

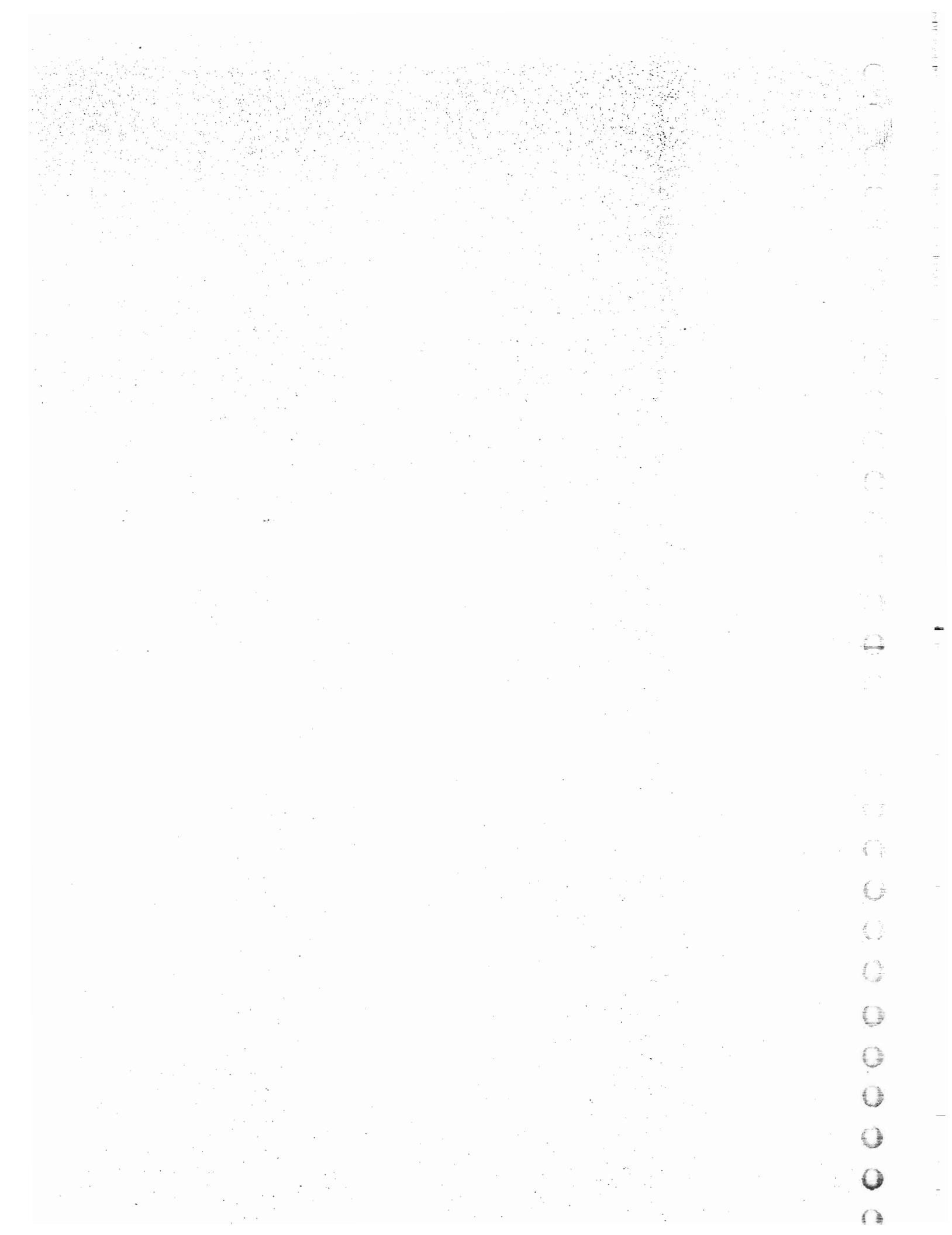
DEVOPS

Material



Flat No: 212, 2nd Floor, Annapurna Block, Aditya Enclave, Ameerpet, Hyderabad, AP.
Ph No: 040 6462 6789, 0998 570 6789 info@kellytechno.com, www.kellytechno.com.

Bangalore: HAL Road, Marathahalli. 080 6012 6789, 0784 800 6789
Online: 001 973 780 6789.





LINUX

BASIC COMMANDS

ABSTRACT

UNIX is a computer Operating System which is capable of handling activities from multiple users at the same time.

Isha Ankush Malode
LINUX

History of linux

What is an Operating system

Operating system is the collection of programs that coordinates the operation of computer hardware & software

functions of operating system:

- 1)process management
- 2)memory management
- 3)Data management
- 4)I/O management

Architecture of linux

Kernel:

kernel is a set of functions that makeup heart of an o/s it is used to provide application interface between programs & physical devices

services provided by kernel :

- controls execution of process
- scheduling process fairly for execution on cpu
- Allocating memory for an executing process

shell:

shell is an interface between human readable language & machine language

Multics project

Multics was started by mainframe GE 645 by the joint effort of AT&T bell labs general electrical Massachusetts Institute of Technology Multics was designed in Assembly Language in 1969 Multics project was dropped in 1969, AT & T redesigned multics and introduced new os that is unics(Uniplex information & Computing system) it is written in 80% of "c" Language and 20% with assembly language by kenthomson & dennis ritchie .Later on totally rewritten in "c" language and renamed as unix(1973)

flavours of Unix

vendor	o/s
AT&T,Bell labs	SYS III -sys V
SUN	Sun os - Solaris
SCO	Sco unix
IBM	Aix
SG	IRIX
HP	HP-Aux
BSD	free BSD linux

In 1988, AT&T shocked the unix community by purchasing a percentage of Sun microsystems which became a threat for other vendors

Quickly the other vendors form a group and named it as OSF(Open Software Foundation)

and former formed their group and named it as UI (Unix International)

In 1990,Linus Torvalds a graduate student from Helsinki university designed unix like kernel 386 intel machine and gave it to OSF

Linux is bundled with many softwares from various distributors and it gave rise to many flavours of Linux

No of companies are providing tech support for linux o/s

RedHat

SUSE

MANDREKE

NOVEL

PUPPY LINUX

TURBOLINUX

SLACKWARE

UBUNTU

KNOPPIX

name of kernel in RhEl5 is vmlinuz 2.6.18-8.el5

Kernel image is initrd

installers name is anaconda-ks.cfg

default shell is /bin/bash

MBR - Master Boot Record

MBR's job is to locate compressed kernel & arrange the architecture logically

default MBR in linux is GRUB : Grand Unified Boot Loader in windows is ntldr

features of linux

1)Open Source(along with source code)

2)Multiuser& Multitasking

3)Enhanced Security (Inbuilt firewalls)

4)Reliability

5)GUI

Biggest servers on this earth are running on linux without restarting from last twelve years

Accessing the command line

Date -Mon Oct 10 12:22:13 IST 2016

Date +%R - 12:23

Date +%x - Monday 10 October 2016

Head - This command is used to read first few lines of any given text.

Tail – This command is used to read last few lines of any given text.

By default it will show 10 lines.

If we want to see particular number line :- tail -n 3 /etc/passwd

Tail -f :- It will show current online information.

Wc :- This command is used to count lines, words, and characters.

Wc -l :- for lines

Wc -w :- for words

Wc -c :- for character

For Ex:- wc -l /etc/passwd.

History :- This command will show a list of previously executed command with command number.

MANAGING FILES FROM COMMAND LINE

/	The root directory, the top-level directory in the FHS. All other directories are subdirectories of root, which is always mounted on some partition. All directories that are not mounted on a separate partition are included in the root directory's partition.
/bin	Essential command line utilities. Should not be mounted separately; otherwise, it could be difficult to get to these utilities when using a rescue disk.
/boot	Includes Linux startup files, including the Linux kernel. Can be small; 16MB is usually adequate for a typical modular kernel. If you use multiple kernels, such as for testing a kernel upgrade, increase the size of this partition accordingly.
/etc	Most basic configuration files.
/dev	Hardware and software device drivers for everything from floppy drives to terminals. Do not mount this directory on a separate partition.
/home	Home directories for almost every user.
/lib	Program libraries for the kernel and various command line utilities. Do not mount this directory on a separate partition.
/mnt	The mount point for removable media, including floppy drives, CD-ROMs, and Zip disks.
/opt	Applications such as WordPerfect or StarOffice.
/proc	Currently running kernel-related processes, including device assignments such as IRQ ports, I/O addresses, and DMA channels.
/root	The home directory of the root user.
/sbin	System administration commands. Don't mount this directory separately.
/tmp	Temporary files. By default, Red Hat Linux deletes all files in this directory periodically.
/usr	Small programs accessible to all users. Includes many system administration commands and utilities.
/var	Variable data, including log files and printer spools.

Pwd :- It display full path name of current location.

Ls :- lists directory contents for specified directory.

Ls -IR :- information about all subdirectorys.

Ls -l :- long listing format.

Ls -a :- All files includes hidden files.

Ls -R :- Recursive to include contents of all subdirectory.

Ls -al :- List current location with file.

Ls a* :- Begin with a.

Ls *a* :- Containing a.

Ls ???? :- Filename at least 4 character in length.

Q :- How many users are there to see ?

Ans :- ls -l /home

Cd – To change directory

Mkdir :- To create directories.

For ex. :- mkdir <dir name>

Cp :- It is use to copy one or more files to become new

For ex.:- cp source/destination/

Mv :- It rename files in same directory, or relocate files to new directory

For ex. :- mv newfile1 newfile2

Rm :- Remove files but not directories

Rm -r :- Delete directory and many subdirectory and files below it

Rmdir :- remove/delete empty directory

USERS AND GROUPS

Id :- This command is used to show information about current logged in user.

Ps :- To view process information.

Ps au :- 'a' option view all process with terminal, 'u' option first column shows username.

Userdel :- remove user from /etc/passwd.

Userdel -r :- It remove user and users home directory.

Useradd :- Adding user.

Usermod :- Modifies existing user.

To lock account :- usermod -L <username>

To unlock account :- usermod -U <username>

Chage -d 0 :- username will force a password update on next login.

Chage -l :- username will list usernames current settings.

Chage -E :- It will expire an account on specific day.

Q :- Determine date 90 days in future and set each of user account to expire on that day ?

Ans :- date -d "+90 days"

Chage -E 2016-06-16 username.

Q :- Change password policy for <user1> account to require new password every 15 days ?

Ans :- chage -M 15 <username>

Chage -l <username>

Q :- How many files are there in linux/unix ?

Ans :- Regular file – Readable, binary, image or compressed files.

Directory file.

Special file (5 subtypes in it)

- Block files (b).
- Character device file (c).
- Named pipe file or just pipe file (p).
- Symbolic link file (l).
- Socket file (s).

Block File (b) :- These files are hardware files, most of them are present in /dev.

How to create :- use fdisk command.

How can we list :- ls -l | grep ^b.

Character device file :- Provides a serial stream of i/o. (crw)

Pipe files in linux :- The other name of pipe is a "named" pipe, which is sometime called a FIFO refers to property that the order of bytes going in is the same coming out (prw)

Symbolic link files :- These are linked files to another files. They are either directory/regular file. The inode number for this file and its parent files are same.

Two types soft and hard link. (lrw)

Socket files in linux :- It is used for passing information between application for common purpose (srw).

Q :- How to find out desired type of file ?

Ans :- Use find command with -type option

Ex. For socket - Find / -type s

Linked - find / -type l

Explain PS command in output

To view process information

%cpu - How much of cpu the process is using.

%MEM – How much of memory the process is using.

ADDR – memory address of process.

C or cp – cpu usage and scheduling information.

COMMAND – name_of process, including arguments, if any.

N1 – nice value.

F – Flags.

PID – process id number.

PPID – ID number of process's parent process.

PRI – Priority of process.

RSS – Real memory usage.

S or STAT – Process status code

START or STIME – Time when process started.

Sz – Virtual memory usage.

TIME – Total cpu usage.

TT or TTY – Terminal associated with process.

UID or user – username of process owner.

WCHAN – memory address of event process is waiting for.

Explain passwd file in linux

" /etc/passwd" file contain account information and look like this.

Smithj : x : 561:561:joe smith : /home /smithj :/bin/bash

- Username upto 8 character.
- An 'x' in password field. Password are stored in "/etc/shadow" file
- Numeric user id.
- Numeric group id.(usually group id will match user id)
- Full name of user (under 30 character)
- User's home directory (ex. /home/smithj)
- User's shell account "/bin/bash".

The "/etc/shadow" file contain password.

Smithj : Ep6mckr0lchf :10063 : 0 : 999999 : 7 :::

Username (8 character)

Password (13 character) a blank entry (eg ::) indicate is not required to log in (usually a bad idea) and a "x" entry (eg :x:) indicate account has been disabled.

The number of days since password was last change.

The number of days before password may be changed (0 indicate it may be changed at any time).

The number of days after which password must be changed. (999999 indicate user can change their password unchanged for many year).

The number of days to warn user of an expiring password.

Q :- How many shells are there in linux ?

Ans :- sh, bash, csh and tcsh, ksh

\$ - Bourne, korn and Bash shells.

% - C shell

CREATING, VIEWING, AND EDITING TEXT FILES

in vi editor we have 3 modes :

- 1)command mode
- 2)insert mode
- 3)execute mode

in command mode we can copy,paste,delete,undo,redo,move,save&quit

in insert mode we can only edit data

in execute mode we can set line numbers,delete line numbers,save&quit,quit with out saving,substitute data

when ever we type vi <filename>

if the file is there it will edit the existing file

if the file is not there it will create a new file

by using vi <filename> it will enter into command mode

to go to insert mode from command mode

i: insert text at current cursor position

I: insert text at the beginning of cursor line

a: append text after cursor position

A: append text at the end of cursor line

o: create a new line below cursor line and append text

O: create a new line above cursor line and append text

s: it removes cursor presented character and append data

to move from insert mode to command mode press "esc key"

command mode :

dd: to delete a line

5dd : to delete five lines

yy: to copy a line

5yy: to copy five lines

p: paste

10p : paste 10 times

u:undo

ctrl+r :redo

shift+g or G : to move last line of a file

gg :to move first line of a file

10g :to move 10th line of a file

shift+zz : save&quit

to move into execute mode from command mode press shift+:

execute mode

q: quit without saving

q!: quit without saving forcefully

wq: save&quit

x: save&quit

wq!: save&quit forcefully

:/<word> : to search for a particular word

ex: /sun : searching for a word sun in the file

:15 :to jump into 15th line of a file

set nu: to set line numbers for a file

set nonu: to remove line numbers of a file

substitution

syntax: <beginline>,<endingline> s /<oldchar>/<newchar>/g

ex: if i want to change LABEL as SMS in /etc/fstab

1,\$ s/LABEL/SMS/g

here 1,\$ indicates from first line to lastline

s for substitute

LABEL is old character

SMS is new character

g for grouping

note: to set automatic line numbers

create a hidden file as follows in /root

vi .vimrc

(type) set nu

save& quit

to create hidden files in linux use . at the begining of filename

ex: cat > .sun

now .sun is a hidden file

clear : used to clear the screen or use ctrl + l

to get more help for vi editor

type vimtutor

CONTROLLING ACCESS TO FILES WITH LINUX SYSTEM

PERMISSION

U, g, o - (user, group, other)

+, -, = - (add, remove, set exactly)

R, w, x - (read, write, execute)

R=4, w=2, x=1

- **Chmod** - for modification

Removing read and write permission for group and other file.

For ex. Chmod go - rw file 1

Add execute permission for everyone of file 2

Ans:- chmod a+x file 2

To set read, write, execute permission

Ans:- chmod all 750

- **Chown** – changing file / directory user or group

For ex. Change ownership of foodir to visitor and group to guest

Ans :- chown visitor :guests foodir

- **Chgrp** - same as chown
- **Umask** – to set permission by default on new file.

MONITORING AND MANAGING LINUX PROCESS

Running jobs in background. Any command can be started in background by appending (&) to command line.

Ps j – It will display job information including initial command

Jcpu – resource consumed by current jobs

Pcpu – current foreground process cpu consumption

To kill process forcefully :– kill -9 PID

To kill process safely :– kill -15 PID

Pkill command like killall can signal multiple processes.

- Top program is a dynamic view of system's process

GREP COMMAND

- grep, which stands for "global regular expression print," processes text line by line and prints any lines which match a specified pattern.
- To connect two commands together, so that the output from one program becomes the input of next program
- The grep program searches a file that have a certain pattern
- The grep searches the named input files
- The basic usage of grep command is to search for a specific string in the specified file as shown below.
- Syntax:
- grep "literal_string" filename

Ex. # grep "this" demo_file

o/p. :- this line is the 1st lower case line in this file

Two lines above this line is empty.

And this is the last line.

- Checking for the given string in multiple files.

Syntax:

```
grep "string" FILE_PATTERN
```

This is also a basic usage of grep command. For this example, let us copy the demo_file to demo_file1. The grep output will also include the file name in front of the line that matched the specific pattern as shown below. When the Linux shell sees the meta character, it does the expansion and gives all the files as input to grep.

Ex. :- # cp demo_file demo_file1

```
# grep "this" demo_*
```

o/p :- demo_file:this line is the 1st lower case line in this file
demo_file:Twoo lines above this line is empty.
demo_file:And this is the last line.
demo_file1:this line is the 1st lower case line in this file
demo_file1:Twoo lines above this line is empty.
demo_file1:And this is the last line.

Q:- How do u display single user from shadow file

Ans.:- # grep <username> /etc/shadow

- Case insensitive search using grep -i

Syntax:

```
grep -i "string" FILE
```

This is also a basic usage of the grep. This searches for the given string/pattern case insensitively. So it matches all the words such as "the", "THE" and "The" case insensitively

Ex.:- # grep -i "the" demo_file

o/p :- THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE

this line is the 1st lower case line in this file

This Line Has All Its First Character Of The Word With Upper Case

And this is the last line.

- Match regular expression in files

Syntax:

`grep "REGEX" filename`

This is a very powerful feature, if you can use use regular expression effectively. In the following example, it searches for all the pattern that starts with "lines" and ends with "empty" with anything in-between. i.e To search "lines[anything in-between]empty" in the demo_file.

Ex. # `grep "lines.*empty" demo_file`

O/p :- Twoo lines above this line is empty.

- Checking for full words, not for sub-strings using grep -w

If you want to search for a word, and to avoid it to match the substrings use -w option. Just doing out a normal search will show out all the lines.

The following example is the regular grep where it is searching for "is". When you search for "is", without any option it will show out "is", "his", "this" and everything which has the substring "is".

Ex.: - `grep -i "is" demo_file`

O/p :- THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE

this line is the 1st lower case line in this file

This Line Has All Its First Character Of The Word With Upper Case

Twoo lines above this line is empty.

And this is the last line.

- The following example is the WORD grep where it is searching only for the word "is". Please note that this output does not contain the line "This Line Has All Its First Character Of The Word With Upper Case", even though "is" is there in the "This", as the following is looking only for the word "is" and not for "this".

```
# grep -iw "is" demo_file
```

O/P :- THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE

this line is the 1st lower case line in this file

Two lines above this line is empty.

