP, but to keep the discussion relevant, this section covers only HTTP.

Note

Three other packages exist for Python: urllib2, urllib3, and urllib. The four packages urllib for Python 2, urllib for Python 3, urllib2, and urllib3 are different implementations of an HTTP client. There are subtle differences between these four packages that are beyond the scope of this book. These differences are not covered here because using the requests package, covered in the next section, is the recommended method of working with HTTP in Python. urllib is covered here to illustrate the different levels of abstraction available in Python, starting with the socket package, then urllib, and finally requests.

The urllib.request module eliminates the need to open a socket manually and then encode the HTTP request. Instead, a single line of code is required to send an HTTP request to a specific URI. Example 7-30 shows a three-line Python script that sends a GET HTTP request to the web page www.example.com. Compare this script with the script in Example 7-28, which uses the socket module.

Example 7-30 *A Simple Python Script Using urllib to Issue a GET Request to www.example.com*

```
import urllib.request


httphand = urllib.request.urlopen('http://www.example.com')


print(httphand.read())
```

After the module urllib.request is imported in the first line, a handle to the URI www.example.com is created by using the **urlopen()** function. A handle, as discussed earlier, is a reference to an object, such as a file or a URI. Then the **read()** function is used in the third line, on the handle **httphand**, to read the content of the URI—that is, to send a GET request and fetch a representation of the resource at the subject URI. Finally, the **print()** function is used to print out to the screen the content found at the URI.

Example 7-31 shows the result of running the script in Example 7-30.

Example 7-31 *The Result of Running the Python Code in Example 7-30*

```
[NetDev@localhost HTTP]$ python http_urllib.py
```

```
b'<!doctype html>\n<html>\n<head>\n    <title>Example Domain</title>\n\n    <meta
charset="utf-8" />\n    <meta http-equiv="Content-type" content="text/html; charset=utf-
8" />\n    <meta name="viewport" content="width=device-width, initial-scale=1" />\n
<style type="text/css">\n    body {\n        background-color: #f0f0f2;\n        margin:
0;\n        padding: 0;\n        font-family: "Open Sans", "Helvetica Neue", Helvetica,
Arial, sans-serif;\n        \n    }\n    div {\n        width: 600px;\n        margin:
5em auto;\n        padding: 50px;\n        background-color: #fff;\n        border-
radius: 1em;\n    }\n    a:link, a:visited {\n        color: #38488f;\n        text-
decoration: none;\n    }\n    @media (max-width: 700px) {\n        body {\n
background-color: #fff;\n        }\n        div {\n            width: auto;\n
margin: 0 auto;\n            border-radius: 0;\n            padding: 1em;\n        }\n
}\n    </style>    \n</head>\n\n<body>\n<div>\n    <h1>Example Domain</h1>\n    <p>This
domain is established to be used for illustrative examples in documents. You may use
this\n    domain in examples without prior coordination or asking for permission.</p>\n
<p><a href="http://www.iana.org/domains/example">More
information...</a></p>\n</div>\n</body>\n</html>\n'


[NetDev@localhost HTTP]$
```

The HTML code in Example 7-31 is found at the URI www.example.com and retrieved (and printed) by the Python script.

Notice in Example 7-31 that the HTTP headers do not appear in the output, as they do with the socket module. The response headers are viewed using the {*file_handle*}**.info()** method, as shown in the script in Example 7-32.

Example 7-32 *Printing the Response Headers by Using the* ***info()*** *Method*

```
import urllib.request


httphand = urllib.request.urlopen('http://www.example.com')


print(httphand.info())
```

Example 7-33 shows the result of running the script in Example 7-21. As you can see, only the response headers are printed out, as intended by the **print(httphand.info())** line.

Example 7-33 *The Result of Running the Script in Example 7-32*

```
[NetDev@localhost HTTP]$ python Scripts/http_urllib_response_headers.py
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Mon, 13 May 2019 21:42:19 GMT
Etag: "1541025663+gzip+ident"
Expires: Mon, 20 May 2019 21:42:19 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (dcb/7EEF)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1270
Connection: close


[NetDev@localhost HTTP]$
```

The second thing to notice in Example 7-31 is that the HTML code prints out as a sequence of characters without any line breaks. This is due to the fact that the **urlopen()** function returns a byte object. A byte object is a sequence of bytes, in contrast to a string, which is a series of characters. The output in Example 7-31 shows the ASCII equivalent of each byte, one after the other, including the **\n** character, which represents a new line.

To decode the byte object received, you use the **decode()** function. Example 7-34 shows the same script as in Example 7-33, except that the **decode()** function is used to decode the UTF-8 byte object back to a string.

Example 7-34 *Using the* ***decode()*** *Function to Decode the Output of the* ***read()*** *Function*

```
import urllib.request


httphand = urllib.request.urlopen('http://www.example.com')
```

```
print(httphand.read().decode('utf-8'))
```

This results in the same output as in Example 7-33, except that the \n character is actually interpreted into a new line. The result is shown in Example 7-35.

Example 7-35 *The Result of Running the Python Code in Example 7-34*

```
[NetDev@localhost HTTP]$ python http_urllib.py
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 50px;
        background-color: #fff;
        border-radius: 1em;
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        body {
            background-color: #fff;
        }
        div {
            width: auto;
            margin: 0 auto;
            border-radius: 0;
            padding: 1em;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents. You may use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
```

```
</div>

</body>

</html>

[NetDev@localhost HTTP]$
```

The **urlopen()** function in the previous examples is used with a URI as an argument. To pass more than just the URI to the function—for example, the HTTP headers or an HTTP message body—you can use the **urlopen()** function with a *request object* as an argument instead of just using the URI. This request object includes all information required to successfully construct an HTTP request. The request object is constructed using the **Request()** class. The following is a simplified form of the **Request()** class syntax:

```
{R_Object}=urllib.request.Request({url}[,data={data}][,headers={headers}][,method={HTTP_method}])
```

The first argument is the URI. The second argument is the data, which is the HTTP request payload body. Data is provided as a byte object. The third argument is the HTTP headers. This is provided as a Python dictionary, where each key/value pair is the header field name and value. Finally, the last argument is the HTTP request method.

To test all the features at once, the script in Example 7-36 can be used to construct an HTTP request that attempts to authenticate to the NX-OS switch.

Example 7-36 *Using urllib to Authenticate to a Nexus Switch*

```
import urllib.request, ssl, json

gcontext = ssl.SSLContext()

uri = "https://sbx-nxos-mgmt.cisco.com/api/aaaLogin.json"

body = {

     "aaaUser": {

       "attributes": {

         "name": "admin",

         "pwd": "Admin_1234!"

        }

    }

}

encoded_to_json = json.dumps(body)

data = encoded_to_json.encode()

headers = {"Content-Type":"application/json","Cache-Control":"no-cache"}

request_object = urllib.request.Request(url=uri,data=data,headers=headers,method='POST')

request = urllib.request.urlopen(request_object,context=gcontext)

print('\n')

print(request.read())
```

Recall that SSL certificate checking needs to be disabled on Postman before you attempt to authenticate to a switch. This is accomplished here via the code line **gcontext = ssl.SSLContext()** and then using **context=gcontext** as an argument in the **urlopen()** function later in the code. (At this point, this is about all you need to know, and you do not need to worry about SSL certificate verification specifics.)

After you import all the required modules, you create the different components required to build the request object. First, you assign the URI to the string variable **uri**. Then you assign the request body to the variable **body** as a nested dictionary. However, the switch is expecting the body of the HTTP request to be encoded in JSON, so the **json.dumps()** method is used to convert the Python dictionary to JSON. Then the **Request()** class is expecting the data argument to be a byte object, so the JSON body is encoded into a byte object using the **encode()** method and assigned to the variable **data**. Finally, the request headers are assigned to the variable **headers** as a dictionary.

When the URI, data and headers are properly constructed and encoded, they are passed to the **Request()** class, along with **method='POST'**, to create the request object **request_object**. Note that if no data is passed to the **Request()** class, the method defaults to GET; otherwise, the method defaults to POST, so, strictly speaking, **method='POST'** is not required, and it is added to the code for illustration purposes only.

The request object is then passed to the **urlopen()** function, resulting in an HTTP request constructed with the required parameters. Example 7-37 shows the result of running this script. Compare the response received here with the response received when the same authentication HTTP request is sent using Postman (refer to Example 7-18).

Example 7-37 *Authenticating to a Nexus Switch*

```
[NetDev@localhost ~]$ python Scripts/auth_python.py




b'{"imdata":[{"aaaLogin": {"attributes": {"token":

"nnrrUpqz4P17nBD26JZkxTwf7vf3ov5zJQV8ahFIFNxbmdIHbChr43ASjOQM4JKRpA0jP2upEb2PUS3XeAEPvwH

dpsVHpLVpF9UNwylo94n6kSRRGUuN048A/Oawde9XjgQkA6UFiaeY3ehEE/iBI0+NT5hHtHYjfZYCGK6j6gg=","

siteFingerprint": "","refreshTimeoutSeconds": "600","guiIdleTimeoutSeconds":

"1200","restTimeoutSeconds": "1920151406","creationTime": "1558145939","firstLoginTime":

"1558145939","userName": "admin","remoteUser": "false","unixUserId": "0","sessionId":

"VPNv91T+yeoBAAAAFwAAAA==","lastName": "","firstName": "","version":

"0.9(14HEAD${version.patch})","buildTime": "Tue Jul 17 15:13:26 PDT
```

```
2018","controllerId": "0"},"children": [{"aaaUserDomain": {"attributes": {"name":
"all","rolesR": "admin","rolesW": "admin"},"children": [{"aaaReadRoles": {"attributes":
{}}},{"aaaWriteRoles": {"attributes": {},"children": [{"role": {"attributes": {"name":
"network-admin"}}}]}]}]}}]}]'
[NetDev@localhost ~]$
```

The requests Package

The requests package has been developed to further abstract and simplify the process of working with HTTP using Python. The Python Software Foundation recommends using the requests package whenever a higher-level HTTP client interface is needed.

The requests package is based on urllib3 and is not part of the standard library (as urllib and urllib2 are), so it has to be manually installed. (Third-party packages are discussed in Chapter 6, "Python Applications.") Example 7-38 shows how to install requests by using **pip3.7**.

Example 7-38 *Installing the requests Package by Using* ***pip***

```
[NetDev@server1 ~]$ sudo pip3.7 install requests
Collecting requests
  Using cached https://files.pythonhosted.org/packages/51/bd/23c926cd341ea6b7dd0b2a00aba99ae0f828be89d7
2b2190f27c11d4b7fb/requests-2.22.0-py2.py3-none-any.whl
Requirement already satisfied: idna<2.9,>=2.5 in /usr/local/lib/python3.7/site-packages
(from requests) (2.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/site-
packages (from requests) (2018.10.15)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python3.7/site-
packages (from requests) (3.0.4)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/site-packages (from requests) (1.24.1)
Installing collected packages: requests
Successfully installed requests-2.22.0
[NetDev@localhost ~]$
```

When the requests package is installed, you are ready to import it into your code.

Using the requests package is primarily based on creating a *response object* and then extracting the required information from that response object. This is the basic syntax for creating a response object:

```
{R_Object}=requests.{method}({uri}[,params={query_parameter}][,headers={headers}][,dat
a={payload_body}])
```

To create an object for a GET request, you use **requests.get**, for a POST request, you use **requests.post**, and so forth.

The **params** parameter holds the value of the key/value pairs that will constitute the query part of the URI. The **headers** parameter holds the headers, and the **data** parameter holds the request message payload (body). All three parameters are Python dictionaries, except for the **data** parameter, which can be provided as a dictionary, a string, or a list. The parameter **data=**{*payload*} may be replaced by **json=**{*payload*}, in which case the payload is encoded into JSON automatically.

You can extract the following response information from the created object:

• {*Response_object*}.**content**: The response from the server as a byte object (not decoded).

• {*Response_object*}.**text**: The decoded response (payload body) from the server. The encoding is chosen automatically based on an educated guess.

• {*Response_object*}.**encoding**: The encoding used to convert {*Response_object*}.**content** to {*Response_object*}.**text**. You can manually set this to a specific encoding of your choice.

• {*Response_object*}.**json()**: The decoded response (payload body) from the server encoded in JSON, if the response resembles a JSON object. (Otherwise, an error is returned.)

• {*Response_object*}.**url**: The actual URI used in the request, with all the different components included as parameters in **requests.**{*method*}**()**.

• {*Response_object*}.**status_code**: The response status code.

• {*Response_object*}.**request.headers**: The request headers.

• {*Response_object*}.**headers**: The response headers.

In Example 7-39, the response object res_obj holds the response data from a POST request sent to https://httpbin.org/post with a custom header named **My-Custom-Header** and a data payload consisting of the string **"THIS IS THE PAYLOAD BODY"**. Then the script prints out all the listed request and response parameter values.

Example 7-39 *POST Request Using the requests Package*

```
import requests


url = 'https://httpbin.org/post'
```

```
headers = {'My-Custom-Header': 'NetDev Doing a POST'}
parameters = {'Key-1':'Value-1','Key-2':'Value-2'}
payload = "THIS IS THE PAYLOAD BODY"

res_obj = requests.post(url,params=parameters,headers=headers,data=payload)

print('The Server Response as a byte object: ','\n\n',res_obj.content,'\n')
print('The decoded response (payload) from the server: ','\n\n',res_obj.text,'\n')
print('The encoding used to convert Response_object.content to Response_object.text:
',',\n\n', res_obj.encoding,'\n')
print('The actual URI used in the request (incl the query component):
',',\n\n',res_obj.url,'\n')
print('The response status code: ','\n\n',res_obj.status_code,'\n')
print('The request headers: ','\n\n',res_obj.request.headers,'\n')
print('The response headers :','\n\n',res_obj.headers,'\n')
```

Example 7-40 shows the result of running the code in Example 7-39.

Example 7-40 *The Information Extracted from the Request Object*

```
[NetDev@localhost Scripts]$ python requests_simple_1.py
The Server Response as a byte object:


 b'{\n  "args": {\n    "Key-1": "Value-1", \n    "Key-2": "Value-2"\n  }, \n  "data":
"THIS IS THE PAYLOAD BODY", \n  "files": {}, \n  "form": {}, \n  "headers": {\n
"Accept":
"*/*", \n    "Accept-Encoding": "gzip, deflate", \n    "Content-
Length": "19",
\n    "Host": "httpbin.org", \n    "My-Custom-Header": "NetDev Doing a POST", \n    "
User-Agent": "python-requests/2.20.1"\n  }, \n  "json": null, \n  "origin":
"51.36.2.121, 51.36.2.121", \n  "url": "https://httpbin.org/post?Key-1=Value-1&Key-
2=Value-2"\n}\n'


The decoded response (payload) from the server:


{
  "args": {
    "Key-1": "Value-1",
    "Key-2": "Value-2"
  },
  "data": "THIS IS THE PAYLOAD BODY",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "19",
    "Host": "httpbin.org",
    "My-Custom-Header": "NetDev Doing a POST",
    "User-Agent": "python-requests/2.20.1"
  },
  "json": null,
  "origin": "51.36.2.121, 51.36.2.121",
  "url": "https://httpbin.org/post?Key-1=Value-1&Key-2=Value-2"
}
```

```
The encoding used to convert Response_object.content to Response_object.text:

 None

The actual URI used in the request (incl the query component):

 https://httpbin.org/post?Key-1=Value-1&Key-2=Value-2

The response status code:

 200

The request headers:

 {'User-Agent': 'python-requests/2.20.1', 'Accept-Encoding': 'gzip, deflate', 'Accept':
'*/*', 'Connection': 'keep-alive', 'My-Custom-Header': 'NetDev Doing a POST', 'Content-
Length': '19'}

The response headers :

 {'Access-Control-Allow-Credentials': 'true', 'Access-Control-Allow-Origin': '*',
'Content-Encoding': 'gzip', 'Content-Type': 'application/json', 'Date': 'Wed, 15 May
2019 02:03:17 GMT', 'Referrer-Policy': 'no-referrer-when-downgrade', 'Server': 'nginx',
'X-Content-Type-Options': 'nosniff', 'X-Frame-Options': 'DENY', 'X-XSS-Protection': '1;
mode=block', 'Content-Length': '307', 'Connection': 'keep-alive'}


[NetDev@localhost Scripts]$
```

You can see from Example 7-40 that the requests package provides a more elegant solution to sending and receiving HTTP messages.

Summary

This chapter covers one of the most fundamental and extensively used protocols in web development and REST API development and use: HTTP. By now, you should be able to successfully construct HTTP requests using proper URIs, headers, and payload bodies and then navigate through the responses received. This chapter also covers the tools most commonly used today for working with HTTP. This chapter discusses the following topics:

- HTTP Overview
- The REST architectural framework and its relationship to the HTTP protocol
- The HTTP connection based on TCP and connection enhancement in HTTP/1.1
- Client request methods and server response codes
- HTTP messages and four types of header fields: general, request, response and entity header fields
- How resources are identified using URIs, URLs and URNs and the syntax rules of valid URIs
- Tools to automate working with HTTP and REST APIs: Postman, cURL, Bash, and the Python socket module, urllib package and requests package

Chapter 8 picks up where this chapter leaves off and covers advanced topics in HTTP such as authentication in HTTP, TLS and HTTPS, and HTTP/2.0.

Chapter 8. Advanced HTTP

This chapter picks up where Chapter 7, "HTTP and REST," leaves off and covers more advanced topics related to HTTP. The first section of the chapter describes the different types of HTTP authentication schemes, which provide a means to verify the identity of a client attempting to access a protected resource and, in some cases, verify the identity of the server back to the client. HTTP is stateless. Cookies provide a workaround to allow servers to maintain state information on client machines. This chapter covers cookies and their use in HTTP. This chapter also discusses the Transport Layer Security (TLS) protocol and HTTP over TLS (HTTPS). Using TLS tunnels, HTTPS provides encryption of HTTP requests and responses and, in turn, data confidentiality and integrity. Finally, this chapter covers the newer versions of HTTP, HTTP/2 and HTTP/3, and the enhancements they have introduced.

HTTP/1.1 Authentication

Authentication is a process through which the identity of a client is verified by an HTTP server and, possibly, vice versa. In addition to verifying the client identity, the HTTP server also needs to make sure that this particular client is authorized to access the resource addressed in its request to the server. The header names used in HTTP, as well as some literature, use the terms *authentication* and *authorization* interchangeably, but there are differences between them: Whereas authentication involves proving that users are who they say they are, authorization involves giving those user access to a particular resource.

There are several ways a server can authenticate clients attempting to access resources on that server. Some of these *authentication schemes* are native to HTTP, or extensions to it, and some are vendor specific. HTTP authentication in general is very extensible. For example, the Basic Authentication scheme uses the native HTTP headers WWW-Authenticate and Authorization to complete the authentication workflow. Alternatively, the product Cisco Meraki uses a vendor-defined header named X-Cisco-Meraki-API-Key to authenticate calls made to its Dashboard API. (You'll learn more about this in Chapter 17, "Programming Cisco Devices.") The value of this header is referred to as an *API key* and is manually generated from the Meraki GUI. The API key is included in each HTTP request to that API endpoint (URI).

This section focuses on standard schemes, which usually provide the foundation for the other, non-standard, schemes. The general concepts underlying HTTP authentication are covered in RFC 7235. The IANA maintains a registry of all registered authentication schemes for HTTP at http://www.iana.org/assignments/http-authschemes. This section covers three schemes:

- Basic Authentication
- OAuth and authentication tokens
- Cookies

Authentication in HTTP may operate using a *challenge/response* model, or it may operate using a direct, *unsolicited* model. In the first model, the client attempts to access a protected resource on a server without providing credentials. The server responds with a "401 Unauthorized" response message, which is the server's way of informing the client that the resource it is trying to access is protected and needs authentication. The server response also includes further details, such as what schemes the server supports and the realm (explained shortly). This is the server *challenge*. The client then re-sends its request, but this time, it includes the required credentials in the request, using the headers relevant to the scheme used. This is the client *response*.

The unsolicited model is used when the client knows beforehand that the resource it is trying to access is protected, knows what scheme the server is using to protect the resource, and has a valid set of credentials. In this case, the client preemptively sends its request, including the credentials required for authentication. The workflow includes only a response and not a challenge.

When a client is authenticating with a proxy server, the proxy uses a "407 Proxy Authentication Required" message to challenge the client instead of a "401 Unauthorized" message.

An HTTP server may define *protection spaces*, also known as *realms*, on the server. These realms partition the server resources into different spaces, and each space may use a different authentication scheme and/or credentials. Using realms is one way to allow a server to not just authenticate a client but also authorize it to access specific resources and not others. In other words, the same client may be authorized to access resources in one realm but not in another.

Figure 8-1 illustrates the authentication workflow in HTTP.