And this is the last line.

```
#grep '^user1' <filename>
```

o/p :- user1:x:1017:1017::/home/user1:/bin/bash

```
# grep 'nologin$' /etc/passwd
```

o/p :- bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

adm:x:3:4:adm:/var/adm:/sbin/nologin

lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin

mail:x:8:12:mail:/var/spool/mail:/sbin/nologin

operator:x:11:0:operator:/root:/sbin/nologin

games:x:12:100:games:/usr/games:/sbin/nologin

ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin

AWK COMMAND

Awk Introduction and Printing Operations

Awk is a programming language which allows easy manipulation of structured data and the generation of formatted reports. Awk stands for the names of its authors "Aho, Weinberger, and Kernighan"

The Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform associated actions.

Some of the key features of Awk are:

Awk views a text file as records and fields.

Like common programming language, Awk has variables, conditionals and loops

Awk has arithmetic and string operators.

Awk can generate formatted reports

Awk reads from a file or from its standard input, and outputs to its standard output. Awk does not get along with non-text files.

Ex. :- By default Awk prints every line from the file.

```
# awk '{print;}' </etc/passwd
o/p :- root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:dæmon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
```

- Print only specific field.

Awk has number of built in variables. For each record i.e line, it splits the record delimited by whitespace character by default and stores it in the \$n variables. If the line has 4 words, it will be stored in \$1, \$2, \$3 and \$4. \$0 represents whole line. NF is a built in variable which represents total number of fields in a record.

Ex.: - awk -F: '{print \$4}' </etc/passwd

o/p :- 0

1

2
4
7
0
0
0
12
0
100

- Ex.: - awk '{print \$2,\$5;}'</etc/passwd
- o/p :- ser:/var/ftp:/sbin/nologin

message
for
mDNS/DNS-SD
IPv4LL

SSH:/var/empty/sshd:/sbin/nologin
Malode:/home/isha.malode:/bin/bash

- Ex.: - awk '{print \$2,\$NF;}'</etc/passwd
- o/p :- root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

```
halt:x:7:0:halt:/sbin:/sbin/halt  
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin  
operator:x:11:0:operator:/root:/sbin/nologin
```

- Find the employees who has employee id greater than 200

Ex.: awk '\$1 > 200' </etc/passwd

```
o/p : isha.malode:x:1000:1000:Isha Malode:/home/isha.malode:/bin/bash  
prince:x:1001:1001::/home/prince:/bin/bash  
bob:x:1002:1002::/home/bob:/bin/bash  
juliet:x:1003:1003::/home/juliet:/bin/bash
```

SED COMMAND

Replacing or substituting string

Sed command is mostly used to replace the text in a file

Ex. :-# sed -n '1p' </etc/passwd
o/p :- root:x:0:0:root:/root:/bin/bash

The tr Command

The tr command is used to translate specified characters into other characters or to delete them.

In contrast to many command line programs, tr does not accept file names as arguments (i.e., input data). Instead, it only accepts inputs via standard input, (i.e., from the keyboard) or from the output of other programs via redirection.

The general syntax of tr is

```
tr [options] set1 [set2]
```

Convert lower case to upper case

The following tr command is used to convert the lower case to upper case

```
# tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

o/p :- thegeekstuff

THEGEEKSTUFF

The following command will also convert lower case to upper case

```
# tr [:lower:] [:upper:]
```

o/p:- thegeekstuff

THEGEEKSTUFF

Translate braces into parenthesis

You can also translate from and to a file. In this example we will translate braces in a file with parenthesis

```
# tr "{}" "{}" < inputfile > outputfile
```

The above command will read each character from "inputfile", translate if it is a brace, and write the output in "outputfile".

3. Translate white-space to tabs

The following command will translate all the white-space to tabs

```
# echo "This is for testing" | tr [:space:] '\t'
```

o/p:- This is for testing

diff command

diff analyzes two files and prints the lines that are different. Essentially, it outputs a set of instructions for how to change one file in order to make it identical to the second file.

It does not actually change the files; however, it can optionally generate a script (with the -e option) for the program ed (or ex which can be used to apply the changes).

Ex:- Let's say we have two files, file1.txt and file2.txt.

If file1.txt contains the following four lines of text:

I need to buy apples.

I need to run the laundry.

I need to wash the dog.

I need to get the car detailed.

...and file2.txt contains these four lines:

I need to buy apples.

I need to do the laundry.

I need to wash the car.

I need to get the dog detailed

...then we can use diff to automatically display for us which lines differ between the two files with this command:

```
diff file1.txt file2.txt
```

...and the output will be:

2,4c2,4

< I need to run the laundry.

< I need to wash the dog.

< I need to get the car detailed.

> I need to do the laundry.

> I need to wash the car.

> I need to get the dog detailed.

- another example. Let's say our two files look like this:

file1.txt:

I need to go to the store.
I need to buy some apples.
When I get home, I'll wash the dog.

file2.txt:

I need to go to the store.
I need to buy some apples.
Oh yeah, I also need to buy grated cheese.
When I get home, I'll wash the dog.

diff file1.txt file2.txt

Output:

2a3

> Oh yeah, I also need to buy grated cheese.

Cut command

Linux command cut is used for text processing. You can use this command to extract portion of text from a file by selecting columns.

- Ex.: - Select Column of Characters

To extract only a desired column from a file use -c option. The following example displays 2nd character from each line of a file test.txt

```
# cut -c2 test.txt
```

a

p

s

- **Select Column of Characters using Range**

Range of characters can also be extracted from a file by specifying start and end position delimited with -. The following example extracts first 3 characters of each line from a file called test.txt

```
$ cut -c1-3 test.txt
```

```
cat  
cp  
ls
```

uniq command

- test file is used in some of the example to understand how uniq command works.

```
# cat test  
aa  
aa  
bb  
bb  
bb  
xx
```

1. Basic Usage

Syntax:

```
$ uniq [-options]
```

For example, when uniq command is run without any option, it removes duplicate lines and displays unique lines as shown below.

```
$ uniq test
```

```
aa
```

```
bb
```

```
xx
```

- Count Number of Occurrences using -c option

This option is to count occurrence of lines in file.

```
$ uniq -c test
```

```
2 aa
```

```
3 bb
```

```
1 xx
```

- Print only Duplicate Lines using -d option

This option is to print only duplicate repeated lines in file. As you see below, this didn't display the line "xx", as it is not duplicate in the test file.

```
$ uniq -d test
```

```
aa
```

```
bb
```

sort command

sort is a simple and very useful command which will rearrange the lines in a text file so that they are sorted, numerically and alphabetically.

Ex.: let's say you have a file, data.txt, which contains the following ASCII text:

```
apples
```

```
oranges
```

pears

kiwis

bananas

To sort the lines in this file alphabetically, use the following command:

sort data.txt

...which will produce the following output:

apples

bananas

kiwis

oranges

pears

CONTROLLING SERVICES AND DAEMONS

Daemons are process that wait or run in the background performing various task.

1. Systemctl - query for all (to verify system startup)
2. Query state of only service units

Systemctl -- type= service

Ex. :- systemctl status sshd.service → to see status of particular service

Systemctl stop sshd.service → To stop the particular service

Systemctl start sshd.service → To start particular service

Systemctl restart sshd.service → To restart the service

Systemctl reload sshd.service → To reload the service

- While reloading service the PID will change.

3. To see active and enabled status

Systemctl is -active sshd

Systemctl is -enabled sshd

Particular service is enabled or not ?

Netstat -napl | grep <port no.>

In /etc/services → To see all ports

4. List active state of all loaded units, limit type of unit

Systemctl list-units --type = service

5. View enabled and disabled setting for all units

Systemctl list – unit – files --type = service

6. View only failed service

Systemctl --failed

7. List all socket units, active and inactive on system

Systemctl list-units --type=socket

8. To see listed daemons are running

Ps -p PID

9. To disable and enable services.

Systemctl enable sshd.service

Systemctl disable sshd.service

**** Disabling service does not stop service ****

CONFIGURING AND SECURING OPENSSH SERVICE

W → It will display list of user currently logged in pc.

This is especially useful to show which users are logged in, using ssh from which remote location.

To store password

/etc/sysconfig/network OR

/etc/hosts

Step 1. Ssh-keygen -t rsa

**** ssh file should be there, if not then create****

Mkdir .ssh

Ls -l

Now try to login into machine with ssh <ip>

IN second machine == Open vim authorized_keys

ANALYZING AND STORING LOGS

/var /log /messages → Most syslog messages are logged here.

/var /log /secure → The log file for security and authentication related messages and error.

/varr /log /maillog →The log file with mail server related messages.

/var /log /cron → The log file related to periodically executed task.

/var /log / boot.log → Messages related to system startup are logged here.

Journal stored in /run/log

Journalctl command shows the full system. journalctl, starting with oldest log entry when run as root user.

Journalctl -p err → command to only list any log entry of priority error or above.

Vim → /etc /logrotate.d

For modification → /etc /logrotate.conf.

MANAGING RED HAT ENTERPRISE LINUX NETWORKING

- **Displaying IP address**

* /sbin /ip command is used to show device and address information

→ ip addr show eth0

* To see statistics about network performance

The received (Rx) and transmitted (Tx) packets.

→ ip -s link show eth0

* Routing information

→ ip route

* To trace path to remote host

→ tracepath <hostname>

Command used to display socket statistics.

ss -ta → show socket status.

ss -n → show number instead of names for interface and ports.

ss -t → Show tcp socket.

ss -u → show udp socket.

ss -l → show only listening socket.

ss -a → show all (listening and established) socket.

ss -p → show process using sockets.

ss -lt → Display listening TCP socket on local system.

* CONFIGURING NETWORK WITH NMCLI *

- To display list of all connection

```
# nmcli con show
```

- To see device details and status

```
# nmcli dev status
```

```
# nmcli dev show eth0
```

CREATING NETWORK CONNECTIONS WITH NMCLI

- Define a new connection named "default" which will autoconnect as an Ethernet connection on the eth0 device using DHCP

```
# nmcli con add con-name "default" type Ethernet ifname eth0
```

- Create a new connection named "static" and specify IP addr and gateway. Do not autoconnect.

```
# nmcli con add con-name "static" ifname eth0 autoconnect no type Ethernet ip4.
```

- The system will autoconnect with DHCP connection at boot change to static connection

```
# nmcli con up "static"
```

- Change back to DHCP connection

```
# nmcli con up "default"
```

SUMMARY OF nmcli COMMANDS

nmcli dev status → list all devices

nmcli con show → list all connections

nmcli con up <ID> → Activate a connection

nmcli con down <ID> → Deactivate a connection

nmcli dev di <DEV> → bring down an interface and temp disable autoconnect

nmcli net off → Disable all managed interfaces

nmcli con add → Add new connection

nmcli con mod <ID> → modify connection

nmcli con del <ID> → delete connection

hostname → this command display or temp modifies the system's fully qualified host name

- Display host name status

hostnamectl status

- To change permanent hostname → vi /etc/hostname (in 7th)
/etc/sysconfig/network (in 6th)

- To see the version

/etc/redhat-release

- To change host name and host name configuration file

sudo hostnamectl set – hostname

- To add router or gateway

ip route add <ip> dev eth0

ARCHIVING AND COPYING FILES BETWEEN SYSTEMS

- Archives files and directories with tar

c → create an archive

t → list the content of an archive

x → extract an archive

f → file name

- To create compressed tar archive

z → For gzip compression (filename.tar.gz)

j → For bzip2 compression (filename.tar.bz2)

J → For xz compression (filename.tar.xz)

Q :- Create the archive with named archive.tar with contents of file1, file2, file3 in home directory

Ans :- tar cf archive.tar file1, file2, file3

Q :- Create tar archive /root/etc.tar with /etc

Ans :- tar cvf /root/etc.tar /etc

Compression of file

Ex. :- 1. Create directory

#mkdir test/

tar -zcvf test.tar.gz test1 tom tomjerry

Uncompressed

tar -xvf test.tar.gz

- To list contents of a tar archive

tar tf /root/etc.tar

COPYING FILES BETWEEN SYSTEMS SECURELY

- For copying file

```
# scp -r <filename> root @ <ip> :/root
```

Ex. :- scp -r abcd root@ 172.24.2.52: /root

- For transferring file remotely

```
# sftp
```

- Synchronizing files between systems securely

```
# rsync
```

INSTALLING AND UPDATING SOFTWARE PACKAGES

Yum → It is used to list repositories, packages group

```
# yum repolist
```

```
# yum list yum*
```

```
# yum list installed
```

```
# yum grouplist
```

FOR INSTALLATION

```
# yum install <packagename>
```

FOR UPDATION

```
# yum update <packagename>
```

FOR REMOVE

```
# yum remove <packagename>
```

Yum group install command will install group which will install its mandatory and default packages

```
# yum group install
```

VIEWING TRANSACTION HISTORY

```
# tail -5 /var/log/yum.log
```

A summary of install and remove transaction can be viewed with yum history

```
# yum history
```

Search for package by keyword

```
# yum search <keyword>
```

To view all available repositories

```
# yum repolist all
```

Display information about package

```
# rpm -q -c NAME
```

List all files included in package

```
# rpm -q -I NAME
```

List config files included in package

```
# rpm -q -c NAME
```

Show a short summary of reason for a new package release

```
# rpm -q --changelog NAME
```

Display the shell scripts included in package

```
# rpm -q --scripts NAME
```

ACCESSING LINUX FILE SYSTEM

Overview about file system mount points and amount of free space available

```
# df  
# df -h  
# df -H
```

For more detailed information about space used by certain directory tree

```
# du /root
```

Use to discover the UUID of newly added partition on server x

```
# blkid
```

To create mount point /mnt/newspace on server

```
# mkdir /mnt /newspace
```

Mount file system by UUID on /mnt/newspace directory of server x machine

```
# mount UUID=' ' /mnt/newspace
```

Change to /mnt/newspace directory on server

```
# cd /mnt/newspace
```

Create a new directory, /mnt/newspace/newdir on server

```
# mkdir newdir
```

Create a new empty file, /mnt/newspace/newdir/newfile, on server

```
# touch newdir /newfile
```

Unmount the file system mounted on /mnt/newspace directory on servers

```
# umount /mnt/newspace
```

INSTALLATION AND CONFIGURING OF APACHE

Yum install httpd*

For configuration

vim /etc/httpd/conf/httpd.conf

For basic installation modification not required

Create a file in /var/www/html

cd /var/www/html

vim index.html

=====

service httpd restart

Server configured.

Go to firefox and check. OR

We can use e-link for that

Install e-link

yum install elinks

If error comes check configuration file

view /etc/httpd/conf/httpd.conf

QUESTIONS AND ANSWERS

Q :- How do you display single user from shadow file ?

Ans:- # vi /etc/shadow

grep <username> /etc/shadow

Q :- View the contents of a directory : A directory may contains visible and invisible files with different file permissions.

Ans: #ls -al

o/p: drw-r--r-- 2 root root 18 Jun 17 14:46 .test

Q :- Viewing system partitions and used space

Ans: #fdisk -l

o/p: Disk /dev/sda: 37.6 GB, 37580963840 bytes, 73400320 sectors

Units = sectors of 1 * 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk label type: dos

Disk identifier: 0x000b08b1

Q :- Know your machine name, OS and Kernel

Ans: #hostnamectl status

o/p: Static hostname: localhost.localdomain

Icon name: computer-vm

Chassis: vm

Machine ID: ef67461d095844519f29acc21b90cf35

Boot ID: 3b427f7acc18458991b51d0a9acf2577

Virtualization: microsoft

Operating System: CentOS Linux 7 (Core)

CPE OS Name: cpe:/o:centos:centos:7

Kernel: Linux 3.10.0-123.el7.x86_64

Architecture: x86-64

Q :- Viewing history

Ans: #history

o/p: 252 nmcli con show

253 nmcli con show --active

254 nmcli con show "static-eth0"

255 nmcli dev status

256 nmcli dev show eth0

257 nmcli dev status

Q :- Being root from your user

Ans: #su

Q :- create Directory

Ans: #mkdir

o/p: isha2 snap2.jpg

Q :- create Files

Ans: # touch, or #cat

o/p: -rw-r--r--. 1 root root 0 Sep 12 17:45 nzfile1

Q :- Changing the file permission

Ans: #chmod

o/p: -rw-r--r--. 1 root root 0 Jul 14 15:46 test1

chmod go-r test1

-rw-----. 1 root root 0 Jul 14 15:46 test1

Q :- Install, Update and maintain Packages

Ans: #using yum

o/p: yum install wget

Loaded plugins: fastestmirror

base	3.6 kB	00:00
extras	3.4 kB	00:00
updates	3.4 kB	00:00

Loading mirror speeds from cached hostfile

* base: mirror.fibergrid.in

* extras: mirror.fibergrid.in

* updates: mirror.fibergrid.in

Resolving Dependencies

--> Running transaction check

--> Package wget.x86_64 0:1.14-10.el7_0.1 will be installed

--> Finished Dependency Resolution

Dependencies Resolved

Package	Arch	Version	Repository	Size
---------	------	---------	------------	------

Installing:

wget	x86_64	1.14-10.el7_0.1	base	545 k
------	--------	-----------------	------	-------

Transaction Summary

Install 1 Package

Q :- Uncompressing a file

Ans:#tar -zvxf file.tar.gz

o/p:

Q :- See current date, time and calendar

Ans: #date +%x

o/p: Monday 12 September 2016

Q :- Print contents of a file on command line

Ans:# cat

o/p: cat> nzfile1

hi

Q :- Copy and Move

Ans:# cp and mv

o/p: mv sum.sh moon.sh

ls moon.sh

moon.sh

Q :- See the working directory for easy navigation

Ans: #pwd

o/p:]# pwd

/root

Q :- Change the working directory, etc...

Ans: #cd

Q :- List out all files and directories in a given directory for ex: on /home

Ans: # ls -l /home

o/p: /home:

abc ateam bob elvis isha.malode prince sspade

alice ateam-text dolly hamlet jerry reba user1

andy bboop dtracy isha juliet romeo

Q :- Finding a file in a given directory

Ans: #ls <filename>

o/p: ls nzfile1

nzfile1

Q :- Searching a file with the given keywords

Ans: #grep <word> <filename>

o/p: grep "number" pali.sh

echo -n "Enter number : "

store number in reverse order

store original number

store previous number and current digit in reverse

Q :- See the current running processes

Ans: #ps

o/p: PID TTY TIME CMD

6426 pts/0 00:00:00 bash

6823 pts/0 00:00:00 su

6843 pts/0 00:00:00 su

6847 pts/0 00:00:00 bash

6935 pts/0 00:00:00 su

Q :- Kill a running process

Ans: # kill -9 pid

Q :- Starting, Ending, Restarting a service

Ans: # systemctl status<service name>.service

Systemctl start <service name>.service

Systemctl stop <service name>.service

Q :- Making and removing of aliases

Ans: #unalias <name> for removing

alias -p <name>

o/p: # alias p="pwd"

[root@localhost ~]# p

/root

Q :- View the disk and space usages

Ans:# df -h and du -s

```
O/p: Filesystem      Size Used Avail Use% Mounted on
/dev/mapper/centos-root 33G 1.4G 32G 5% /
devtmpfs        488M  0 488M 0% /dev
tmpfs          494M  0 494M 0% /dev/shm
tmpfs          494M 264K 494M 1% /run
tmpfs          494M  0 494M 0% /sys/fs/cgroup
/dev/sda1       497M 96M 402M 20% /boot
```

Q :- Removing a file and/or directory

Ans: #rm -r