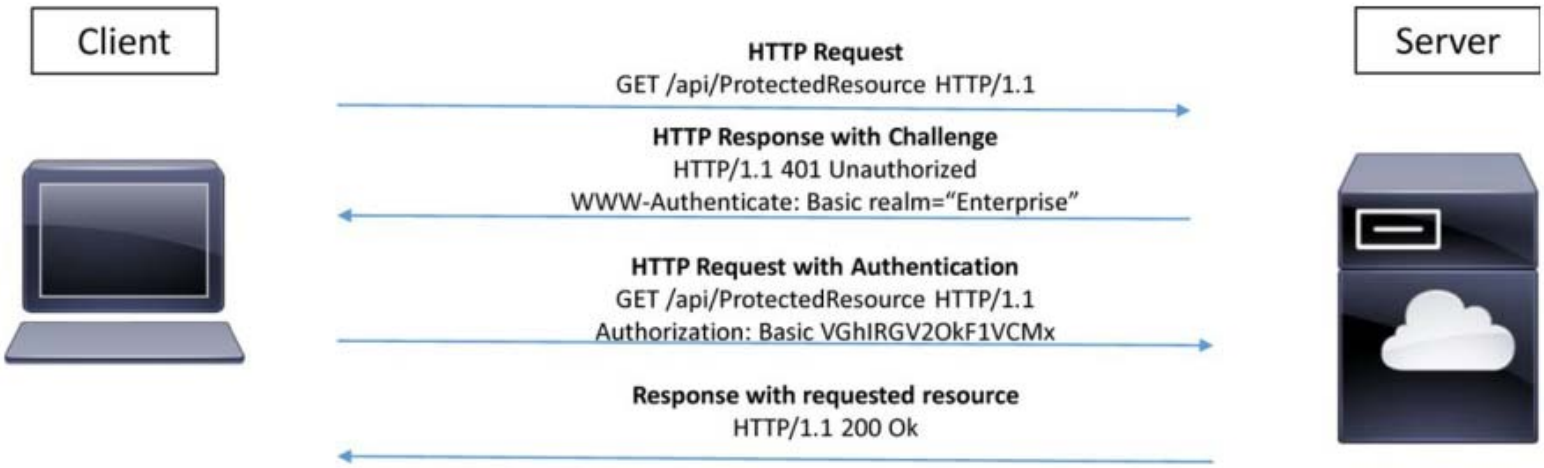


Figure 8-1 *HTTP Authentication Workflow*

The server challenge is a "401 Unauthorized" HTTP response message that includes the header WWW-Authenticate. The header field value is equal to the challenge value. The general format of the header field, which may contain one of more challenges, is as follows:

```
WWW-Authenticate: {challenge_1}[, challenge_2], . . .[, challenge_n]
```

Each of the challenges in the header field has the following general format:

```
{scheme} {parameter_1}={value_1}[, {parameter_2}={value_2},. .
., {parameter_n}={value_n}]
```

Each challenge is composed of the scheme name followed by a whitespace and then a list of parameter name/value pairs. The following is an example of a WWW-Authenticate header containing two challenges:

```
WWW-Authenticate: Digest realm="dev ", algorithm=SHA-256, Basic realm= "prod",

charset="UTF-8"
```

The first challenge in this example is for the Digest scheme and has two parameters: realm and algorithm. The second challenge is for the Basic scheme and has two parameters: realm and charset. Do not worry about the meaning of each of these parameters just yet; they are covered in detail later in this chapter. The goal here is to showcase the header field format only.

In the case of a client authenticating with a proxy server, the WWW-Authenticate header is replaced with the Proxy-Authenticate header that has the same format.

When the client has received the challenge, it should respond with the same request it sent earlier (that is, the one that triggered the challenge), but this time it should add an Authorization header field, which has the following general format:

```
Authorization: {scheme} {credentials}
```

Basic Authentication

Basic Authentication, as the name implies, is the most basic method of authenticating users attempting to access a resource on an HTTP server. Basic Authentication is covered in RFC 7617, and the scheme is identified by the scheme name Basic.

This scheme uses exactly the same response codes and header fields discussed earlier in this chapter for server HTTP challenge messages. The server uses the WWW-Authenticate header field in a "401 Unauthorized" message to challenge the client (or the Proxy-Authenticate header field in a "407 Proxy Authentication Required" message in the case of a proxy server). The general format of the WWW-Authenticate header field is as follows:

```
WWW-Authenticate: Basic realm="{realm_value}"[, charset="UTF-8"]
```

Apart from the scheme name, this scheme mandates that the protection space value be specified as the value of the realm parameter. The other parameter, charset, is optional and has only one valid value, UTF-8, because it is used to advise the client to use UTF-8 encoding when generating its Base64-encoded credentials to the server. (This is explained in detail shortly.) The scheme name and parameter names and values are all case insensitive, except for the realm value. The parameter values may also be expressed as quoted-string values, or tokens (without quotes).

The client responds with an HTTP request that includes the Authorization header field containing the required credentials (or a Proxy-Authorization header field in the case of authenticating with a proxy server). The Authorization header field used by the Basic scheme has the following general format:

```
Authorization: Basic {base64-encoded-credentials}
```

Base64 is an encoding scheme defined in RFC 4648 that was designed to represent a sequence of octets using a 65-character subset of the U.S. ASCII; this representation was not designed to be human readable. Base64 encoding is one type of *base encoding*, originally developed to support environments and systems that only support the U.S. ASCII character set. A simple search on the Internet will return several online Base64 encoders that automate the process of converting a string to its Base64 equivalent. It can be helpful to understand where the strange-looking string in the Authorization header came from and how it is generated.

The process to generate the *base64-encoded-credentials* token involves a few simple steps. Let's say that the client username is TheDev, and the password is AuT#1. The first step is to concatenate the username and password, separated by a colon—in this case, TheDev:AuT#1.

The next step is to encode this string using a data encoding scheme. There is no default encoding specified by the Base64 standard. The only restriction is that the encoding used has to be compatible with U.S. ASCII, which means that each U.S. ASCII character needs to map to one single byte. Table 8-1 shows how this works with UTF-8.

Table 8-1 *Mapping the Letters of the Username and Password to Their UTF-8 Codes*

Letter	UTF-8 Encoding (Decimal)	UTF-8 Encoding (Binary)
T	84	01010100
h	104	01101000
e	101	01100101
D	68	01000100
e	101	01100101
v	118	01110110
:	58	00111010
A	65	01000001
u	117	01110101
T	84	01010100
#	35	00100011
1	49	00110001

Note

Fore more on UTF-8 encoding, see https://unicode.org/faq/utf_bom.html#UTF8.

The last step is to encode the string into a Base64 token. This is done in four substeps:

1. Concatenate all 12 bytes:

```
01010100011010000110010101000100011001011011101100

01110100100000101110101010101000010001100110001
```

2. Split the bytes into 6-bit words:

```
010101-000110-100001-100101-010001-000110-010101-

110110-001110-100100-000101-110101-010101-000010-

001100-110001
```

3. Translate each 6-bit binary word into its decimal equivalent:

```
21-6-33-37-17-6-21-54-14-36-5-53-21-2-12-49
```

4. Using Table 8-2, match each decimal value from step 3 to its corresponding letter:

```
VGhIRGV2OkF1VCMx
```

For this example, the Authentication header field is as follows:

```
Authentication: Basic VGhIRGV2OkF1VCMx
```

Table 8-2 *Base64 Alphabet*

Value	Encoding		Value	Encoding		Value	Encoding		Value	Encoding
0	A		17	R		34	i		51	z
1	B		18	S		35	j		52	0
2	C		19	T		36	k		53	1
3	D		20	U		37	l		54	2
4	E		21	V		38	m		55	3
5	F		22	W		39	n		56	4
6	G		23	X		40	o		57	5
7	H		24	Y		41	p		58	6
8	I		25	Z		42	q		59	7
9	J		26	a		43	r		60	8
10	K		27	b		44	s		61	9
11	L		28	c		45	t		62	+
12	M		29	d		46	u		63	/
13	N		30	e		47	v			
14	O		31	f		48	w		(pad)	=
15	P		32	g		49	x			
16	Q		33	h		50	y			

As you can see, the Authentication header contains the Base64 encoding of the credentials in plaintext. Any entity capable of inspecting this HTTP response from the client can easily decode the string VGhlRGV2OkF1VCMx and learn the user credentials. This is one of the reasons Basic Authentication is almost never used on its own when security is considered to be essential. Basic Authentication is usually coupled with technologies that provide data confidentiality, such as TLS, and then used as an intermediate step in a more sophisticated authentication workflow, such as that used in OAuth 2.0. Both TLS and OAuth are covered in later sections of this chapter.

OAuth and Bearer Tokens

The OAuth protocol, or the OAuth *authorization framework*, as RFC 6749 calls it, was developed with a very specific authorization use case in mind: to enable a third-party application to access resources owned by another entity (which the RFC calls *restricted resources*) without the entity that owns these resources having to share its credentials with the application attempting to access the resources. That was a mouthful! But it is not as complicated as it sounds.

Let's consider an example. Say that you are trying to link your fitness application with your Facebook account so that the fitness app can retrieve your list of Facebook friends and check whether any of them uses the same app and then connect you together on the app. The app would also want to post your workouts to your Facebook timeline. The OAuth protocol is used to get your approval to allow the fitness app to access some or all of your Facebook data and then to authenticate and authorize the app with the Facebook authentication server. The app can then retrieve data from or post data to your Facebook account. The fitness app in this case is the third-party application attempting to access data, owned by you, residing on the Facebook server, without you having to share your Facebook credentials with the fitness app.

In order to describe the workflow for OAuth, the protocol defines four different *roles*. These roles are defined for a workflow that involves a resource hosted on a *resource server* (role 1). The resource is owned by a *resource owner* (role 2), and this owner may wish to grant a third-party application, referred to as the *client* (role 3), access to that resource; it does so by leveraging the services of an *authorization server* (role 4). We can map these roles to the previous example: The fitness app is the client. You are the resource owner. The Facebook server hosting your data is the resource server. The Facebook server that will authenticate you as the resource owner as well as the client attempting to access your data is the authorization server.

But what does all this have to do with network programmability and automation? The automation ecosystem involves a lot of *integrations*. For example, the Cisco Webex line of products depends heavily on OAuth to integrate with third-party apps. Let's say you developed an application called App X, and you need to integrate it with Webex Teams so that this app can do API calls on behalf of a user, so as to post to a space (that is, a chat room in Webex Teams) without the user providing her credentials to your app. This is a classic case for using OAuth. OAuth makes integration between applications seamless and much more secure than do other alternatives.

Note

For more on Webex integrations, see https://developer.webex.com/docs/integrations.

Keep in mind that these roles are just functions and that one single entity may perform the functions of one or more roles. A client may also be the resource owner, for instance, or the resource server may be the same as the authorization server. There are therefore many variations of the OAuth workflow and quite a number of nuts and bolts to the protocol. This section covers OAuth in enough detail to help you understand the majority of the service documentation for integrations provided by network vendors and omits the fine details typically required by software developers creating or maintaining commercial-grade software.

The latest version of the OAuth protocol, Version 2.0, is defined in RFC 6749; using Bearer Tokens with the OAuth protocol is defined in RFC 6750. Although this chapter does not explicitly cover native applications, it is worth mentioning that RFC 8252 discusses the best practice of only using external user agents, such as web browsers, for making OAuth requests from native apps. (Native apps are apps running natively on a system, such as Microsoft Office applications installed on a PC; in contrast, web apps are apps that run on a remote server with only a user interface on the client machine.) The high-level workflow of OAuth involves four steps:

1. Client Registration: The client (that is, the third-party application) registers with the authorization server.

2. Authorization Grant: The client requests authorization from the resource owner and receives an *authorization grant* in response to its request. The grant may come from the resource owner directly or from the authorization server.

3. Access Token: The client sends the authorization grant to the authorization server and receives an *access token*.

4. Resource Access: The client uses the access token in its API call to the resource server in order to gain access to the protected resource(s) residing on the server and owned by the resource owner.

The high-level workflow in Figure 8-2 is adapted from the workflow in RFC 6749.

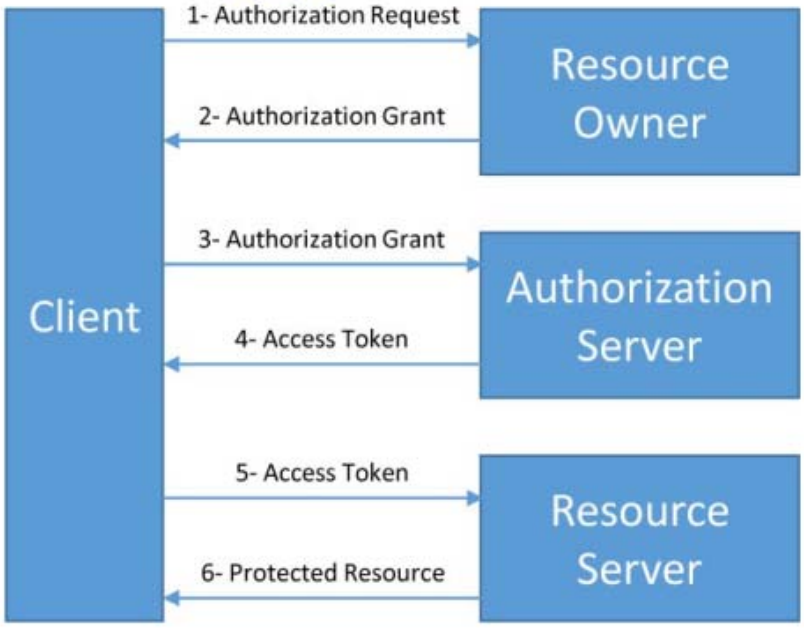


Figure 8-2 *High-Level OAuth Workflow*

The OAuth workflow described here comes in several flavors and involves a lot of fine details. A lot of abstraction is also embedded in the four steps just described. The following sections touch on each of the four steps and provide details related to network automation and programmability.

Client Registration

First, and as a prerequisite to any other OAuth-related activity, the client—whether it is a web, native, or mobile application—should register itself with the authorization server. The authorization server needs to know up front what applications are integrated with it—that is, what applications may leverage its services to request access to resources owned by other entities.

For instance, App X, which you have developed and would like to post on behalf of Webex Teams users, needs to be registered with Webex Teams (acting as the authorization server) through the registration page at https://developer.webex.com/my-apps/new/integration. The client is required to provide specific information as part of the registration process.

Each authorization server requests different registration information from the client, based on the nature of that particular integration. This is also the case with the information that the server sends back to the client. At a minimum, the client needs to specify the *redirection endpoint URI* (covered in detail in the next section). The client may also be required to provide a *token endpoint URI* (covered in detail in the "Access Token" section), an application logo, an email, a description of what the client does, and the scope. (The scope is the resources that the client needs access to, and whether this access is read-only or read/write.)

In return, the information received back from the authorization server will be, at a minimum, a *client identifier* that the server will later use to uniquely identify this particular integration and the *authorization endpoint URI* (covered in detail in the next section). The server may also send back to the client a password for authentication.

Figure 8-3 illustrates the client registration process with the authorization server.

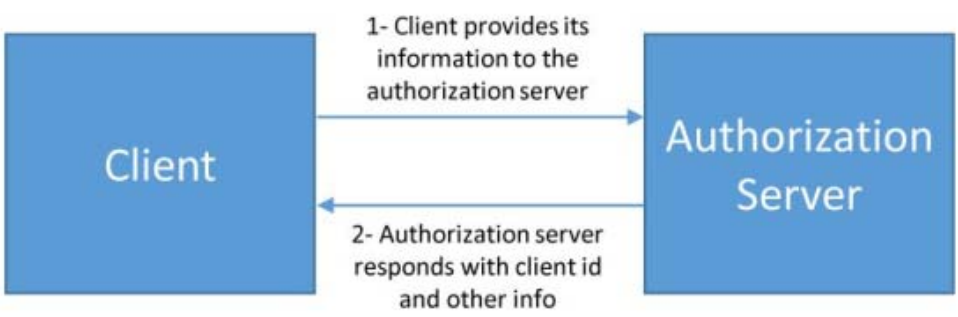


Figure 8-3 *Client Registration with the Authorization Server*

Authorization Grant

When the client is registered and integrated with the authorization server, the next step is for the client to seek authorization from the resource owner; this results, in a best-case scenario, in the resource owner issuing the authorization grant to the client. The client may request this authorization in a number of different ways, and the authorization grant that it receives back comes as one of several different grant types. This section starts with one specific use case and then branches out to describe some other variations to this particular step in the OAuth workflow.

Imagine a Webex Teams user (the resource owner) using App X (the client), mentioned earlier in this chapter, on her laptop. App X is a time-management tool. Since App X is a web application, the user accesses it through Google Chrome, which is referred to as the *user agent*, as you can see in step 1 in Figure 8-4. This figure illustrates the authorization grant workflow for this particular use case—that is, a web app attempting to integrate with Webex Teams (the authorization server).

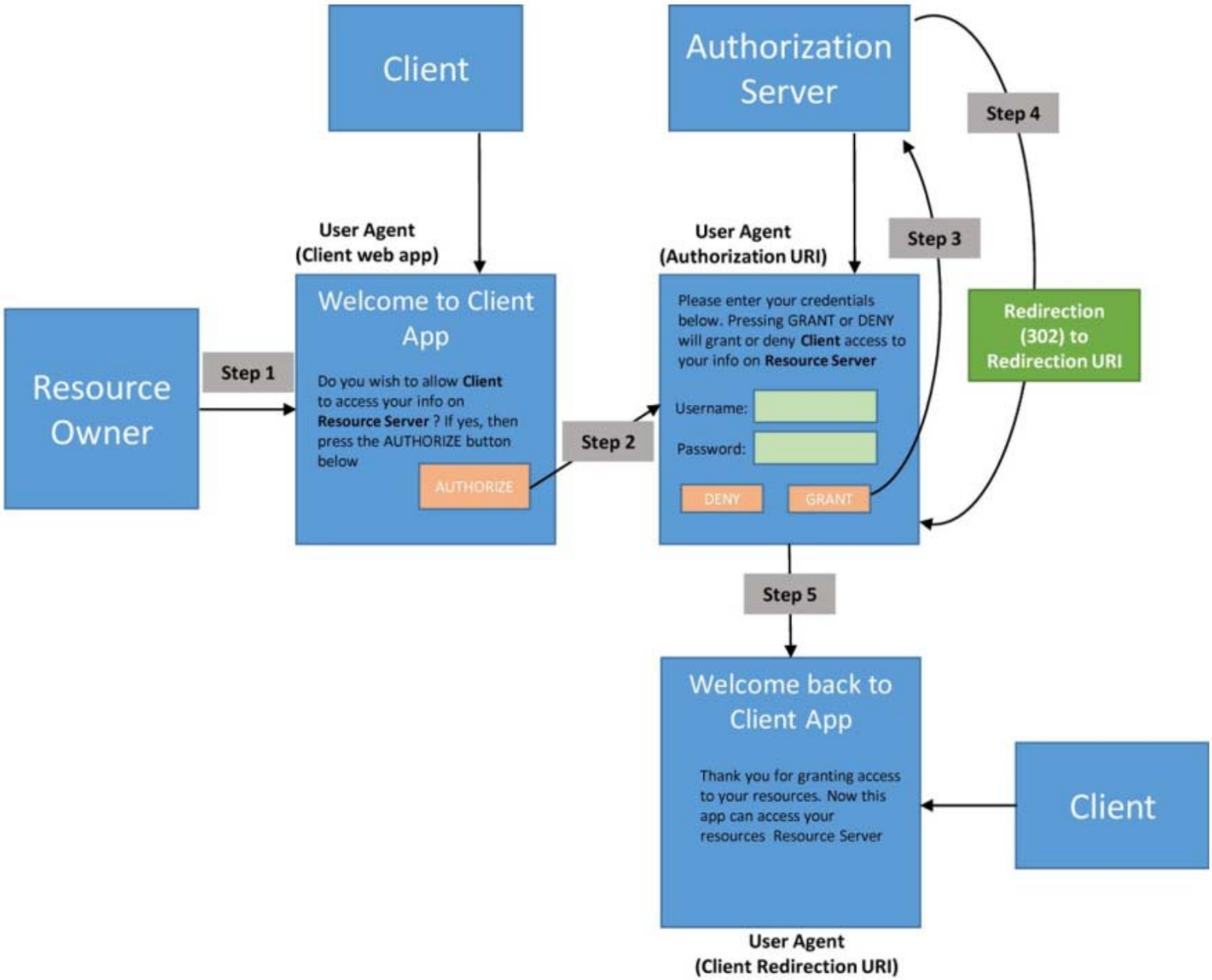


Figure 8-4 *Authorization Grant Workflowfor a Web Application*

While using App X, the user decides at some point to leverage an option in the app that lets the app send messages to a Webex Teams space that is shared with the user's team at work, on behalf of the user, when the user schedules a meeting on App X. App X will be doing that through one or more API calls to the Webex Teams server (the resource server). So the user decides to click the button on the app's web page labeled, for example, AUTHORIZE, as illustrated in step 2 in Figure 8-4.