```
o/p: drwxr-xr-x. 2 root root   6 Jun 17 10:39 family
drwxr-xr-x. 2 root root   6 Jun 17 10:39 friend
o/p: drwxr-xr-x. 2 root root   6 Jun 17 10:39 friend
drwxr-xr-x. 5 root root   45 Jun  7 12:53 glob
```

Q :- Changing password of on-self and other's, if you are root.

Ans:# passwd

o/p: passwd

Changing password for user root.

New password:

Q :- Compare two files

Ans: #diff file1 file2

```
o/p: [root@localhost ~]# diff vyankat tom
```

1,66d0

<

< 1.Insert Mode

< 2.Escape Mode

< 3.Colon Mode

Q :- Download a file, the Linux way (wget)

Ans:#wget <file name>

```
o/p: ]# wget http://website.com/files/file.zip
```

--2016-09-12 18:39:14-- <http://website.com/files/file.zip>

Resolving website.com (website.com)... 65.61.198.201

Connecting to website.com (website.com)|65.61.198.201|:80... connected.

HTTP request sent, awaiting response... 301 Moved Permanently

Location: <http://www.website.com/files/file.zip> [following]

--2016-09-12 18:39:15-- <http://www.website.com/files/file.zip>

Resolving www.website.com (www.website.com)... 65.61.198.201

Reusing existing connection to website.com:80.

Q :- Mount a block / partition / external HDD

Ans:# mount

Q :- Configuring Network Interface

Ans:# ifconfig

Q :- Viewing custom Network Related information

Ans:# ip addr show

o/p: 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid_lft forever preferred_lft forever

inet6 ::1/128 scope host

valid_lft forever preferred_lft forever

Q :- Digging DNS (how to check IP address of specific URL. for ex: to get IP address of www.google.com)

Ans:# nslookup google.com

o/p: Server: 192.168.0.105

Address: 192.168.0.105#74

Q :- Knowing Your System uptime

Ans:# uptime

o/p: 17:51:36 up 3 days, 4:48, 1 user, load average: 0.00, 0.01, 0.05

Q :- Renaming a file

Ans:# mv

o/p: mv newfile3 newfile6

o/p: -rw-rw---- 1 root root 0 Jul 15 16:54 newfile6

Q :- Text File editors like vi, emacs and nano

Ans:# vim /etc/exports (for configuring nfs)

O/p:

Q :- Copy folder from one location to another location with all subdirectories and files

Ans:# cp -R source/destination/

o/p: cp -R video newmoon

[root@localhost ~]# ls newmoon

watched

Q :- check free and available memory

Ans:#free

o/p:

	total	used	free	shared	buffers	cached
Mem:	1010924	889356	121568	240	20	174784
-/+ buffers/cache:	714552	296372				
Swap:	2129916	1732	2128184			

Ext2

- Ext2 stands for second extended file system.
- It was introduced in 1993. Developed by Rémy Card.
- This was developed to overcome the limitation of the original ext file system.
- Ext2 does not have journaling feature.
- On flash drives, usb drives, ext2 is recommended, as it doesn't need to do the overhead of journaling.
- Maximum individual file size can be from 16 GB to 2 TB
- Overall ext2 file system size can be from 2 TB to 32 TB

Ext3

- Ext3 stands for third extended file system.
- It was introduced in 2001. Developed by Stephen Tweedie.
- Starting from Linux Kernel 2.4.15 ext3 was available.
- The main benefit of ext3 is that it allows journaling.
- Journaling has a dedicated area in the file system, where all the changes are tracked.
- When the system crashes, the possibility of file system corruption is less because of journaling.
- Maximum individual file size can be from 16 GB to 2 TB
- Overall ext3 file system size can be from 2 TB to 32 TB
- There are three types of journaling available in ext3 file system.
 - Journal – Metadata and content are saved in the journal.
 - Ordered – Only metadata is saved in the journal. Metadata are journaled only after writing the content to disk. This is the default.
 - Writeback – Only metadata is saved in the journal. Metadata might be journaled either before or after the content is written to the disk.
- You can convert a ext2 file system to ext3 file system directly (without backup/restore).

Ext4

- Ext4 stands for fourth extended file system.
- It was introduced in 2008.
- Starting from Linux Kernel 2.6.19 ext4 was available.
- Supports huge individual file size and overall file system size.
- Maximum individual file size can be from 16 GB to 16 TB

- Overall maximum ext4 file system size is 1 EB (exabyte). 1 EB = 1024 PB (petabyte). 1 PB = 1024 TB (terabyte).
- Directory can contain a maximum of 64,000 subdirectories (as opposed to 32,000 in ext3)
- You can also mount an existing ext3 fs as ext4 fs (without having to upgrade it).
- Several other new features are introduced in ext4: multiblock allocation, delayed allocation, journal checksum, fast fsck, etc. All you need to know is that these new features have improved the performance and reliability of the filesystem when compared to ext3.
- In ext4, you also have the option of turning the journaling feature “off”.

Here is a short list of advantages of Ext4 over Ext3:

- extents (reduce overhead for large files, reduce fragmentation and improve performance)
 - flexible block groups (fast fsck)
 - support for huge files, huge total filesystem size
 - more subdirectories
 - journal checksumming
- Overall, Ext4 provides better performance, reliability and scalability.



The Chef Server

The Chef server is the primary mode of communication between the workstations where your infrastructure is coded, and the nodes where it is deployed. All configuration files, cookbooks, metadata, and other information are stored on the server. The Chef server also keeps information regarding the state of all nodes at the time of the last `chef-client` run.

Any changes made must pass through the Chef server to be deployed. Prior to accepting or pushing changes, it verifies that the nodes and workstations are paired with the server through the use of authorization keys, and then allows for communication between the workstations and nodes.

Workstations

Workstations are where users create, test, and maintain cookbooks and policies that will be pushed to nodes. Cookbooks created on workstations can be used privately by one organization, or uploaded to the Chef Supermarket for others to use. Similarly, workstations can be used to download cookbooks created by other Chef users and found in the Supermarket.

Workstations are set up to use the *Chef Development Kit* (ChefDK), and can be located on virtual servers or on physical workstation computers. Workstations are set to interact with only one Chef server, and most work will be done in the `chef-repo` directory located on the workstation.

Nodes

A node is a system configured to run the `chef-client`. This can be any system, as long as it is being maintained by Chef.

Nodes are validated through the `validator.pem` and `client.pem` certificates that are created on the node when it is bootstrapped. All nodes must be bootstrapped over SSH as either the root user or a user with elevated privileges.

Nodes are kept up-to-date through the use of the `chef-client`, which runs a convergence between the node and the Chef server. What cookbooks and roles the node will take on depends on the run list and environment set for the node in question.

`chef-client`

The `chef-client` checks the current configuration of the node against the recipes and policies stored in the Chef server and brings the node up to match. The process begins with the `chef-client` checking the node's run list, loading the cookbooks required, then checking and syncing the cookbooks with the current configuration of the node.

The `chef-client` must be run with elevated privileges in order to properly configure the node, and should be run periodically to ensure that the server is always up to date – often this is achieved through a cron job or by setting up the `chef-client` to run as a service.

•

`chef-repo`

The `chef-repo` directory is the specific area of the workstation where cookbooks are authored and maintained. The `chef-repo` is always version-controlled, most often through the use of Git, and stores information and history that will be used on nodes, such as cookbooks, environments, roles, and data

bags. Chef is able to communicate with the server from the `chef-repo` and push any changes via the use of the `knife` command, which is included in the ChefDK.

Originally the `chef-repo` had to be pulled from GitHub using git commands, but that action is now integrated into Chef through the use of the `chef generate repo chef-repo` command.

Knife

The `knife` command communicates between the `chef-repo` located on a workstation and the Chef server. `knife` is configured with the `knife.rb` file, and is used from the workstation:

`~/chef-repo/.chef/knife.rb`

```
1 log_level:info
2 log_locationSTDOUT
3 node_name'username'
4 client_key'~/chef-repo/.chef/username.pem'
5 validation_client_name'shortname-validator'
6 validation_key'~/chef-repo/.chef/shortname.pem'
7 chef_server_url'https://123.45.67.89/organizations/shortname'
8 syntax_check_cache_path'~/chef-repo/.chef/syntax_check_cache'
9 cookbook_path['~/chef-repo/cookbooks']
```

The default `knife.rb` file is defined with the following properties:

- **log_level:** The amount of logging that will be stored in the log file. The default value, `:info`, notes that any informational messages will be logged. Other values include `:debug`, `:warn`, `:error`, and `:fatal`.
- **log_location:** The location of the log file. The default value, `STDOUT` is for *standard output logging*. If set to another value standard output logging will still be performed.
- **node_name:** The username of the person using the workstation. This user will need a valid authorization key located on the workstation.
- **client_key:** The location of the user's authorization key.
- **validation_client_name:** The name for the server validation key that will determine whether a node is registered with the Chef server. These values must match during a chef-client run.
- **validation_key:** The path to your organization's validation key.
- **chef_server_url:** The URL of the Chef server, with `shortname` being the defined shortname of your organization. This can also be an IP address. `/organizations/shortname` must be included in the URL.
- **syntax_check_cache_path:** The location in which `knife` stores information about files that have been checked for appropriate Ruby syntax.
- **cookbook_path:** The path to the cookbook directory.

Run Lists

Run lists define what cookbooks a node will use. The run list is an ordered list of all cookbooks and recipes that the chef-client needs to pull from the Chef server to run on a node. Run lists are also used to define roles, which are used to define patterns and attributes across nodes.

Ohai

Ohai collects information regarding nodes for the Chef server. It is required to be present on every node, and is installed as part of the bootstrap process.

The information gathered includes network and memory usage, CPU data, kernel data, hostnames, FQDNs, and other automatic attributes that need to remain unchanged during the chef-client run.

Cookbooks

Cookbooks are the main component of configuring nodes on a Chef infrastructure. Cookbooks contain values and information about the *desired state* of a node, not how to get to that desired state – Chef does all the work for that, through their extensive libraries.

Cookbooks are comprised of recipes, metadata, attributes, resources, templates, libraries, and anything else that assists in creating a functioning system, with attributes and recipes being the two core parts of creating a cookbook. Components of a cookbook should be modular, keeping recipes small and related.

Cookbooks can and should be version controlled. Versions can help when using environments and allow for the easier tracking of changes that have been made to the cookbook.

Recipes

Recipes are the fundamental part of cookbooks. Recipes are written in Ruby and contain information in regards to everything that needs to be run, changed, or created on a node. Recipes work as a collection of *resources* that determine the configuration or policy of a node, with resources being a configuration element of the recipe. For a node to run a recipe, it must be on that node's run list.

Attributes

Attributes define specific values about a node and its configuration. These values are used to override default settings, and are loaded in the order cookbooks are listed in the run list. Often attributes are used in conjunction with templates and recipes to define settings.

Files

These are static files that can be uploaded to nodes. Files can be configuration and set-up files, scripts, website files – anything that does not need to have different values on different nodes.

Libraries

Although Chef comes with a number of libraries built in, additional libraries can be defined. Libraries are what bring recipes to life: If a recipe is the *desired state* of a node, then added libraries contain the behind-the-scenes information Chef needs for the nodes to reach this state. Libraries are written in Ruby, and can also be used to expand on any functionalities that Chef already contains.

Providers and Resources

DEVOPS MATERIAL

Providers and resources are also used to define new functionality to use in Chef recipes.

A *resource* defines a set of actions and attributes, whereas *provider* informs the chef-client how to commit each action.

Templates

Templates are embedded Ruby files (.erb) that allows for content based on the node itself and other variables generated when the chef-client is run and the template is used to create or update a file.

Environments

Chef environments exist to mimic real-life workflow, allowing for nodes to be organized into different “groups” that define the role the node plays in the fleet. This allows for users to combine environments and versioned cookbooks to have different attributes for different nodes. For example, if testing a shopping cart, you may not want to test any changes on the live website, but with a “development” set of nodes.

Environments are defined in chef-repo/environments and saved as Ruby or JSON files.
As a Ruby file:

chef-repo/environments/environmentname.rb

```
1 name "environmentname"
2 description "environment_description"
3 cookbook_versions "cookbook" => "cookbook_version"
4 default_attributes "node" => { "attribute" => ["value", "value", "etc."] }
5 override_attributes "node" => { "attribute" => ["value", "value", "etc."] }
```

As a JSON:

chef-repo/environments/environmentname.json

```
1 {
2   "name": "environmentname",
3   "description": "a description of the environment",
4   "cookbook_versions": {
5     "json_class": "Chef::Environment",
6     "chef_type": "environment",
7     "default_attributes": {
8       "override_attributes": {
9         "10": {
11       },
12     }
13   }
14 }
```

DEVOPS MATERIAL

All nodes are automatically set to the “default” environment upon bootstrap. To change this, the environment should be defined in the client.rb file found in/etc/chef on thenodes.

Bookshelf

The Bookshelf is a versioned repository where cookbooks are stored on the Chef server (generally located at /var/opt/opscode/bookshelf; full root access is needed). When a cookbook is uploaded to the Chef server, the new version is compared to the one already stored; if there are changes, a new version is stored. The Chef server only stores one copy of a file or template at once, meaning if resources are shared between cookbooks and cookbook versions, they will not be stored multiple times.

Install chef

```
=====
curl -L https://www.opscode.com/chef/install.sh | bash
Check
=====
# chef-solo -v
Chef: 11.6.2
Setup chef repository
=====
#wget http://github.com/opscode/chef-repo/tarball/master
#tar zxvf master
opscode-chef-repo-f9d4b0c/
opscode-chef-repo-f9d4b0c/.gitignore
opscode-chef-repo-f9d4b0c/LICENSE
opscode-chef-repo-f9d4b0c/README.md
opscode-chef-repo-f9d4b0c/Rakefile
opscode-chef-repo-f9d4b0c/certificates/
opscode-chef-repo-f9d4b0c/certificates/README.md
opscode-chef-repo-f9d4b0c/chefignore
opscode-chef-repo-f9d4b0c/config/
opscode-chef-repo-f9d4b0c/config/rake.rb
opscode-chef-repo-f9d4b0c/cookbooks/
opscode-chef-repo-f9d4b0c/cookbooks/README.md
opscode-chef-repo-f9d4b0c/data_bags/
opscode-chef-repo-f9d4b0c/data_bags/README.md
opscode-chef-repo-f9d4b0c/environments/
opscode-chef-repo-f9d4b0c/environments/README.md
opscode-chef-repo-f9d4b0c/roles/
opscode-chef-repo-f9d4b0c/roles/README.md
#mv opscode-chef-repo-f9d4b0c/ chef-repo
# ls chef-repo/
certificates config data_bags LICENSE README.md
chefignore cookbooks environments Rakefile roles
Notice the directory structure created in the chef repository?
Create .chef directory inside chef-repo
=====
# mkdir chef-repo/.chef
```

Setup a local cookbook path

```
# vi chef-repo/.chef/knife.rb
cookbook_path[ '/root/chef-repo/cookbooks' ]
Create your first cookbook
=====
# knife cookbook create ntp
** Creating cookbook ntp
** Creating README for cookbook: ntp
** Creating CHANGELOG for cookbook: ntp
** Creating metadata for cookbook: ntp
Here's the cookbook
=====
# ls chef-repo/cookbooks/ntp/
attributes definitions libraries providers recipes
templates
CHANGELOG.md files metadata.rb README.md resources
Writing your first recipe
=====
```

```
# vi chef-repo/cookbooks/ntp/recipes/default.rb
package 'ntp'
This will install ntp if it is not installed. If it is already
installed, it will do nothing.
```

Now configure your server using chef-solo

```
=====
# vi chef-repo/solo.rb
file_cache_path "/root/chef-solo"
cookbook_path "/root/chef-repo/cookbooks"
# vi web.json
{
  "run_list": [ "recipe[ntp]" ]
}
Ensure ntp is not installed
=====
```

```
# yum remove -y ntp
Use chef-solo to configure your server
=====
```

```
# chef-solo -c chef-repo/solo.rb -j chef-repo/web.json
Starting Chef Client, version 11.6.2
Compiling Cookbooks...
Converging 1 resources
Recipe: ntp::default
* package[ntp] action install
- install version 4.2.4p8-3.el6.centos of package ntp
Chef Client finished, 1 resources updated
Confirm ntp is installed
=====
```

```
# yum info ntp
Notes
=====
Uninstall chef
# yum remove -y chef
Other ways to install chef
# wget https://www.opscode.com/chef/install.sh
# bash install.sh
or
Go http://www.opscode.com/chef/install/ and select your distro, distro
=====
```

version and architecture. Download the package.

In my case, I use CentOS:

```
# wget https://opscode-omnibuspackages.s3.amazonaws.com/el/6/x86_64/chef-11.6.2-1.el6.x86_64.rpm  
# yum localinstall -y chef-11.6.2-1.el6.x86_64.rpm
```

Another to setup your chef repo

```
# git clone git://github.com/opscode/chef-repo.git  
# ls chef-repo/  
certificates chefignore config cookbooks data_bags environments  
LICENSE Rakefile README.md roles
```

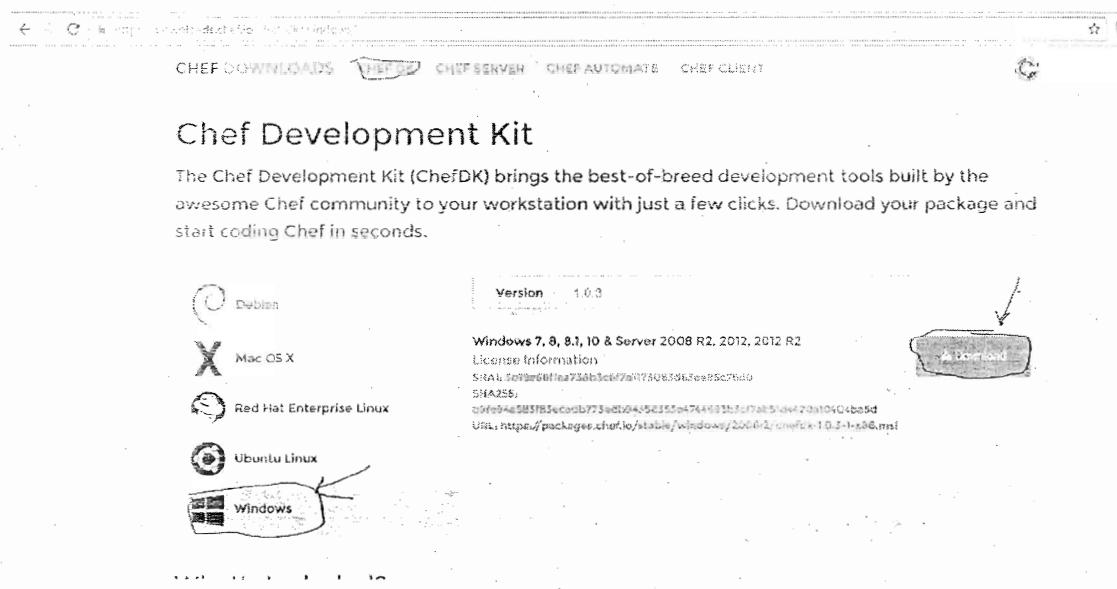
References

1. <http://gettingstartedwithchef.com/first-steps-with-chef.html>
2. <https://learnchef.opscode.com/starter-use-cases/ntp/>

Chef workstation setup on windows

On your workstation open browser and go to below link

<https://downloads.chef.io/chef-dk/windows/>



The screenshot shows the Chef Development Kit (ChefDK) download page. At the top, there are navigation links: CHEF DOWNLOADS, CHEF SERVER, CHEF AUTOMATE, and CHEF CLIENT. Below these, a heading reads "Chef Development Kit". A sub-section titled "Windows 7, 8, 8.1, 10 & Server 2008 R2, 2012, 2012 R2" is shown, indicating the supported operating systems. It includes a "Version 1.0.3" link, a "License Information" link, and a "SHA1: 5e396bf1ea734b3c6f7a0173083d5eae85c796d05fa255" link. There is also a "URL: https://package.chef.io/stable/windows/2008r2/chefdk-1.0.3-x64.msi" link. On the left, there are icons for Debian, Mac OS X, Red Hat Enterprise Linux, Ubuntu Linux, and Windows. A large "Download" button with an arrow pointing down is located on the right side of the page.

Open command prompt

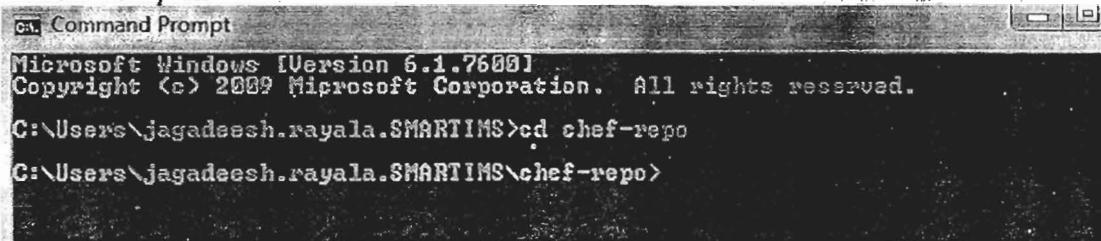
Create chef-repo directory under your home directory