The code attached to the button on the client web page will navigate the resource owner to the authorization endpoint URI that was provided by the server to the client during registration, and at that URI, the resource owner will provide her Webex Teams account credentials. In step 3 in Figure 8-4, the user enters her credentials and clicks the button labeled GRANT. The code attached to this button sends the credentials to the authorization server (Webex Teams) for validation.

Or, to use our earlier example, the fitness app on your mobile phone would request access to your Facebook account in order to be able to post your workouts to your timeline. The resource owner (you in this case) will click a button or link that will take you to the authorization endpoint provided by the Facebook server to the fitness app during registration, and you will enter your Facebook username and password.

Back to the Webex Teams use case: If the user credentials are correct and if the user clicks the button labeled GRANT, the authorization server redirects the user to the redirection endpoint URI that the client provided to the server during the registration process. This redirection happens in step 4 in Figure 8-4 via an HTTP response from the authorization server to the user agent. In step 5, the user agent navigates back to the client application to resume her work (the redirection URI). The user or the resource owner's involvement in the OAuth workflow ends at this point.

Now that the general flow for requesting the authorization grant is clear, let's dig a little deeper, particularly into what happens in steps 2 and 4.

In step 2 in Figure 8-4, the resource owner clicks the button labeled AUTHORIZE. The code attached to this button actually generates an HTTP GET request. The target URI of this request will be the authorization endpoint URI, which is composed of the host and path parts, in addition to a number of query parameters that the client application needs to pass on to the authorization server. In the case of the Webex Teams integration with App X, the target URI will look something like this:

```
https://webexapis.com/v1/authorize?client_id={client_id}&response_type=code[&redirect_

uri={redirection_endpoint_uri}][&scope={scope}][&state={state}]
```

The authorization URI includes five query parameters and their values:

• **response_type**: As mentioned earlier, there are several types of authorization grants. The four standard types defined by RFC 6749 (and discussed later in this section) are authorization code, implicit, resource owner password credentials, and client credentials. This query parameter specifies what type of authorization grant the client is requesting. Webex Teams integrations support only the authorization code grant type, and therefore the value of this parameter is code.

• **client_id**: This is the client identifier value assigned by the authorization server to this specific integration during registration.

• **redirect_uri**: This is the redirection endpoint URI that the client provides to the authorization server during registration. It may be (optionally) included as a query parameter value for verification purposes, and it must match the value of the URI provided during registration.

• **scope**: This optional query parameter value defines the scope of the grant. Simply put, the scope parameter defines what resources and what kind of access to those resources the client gets when the whole OAuth workflow is complete. For the Webex Teams integration example, the value of this parameter defines such things as whether the client will have access to the user's teams, rooms, or people and whether the client will be able to read, write, or manage these objects, among other things. Webex Teams provides 44 different scopes, and the client may choose 1 or more scopes during the registration. The value of this query parameter is determined by the scopes the client chooses during registration.

• **state**: This optional parameter is an arbitrary value chosen by the client. The same value will be used when redirecting the resource owner's user agent back to the client application after authentication (step 4 in Figure 8-4), and it helps the client identify each resource owner that is directed to the authorization server when that owner is redirected back to the client application after the authentication process (step 5).

The HTTP response to this GET request is typically the content of the https://webexapis.com/v1/authorize web page, possibly with customized content to match the client application, based on the client_id parameter value received in the GET request. This corresponds to the web page with the GRANT and DENY buttons in Figure 8-4.

Moving on to step 4, the HTTP response from the authorization server will use a "302 Found" response code. The response will have a Location header field with a value equal to the redirection endpoint URI plus a query parameter named code that will hold the authorization grant value in the form of an *authorization code*. The net effect of this is that the user agent, Google Chrome in this case, will navigate the user back to the client application (because the redirection URI points back to a client web page) so the user can continue using the client app and the client will have received the authorization grant as a query parameter. A typical redirection URI holding the authorization code grant may look like this:

```
https://clientapp.example.com/oauth?code={authorization_code_grant_value}&state={state}
```

So far, we have looked at only one type of authorization grant: the authorization code grant. Four types of authorization grants are defined by RFC 6749:

• **Authorization code**: This grant type is provided to the client typically in the form of an alphanumeric string. The distinctive characteristic of this type of grant, as you have seen from the description in this section, is that it is provided by the authorization server, not directly by the resource owner, to the client. This is the most common authorization grant type due to its inherent security benefits. The resource owner provides her credentials to the authorization server for authentication without exposing those credentials to the client (step 3 in Figure 8-4). The authorization code is also, typically, extracted by the client from the URI in the redirection HTTP request before the landing page is displayed to the resource owner (steps 4 and 5 in Figure 8-4) so that the authorization code is not exposed to the resource owner, and thus it is safe from any security breaches or exposures that the resource owner may be experiencing.

• **Implicit**: This is a more efficient but less secure grant type than the authorization code and is primarily targeted toward clients that have no way of authenticating themselves with the authorization server. The authorization server moves through all the steps in the previous workflow, and in step 4 in Figure 8-4, it sends an access token instead of a grant to the client. For this type, the response_type query parameter in the authorization endpoint URI has the value token. The use of this type is not recommended, and, as per the OAuth 2.0 Security Best Current Practice draft-ietf-oauth-security-topics (https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15), the use of the implicit type has been replaced with the authorization code type with extensions for enhanced security, such as the proof key for code exchange extension (RFC 7636).

• **Resource owner password credentials**: This type of grant is exactly what its name implies. The credentials that the resource owner provides to the authorization server in the first grant type will themselves be the authorization grant that the client uses to request an access token (discussed in the next section). So instead of the client redirecting the resource owner to the authorization server to get her credentials, the client requests those credentials from the owner directly through the client application interface. This grant type is used when there is a level of trust between the resource owner and the client, such as when the client application is a highly trusted native application running on the resource owner's machine. The client will then use these credentials to request an access token from the authorization server without further involvement from the owner.

• **Client credentials**: This type of grant may be used when the client application has a way to authenticate itself with the authorization server. For example, the client may receive a client secret along with the client identifier after registration. Or perhaps the client application is the same entity as the resource owner, and that entity opts for using OAuth for authentication for some application architectural requirement. With this authorization grant type, the resource owner is not involved in the workflow at all, and the client simply sends its credentials to the authorization server to get an access token.

Note

Each of these authorization grant types has a different workflow from the one described in this section. However, we discuss only the authorization code grant workflow in this chapter because it is the recommended and most commonly used authorization grant type. You can consult RFC 6749 for detailed descriptions of the other grant types.

Access Token

The client now has an authorization grant and will use it to get an access token. The client sends the grant it received (from the authorization server or directly from the client) to the *token endpoint URI* residing on the authorization server and receives an *access token* and, optionally, a *refresh token*.

The token endpoint URI is provided to the client during the registration phase or is available in the service documentation. For example, the token endpoint URI for the Webex Teams server is https://webexapis.com/v1/access_token, and this information is documented in the Webex Teams integration documentation at https://developer.webex.com/docs/integrations#getting-an-access-token.

The access token is what the client will eventually send to the resource server in order to authenticate its API calls on behalf of the resource owner (discussed in the next section). As a security measure, the authorization grant should have a lifetime, and it may only be used once to request an access token. The same applies to the access token itself: It will have a lifetime, after which it will expire and not be valid anymore, but it may be used to authenticate multiple API calls. When (or before) the access token expires, the client may use the refresh token to request a new access token from the authorization server by sending the refresh token to the token endpoint URI.

Figure 8-5 illustrates the workflow involved in using a valid access grant to receive the access and refresh tokens and then using the refresh token to get a fresh access token.

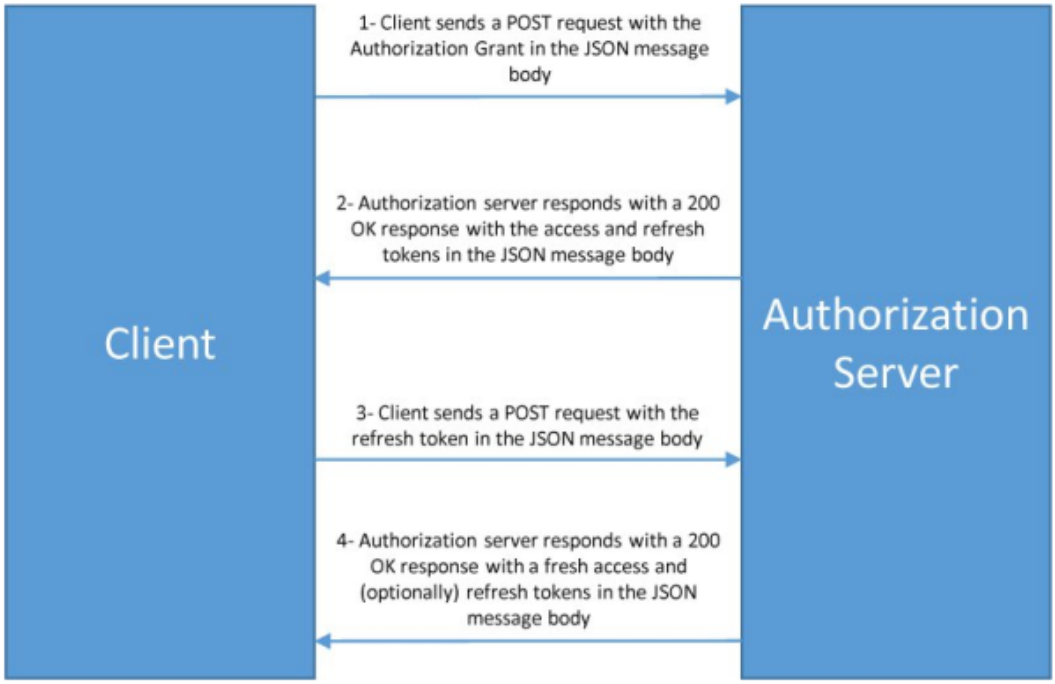


Figure 8-5 *Workflow to Issue Access and Refresh Tokens to the Client*

The request sent by the client to the authorization server to get an access token should be a POST request to the token endpoint URI with no query parameters and a JSON-encoded message body that contains the required request parameters. The parameters in the client request will be the grant_type parameter having a value of authorization_code (assuming the first grant type is used), the code parameter having a value equal to the authorization grant, and then the redirect_uri and client_id parameters.

If the client authenticates successfully with the authorization server and sends a valid authorization grant and a redirection URI that matches the one sent during the authorization grant request, the server responds to the client with a "200 OK" HTTP response message with a JSON-encoded message body.

The response message body will be composed of parameters with the values of the access token and, optionally, refresh token (access_token and refresh_token), the lifetime of the access token, in seconds (expires_in), the type of the access token (token_type), and, optionally, the scope (scope). The value of the scope parameter here is mandatory if the actual scope of the provided token is different from the scope requested by the client when requesting the authorization grant. If the scopes are equal, this parameter may be omitted. The token_type parameter defines the type of token used in this OAuth workflow. The most common type is the Bearer Token type, where the value of the parameter should be bearer.

Now if the server issued a refresh token to the client, when the access token expires the client will send an access token request to the token endpoint URI (a POST HTTP request) with no URI query parameters. The message body will include the parameter grant_type with the value refresh_token, the parameter refresh_token with a value equal to the value of the refresh token, and optionally, the scope parameter. The authorization server will respond with a "200 OK" HTTP response message that contains the details of the new access token and possibly a new refresh token. Again, the client needs to be authenticated with the authorization server in order to receive a new access token.

API Call to the Resource Server

Now that the client has a valid access token, this token is sent to the resource server in every HTTP request made by the client as a means to authorize this request. Figure 8-6 shows the workflow for accessing a resource on the resource server using the valid access token.

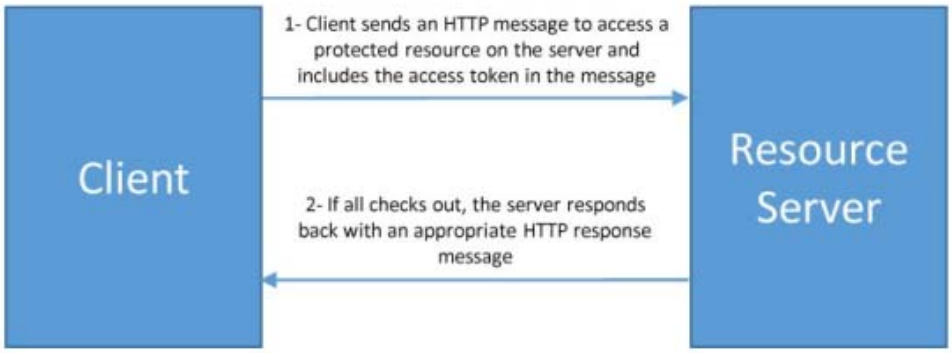


Figure 8-6 *Workflow for Accessing a Protected Resource on the Resource Server by the Client*

How the token will be included in the HTTP request message from the client depends on the token type. In the case of Bearer Tokens, the Authorization header field in the request message will use be used with the Bearer authorization scheme followed by a whitespace and then the token value, similar to the following:

```
Authorization: Bearer {access_token_value}
```

State Management Using Cookies

HTTP is a stateless protocol. This means that the server generates an HTTP response to each HTTP request it receives from a client, and this marks the end of the particular transaction. Of course, there are a few caveats to this statement. As you saw in Chapter 7, sometimes the server responds with codes such as the "1xxx Informational" or "3xx Redirection" codes that signal the client to take further action. However, even in those cases, that further action requires a new HTTP request to be generated and sent by the client.

RFC 6265, "HTTP State Management Mechanism," defines a method by which a server can store state information on a client machine. A server does this by including a data structure called a *cookie* as the value of the Set-Cookie header field in its response to the client. The client saves the information received locally, along with some other information, such as the time the cookie was received. The client then includes a Cookie header field in its subsequent HTTP requests to that server, which the server uses to identify that particular client, in addition to other information specific to that client, such as that client's browsing preferences or location information.

Cookies are not typically used for authentication on their own and work in tandem with other authentication schemes. A typical scenario would involve the client using the Basic, Digest, or any other authentication scheme mandated by the server. Upon successful authentication, the server sends one or more cookies to the client to save specific information related to that client. Further HTTP requests from the client will not need to be authenticated as the cookie information will be sufficient to identify the client to the server. This information will remain valid until the cookie expires, in which case the client re-authenticates to the server and receives a new cookie to be used in the following HTTP requests, and so forth.

Figure 8-7 illustrates a typical workflow involving state management using cookies.

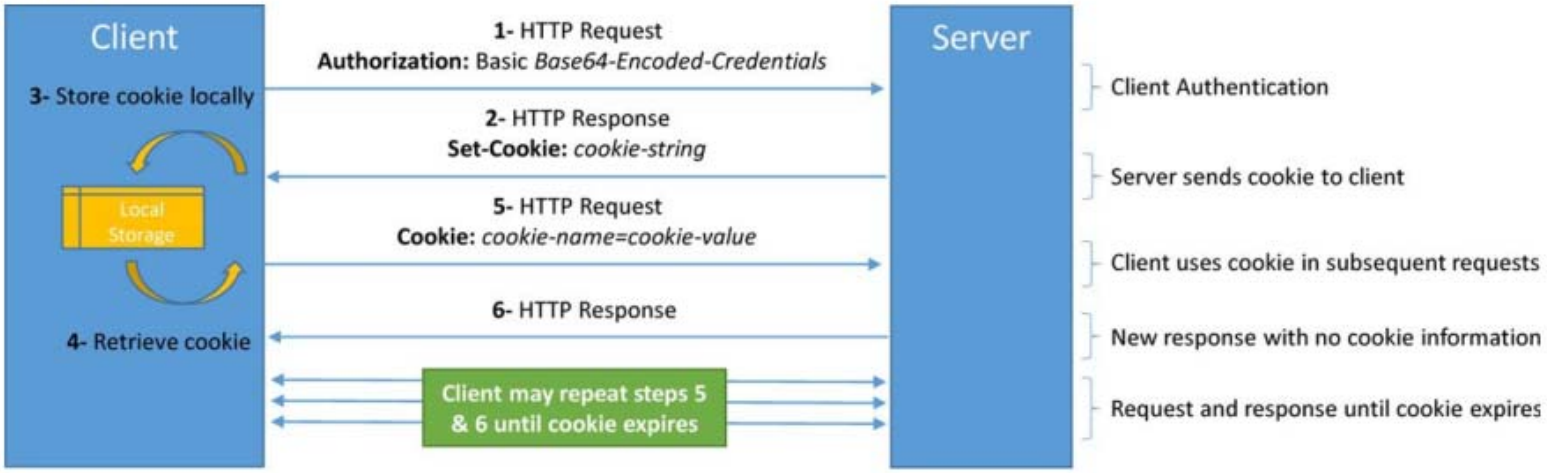


Figure 8-7 *State Management Using Cookies: A Typical Workflow*

The figure depicts six steps in the workflow. The first step is authentication. In the example, the client authenticates with the server using Basic Authentication, but it can be any authentication scheme, whether standard or proprietary. The workflow may also not involve authentication at all: The server may simply wish to tag that particular client with some information using cookies. A typical example not involving authentication is a customer visiting a public website, and that website wishing to save the customer's browsing history in order to personalize site settings or recommendations for that particular customer.

Assuming that the authentication, if any was required, was successful, the server sends back a response that includes the Set-Cookie header field, which has the following format:

```
Set-Cookie: {cookie-name}={cookie-value}[; Expires={expiry-date}][; Max-Age={seconds}][;
Domain={domain-value}][; Path={path-value}][; Secure][; HttpOnly]
```

cookie-name and *cookie-value* are arbitrary values.

The Expires attribute value is the expiration date of the cookie. The client may use this cookie up until the expiration date, but it may choose to discard it any time before that. The Max-Age attribute does the seemingly redundant task of indicating the cookie validity lifetime, but it does so in seconds until the cookie expires, not as a date. The Max-Age attribute has precedence over the Expires attribute if both of them are included in the Set-Cookie header. In the absence of both headers, the cookie expires at the end of the session in which the cookie has been received.

The Domain attribute defines which servers this cookie is good for. If the Domain attribute value sent back to the client is cisco.com, then this cookie should be valid for both servers: sandboxdnac.cisco.com and sbx-nxos-mgmt.cisco.com. A server should not send a Domain attribute with a value that does not encompass its own address. For example, neither server just mentioned can use the Domain value apache.org in its Set-Cookie header.

At the receiving end, the peer calculates its own value for the tag, using the received ciphertext and associated data. If the received tag value is equal to the calculated value, the ciphertext is decrypted back to the plaintext payload.

Digital Signatures and Peer Authentication

An endpoint proves its identity to the other peer over a connection via a signature, which is a cryptographic value calculated using (as you might have already guessed) a key. Digital signature algorithms use asymmetric keys, each composed of a private key and a public key. The endpoint that needs to prove its identity maintains a private, confidential key. The peer that will verify the identity of the endpoint needs to know the public key of the key pair, which is not a confidential value. Digital signatures work as follows:

1. The endpoint that needs to prove its identity uses a digital signature algorithm that takes the message and private key as inputs, and it outputs a digital signature. This endpoint attaches the signature to the message and sends it over the channel.

2. At the receiving end, the peer uses a signature verification algorithm that take the message, public key, and signature as inputs and then outputs the verification result—that is, whether or not the message, public key, and signature all match; in doing so, it effectively verifies whether the message originated from the endpoint that generated the signature and owns that public key.

Note

One point to note here is that given the message and public key, the signature cannot be re-created. The only way the signature can be generated is by using the private key.

In TLS 1.3, server authentication to the client is mandatory, and client authentication to the server is optional. While certain protocols are not considered secure enough to generate keys to be used for data encryption, these same protocols may be used to generate keys used for authentication. Authentication is performed using either a PSK or a certificate that uses a digital signature. TLS 1.3 supports the digital signature algorithms RSA, ECDSA, and EdDSA. These three protocols are used to generate public and private key pairs and define signature generation and verification algorithms.

A client connecting to a server over the Internet may receive a message with a correct signature that checks out using the public key of the server. But what proves to the client that it is actually speaking with the server that claims to be who it is? A malicious party can generate a private and public key pair and share the public key with the client; because the private key was generated by the same entity, the signature checks out. This malicious party, impersonating a shopping website, for instance, may then trick the client into providing credit card information or other confidential information. For this reason, public servers over the Internet typically use X.509 certificates to authenticate themselves to clients connecting to them over TLS. These certificates are issued and signed by certificate authorities (CAs). A certificate contains information such as the certificate serial number, certification expiration date, server and organization details, server public key, and CA name and signature. Clients, such as well-known web browsers, ship preconfigured with a list of well-known and trusted CAs.

When a client receives a server certificate while, for example, loading a web page over HTTPS, it makes sure the certificate was issued by an entity that it trusts, such as a well-known CA. If it was not issued by a trusted CA (as in the case of a self-issued certificate), it may prompt the user to either proceed to the website or terminate the connection. The client also checks that the expiration date of the certificate has not passed. The certificate contains the domain name of the certificate owner, so the client can verify that it is communicating with the same domain stated in the certificate. The client then uses the server's public key listed in the certificate to verify the signature attached to each message using the verification algorithm described at the beginning of this section.