```
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Users\jagadeesh.rayala.SMARTIMS>cd chef-repo
```

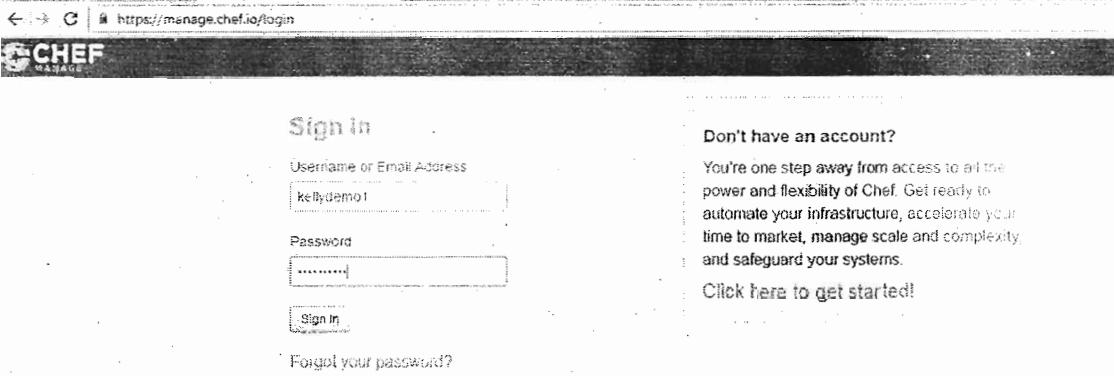
Go to chef-rep dir



```
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Users\jagadeesh.rayala.SMARTIMS>cd chef-repo
C:\Users\jagadeesh.rayala.SMARTIMS\chef-repo>
```

Login to chef console



The screenshot shows a web browser window with the URL <https://manage.chef.io/login> in the address bar. The page is titled "CHEF MANAGE". It features a "Sign in" form with fields for "Username or Email Address" containing "kellydemo1" and "Password" containing ".....". Below the form is a "Sign In" button. To the right of the form is a "Don't have an account?" section with a descriptive paragraph and a "Click here to get started!" link.

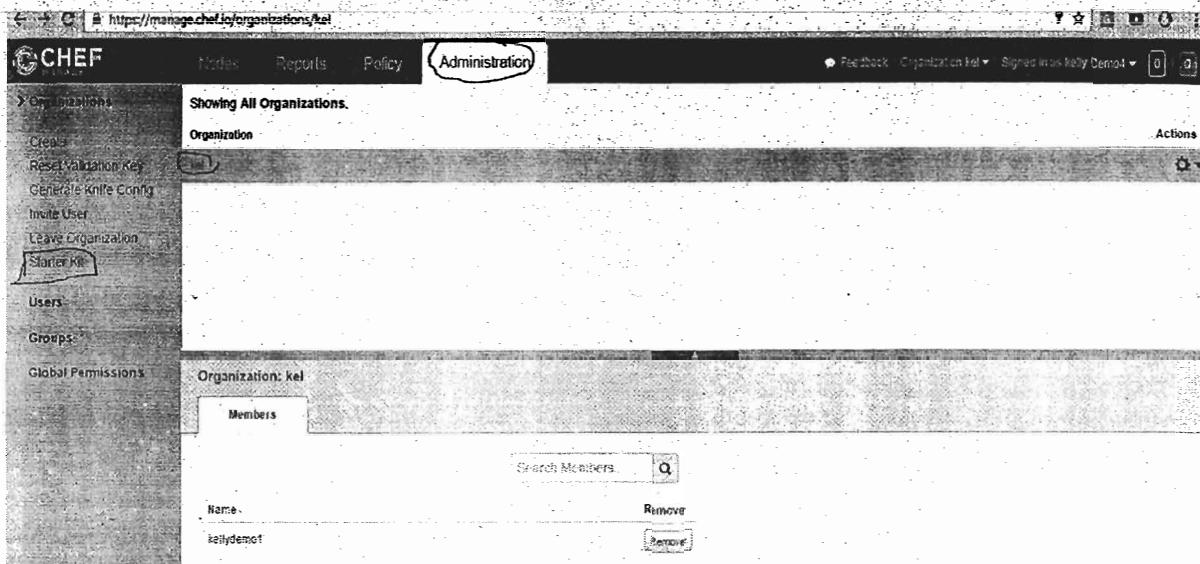
Sign in

Username or Email Address

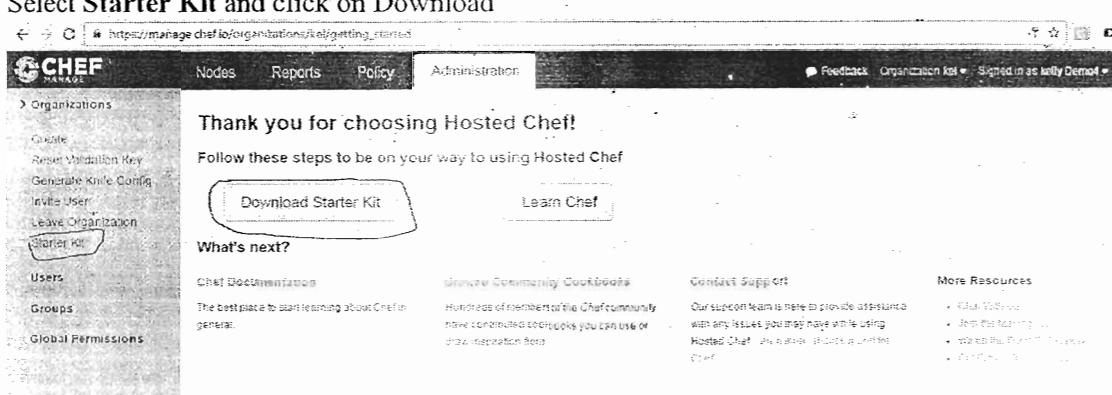
Password

Forgot your password?

Don't have an account?
You're one step away from access to all the power and flexibility of Chef. Get ready to automate your infrastructure, accelerate your time to market, manage scale and complexity, and safeguard your systems.
[Click here to get started!](#)



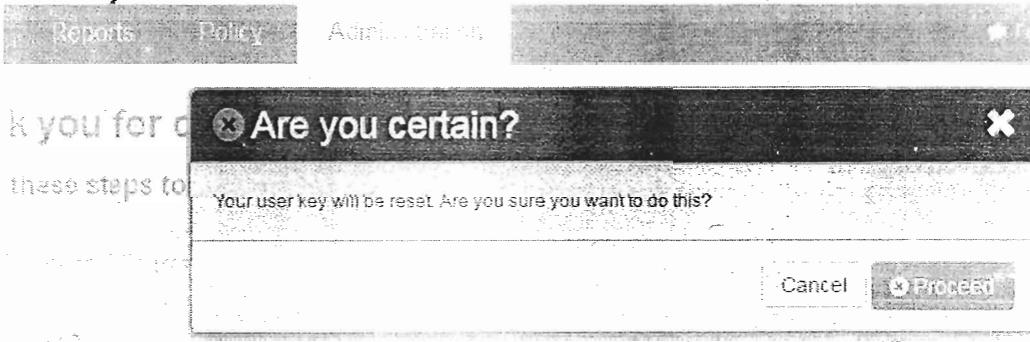
Select **Administration** click on organization name and
Select **Starter Kit** and click on Download



What's next?

Chef Documentation	Community Cookbooks	Contact Support	More Resources
The best place to start learning about Chef in general.	Hundreds of members of the Chef community have contributed cookbooks you can use or draw inspiration from.	Our support team is here to provide assistance with any issues you may have while using Hosted Chef. We answer tickets in under the hour.	<ul style="list-style-type: none"> • Chef Workstation • Join the Chef Community • Watch the Playbook • Get Help

Click on proceed



You will get Zip format **chef-starter** folder



Unzip it. After unzip you will see below folder.



Go to the **chef-starter** folder

You will see chef-repo folder go to that folder

Copy all the file under chef-repo folder to chef-repo folder in your home directory

	.chef
	cookbooks
	roles
	README.md

12/2/2016 7:29 AM	File folder
12/2/2016 7:16 AM	File folder
11/30/2016 2:27 AM	File folder
11/30/2016 2:27 AM	Text Document
11/30/2016 2:27 AM	MD File
	1 KB
	3 KB

Now check whether workstation is communicating to server

Run below command

```
C:\Users\jagadeesh.rayala.SMARTIMS>cd chef-repo
C:\Users\jagadeesh.rayala.SMARTIMS\chef-repo>knife client list
[output]
  [redacted]
```

You will see output organization-validator.

Now you successfully connected to your chef server.

Setup a Chef Workstation on Linux

Install some dependencies

```
bash-shell# yum install-ygit
```

Install Chef

```
bash-shell# curl -L https://www.opscode.com/chef/install.sh |bash
```

Verify Chef install

```
bash-shell# chef-client -v
```

Clone Chef repo

```
bash-shell# cd ~
bash-shell# git clone git://github.com/opscode/chef-repo.git
```

Configure Chef

```
bash-shell# cd chef-repo/
bash-shell# mkdir-p ~/chef-repo/.chef
bash-shell# echo'.chef'>> .gitignore
bash-shell# echo'export PATH="/opt/chef/embedded/bin:$PATH"'>> ~/.bash_profile && source ~/.bash_profile
bash-shell# cd ~/chef-repo/.chef
```

Copy your chef-validator.pem and admin.pem Chef server /etc/chef-server to ~/chef-repo/.chef/

```
bash-shell# chmod600 ~/chef-repo/.chef/*.pem
```

Create a knife.rb configuration file

```
bash-shell# knife configure --initial

WARNING: No knife configuration file found
Where should I put the config file? ~/chef-repo/.chef/knife.rb
Please enter the chef server URL: https://us-east-chef-server-01.givit.com:443
Please enter a name for the new user: knife
Please enter the existing admin name. [admin]
Please enter the location of the existing admin's private key: ~/chef-repo/.chef/admin.pem
Please enter the validation clientname: [chef-validator]
Please enter the location of the validation key: ~/chef-repo/.chef/chef-validator.pem
Please enter the path to a chef repository (or leave blank):
Creating initial API user...
Please enter a password for the new user:
Created user[knife]
Configuration file written to ~/chef-repo/.chef/knife.rb
```

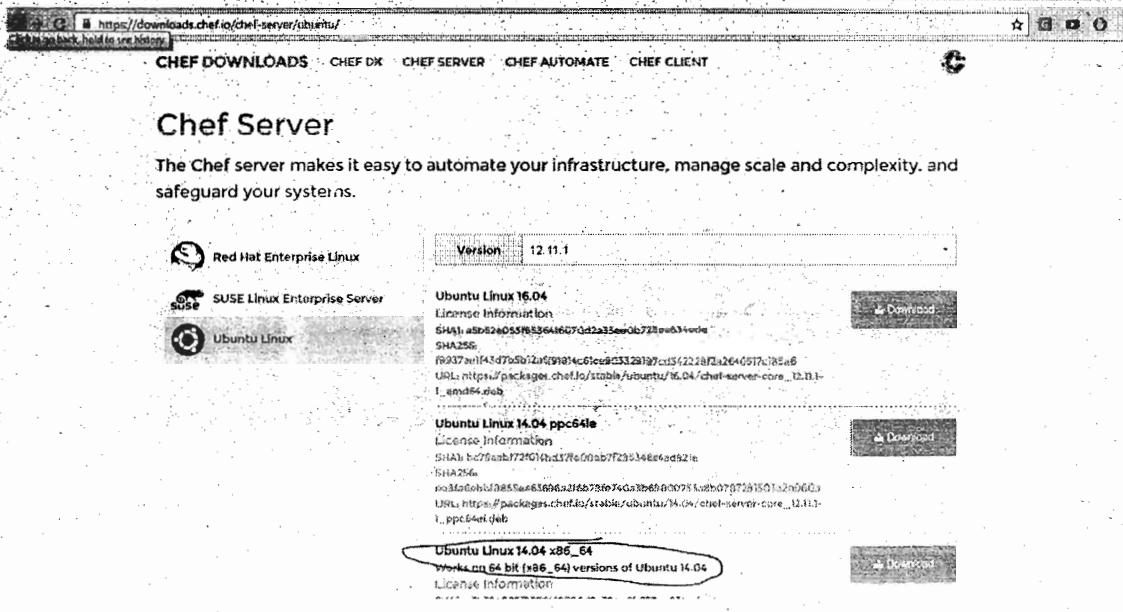
Verify your workstation install

```
bash-shell# knife client list
chef-validator
```

Install Chef server on Amazon EC2 Ubuntu 14.04

The **Chef server** is the hub of interaction between all **workstations** and **nodes** using Chef. Changes made through workstations are uploaded to the Chef server, which is then accessed by the chef-client and used to configure each individual node.

DEVOPS MATERIAL



The screenshot shows the 'CHEF DOWNLOADS' section for the Chef Server. It lists three download options for Ubuntu 14.04:

- Ubuntu Linux 12.11.1**: Version 12.11.1, Release Date: 2013-07-22, SHA256: 93037f1f43d705b012a79a074c61ce933323197c15f2228f2a2e4057c185a8, URL: https://packages.chef.io/stable/ubuntu/14.04/chef-server-core_12.11.1_amd64.deb, Download button.
- Ubuntu Linux 14.04 ppc64le**: Version 14.04, Release Date: 2014-07-22, SHA256: b7c79aeb72401a217fe001ab7f23534ee4ad52fe, URL: https://packages.chef.io/stable/ubuntu/14.04/chef-server-core_14.04_ppc64le.deb, Download button.
- Ubuntu Linux 14.04 x86_64**: Version 14.04, Release Date: 2014-07-22, SHA256: 0855ae63696a2f6e78fe74ca2b69000751a2bd721501a2h0603, URL: https://packages.chef.io/stable/ubuntu/14.04/chef-server-core_14.04_x86_64.deb, Download button.

```
ubuntu@ip-172-31-10-101:~$ wget https://packages.chef.io/stable/ubuntu/14.04/chef-server-core_12.11.1-1_amd64.deb
```

Install the Chef server:

1.

```
ubuntu@ip-172.31.10.101:~$ sudo dpkg -i chef-server*
```

This will install the base Chef 12 system onto the server.

2. Once the installation is complete, we should run `chef-server-ctl reconfigure` command to start the Chef server services. It configures the components that make up the server to work together in our specific environment:
3.

```
ubuntu@ip-172.31.10.101:~$ sudo chef-server-ctl reconfigure
```
4. ...
5. Chef Server Reconfigured!

Once the installation is complete, you must call the command, which configures the components that make up the server to work together in your specific environment:

6. ubuntu@ip-172.31.10.101:~\$ sudo chef-server-ctl status
7. run: bookshelf: (pid 4486) 39s; run: log: (pid 4526) 39s
8. run: nginx: (pid 4269) 44s; run: log: (pid 4679) 35s
9. run: oc_bifrost: (pid 4199) 46s; run: log: (pid 4246) 45s
10. run: oc_id: (pid 4253) 45s; run: log: (pid 4258) 44s
11. run: opscode-erchef: (pid 4591) 36s; run: log: (pid 4580) 38s
12. run: opscode-expander: (pid 4367) 41s; run: log: (pid 4469) 40s
13. run: opscode-expander-reindexer: (pid 4435) 40s; run: log: (pid 4475) 40s
14. run: opscode-solr4: (pid 4308) 42s; run: log: (pid 4344) 42s
15. run: postgresql: (pid 4151) 46s; run: log: (pid 4174) 46s
16. run: rabbitmq: (pid 1685) 136s; run: log: (pid 4138) 47s
17. run: redis_lb: (pid 4086) 64s; run: log: (pid 4673) 35s
- 18.
19. ubuntu@ip-172.31.10.101:~\$ sudo chef-server-ctl test
- 20.

The Chef core server is now installed and started. The next steps is to configure it to allow us to log in.

Create a default user and organization

DEVOPS MATERIAL

The next step is to create a default user and organization for the chef-server.

Next, we need to create an admin user. This will be the username that will have access to make changes to the infrastructure components in the organization we will be creating.

We can do this using the **user-create** subcommand of the **chef-server-ctl** command. The command requires a number of fields to be passed in during the creation process.

We will create a user with the following information:

1. Username: admin
2. First Name: admin
3. Last Name: admin
4. Email: admin@testorg.com
5. Password: password
6. Filename: admin.pem

Also, we will create an organization with the following information:

1. Short Name: testorg
2. Long Name: testorg.com
3. Association User: admin
4. Filename: testorg.pem

1. In order to link workstations and nodes to the Chef server, an administrator and an organization need to be created with associated RSA private keys. From the home directory, create a **.chef** directory to store the keys:

2. `ubuntu@ip-172-31-10-101: ~$ mkdir .chef`

3. Let's create an administrator:

4. `ubuntu@ip-172-31-10-101: ~$ sudo chef-server-ctl user-create admin adminadmin admin@testorg.com password -f ~/.chef/admin.pem`

5.

6. We should now have a private key called `admin.pem` in `~/.chef/` directory.

7. Create an organization with the `org-create` subcommand:

8. `ubuntu@ip-172-31-10-101:~/chef$ sudo chef-server-ctl org-create testorg "testorg.com" --association_user admin -f ~/.chef/testorg.pem`

Now, we should have two .pem key files in `~/.chef/` directory:

```
ubuntu@ip-172-31-10-101:~/chef$ ls
```

```
admin.pem testorg.pem
```

We will need to connect to this server and download these keys to our workstation momentarily. For now though, our Chef server installation is complete.

Opscode Manage (GUI)

Let's install the GUI plugin for the Chef:

```
ubuntu@ip-172-31-10-101:~/chef$ sudo chef-server-ctl install opscode-manage
```

```
ubuntu@ip-172-31-10-101:~/chef$ sudo opscode-manage-ctl reconfigure
```

```
ubuntu@ip-172-31-10-101:~/chef$ sudo chef-server-ctl reconfigure
```



DEVOPS MATERIAL

To create a file in Microsoft Windows, be sure to add an escape character—\—before the backslashes in the paths:

```
file 'C:\\tmp\\something.txt' do
  rights :read, 'Everyone'
  rights :full_control, 'DOMAIN\\User'
  action :create
end
```

Remove a file

```
file '/tmp/something' do
  action :delete
end
Set file modes
file '/tmp/something' do
  mode '0755'
end
```

```
directory '/a/b/c' do
  owner 'admin'
  group 'admin'
  mode '0755'
  action :create
  recursive true
end
```

Linux Commands execution chef

```
bash 'install_something' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  wget http://www.example.com/tarball.tar.gz
  tar -zxf tarball.tar.gz
  cd tarball
  ./configure
  make
  make install
  EOH
End
```

Start a service

```
service 'example_service' do
  action :start
end
```

Start a service, enable it

```
service 'example_service' do
  supports :status => true, :restart => true, :reload => true
  action [ :enable, :start ]
end
```

Use a pattern

```
service 'samba' do
  pattern 'smbd'
  action [:enable, :start]
end
```

Use the :nothing common action

```
service 'memcached' do
  action :nothing
  supports :status => true, :start => true, :stop => true, :restart => true
end
```

Use the supports common attribute

```
service 'apache' do
  supports :restart => true, :reload => true
  action :enable
end
```

```
#####
Manage a service, depending on the node platform
```

```
service 'example_service' do
  case node['platform']
  when 'centos', 'redhat', 'fedora'
    service_name 'redhat_name'
  else
    service_name 'other_name'
  end
  supports :restart => true
  action [ :enable, :start ]
end
```

For example, installing multiple packages:

```
package ['package1', 'package2']
```

Upgrading multiple packages:

```
package ['package1', 'package2'] do
  action :upgrade
end
```

Removing multiple packages:

```
package ['package1', 'package2'] do
  action :remove
end
```



DEVOPS MATERIAL

end

```
user 'a user' do
  comment 'A random user'
  uid '1234'
  gid '1234'
  home '/home/random'
  shell '/bin/bash'
  password '$1$JJsvHslasdfjVEroftprNn4JHtDi'
end
```

Connect to git

```
git "/path/to/check/out/to" do
  repository "git://github.com/opscode/chef.git"
  reference "master"
  action :sync
end
```

Why do we need classes?

We don't want a huge `/etc/puppet/manifests/site.pp` file, and the files should be splitted into chunks of logically related code out into their own files, and then refer to those chunks by name when we need them.

"Classes are Puppet's way of separating out chunks of code, and modules are Puppet's way of organizing classes so that you can refer to them by name." - Learning Puppet - Modules and Classes

classes

1. Defining a class makes it available by name, but doesn't automatically evaluate the code inside it.

Before we can use a class, we must define it, which is done with the `class` keyword, a name, curly braces, and a block of code:

```
2. class my_class {  
3.     ... puppet code ...  
4. }
```

This manifest does nothing.

5. Declaring a class evaluates the code in the class, and applies all of its resources.

This one actually does something.

```
6. class my_class {  
7.     ... puppet code ...  
8. }  
9. include my_class
```

class names

Class names must start with a lowercase letter, and can contain lowercase letters, numbers, and underscores. Class names can also use a double colon (::) as a namespace separator.

Class & Variable Scope

Each class definition introduces a new variable scope. Any variables we assign inside the class won't be accessible by their short names outside the class; to get at them from elsewhere, we have to use the fully-qualified name, e.g. \$ntp::service_name, as shown in the example below (**modules/ntp/manifests/init.pp**):

```
classntp {  
    case $operatingsystem {  
        centos, redhat: {  
            $service_name = 'ntpd'  
            $conf_file   = 'ntp.conf.el'  
        }  
        debian, ubuntu: {  
            $service_name = 'ntp'  
            $conf_file   = 'ntp.conf.debian'  
        }  
    }  
    package { 'ntp':
```

```
ensure => installed,  
}  
  
file { 'ntp.conf':  
    path  => '/etc/ntp.conf',  
    ensure => file,  
    require => Package['ntp'],  
    source => "/root/examples/answers/${conf_file}"  
}  
  
service { 'ntp':  
    name    => $service_name,  
    ensure   => running,  
    enable   => true,  
    subscribe => File['ntp.conf'],  
}  
}
```

We can assign new, local values to variable names that were already used at top scope. For example, we could specify a new local value for \$var.

define vs declaration

Defining makes a class available, and **declaring** evaluates it. We can see that in action by trying to apply our manifest in the previous section:

```
ubuntu@ip-172-31-45-62:/etc/puppet$ sudo puppet apply modules/ntp/manifests/init.pp
```

```
Notice: Compiled catalog for ip-172-31-45-62.ec2.internal in environment production in 0.02 seconds
```

```
Notice: Finished catalog run in 0.02 seconds
```

We can see it does nothing, because we only defined the class.

To declare a class, we should use the **include** function with the class's name:

```
node 'puppet-agent' {  
  includentp  
  
  ...  
}
```

Run puppet:

```
ubuntu@puppet-agent: ~ $ sudo puppet agent --test
```

```
Info: Retrieving plugin
```

```
Info: Caching catalog for puppet-agent.ec2.internal
```

```
Info: Applying configuration version '1419831895'
```

```
Notice: /Stage[main]/Exec[Exec[Run a command]]/returns: executed successfully
```

```
Notice: Finished catalog run in 0.18 seconds
```

Modules

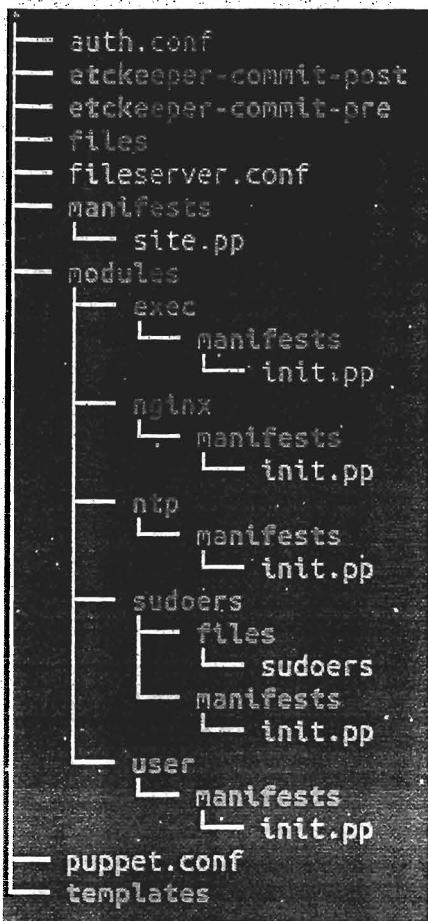
DEVOPS MATERIAL

Now we know how to define and declare classes. We could have done everything in a single manifest (`site.pp`), but we actually splitted up our manifests into an easier to understand structure, and used **manifests/site.pp**.

```
node 'puppet-agent' {  
    includentp  
    includenginx  
    include user  
    includesudoers  
    include exec  
  
    Exec {  
        path => ['/bin', '/usr/bin'],  
    }  
}
```

By stowing the implementation of a feature in a module, our main manifest can become much smaller, more readable, and policy-focused - we can tell at a glance what will be configured on our nodes, and if we need implementation details on something, we can delve into the module.

Module Structure



1. A module is a directory.
2. The module's name must be the name of the directory.
3. It contains a manifests directory, which can contain any number of .pp files.
4. The manifests directory should always contain an **init.pp** file.
1. This file must contain a single class definition. The class's name must be the same as the module's name.

Puppet Master and Client Installation steps

Note: run all these commands from root. If you are running from other users you need to use #sudo<command>

Change machine hostnames like bellow

DEVOPS MATERIAL

Server hostname change:

#view /etc/hostname (Don't copy # from this line. You need to take command line only not #)

Add "server.example.com" and close the file

After adding press escape, colon, wq!

:wq! (to save and quit)

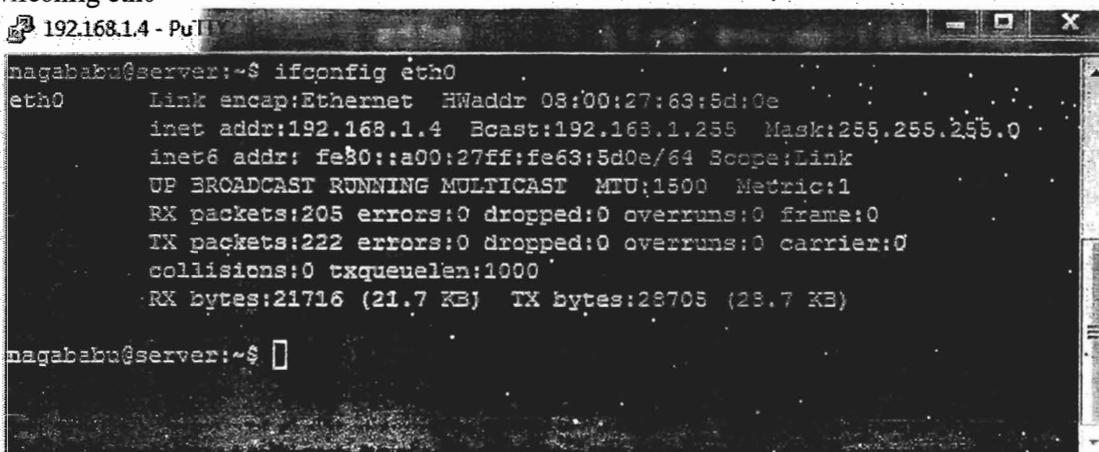
Do it on client machine also.

#view /etc/hostname

Add "client.example.com" and close the file

Get the eth0 IP information from both server and client using bellow command

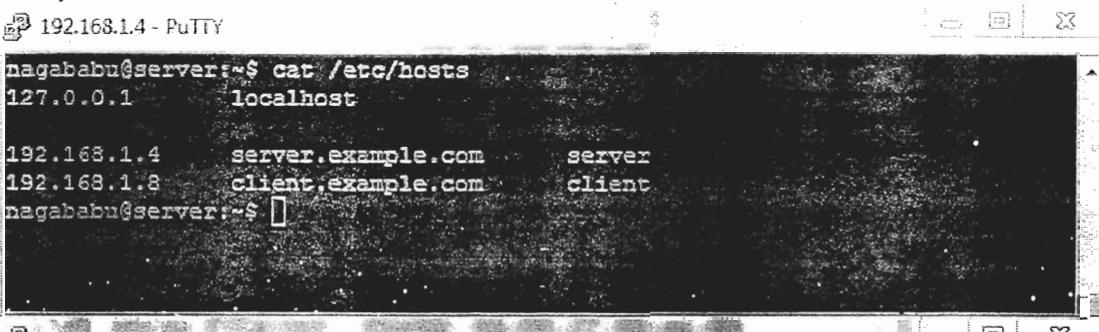
#ifconfig eth0



```
nagababu@server:~$ ifconfig eth0
eth0      Link encap:Ethernet HWaddr 08:00:27:63:5d:0e
          inet addr:192.168.1.4 Bcast:192.168.1.255 Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe63:5d0e/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
             RX packets:205 errors:0 dropped:0 overruns:0 frame:0
             TX packets:222 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:21716 (21.7 KB) TX bytes:28705 (28.7 KB)

nagababu@server:~$
```

Add Ip address and hostname in "/etc/hosts" file like below on both client and server machines.



```
nagababu@server:~$ cat /etc/hosts
127.0.0.1   localhost
192.168.1.4   server.example.com   server
192.168.1.8   client.example.com   client
nagababu@server:~$
```



```
nagababu@client:~$ cat /etc/hosts
127.0.0.1   localhost
192.168.1.4   server.example.com   server
192.168.1.8   client.example.com   client
nagababu@client:~$
```

Now we need to make sure that both server and client should sync same time.

Run bellow commands from root user

```
# ntpdate pool.ntp.org ; apt-get update && sudo apt-get -y install ntp ; service ntp restart
```

Puppet master installation

To install Puppet master we need latest repository and install the Puppet as follows:

```
# cd /tmp  
# wget https://apt.puppetlabs.com/puppetlabs-release-trusty.deb  
# dpkg -i puppetlabs-release-trusty.deb  
# apt-get update  
# apt-get install puppetmaster
```

Check the puppet version as:

```
# puppet -V
```

3.8.6

We have puppet version as 3.8.6. Now we need to lock the puppet version update as this will hamper the configurations while updating the puppet. It will be done by editing the file as follows:

```
# view /etc/apt/preferences.d/00-puppet.pref
```

```
# /etc/apt/preferences.d/00-puppet.pref  
Package: puppet puppet-common puppetmaster-passenger  
Pin: version 3.8*  
Pin-Priority: 501
```

Add the entries
in the newly
created above
file as:

It will not update the Puppet while running updates in the system.

Next we will change the configuration file as follows:

```
# view /etc/puppet/puppet.conf
```

192.168.1.4 - PuTTY

```
nagababu@server:~$ cat /etc/puppet/puppet.conf  
[main]  
logdir=/var/log/puppet  
vardir=/var/lib/puppet  
ssldir=/var/lib/puppet/ssl  
rundir=/var/run/puppet  
factpath=$vardir/lib/facter  
#templatedir=$confdir/templates  
  
[master]  
# These are needed when the puppetmaster is run by passenger  
# and can safely be removed if webrick is used.  
ssl_client_header = SSL_CLIENT_S_DN  
ssl_client_verify_header = SSL_CLIENT_VERIFY  
certname = puppet  
dns_alt_names = puppet,puppet.example.com,server.example.com  
nagababu@server:~$
```

Now Restart the puppetmaster service

```
#servicepuppetmaster restart
```

Now the puppet master server is ready.

Puppet client installation

In above steps we have already configured /etc/hosts file and installed ntp packages. Now we are going to install puppet client package and configure puppet.conf for client

```
#cd /tmp
```

```
#wget https://apt.puppetlabs.com/puppetlabs-release-trusty.deb
```

```
#dpkg -i puppetlabs-release-trusty.deb
```

```
#apt-get update
```

To install client package:

```
#apt-get install puppet
```

Check the puppet version as:

```
#puppet -V
```

3.8.6

Now we need to lock the puppet version using below steps

Create below file

```
#view /etc/apt/preferences.d/00-puppet.pref
```

Add the entries in the newly created file as:

```
# /etc/apt/preferences.d/00-puppet.pref
Package: puppet puppet-common puppetmaster-passenger
Pin: version 3.8*
Pin-Priority: 501
```

It will not
update the

Puppet while running updates in the system.

Next we will change the configuration file as follows:

```
#view /etc/puppet/puppet.conf
```

```
nagababu@client:~$
```

```
nagababu@client:~$ cat /etc/puppet/puppet.conf
[main]
logdir=/var/log/puppet
vardir=/var/lib/puppet
ssldir=/var/lib/puppet/ssl
rundir=/var/run/puppet
factpath=$vardir/lib/facter
$templatedir=$confdir/templates

#[master]
# These are needed when the puppetmaster is run by passenger
# and can safely be removed if webrick is used.
#$ssl_client_header = SSL_CLIENT_S_DN
#$ssl_client_verify_header = SSL_CLIENT_VERIFY

[agent]
server = server.example.com
nagababu@client:~$
```

Next we need to edit the file vi /etc/default/puppet and make changes from no to yes as shown below:

```
#view /etc/default/puppet
```

```
nagababu@client:~$ cat /etc/default/puppet
# Defaults for puppet - sourced by /etc/init.d/puppet

# Enable puppet agent service?
# Setting this to "yes" allows the puppet agent service to run.
# Setting this to "no" keeps the puppet agent service from running.
START=yes

# Startup options
DAEMON_OPTS=""
```

Now we are ready to start the puppet service, it will be done as follows:

```
#service puppet start
```

Now our client machine is ready to communicate with Puppet Master server.

Cert exchange from Puppet master to puppet client

After the successful configuration Puppet client Ubuntu Desktop will search Puppet master Server and ask for cert request before accepting any administrative instructions from Master puppet server.

To view such cert request run the command at Puppet Master Ubuntu server.

```
# puppet cert list
```

```
nagababu@client:~$ 
nagababu@server:~$ sudo puppet cert list
"client.example.com" (SHA256) 7F:65:64:FE:82:17:18:6C:10:BD:8D:2F:DF:AB:A8:08:8A:3A:E7:04:68:F9
4:4B:E9
nagababu@server:~$ 
```

It means that there is a machine named as client.example.com which came into existence and asking for cert request. Now the puppet master server must sign the cert requested from puppet client. It can be done as follows:

```
# puppet cert sign client.example.com
```

The output will be like this:

```
nagababu@server:~$ sudo puppet cert sign client.example.com
Notice: Signed certificate request for client.example.com
Notice: Removing file Puppet::SSL::CertificateRequest client.example.com at '/var/lib/puppet/ssl/ca/requests/client.example.com.pem'
nagababu@server:~$ 
```

It means the request from client machine is accepted at Puppet master machine. We can check that with command as well:

```
# puppet cert list --all
```

```
nagababu@server:~$ sudo puppet cert list -all
+ "client.example.com" (SHA256) 77:B5:04:85:72:8D:70:88:89:04:07:F9:41:42:4F:0B:4D:A7:82:10:25:2E
:66:36:F0:19:B4:46:EC:ED:F4:3D
+ "puppet" (SHA256) 65:4C:53:C5:03:06:25:63:4B:88:20:E3:C6:79:A0:4A:AD:F2:7F:90:D1:87
:12:CE:BD:68:DB:82:44:85:8D:61 (alt names: "DNS:puppet", "DNS:puppet.example.com", "DNS:server.ex
ample.com")
+ "server.example.com" (SHA256) 22:20:A7:18:CD:B0:D8:C2:58:C0:1C:1F:72:D7:CD:E9:54:17:23:09:62:D4
:3F:23:55:00:D0:95:49:B1:14:86 (alt names: "DNS:puppet", "DNS:puppet.example.com", "DNS:server.ex
ample.com")
nagababu@server:~$ 
```

The above + sign in the output shows successful certificate signing at Puppet master Ubuntu Server.

Similarly we can add any number of clients with Puppet master Ubuntu Server and sign the cert requests from the clients.

Note: Don't run this now. Only run when you need to remove client certification from master

If for any administrative requirement you wish to revoke the certs from the Puppet master Ubuntu Server we can run:

```
# puppet cert clean client.example.com
```

In Puppet, the combined configuration to be applied to a host is called a *catalog*, and the process of applying it is called a *run*.

The Puppet **client** software is called the **agent**. Puppet calls the definition of the host itself **anode**. The Puppet **server** is called the **master**.

Packages, Files, and Services

The most common types of resources we'll manage with Puppet are packages, files, and services.

Packages

With Puppet, to install nginx, we give a resource declaration like this:

```
# /etc/puppet/manifests/site.pp

node 'puppet-agent' {

  package { 'nginx':
    ensure => installed,
  }
}
```

DEVOPS MATERIAL

Then, Puppet will take the necessary actions by running **apt-get** behind the scenes.

Let's look at the code in detail:

```
node 'puppet-agent' {  
    ...  
}  
}
```

Note that the **node** keyword introduces a node declaration, a list of resources that are to be applied only to node 'puppet-agent'.

```
package { 'nginx':  
    ensure => installed,  
}
```

In our case, there is one type of resource, package. As with the file resource, the resource declaration consists of the following:

1. The type of resource: package
2. The name of the instance: nginx
3. A list of attributes

Each resource type has a different list of attributes that we can control. A useful attribute for package resources is **ensure**. We use this attribute to install (or sometimes remove) packages.

4. **ensure => installed,**

Run Puppet:

```
root@puppet-agent:~# puppet agent --test
```

Info: Retrieving plugin

Info: Caching catalog for puppet-agent.ec2.internal

Info: Applying configuration version '1419668689'

Notice: /Stage[main]/Main/Node[puppet-agent]/Package[nginx]/ensure: ensure changed 'purged' to 'present'

Notice: Finished catalog run in 4.08 seconds

When we apply the manifest, Puppet checks whether the nginx package is installed. If this is the first time we've applied the manifest, the package probably won't be present, so Puppet prints a message telling us that the package is being installed as shown above

Once the resource has been created the first time, subsequent Puppet runs will do nothing because the state of the system already matches the manifest:

```
root@puppet-agent:~# puppet agent --test
```

Info: Retrieving plugin

Info: Caching catalog for puppet-agent.ec2.internal

Info: Applying configuration version '1419668689'

Notice: Finished catalog run in 0.03 seconds

Removing Package

Sometimes we need to make sure a package is removed entirely from a machine:

```
package { 'nginx':
```

```
  ensure => absent,
```

{

Using **ensure => absent** will remove the package if it's installed.

Updating Package

Another value that ensure can take on a package resource is latest. This will cause Puppet to check which version of the package is available in the repository (if we're using Ubuntu, this includes any additional APT sources that we may have configured, such as the Puppet Labs repo). If it is newer than the installed version, Puppet will upgrade the package to the latest version:

```
package { 'nginx':  
  ensure => latest,  
}
```

Modules

To make our Puppet manifests more readable and maintainable, it's a good idea to arrange them into modules. A Puppet module is a way of grouping related resources. In our example, we're going to make an nginx module that will contain all Puppet code relating to nginx.

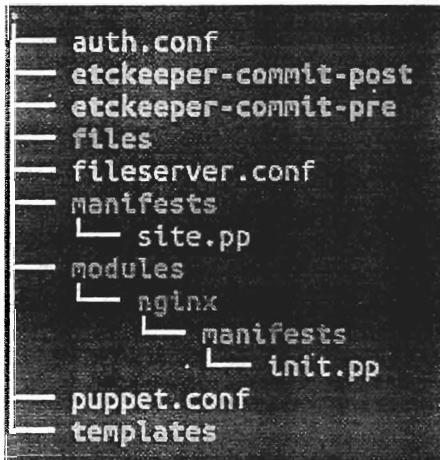
Create the file **modules/nginx/manifests/init.pp** with the following contents:

```
# /etc/puppet/modules/nginx/manifests/init.pp
```

```
classnginx {  
    package { 'nginx':  
        ensure => installed,  
    }  
}
```

Edit the **manifests/sites.pp** file as follows:

```
# /etc/puppet/manifests/site.pp  
  
node 'puppet-agent' {  
    includenginx  
}  
  
}
```



Run puppet:

```
ubuntu@puppet-agent:~$ sudo puppet agent --test
```

```
Info: Retrieving plugin
```

Info: Caching catalog for puppet-agent.ec2.internal

Info: Applying configuration version '1419714937'

Notice: Finished catalog run in 0.04 seconds

Note that we've modified our puppet code for node from:

```
# /etc/puppet/manifests/site.pp

node 'puppet-agent' {

  package { 'nginx':
```

```
    ensure => installed,
```

```
}
```

```
}
```

to:

```
# /etc/puppet/manifests/site.pp

node 'puppet-agent' {

  includenginx
```

```
}
```

As we can see the nginx resource has been replaced by the line **include nginx**. We're telling Puppet, "Look for a class called nginx and include all the resources on this node."

The **class** keyword declares a group of resources (here, the package resource for nginx) identified by the name nginx. We can then use the **include** keyword elsewhere to include all the resources in the class at once.

But why we do this?

That's because this enables us to include the nginx class on many nodes without repeating the same resource declarations over and over as shown below:

```
node 'node1' {
```

```
    includenginx
```

```
}
```

```
node 'node2' {
```

```
    includenginx
```

```
}
```

```
node 'node3' {
```

```
    includenginx
```

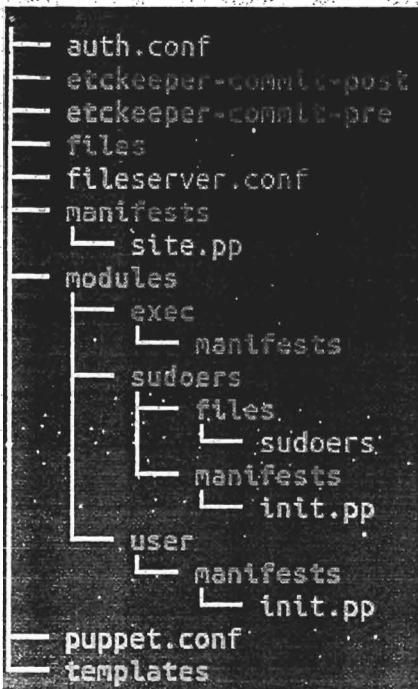
```
}
```

Also, by grouping resources into classes and modules helps us organize our code, and it's easy to read and maintain.

exec resources

We want Puppet to run a certain command directly using an **exec resource**. This is a very flexible and powerful resource, and we can use it to implement almost anything in Puppet.

```
ubuntu@ip-172-31-45-62:/etc/puppet$ sudo mkdir -p modules/exec/manifests
```



```

# modules/exec/manifests/init.pp

class exec {

  exec { 'Run a command':
    command => '/bin/echo `bin/date` >/tmp/output.txt',
  }
}
  
```

The line

```
exec { 'Run a command':
```

declares an **exec** resource with the name 'Run a command'. The name can be anything; it's not otherwise used by Puppet, except that like all resource names it can't be the same as another instance of the same resource type.

echo and **date**, are specified with their full path. This is because Puppet wants to be sure exactly which command we mean. When Puppet runs, it applies the **exec** resource by running the command:

```
command => '/bin/echo `date` >/tmp/output.txt',
```

This command will write the following text to **/tmp/output.txt**:

With the node definition:

```
# manifests/site.pp

node 'puppet-agent' {

  include user

  includesudoers

  include exec

}
```

Run Puppet:

```
ubuntu@puppet-agent:/etc/puppet$ sudo puppet agent --test

Info: Retrieving plugin

Info: Caching catalog for puppet-agent.ec2.internal

Info: Applying configuration version '1419811309'

Notice: /Stage[main]/Exec[Run a command]/returns: executed successfully

Notice: Finished catalog run in 0.11 seconds
```

Check the output produced:

```
ubuntu@puppet-agent:/etc/puppet$ cat /tmp/output.txt
```

```
Mon Dec 29 00:01:59 UTC 2014
```

Command search paths - I

As in the previous section, Puppet requires us to specify the full path to any command referenced in an exec resource. However, we can provide a list of paths for Puppet to search for commands, using the **path** attribute. For example:

```
class exec {  
  
  exec { 'Run a command':  
  
    #command => '/bin/echo `/bin/date` >/tmp/output.txt',  
  
    command => 'echo `date` >/tmp/output.txt',  
  
    path => ['/bin', '/usr/bin'],  
  
  }  
  
}
```

Now when Puppet sees a command name, it will search the directories you specify looking for the matching commands.

Command search paths - II

If we want to specify a set of default search paths for all exec resources, we can put this in our **manifests/site.pp** file:

```
Exec {  
  path => ['/bin', '/usr/bin'],  
}
```

Note the capital 'E' for **Exec**. This means "make this the default for all exec resources." Then we can use unqualified commands without an explicit path attribute. So, the **modules/exec/manifests/init.pp** now looks like this:

```
class exec {  
  exec { 'Run a command':  
    command => 'echo `date` >/tmp/output.txt',  
  }  
}
```

Check a request on master

```
$ sudo puppet cert list -all  
  
"ip-172-31-53-211.ec2.internal" (SHA256)  
13:2E:06:C1:D8:51:E6:34:61:85:3C:B5:82:99:F4:FB:96:04:B3:CB:6F:05:67:AD:D6:93:E7:C7:E4:2  
E:BB:A0  
  
+ "ip-172-31-53-212.ec2.internal" (SHA256)  
07:C0:0E:C0:B0:30:51:89:02:AE:1B:AF:3C:C5:40:1D:5A:E5:DF:BE:00:18:23:22:2E:FA:8B:6A:7B  
:E3:89:60 (alt names: "DNS:ip-172-31-53-212.ec2.internal", "DNS:puppet",  
"DNS:puppet.ec2.internal")
```

Sign a request

As mentioned in previous sections, in an agent/master deployment, an admin must approve a certificate request for each agent node before that node can fetch configurations. Agent nodes will request certificates the first time they attempt to run.

To sign a certificate request, use the `puppet cert sign` command, with the hostname of the certificate we want to sign. For example, to sign '`puppet_agent.example.com`', we would use the following command:

```
$ sudo puppet cert sign ip-172-31-53-211.ec2.internal
```

```
Notice: Signed certificate request for ip-172-31-53-211.ec2.internal
```

```
Notice: Removing file Puppet::SSL::CertificateRequest ip-172-31-53-211.ec2.internal at
'/var/lib/puppet/ssl/ca/requests/ip-172-31-53-211.ec2.internal.pem'
```

Once Puppet agent has been signed, the Puppet master can now communicate and control the node that the signed certificate belongs to.

Puppet Forge modules

The modules in the Puppet Forge repository is publically-available modules, and they can be useful when we develop our own infrastructure. The Puppet Forge modules can be quickly installed with built-in `puppet module` command.

Since Apache and MySQL are available via Puppet Forge, we will demonstrate how they can be used to help us set up our lamp stack.

Install Apache and MySQL modules

On our Puppet master, install the **puppetlabs-apache** module:

```
$ sudo puppet module install puppetlabs-apache
```

We will see the following output, which indicates the modules installed correctly:

```
Notice: Preparing to install into /etc/puppet/modules ...
```

```
Notice: Downloading from https://forgeapi.puppetlabs.com ...
```

```
Notice: Installing -- do not interrupt ...
```

```
/etc/puppet/modules
```

```
--- puppetlabs-apache (v1.1.1)
```

```
    --- puppetlabs-concat (v1.1.2)
```

```
    --- puppetlabs-stdlib (v4.3.2)
```

Also, we need to install the **puppetlabs-mysql** module:

```
$ sudo puppet module install puppetlabs-mysql
```

```
Notice: Preparing to install into /etc/puppet/modules ...
```

```
Notice: Downloading from https://forgeapi.puppetlabs.com ...
```

```
Notice: Installing -- do not interrupt ...
```

```
/etc/puppet/modules
```

```
--- puppetlabs-mysql (v2.3.1)
```

```
    --- puppetlabs-stdlib (v4.3.2)
```

Now, the apache and mysql modules are available.

Main manifest

Now let's edit our main manifest so it uses the new modules to install our lamp stack. On the Puppet master, edit the main manifest (`/etc/puppet/manifests/site.pp`):

```
node default { }

node 'ip-172-31-53-211' {

    class { 'apache':          # use the "apache" module
           default_vhost => false,      # don't use the default vhost
           default_mods => false,      # don't load default mods
           mpm_module => 'prefork',    # use the "prefork" mpm_module
           }

    include apache::mod::php      # include mod php

    apache::vhost{ 'example.com': # create a vhost called "example.com"
        port   => '80',          # use port 80
        docroot => '/var/www/html', # set the docroot to the /var/www/html
    }

    class { 'mysql::server':
```

```
root_password => 'password',  
}  
  
file { 'info.php':  
    ensure => file,  
    path => '/var/www/html/info.php',  
    require => Class['apache'],  
    source => 'puppet:///modules/apache/info.php',  
}  
}  
}
```

The apache module can be passed parameters that override the default behavior of the module. We are passing in some basic settings that disable the default virtual host that the module creates, and make sure we create a virtual host that can use php.

Using the MySQL module is similar to using the Apache module. We will keep it simple since we are not actually using the database this time.

The file resource ensures **info.php** gets copied to the proper location. This time, we will use the source parameter to specify a file to copy. This file resource declaration is slightly different from the one we used in our previous example. The main difference is that we are specifying the source parameter instead of the content parameter. Source tells puppet to copy a file over, instead of simply specifying the file's contents. The specified **source,puppet:///modules/apache/info.php** gets interpreted by Puppet into **/etc/puppet/modules/apache/files/info.php**, so we must create the source file in order for this resource declaration to work properly.

Let's create the **info.php** file with the following command:

```
sudosh -c 'echo "<?php phpinfo(); ?>" > /etc/puppet/modules/apache/files/info.php'
```

Pulling the configuration

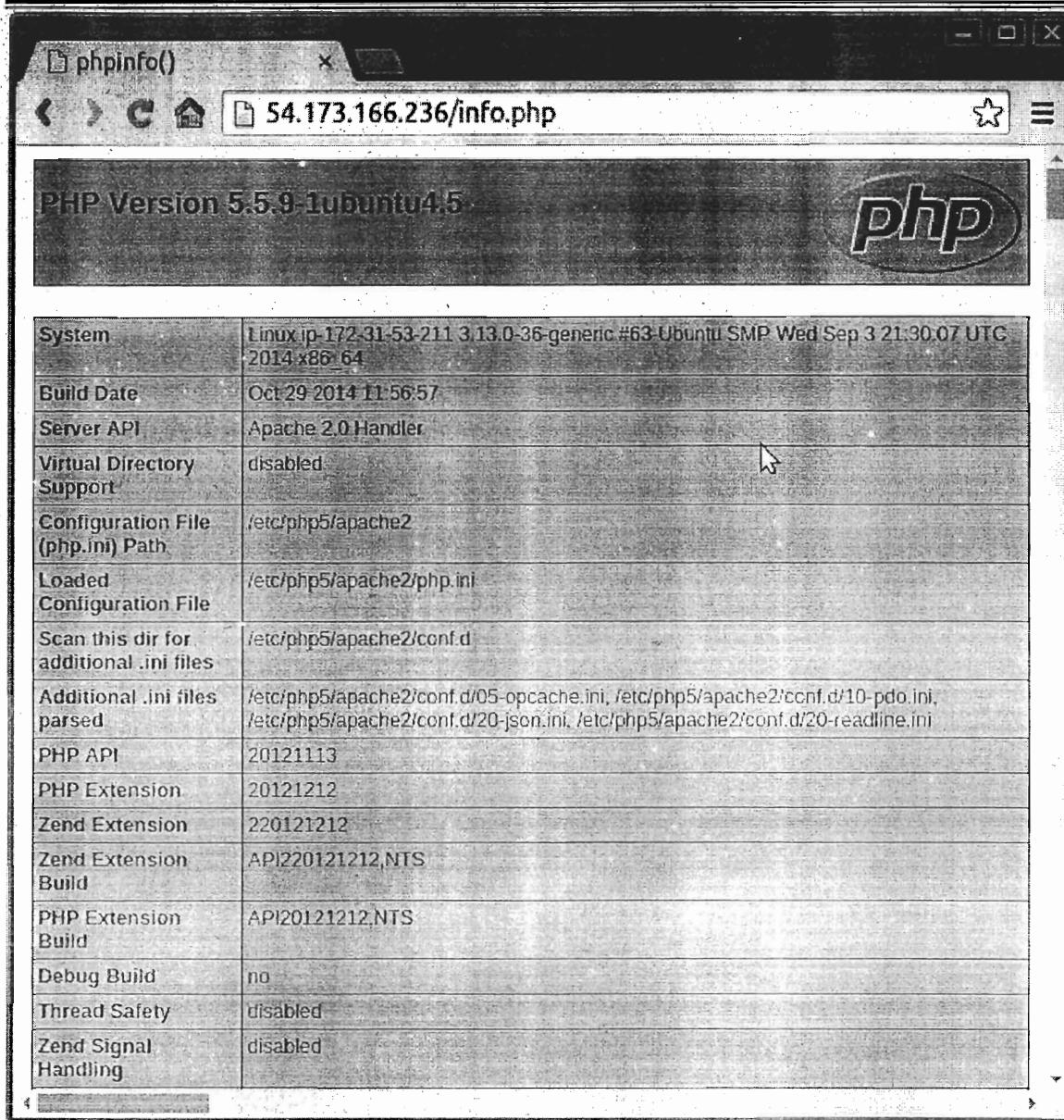
The next time our 'ip-172-31-53-211' Puppet agent node pulls its configuration from the master ('ip-172-31-53-212'), it will evaluate the main manifest and apply the module that specifies a lamp stack setup.

But if we want to try it out immediately, run the following command on the 'ip-172-31-53-211' Puppet agent node:

```
$ sudo puppet agent --test
```

Once it completes, we can see that a basic lamp stack is set up:

Master node:



phpinfo()

54.173.166.236/info.php

PHP Version 5.5.9-Lubuntu4.5

System: Linux ip-172-31-53-211 3.13.0-36-generic #63-Ubuntu SMP Wed Sep 3 21:30:07 UTC 2014 x86_64

Build Date: Oct 29 2014 11:56:57

Server API: Apache 2.0 Handler

Virtual Directory Support: disabled

Configuration File (php.ini) Path: /etc/php5/apache2

Loaded Configuration File: /etc/php5/apache2/php.ini

Scan this dir for additional .ini files: /etc/php5/apache2/conf.d

Additional .ini files parsed: /etc/php5/apache2/conf.d/05-opcache.ini, /etc/php5/apache2/conf.d/10-pdo.ini, /etc/php5/apache2/conf.d/20-json.ini, /etc/php5/apache2/conf.d/20-readline.ini

PHP API: 20121113

PHP Extension: 20121212

Zend Extension: 220121212

Zend Extension Build: API20121212,NTS

PHP Extension Build: API20121212,NTS

Debug Build: no

Thread Safety: disabled

Zend Signal Handling: disabled

Why do we need classes?

We don't want a huge `/etc/puppet/manifests/site.pp` file, and the files should be splitted into chunks of logically related code out into their own files, and then refer to those chunks by name when we need them.

"Classes are Puppet's way of separating out chunks of code, and modules are Puppet's way of organizing classes so that you can refer to them by name." - Learning Puppet - Modules and Classes

classes

1. Defining a class makes it available by name, but doesn't automatically evaluate the code inside it.

Before we can use a class, we must define it, which is done with the `class` keyword, a name, curly braces, and a block of code:

2. `class my_class {`
3. `... puppet code ...`
4. `}`

This manifest does nothing.

5. Declaring a class evaluates the code in the class, and applies all of its resources.
This one actually does something.

6. `class my_class {`
7. `... puppet code ...`
8. `}`
9. `include my_class`

class names

Class names must start with a lowercase letter, and can contain lowercase letters, numbers, and underscores. Class names can also use a double colon (::) as a namespace separator.

Class & Variable Scope

Each class definition introduces a new variable scope. Any variables we assign inside the class won't be accessible by their short names outside the class; to get at them from elsewhere, we have to use the fully-qualified name, e.g. \$ntp::service_name, as shown in the example below (**modules/ntp/manifests/init.pp**):