Later in this chapter you will learn more about the phase of the TLS 1.3 workflow in which the server sends its certificate to the client.

Wrapping It Up

Note

Before we connect the dots and discuss TLS 1.3, make sure you understand the following concepts covered in this section:

• PSK

• Key generation algorithms (FFDHE/ECDHE)

• HKDF

• Key schedule and keying material

• Block and stream cipher algorithms (such as AES-128/256 and ChaCha20)

• Block cipher modes (CCM/GCM)

• Hash functions (such as SHA-256/384)

• Message authentication code (CBC-MAC, GHMAC, and Poly1305)

• AEAD protocols

• Authentication protocols and certificates

TLS 1.3 Protocol Operation

TLS 1.3 supports only Finite Field DHE (FFDHE) and Elliptic Curve DHE (ECDHE) key exchange algorithms. TLS 1.3 also supports the use of PSKs. A PSK may either be agreed on out of band or in band over an already established secure session, to be used for another future session. A session may use a PSK alone, DHE alone, or both together.

Traditionally, a number of different protocols grouped together—that is, a specific key exchange protocol, a specific block/stream encryption protocol, a MAC protocol, and a hash function—was referred to as a *cipher suite*. Because TLS 1.3 only supports AEAD algorithms, the TLS 1.3 RFC (RFC 8446) redefines a cipher suite as only the AEAD encryption protocol along with the name of the hash function used by the HKDF. A major enhancement in TLS 1.3 is the exclusion of the vast majority of cryptographic algorithms that were deemed insecure; TLS 1.3 retains support for only five cipher suites:

• TLS_AES_128_GCM_SHA256

• TLS_AES_256_GCM_SHA384

• TLS_AES_128_CCM_SHA256

• TLS_AES_128_CCM_8_SHA256

• TLS_CHACHA20_POLY1305_SHA256

The first four of these cipher suites uses the AES block encryption protocol with one of two different key sizes (128 or 256), and one of two different modes (CCM or GCM). Three of these cipher suites use SHA-256, and one uses SHA-384 as the hash function used by the HKDF. The fifth cipher suite uses the ChaCha20 stream encryption protocol with the poly1305 MAC algorithm and the SHA-256 hash function. (By now, you should know exactly what each of these protocols does and where it fits in the picture.)

In a nutshell, TLS 1.3 performs the following tasks toward establishing a secure channel for application data, such as HTTP:

1. The peers use a PSK and/or a key generation algorithm to generate and agree on a symmetric key.

2. The HKDF generates all the keying material required by TLS moving forward, as per the key schedule of the protocol.

3. The keying material is used by the cipher specified in the cipher suite to encrypt *some* of the handshake messages and *all* the application data messages to maintain data confidentiality.

4. A MAC is generated and attached to each message to maintain message integrity and authenticity. The MAC may be generated based on the plaintext or the ciphertext, and it may use a cipher or hash function, depending on the chosen cipher suite.

5. Finally, the server is authenticated to the client, and optionally, the client is authenticated to the server, either using a PSK or certificates.

TLS comprises three sub-protocols, each managing a different phase of the protocol workflow:

• **Handshake protocol**: This protocol is responsible for negotiating the TLS version, negotiating the key exchange protocol and/or PSK, negotiating the choice of cipher suite, exchanging the public key values and establishing the shared keying material, authenticating the server to the client, and, optionally, authenticating the client to the server.

• **Record protocol**: This protocol divides the data to be transmitted from one endpoint into blocks called *records* and reconstructs the records into data at the receiving peer. The record protocol leverages the parameters established by the handshake protocol to secure the transmission of some of these records over the channel. The handshake protocol requires the record protocol to manage the transmission and receipt of the handshake messages, whether encrypted or not.

• **Alert protocol**: This protocol manages the protocol alert messages that signal connection closures or errors.

The TLS workflow has many variations, based on many variables. The TLS workflow shown in Figure 8-10 is the typical workflow when everything goes well between the client and server. The messages with a trailing asterisk (*) are optional messages that are sent in some cases, and the rest of the messages are mandatory, as discussed in the following sections.

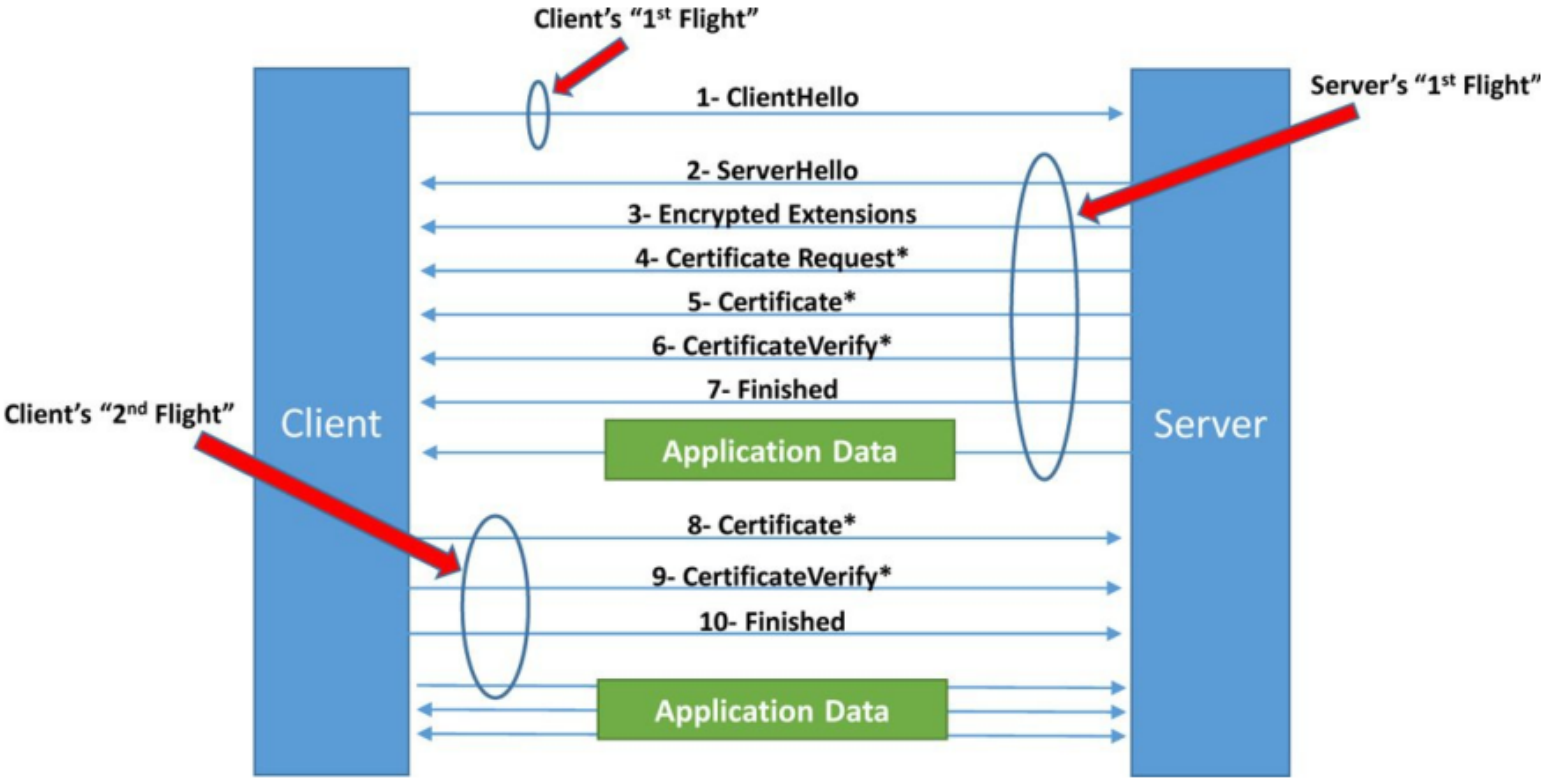


Figure 8-10 *Simple 1-RTT TLS Workflow*

The TLS Version 1.3 Handshake

In all versions of the workflow, the client initiates the TLS connection to the server by sending a ClientHello message. This message includes a few fields and several extensions. The fields are used to enable a TLS 1.3 endpoint to speak with a peer that only supports an earlier version of TLS, until the protocol version is negotiated, and the majority of information related to TLS 1.3 will be communicated via extensions.

The ClientHello is basically the client's proposed TLS version and the cryptographic protocols, modes, and parameters, such as the supported DHE groups, the public key values of some or all of these groups (such as the value of the $A=g^a \bmod p$ for FFDHE groups, discussed in the "Key Generation and Exchange" section), and the supported cipher suites. It also contains an extension that includes the list of supported signature algorithms for authentication. If a PSK (or more than one) is already known to the client and server (from an out-of-band channel or an earlier session), the PSK label is also included. The message also includes a random nonce called ClientHello.random that is unique to the particular session.

The server responds with a ServerHello message that establishes which of the client-proposed protocols and modes the server agrees to use and its own parameters for these protocols and modes, such as the value of its own public key share used by DHE (for example, the value of $B=g^b \bmod p$ if an FFDHE group is chosen), and which cipher suite to use. The ServerHello also uses fields and extensions in a similar fashion

to the ClientHello. Up to this point, the channel is still insecure, and the parameters exchanged over it are publicly available to any third party that is listening in on the channel. But this is not a problem; this is actually how key exchange protocols such as DHE work, as you have seen earlier in this chapter.

After the server sends out the ServerHello, both the client and server have sufficient information to start generating the keying material required to start encrypting any further data exchanged over the channel. Following the ServerHello, the server sends out a second message, called the EncryptedExtensions message. As the name implies, and because this is feasible at this point in the handshake, this message is encrypted using a key called server_handshake_traffic_secret. The information in this message includes all the handshake information that is *not* necessary to establish the cryptographic parameters. In other words, the ServerHello message communicates just enough information for the server to be able to start encrypting data. Any other information required to complete the handshake is then sent encrypted. All subsequent messages are also encrypted. This is a major enhancement in TLS 1.3 over earlier versions, where the full handshake was performed without any encryption.

If the server requires a certificate from the client, it sends out a third message, the CertificateRequest message. If the server authenticates itself to the client using a certificate, it sends out a fourth message, the Certificate message, which contains the server's public key, and a fifth message, the CertificateVerify message, which contains a signature over the entire handshake using the server's private key. If the server is not authenticating using a certificate, both the Certificate and CertificateVerify messages are omitted.