```
classntp {  
    case $operatingsystem {  
        centos, redhat: {  
            $service_name = 'ntpd'  
            $conf_file   = 'ntp.conf.el'  
        }  
        debian, ubuntu: {  
            $service_name = 'ntp'  
            $conf_file   = 'ntp.conf.debian'  
        }  
    }  
  
    package { 'ntp':  
        ensure => installed,  
    }
```

```
}

file { 'ntp.conf':
  path  => '/etc/ntp.conf',
  ensure => file,
  require => Package['ntp'],
  source => "/root/examples/answers/${conf_file}"
}

service { 'ntp':
  name    => $service_name,
  ensure  => running,
  enable   => true,
  subscribe => File['ntp.conf'],
}

}
```

We can assign new, local values to variable names that were already used at top scope. For example, we could specify a new local value for \$var.

define vs declaration

Defining makes a class available, and **declaring** evaluates it. We can see that in action by trying to apply our manifest in the previous section:

```
ubuntu@ip-172-31-45-62:/etc/puppet$ sudo puppet apply modules/ntp/manifests/init.pp
```

Notice: Compiled catalog for ip-172-31-45-62.ec2.internal in environment production in 0.02 seconds

Notice: Finished catalog run in 0.02 seconds

We can see it does nothing, because we only defined the class.

To declare a class, we should use the **include** function with the class's name:

```
node 'puppet-agent' {
  includentp
  ...
}
```

Run puppet:

```
ubuntu@puppet-agent:~$ sudo puppet agent --test
```

Info: Retrieving plugin

Info: Caching catalog for puppet-agent.ec2.internal

Info: Applying configuration version '1419831895'

Notice: /Stage/main/Exec[Exec[Run a command]]/returns: executed successfully

Notice: Finished catalog run in 0.18 seconds

Modules

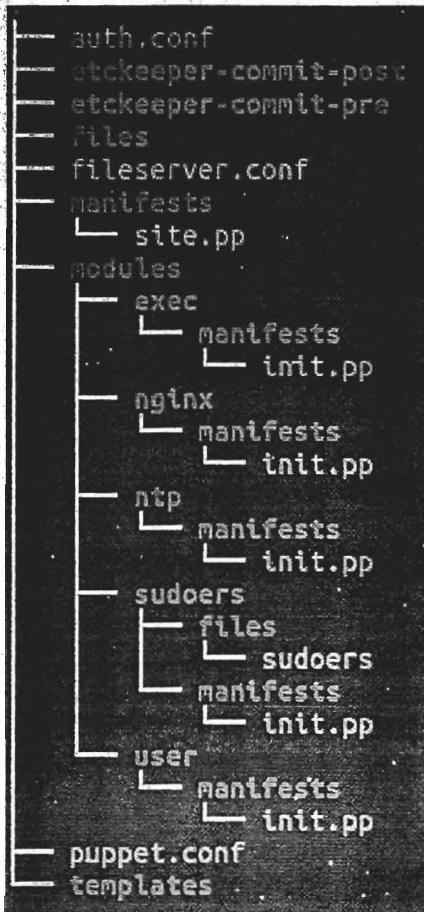
DEVOPS MATERIAL

Now we know how to define and declare classes. We could have done everything in a single manifest (`site.pp`), but we actually splitted up our manifests into an easier to understand structure, and used `manifests/site.pp`.

```
node 'puppet-agent' {  
    includentp  
    includenginx  
    include user  
    includesudoers  
    include exec  
  
    Exec {  
        path => ['/bin', '/usr/bin'],  
    }  
}
```

By stowing the implementation of a feature in a module, our main manifest can become much smaller, more readable, and policy-focused - we can tell at a glance what will be configured on our nodes, and if we need implementation details on something, we can delve into the module.

Module Structure



1. A module is a directory.
2. The module's name must be the name of the directory.
3. It contains a manifests directory, which can contain any number of .pp files.
4. The manifests directory should always contain an **init.pp** file.
 1. This file must contain a single class definition. The class's name must be the same as the module's name.

Services

To manage services, use the service resource type. The ensure attribute controls whether or not the service should be running. To specify that the service should be running, use

ensure => running

To specify that it should be stopped, use

ensure => stopped

The enable attribute controls whether or not a service is started at boot time. To start the service at boot time, use

enable => true

If we don't want it to start on boot, use

enable => false

Resource dependencies

We can specify a dependency between resources using the require attribute:

require => Package['nginx']

If resource 'resB' requires resource 'resA', then Puppet will make sure the resources are applied in the right order.

Files

We can have Puppet deploy a copy of a file using the source attribute:

```
file { '/etc/nginx/sites-enabled/default':  
    source => 'puppet:///modules/nginx/myfile',  
}
```

File resources can trigger a service to be restarted using the notify attribute. This is useful for configuration files, for which changes often don't take effect until the relevant service is restarted:

```
notify => Service['nginx'],
```

Services

The following code declares a resource of type service:

```
classnginx {  
    package { 'apache2.2-common':  
        ensure => absent,  
    }  
    package { 'nginx':  
        ensure => installed,  
        require => Package['apache2.2-common'],  
    }  
}
```

```
}
```

```
service { 'nginx':
```

```
    ensure => running,
```

```
    require => Package['nginx'],
```

```
}
```

```
}
```

The first part of the code:

```
package { 'apache2.2-common':
```

```
    ensure => absent,
```

```
}
```

On Ubuntu, the default setup includes the Apache web server, which would conflict with nginx if we tried to run it at the same time. So by specifying `ensure => absent`, we remove the Apache package.

The middle section declares the nginx package:

```
package { 'nginx':
```

```
    ensure => installed,
```

```
    require => Package['apache2.2-common'],
```

```
}
```

The require attribute tells Puppet that this resource depends on another resource, which must be applied first. In this case, we want the removal of **Apache** to be applied before the installation of **nginx**.

In the last part, we declare the nginx service:

```
service { 'nginx':  
    ensure => running,  
    require => Package['nginx'],  
}
```

Service resources manage daemons, or background processes, on the server.

The **ensure** attribute tells Puppet what state the service should be in:

```
ensure => running,
```

Files

Though nginx is installed and running, but it's not serving a website yet. To do that, we have to have Puppet install a config file on the server to define an nginx **virtual host**. This will tell nginx how to respond to requests for the 'bogo' website.

We'll create a simple website for nginx to serve:

1. Create the directory `/var/www/bogo`:
2. `ubuntu@ip-172-31-45-62:~$ sudo mkdir -p /var/www/bogo`
3. Add an HTML file:
4. `ubuntu@ip-172-31-45-62:~$ sudo su -c 'echo "sample site" >/var/www/bogo/index.html'`
5. Now we want to create the virtual host file for Puppet to deploy.
Create the directory **modules/nginx/files**:

6. ubuntu@ip-172-31-45-62:~\$ sudo mkdir -p modules/nginx/files

7. Create the file **modules/nginx/files/bogo.conf** with the following contents:

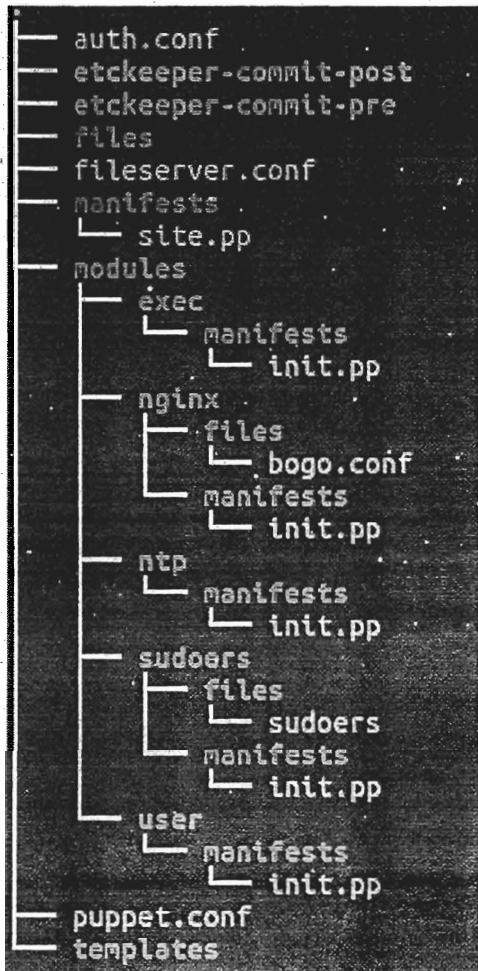
```
8.       server {  
9.           listen 80;  
10.          root /var/www/bogo;  
11.          server_name bogo.com;  
12.      }
```

13. Edit the file **modules/nginx/manifests/init.pp** so it looks like this:

```
14.       class nginx {  
15.          package { 'nginx':  
16.              ensure => installed,  
17.          }  
18.  
19.          service { 'nginx':  
20.              ensure => running,  
21.              require => Package['nginx'],  
22.          }  
23.  
24.          file { '/etc/nginx/sites-enabled/default':  
25.              source => 'puppet:///modules/nginx/bogo.conf',  
26.              notify => Service['nginx'],  
27.          }
```

28.

}



```
file { '/etc/nginx/sites-enabled/default':
```

This line declares a file resource with the path /etc/nginx/sites-enabled/default.

```
source => 'puppet:///modules/nginx/bogo.conf',
```

source is a file attribute that tells Puppet where to find a copy of the file:

```
puppet:///modules/nginx/bogo.conf
```

This looks like a URL, but it tells Puppet to look in the **modules/nginx/files** directory for a file named cat-pictures.conf.

notify is an attribute that tells Service['nginx'] to restart whenever there is change.

```
notify => Service['nginx'],
```

29. Run puppet and make sure everything worked properly, request the website:

30. `ubuntu@puppet-agent:~$ sudo puppet agent --test`

31. Info: Retrieving plugin

32. Info: Caching catalog for puppet-agent.ec2.internal

33. Info: Applying configuration version '1419914163'

34. Notice: /Stage/main/Exec/Exec[Run a command]/returns: executed successfully

35. Notice: Finished catalog run in 0.29 seconds

36.

37. `ubuntu@puppet-agent:~$ netstat -antlp`

38. (No info could be read for "-p": geteuid()=1000 but you should be root.)

39. Active Internet connections (servers and established)

40.	name	Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program
41.		tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN	-
42.		tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	-
43.		tcp	0	0	172.31.43.38:22	64.71.28.178:37040	ESTABLISHED	-
44.		tcp6	0	0	::22	::*	LISTEN	-

```
45.  
46.      ubuntu@puppet-agent:~$ curl localhost  
47.      <html>  
48.          <head><title>404 Not Found</title></head>  
49.          <body bgcolor="white">  
50.              <center><h1>404 Not Found</h1></center>  
51.              <hr><center>nginx/1.4.6 (Ubuntu)</center>  
52.          </body>  
53.      </html>
```

package-file-service pattern

The following covers most services:

```
class NAME {  
    package { NAME:  
        ensure => installed,  
    }  
  
    service { NAME:  
        ensure => running,  
        require => Package[NAME],  
    }  
}
```

```
file { '/etc/NAME.conf':  
    source => 'puppet:///modules/NAME/NAME.conf',  
    notify => Service[NAME],  
}  
}  
}
```

1. The service NAME should be running
 2. Before the service NAME is started, the package NAME should be installed
 3. Before the service NAME is started, the file /etc/NAME.conf should be present
(remember that "A notifies B" implies "B requires A")
 4. If the file /etc/NAME.conf changes, restart the service NAME
-
-

templates

In the chapter, Puppet packages, services, and files II with nginx, we had Puppet deploy an nginx virtual host file for the 'SAMPLE' application where we simply used a file resource with the **SAMPLE.conf** file distributed from Puppet master to our EC2 agent node.

In this chapter, we'll learn how to template configuration files out with Puppet, filling in variables with the managed node's facts.

Often we may want to maintain configuration files for applications that are different between servers. If we have a couple of configurations, it's not touch to maintain multiple files, but what if we have a very large number of differing configurations? We can manage this situation by writing ERB templates and populating the templates with node-specific information. This can be done in Puppet with the **template()** function:

Puppet supports templates written in the **ERB** templating and it can be used to specify the contents of files.

Now we want to manage many different websites, it would quickly become tedious to supply an almost identical virtual host file for each site, altering only the name and domain of the site.

The best way of doing this is to give Puppet master a template file into which it could just insert these variables for each different site. The **template()** function serves just this purpose. Anywhere we have multiple files that differ only slightly, or files that need to contain dynamic information, we just use a template.

template evaluation

Templates are evaluated via a simple function as shown in the following example:

```
$value = template("my_module/my_template.erb")
```

Template files should be stored in the **templates** directory of a Puppet module, which allows the template function to locate them with the simplified path format shown above. For example, the file referenced by **template("my_module/my_template.erb")** would be found on disk at **/etc/puppet/modules/my_module/templates/my_template.erb**.

Templates are always evaluated **by the parser, not by the client**. This means that if we're using a puppet master server, then the templates only need to be on the server, and we never need to download them to the client. The client sees no difference between using a template and specifying all of the text of the file as a string. - Docs: Using Puppet Templates

conf file - deploying a virtual host

Suppose now we want to build three new sites: SAMPLE1.com, SAMPLE2.com, and SAMPLE3.com. To prepare for this, we need to change the Puppet config for SAMPLE.com to use a template so that we can later use the same template for the new sites.

1. Here is our new **modules/nginx/manifests/init.pp** file:

```
2. class nginx {  
3.     package { 'nginx':  
4.         ensure => installed,  
5.     }  
6.  
7.     service { 'nginx':  
8.         ensure => running,  
9.         enable => true,  
10.        require => Package['nginx'],  
11.    }  
12.  
13.    file { '/etc/nginx/sites-enabled/default':  
14.        ensure => 'absent',  
15.    }  
16. }
```

Since we previously used the file **/etc/nginx/sites-enabled/ default** as the virtual host for SAMPLE.com (Puppet packages, services, and files II with nginx), we need to remove that now:

```
file { '/etc/nginx/sites-enabled/default':
  ensure => absent,
}
```

17. We want to reate a new templates directory in the nginx module:

```
18. ubuntu@ip-172-31-45-62:/etc/puppet$ sudo mkdir -p modules/nginx/templates
```

19. Create the file **modules/nginx/templates/vhost.conf.erb** with the following contents. The template file is for the virtual host definition:

```
20. server {
21.   listen 80;
22.   root /var/www/<%=@site_name%>;
23.   server_name<%=@site_domain%>;
24. }
```

The <%=%> signs mark where parameters will go; we will supply **site_name** and **site_domain** later, when we use the template. Puppet will replace <%=@site_name%> with the value of the **site_name** variable.

25. Then in the **site.pp** file, we include the nginx module on the node:

```
26. node 'puppet-agent' {
27.   include nginx
28.   $site_name = 'SAMPLE'
29.   $site_domain = 'SAMPLE.com'
30.   file { '/etc/nginx/sites-enabled/SAMPLE.conf':
31.     content => template('nginx/vhost.conf.erb'),
```

```
32.      notify => Service['nginx'],
33.    }
34.    ...
35.  }
```

Note that before using the template, we need to set values for the variables `site_name` and `site_domain`:

```
$site_name = 'SAMPLE'  
$site_domain = 'SAMPLE.com'
```

Also note that when we refer to these variables in Puppet code, we use a `$` prefix (`$site_name`), but in the template it's an `@` prefix (`@site_name`). This is because in templates we're actually writing Ruby, not Puppet!

Now we can use the template to generate the nginx virtual host file:

```
file { '/etc/nginx/sites-enabled/SAMPLE.conf':
  content => template('nginx/vhost.conf.erb'),
  notify => Service['nginx'],
}
```

This looks just like any other file resource, with a content attribute. Though we can give the contents of the file as a literal string:

```
content => "Hello, SAMPLEtoSAMPLE\n",
```

However, here we call the template function:

```
content => template('nginx/vhost.conf.erb'),
```

The argument to template tells Puppet where to find the template file. The path

```
nginx/vhost.conf.erb
```

Translates to

```
modules/nginx/templates/vhost.conf.erb
```

36. Run puppet:
37. ubuntu@ip-172-31-45-62:/etc/puppet\$ sudo puppet apply modules/nginx/manifests/init.pp
38. Notice: Compiled catalog for ip-172-31-45-62.ec2.internal in environment production in 0.02 seconds
39. Notice: Finished catalog run in 0.02 seconds
40. Check the resulting virtual host file on agent node:

```
41. ubuntu@puppet-agent:~$ cat /etc/nginx/sites-enabled/SAMPLE.conf
42. server {
43.   listen 80;
44.   root /var/www/SAMPLE;
45.   server_name SAMPLE.com;
46. }
```

Puppet now evaluated the template, inserted the values of any variables referenced in <%=%> signs in **modules/nginx/templates/vhost.conf.erb**, and generated the final output as we can see in the **SAMPLE.conf**.

garethr/docker from Puppet forge

Let's install garethr/docker from Puppet forge. It's a module for installing and managing docker.

```
$ sudo puppet module install garethr-docker

Notice: Preparing to install into /etc/puppet/modules ...

Notice: Downloading from https://forgeapi.puppetlabs.com ...

Notice: Installing -- do not interrupt ...

/etc/puppet/modules

|-- garethr-docker (v4.1.1)
   |-- puppetlabs-apt (v2.1.1)
   |-- puppetlabs-stdlib (v4.9.0)
   |-- stahnma-epel (v1.1.1)
```

docker_example.pp

The /etc/puppet/manifests/docker_example.pp should look like as shown below:

```
include 'docker'

docker::image { 'ubuntu':
  image_tag => 'trusty',
```

}

The module includes a single class:

```
include 'docker'
```

By default, this sets up the docker hosted repository if necessary for our OS, and installs the docker package and on Ubuntu, any required Kernel extensions.

```
docker::image { 'ubuntu':  
    image_tag => 'trusty'  
}
```

The image tag is equivalent to running **docker pull -t="trusty" ubuntu**. Note that the image will only install if an image of that name does not already exist.

manifest apply

Let's apply the puppet manifest (**/etc/puppet/manifests/docker_example.pp**) in order to get docker installed on our puppet master:

```
$ sudo puppet apply site.pp
```

```
...
```

```
Notice: Finished catalog run in 46.40 seconds
```

Check if docker is installed

Let's check if docker has been installed:

```
$ sudo docker version
```

Client version: 1.7.1

Client API version: 1.19

Go version (client): go1.4.2

Git commit (client): 786b29d

OS/Arch (client): linux/amd64

Server version: 1.7.1

Server API version: 1.19

Go version (server): go1.4.2

Git commit (server): 786b29d

OS/Arch (server): linux/amd64

Also, we can check there is no running docker:

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			

Check the installation of the ubuntu docker image:

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	trusty	91e54dfb1179	3 weeks ago	188.4 MB

It's looking good, and we have an image to work with.

manifest update

We can launch containers:

```
docker::run { 'helloworld':  
    image  => 'ubuntu',  
    command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',  
}
```

which is equivalent to running the following:

```
docker run -d base /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

This will launch a Docker container managed by the local init system.