Finally, the server always sends out a sixth message, the Finished message, to signal the end of the handshake protocol phase from its side. The Finished message contains a MAC that is used to verify the authenticity and integrity of the complete handshake.

Although some messages are optional and others mandatory, the messages must be sent in the order described, and this is the reason for the intentional numbering of messages (first, second, and so on). A message received out of order triggers the alert protocol at the receiving peer to issue an unexpected_message alert and abort the handshake.

The Finished message concludes a successful handshake phase from the perspective of the endpoint sending it, with the next record of data being the application data. Notice that the very first application data is sent on the server's first flight, which is practically the first round trip since the client initiated the session. For this reason, this is called the 1-RTT handshake. Besides enhanced security, one of the drivers of TLS 1.3 was better performance. The 1-RTT mode and the 0-RTT mode (which is discussed shortly) are great improvements over TLS 1.2, with which the first application data is sent after two complete round trips of handshake messages.

Application data is encrypted using a key called server_application_traffic_secret that is different from the key used to encrypt the handshake messages. Recall that TLS generates several keys called the keying material, with each key used at a different phase of the workflow. These keys are generated via the HKDF according to the protocol key schedule.

At this point in the workflow, the server is sending encrypted application data to the client. However, the client has not been authenticated yet. Recall that client authentication is optional in TLS 1.3. If the server requires that the client have a certificate and sends a CertificateRequest message, the client will send back the two messages: Certificate and CertificateVerify. Finally, the client will send its own Finished message, followed by the first transmitted application data. These three messages sent by the client contain, more or less, content similar to that of the corresponding messages sent by the server.

Keep in mind that the first three messages are encrypted using the client_handshake_traffic_secret key, and the application data is encrypted using the client_application_traffic_secret key.

In its ClientHello, the client includes two significant extensions: the supported_groups and key_share extensions. The supported_groups extension contains an ordered list of DHE named groups supported by the client, from most preferred to least preferred. This list includes all supported groups, whether FFDHE or ECDHE. The key_share extension, on the other hand, includes the parameters for some or all of the groups listed in the former extension. This extension includes a list of records, each record specifying the group name and the parameters for the group.

For example, the supported_groups extension may comprise a list, with one entry specifying the group ffdhe2048. There should be a corresponding record in the key_share extension with the name of the group and the value of the variable A calculated as $A=g^a \bmod p$, where the values of p and g are well-known values for that particular group, and a is the secret key that the client does not share with any other entity, including the server. The client typically includes a record in the key_share extension for every group listed in the supported_groups extension.

But what if the server agrees to use one of the groups in the supported_groups extension but does not find a record in the key_share extension? In this case, the server responds to the ClientHello with a HelloRetryRequest message. This message is sent back to the client when the server cannot find sufficient information in the ClientHello to complete the handshake. It has the same format as the ServerHello message and includes, among other extensions, the server's key_share extension for the server's chosen group. The server can only choose a group initially proposed in the first ClientHello—and not any other.

Once the client receives the new key share, it should respond with a new ClientHello that includes the client's updated (and complete) key share for the group chosen by the server. This second hello is a part of the same handshake; the session is not reset. Now if the server agrees with the client on the parameters shared in the second hello message, the workflow continues normally, as in Figure 8-10. If not, the server aborts the handshake and sends an appropriate alert message to the client.

One HelloRetryRequest message is allowed per session. If a client receives a second HelloRetryRequest in the same session, it must abort the handshake with an unexpected_message alert.

0-RTT and Early Data

As mentioned earlier in this chapter, a PSK may be used to bootstrap a TLS connection without having to perform a full handshake. A PSK may be established out of band, but it may also be generated by the server and communicated over an already secure channel to the client to be used in another future session. In the latter case, the server sends the PSK details by using a NewSessionTicket message that is encrypted using the server_application_traffic_secret key. This message is sent right after receiving the client's Finished message.

In the new session, the client communicates the PSK details in its ClientHello by using an extension called pre_shared_key. The same extension is used by the server in its ServerHello back to the client. The client may, optionally, also include a key_share extension to provide the option for the server to fall back to a full handshake, if required. In addition to the PSK, the server may still respond with a key_share in its ServerHello and use both the PSK and DHE keys to guarantee perfect forward secrecy. When using a PSK, the client and server do not need a certificate for authentication and use the PSK for that purpose. Therefore, the Certificate and CertificateVerify messages are not used. Using a PSK to bootstrap a TLS session is called *session resumption*.

When using a PSK for session resumption, the client has the option to send application data on its first flight, right after the ClientHello, without waiting for any messages from the server first. This is made possible because this *early data* is encrypted using a key called the client_early_traffic_secret that is generated directly from the PSK. The endpoints also use the PSK for authentication. It should be noted that since the early data encryption key is generated from the PSK, the early data lacks perfect forward secrecy. This does not apply to the rest of the application data exchanged after the server Finished message. This mode of operation is called 0-RTT.

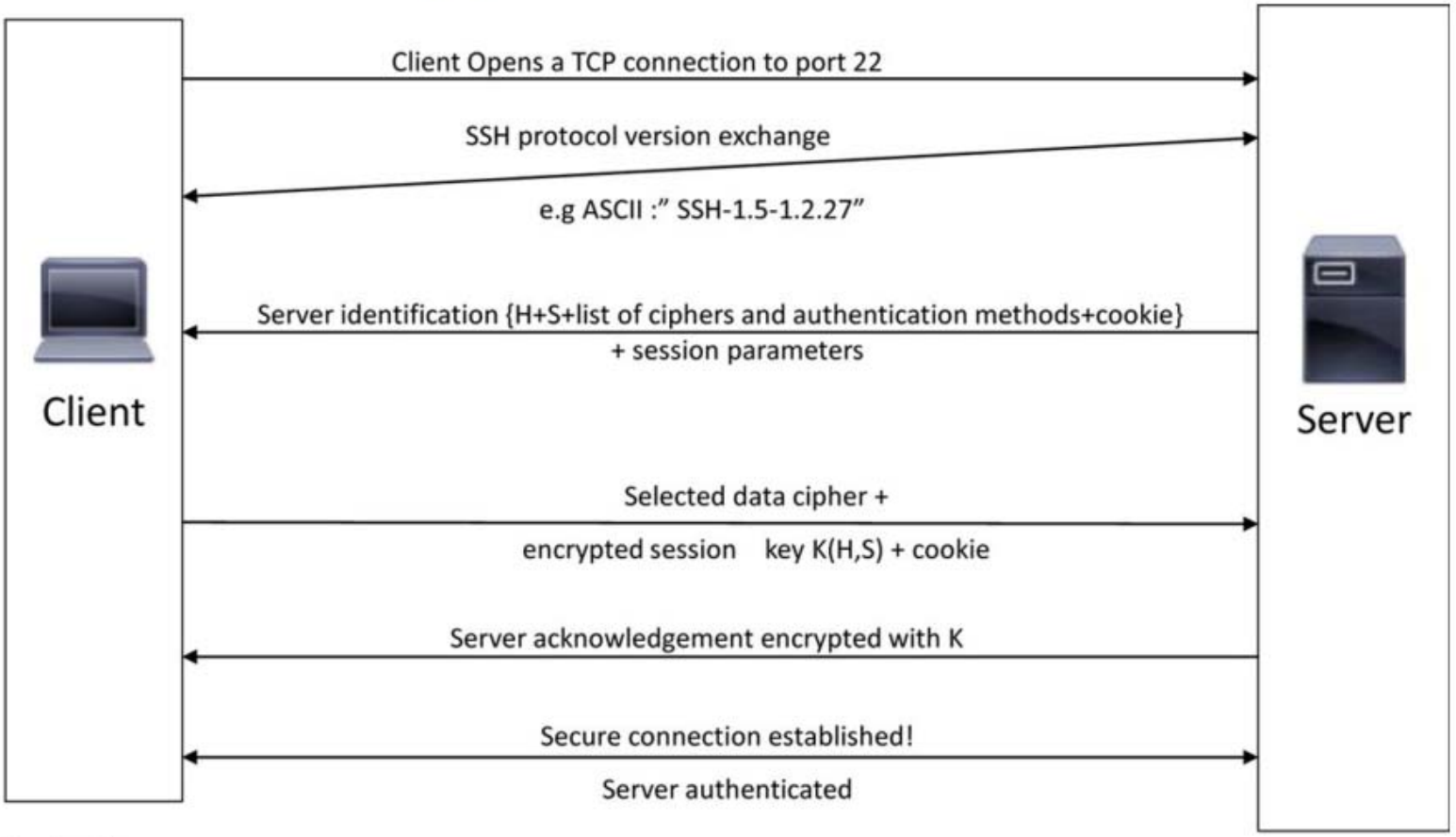
The Record Protocol

The record protocol is responsible for dividing the data to be transmitted from one endpoint into blocks called *records*, optionally protecting

Note

It is important to note that the version number should match on both sides as SSH1 is not compatible with SSH2, especially when the server is running SSH1. In other words, An SSH server running SSH2 can accept connections from SSH1 clients, but a server running SSH1 cannot accept SSH2 requests.

After the authentication method between the client and the server has been negotiated, a secure connection is established. With SSH1, integrity checking is provided by means of a weak CRC-32 that is collision prone. This means that, given a CRC, somebody can provide another input that matches the same CRC value. Figure 9-2 illustrates the SSH1 protocol packet exchange between a client and a server.



H – host key ;

S- Server Key ;

cookie- sequence of 8 random bytes;

K- session key

Figure 9-2 *SSH1 Protocol Packet Exchange*

With SSH1, if the client and the server negotiate for compression of the payload, the payload is compressed using the Deflate algorithm, which is used by the GNU **gzip** utility. Compression is primarily used by file transfer utilities such as Secure Copy Protocol (SCP).

SSH2

The SSH2 protocol is documented in RFCs 4250 through 4254. Unlike SSH1, SSH2 is designed in a modular way, and the protocol functionality is implemented into three separate protocol modules:

• SSH Transport Layer Protocol (SSH-TRANS)

• SSH Authentication Protocol (SSH-USERAUTH)

• SSH Connection Protocol (SSH-CONNECT)

All these protocols run on top of TCP.

Figure 9-3 illustrates the SSH2 protocol architecture. At the bottom of the protocol stack is TCP, which provides and ensures reliable connectivity across multiple networks. Next, the SSH protocol provides server authentication, confidentiality, and integrity of data. The SSH Authentication Protocol provides host-based authentication and user authentication using public keys, passwords, and so on. The SSH Connection Protocol enables session multiplexing and helps with features such as X11 and port forwarding and remote command execution.