Launch docker container II

Let's run the updated puppet manifests:

```
$ sudo puppet apply docker_example.pp
```

Notice: Compiled catalog for puppet in environment production in 0.57 seconds

Notice: /Stage[main]/Main/Docker::Run[helloworld]/File[/etc/init.d/docker-helloworld]/ensure: created

Notice: /Stage[main]/Main/Docker::Run[helloworld]/Service[docker-helloworld]/ensure: ensure changed 'stopped' to 'running'

Notice: Finished catalog run in 2.47 seconds

We can check if it ran successfully:

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
f7d43960cb62	ubuntu	"/bin/sh -c 'while t	About a minute ago	Up About a minute
	helloworld			

If we want, we can attach it:

```
$ sudo docker attach f7d43960cb62
```

hello world

hello world

hello world

Example 1: Install LAMP with a Single Manifest

If you have not ever written a Puppet manifest before, this example is a good place to start. The manifest will be developed on a Puppet agent node, and executed via `puppet apply`, so an agent/master setup is not required.

You will learn how to write a manifest that will use following types of resource declarations:

- **exec**: To execute commands, such as `apt-get`
- **package**: To install packages via `apt`
- **service**: To ensure that a service is running
- **file**: To ensure that certain files exist

Create Manifest

On a fresh `lamp-1` VPS, create a new manifest:

```
sudo vi /etc/puppet/manifests/lamp.pp
```

Add the following lines to declare the resources that we just determined we wanted. The inline comments detail each resource declaration:

```
# execute 'apt-get update'
exec { 'apt-update':           # exec resource named 'apt-update'
  command => '/usr/bin/apt-get update' # command this resource will run
}

# install apache2 package
package { 'apache2':
  require => Exec['apt-update'],    # require 'apt-update' before installing
  ensure => installed,
}

# ensure apache2 service is running
service { 'apache2':
  ensure => running,
}

# install mysql-server package
package { 'mysql-server':
  require => Exec['apt-update'],    # require 'apt-update' before installing
  ensure => installed,
}
```

```
# ensure mysql service is running
service { 'mysql':
  ensure => running,
}

# install php5 package
package { 'php5':
  require => Exec['apt-update'],    # require 'apt-update' before installing
  ensure => installed,
}

# ensure info.php file exists
file { '/var/www/html/info.php':
  ensure => file,
  content => '<?php phpinfo(); ?>',  # phpinfo code
  require => Package['apache2'],      # require 'apache2' package before creating
}
```

Save and exit.

Apply Manifest

Now you will want to use the puppet apply command to execute the manifest. On *lamp-1*, run this:

```
sudo puppet apply --test
```

You will see many lines of output that show how the state of your server is changing, to match the resource declarations in your manifest. If there were no errors, you should be able to visit the public IP address (or domain name, if you set that up), and see the PHP info page that indicates that Apache and PHP are working. You can also verify that MySQL was installed on your server (it has not been secured, but we're not going to worry about that for now). Congrats! You set up a LAMP stack with Puppet.

This particular setup isn't too exciting, because we did not take advantage of our agent/master setup. The manifest is currently not available to other agent nodes, and Puppet is not continuously checking (every 30 minutes) that our server is in the state that the manifest described.

Now we want to convert the manifest that we just developed into a module, so it can be used by your other Puppet nodes.

Example 2: Install LAMP by Creating a New Module

Now let's create a basic module, based on the LAMP manifest that was developed in example 1. We will do this on the Puppet *master* node this time. To create a module, you must create a directory (whose name matches your module name) in Puppet's modules directory, and it must contain a directory called *manifests*, and that directory must contain an *init.pp* file. The *init.pp* file must only contain a Puppet class that matches the module name.

Create Module

On the Puppet *master*, create the directory structure for a module named *lamp*:

```
cd /etc/puppet/modules  
sudo mkdir -p lamp/manifests
```

Now create and edit your module's *init.pp* file:

```
sudo vi lamp/manifests/init.pp
```

Within this file, add a block for a class called "lamp", by adding the following lines:

```
class lamp {  
}  
}
```

Copy the contents of LAMP manifest that you created earlier (or copy it from example 1 above) and paste it into the *lamp* class block. In this file, you created a class definition for a "lamp" class. The code within the class is will not be evaluated at this time, but it is available to be declared. Additionally, because it complies with the Puppet conventions for defining a module, this class can be accessed as a module by other manifests.

Save and exit.

Use Module in Main Manifest

Now that we have a basic *lamp* module set up, let's configure our main manifest to use it to install a LAMP stack on *lamp-1*.

On the Puppet *master*, edit the main manifest:

```
sudo vi /etc/puppet/manifests/site.pp
```

Assuming the file is empty, add the following *node* blocks (replace "lamp-1" with the hostname of the Puppet agent that you want to install LAMP on):

```
node default { }

node 'lamp-1' { }
```

A node block allows you to specify Puppet code that will only apply to certain agent nodes.

The *default* node applies to every agent node that does not have a node block specified--we will leave it empty. The *lamp-1* node block will apply to your *lamp-1* Puppet agent node.

In the *lamp-1* node block, add the following code to use the "lamp" module that we just created:

```
include lamp
```

Now save and exit.

The next time your *lamp-1* Puppet agent node pulls its configuration from the master, it will evaluate the main manifest and apply the module that specifies a LAMP stack setup. If you want to try it out immediately, run the following command on the *lamp-1* agent node:

```
sudo puppet agent --test
```

Once it completes, you will see that a basic LAMP stack is set up, exactly like example 1. To verify that Apache and PHP are working, go to *lamp-1*'s public IP address in the a web browser:

http://lamp_1_public_IP/info.php

You should see the information page for your PHP installation.

Note that you can reuse the "lamp" module that you created by declaring it in other node blocks. Using modules is the best way to promote Puppet code reuse, and it is useful for organizing your code in a logical manner.

Now we will show you how to use pre-existing modules to achieve a similar setup.

Example 3: Install LAMP with Pre-existing Modules

There is a repository of publically-available modules, at the Puppet Forge, that can be useful when trying to develop your own infrastructure. The Puppet Forge modules can be quickly installed with built-in `puppet module` command. It just so happens that modules for installing and maintaining Apache and MySQL are available here. We will demonstrate how they can be used to help us set up our LAMP stack.

Install Apache and MySQL Modules

On your Puppet *master*, install the `puppetlabs-apache` module:

```
sudo puppet module install puppetlabs-apache
```

You will see the following output, which indicates the modules installed correctly:

```
Notice: Preparing to install into /etc/puppetlabs/puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppet/modules
└── puppetlabs-apache (v1.0.1)
    ├── puppetlabs-concat (v1.0.0) [/etc/puppet/modules]
    └── puppetlabs-stdlib (v3.2.0) [/etc/puppet/modules]
```

Also, install the `puppetlabs-mysql` module:

```
sudo puppet module install puppetlabs-mysql
```

Now the *apache* and *mysql* modules are available for use!

Edit the Main Manifest

Now let's edit our main manifest so it uses the new modules to install our LAMP stack.

On the Puppet *master*, edit the main manifest:

```
sudo vi /etc/puppet/manifests/site.pp
```

Assuming the file is empty, add the following node blocks (if you followed example 2, just delete the contents of the *lamp-1* node block):

```
node default { }
```

```
node 'lamp-1' { }
```

}

Within the *lamp-1* node block, use a resource-like class declaration to use the *apache* module (the in-line comments explain each line):

```
class { 'apache':           # use the "apache" module
  default_vhost =>false,    # don't use the default vhost
  default_mods =>false,     # don't load default mods
  mpm_module =>'prefork',   # use the "prefork" mpm_module
}
include apache::mod::php# include mod php
apache::vhost{ 'example.com': # create a vhost called "example.com"
  port  =>'80',            # use port 80
  docroot =>'/var/www/html', # set the docroot to the /var/www/html
}
```

The *apache* module can be passed parameters that override the default behavior of the module. We are passing in some basic settings that disable the default virtual host that the module creates, and make sure we create a virtual host that can use PHP. For complete documentation of the PuppetLabs-Apache module, check out its readme.

Using the MySQL module is similar to using the Apache module. We will keep it simple since we are not actually using the database at this point. Add the following lines within the node block:

```
class { 'mysql::server':
  root_password =>'password',
}
```

Like the Apache module, the MySQL module can be configured by passing parameters (full documentation here).

Now let's add the file resource that ensures info.php gets copied to the proper location. This time, we will use the *source* parameter to specify a file to copy. Add the following lines within the node block:

```
file { 'info.php':           # file resource name
  path =>'var/www/html/info.php',      # destination path
  ensure => file,
  require =>Class['apache'],          # require apache class be used
  source =>'puppet:///modules/apache/info.php', # specify location of file to be copied
}
```

This file resource declaration is slightly different from before. The main difference is that we are specifying the *source* parameter instead of the *content* parameter. *Source* tells puppet to copy a file over, instead of simply specifying the file's contents. The specified source, `puppet:///modules/apache/info.php` gets interpreted by Puppet into `/etc/puppet/modules/apache/files/info.php`, so we must create the source file in order for this resource declaration to work properly.

Save and exit `site.pp`.

Create the `info.php` file with the following command:

```
sudosh -c 'echo "<?php phpinfo(); ?>" > /etc/puppet/modules/apache/files/info.php'
```

The next time your *lamp-1* Puppet agent node pulls its configuration from the master, it will evaluate the main manifest and apply the module that specifies a LAMP stack setup. If you want to try it out immediately, run the following command on the *lamp-1* agent node:

```
sudo puppet agent --test
```

Once it completes, you will see that a basic LAMP stack is set up, exactly like example 1. To verify that Apache and PHP are working, go to *lamp-1*'s public IP address in the a web browser:

http://lamp_1_public_IP/info.php

Puppet creating and managing user accounts with SSH access:-

Security and access control

To have good security and access control practices, we need to use the following policies:

1. Everyone who needs access to a machine has his/her own user account with an SSH key (not a password).

DEVOPS MATERIAL

2. Access to special-purpose accounts, such as those used to deploy and run applications, or a database, is controlled by authorizing specific SSH keys, rather than using passwords.
3. Accounts that need certain, specific superuser privileges can get them via the sudo mechanism.
4. The root account is not accessible via the network (but there is secure, out-of-band access to the system console).
5. Third parties, such as contractors and support staff, get temporary access with limited privileges, which can be revoked once a job is finished.

Setting up policies listed above, while highly desirable from a security point of view, is time-consuming to do by hand and difficult to maintain. If a new user arrives, someone has to add and configure his account on every server. If a user leaves, the accounts have to be removed or locked everywhere.

Puppet can make it quicker and easier to manage user accounts securely across a large network. We can add or remove individual and shared accounts, control their access via SSH, manage their privileges via sudo, and have the changes immediately applied to every machine under Puppet's control, all without logging into a single server.

Puppet provides a couple of ways to help us manage users. The `user` resource type controls user accounts, and the `ssh_authorized_key` resource type controls SSH access to accounts. We can use Puppet to control user privileges by managing the `sudoers` file.

Creating a user

Edit our `manifests/site.pp` file as follows:

```
node 'puppet-agent' {
  include user
}
```

Also, we need to edit our `modules/user/manifests/init.pp` file:

```
class user {  
    user { 'testuser':  
        ensure => present,  
        comment => 'user',  
        home => '/home/testuser',  
        managehome => true  
    }  
}
```

Run puppet:

```
ubuntu@puppet-agent:~$ sudo puppet agent --test  
Info: Retrieving plugin  
Info: Caching catalog for puppet-agent.ec2.internal  
Info: Applying configuration version '1419727849'  
Notice: /Stage/main/User/User[testuser]/ensure: created  
Notice: Finished catalog run in 0.22 seconds  
ubuntu@puppet-agent:~$ cd /home  
ubuntu@puppet-agent:/home$ ls  
testuser ubuntu
```

Puppet's user resource type creates a user or modifies it if the user already exists. The following line declares a user whose login name is 'testuser':

```
user { 'testuser':
```

The user should be present:

```
ensure => present,
```

We can also specify here some information about the user:

```
comment => 'user',
```

The comment attribute sets the user's full name.

```
home => '/home/testuser',
```

The home attribute sets the path to the user's home directory. Puppet will not create this directory for us unless we also set the managehome attribute:

```
managehome => true,
```

So the manifest says that a user named 'testuser' should exist, whose full name is ' user', and that the home directory should be '/home/testuser', and that that directory should exist. Note that we have not specified a password for the user, and as a result 'testuser' will not yet be able to log in. Although Puppet can set passwords for users, SSH authentication is recommended.

Removing a user

To remove a user from the system altogether, use the **ensure => absent** attribute:

```
user { 'testuser':  
    ensure => absent,  
}
```

When we run Puppet, the 'testuser' account will be removed though testuser's home directory and any files he owned will remain.

Access control

Now that we've created the user's account, we now need to provide a secure way for a user to log in. We can do this using the SSH protocol.

In this section, the Puppet master will put a public key of my labtop into **authorized_keys** of Puppet agent so that I can login to the agent node from my laptop computer via ssh.

Puppet can manage SSH public keys and authorize them for user accounts, using the **ssh_authorized_key** resource type.

We'll need our own SSH public key for this. If we already have one on our own computer, display the contents:

If we don't have an SSH key, we can generate one for this exercise:

```
k@laptop:~/ssh$ ssh-keygen
```

Generating public/private rsa key pair.

Enter file in which to save the key (/home/testuser/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):

DEVOPS MATERIAL

Enter same passphrase again:

Your identification has been saved in /home/testuser/.ssh/id_rsa.

Your public key has been saved in /home/testuser/.ssh/id_rsa.pub.

The key fingerprint is:

bf:af:d1:85:af:a6:5f:f9:19:ad:cf:94:df:7d:21:d1 testuser@laptop

The key's randomart image is:

+--[RSA 2048]---

```
| . . |  
| . . |  
| . . |  
| o E |  
| S . o |  
| .. + oo|  
| o . =o+|  
| o.o =R|  
| +*+.oO|  
+-----+
```

Now display the **id_rsa.pub** file to see the public key:

```
k@laptop:~$ cat ~/.ssh/id_rsa.pub
```

```
ssh-rsa AAAAB3...jjQfJ7 testuser@laptop
```

DEVOPS MATERIAL

The key itself is the long string of numbers and letters, without the **ssh-rsa** part at the beginning, or the **testuser@laptop** part at the end. It's this string we'll put into the Puppet manifest in the next step.

Edit our **modules/user/manifests/init.pp** file as follows using our own key string as the value for key:

```
class user {  
    user { 'testuser':  
        ensure => present;  
        comment => 'user';  
        home => '/home/testuser';  
        managehome => true  
    }  
  
    ssh_authorized_key { 'testuser_ssh':  
        user => 'testuser';  
        type => 'rsa';  
        key => 'AAAA...GjjQfJ7';  
    }  
}
```

Run Puppet:

```
root@puppet-agent:~# puppet agent --test  
Info: Retrieving plugin  
Info: Caching catalog for puppet-agent.ec2.internal  
Info: Applying configuration version '1419754797'
```

```
Notice: /Stage[main]/User/Ssh_authorized_key[testuser_ssh]/key: key changed 'AAA...iQ8JkZV1F' to  
'AAAA...jjQfJ7'
```

```
Notice: Finished catalog run in 0.03 seconds
```

Actually, we got the output after changing the pub key.

At this point, Puppet has added the key to the file **/home/testuser/.ssh/authorized_keys** on Puppet agent node. When we try to log in to testuser's account via SSH, the system will look in this file to see if our private key matches any of the public keys listed there.

Assuming it does, we'll be able to log in to Puppet agent node of AWS via ssh:

```
k@laptop:~/ssh$ ssh testuser@54.88.104.246
```

```
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-36-generic x86_64)
```

* Documentation: <https://help.ubuntu.com/>

System information as of Sat Dec 27 19:11:48 UTC 2014

System load: 0.0 Processes: 104

Usage of /: 11.8% of 7.74GB Users logged in: 0

Memory usage: 11% IP address for eth0: 172.31.43.38

Swap usage: 0%

```
$ uname -a
```

```
Linux puppet-agent 3.13.0-36-generic #63-Ubuntu SMP Wed Sep 3 21:30:07 UTC 2014 x86_64
```

```
x86_64x86_64 GNU/Linux
```

```
$ cd /home
```

```
$ ls
```

```
tesuser ubuntu
```

we'll learn how to template configuration files out with Puppet, filling in variables with the managed node's facts.

Often we may want to maintain configuration files for applications that are different between servers. If we have a couple of configurations, it's not touch to maintain multiple files, but what if we have a very large number of differing configurations? We can manage this situation by writing ERB templates and populating the templates with node-specific information. This can be done in Puppet with the **template()** function:

Puppet supports templates written in the **ERB** templating and it can be used to specify the contents of files.

Now we want to manage many different websites, it would quickly become tedious to supply an almost identical virtual host file for each site, altering only the name and domain of the site.

The best way of doing this is to give Puppet master a template file into which it could just insert these variables for each different site. The **template()** function serves just this purpose. Anywhere we have multiple files that differ only slightly, or files that need to contain dynamic information, we just use a template.

template evaluation

Templates are evaluated via a simple function as show in the following example:

```
$value = template("my_module/my_template.erb")
```

Template files should be stored in the **templates** directory of a Puppet module, which allows the template function to locate them with the simplified path format shown above. For example, the file referenced by **template("my_module/my_template.erb")** would be found on disk at **/etc/puppet/modules/my_module/templates/my_template.erb**.

Templates are always evaluated **by the parser, not by the client**. This means that if we're using a puppet master server, then the templates only need to be on the server, and we never need to download them to the client. The client sees no difference between using a template and specifying all of the text of the file as a string. - Docs: Using Puppet Templates

conf file - deploying a virtual host

Suppose new we want to build three new sites: test1.com, test2.com, and test3.com. To prepare for this, we need to change the Puppet config for test.com to use a template so that we can later use the same template for the new sites.

1. Here is our new **modules/nginx/manifests/init.pp** file:

```
2. class nginx {  
3.   package { 'nginx':  
4.     ensure => installed,
```

```
5.      }
6.
7.      service { 'nginx':
8.          ensure => running,
9.          enable => true,
10.         require => Package['nginx'],
11.      }
12.
13.     file { '/etc/nginx/sites-enabled/default':
14.         ensure => 'absent',
15.     }
16. }
```

Since we previously used the file **/etc/nginx/sites-enabled/ default** as the virtual host for **test.com** (Puppet packages, services, and files II with nginx), we need to remove that now:

```
file { '/etc/nginx/sites-enabled/default':
ensure => absent,
}
```

17. We want to reate a new templates directory in the nginx module:

```
18. ubuntu@ip-172-31-45-62:/etc/puppet$ sudo mkdir -p modules/nginx/templates
```

19. Create the file **modules/nginx/templates/vhost.conf.erb** with the following contents. The template file is for the virtual host definition:

```
20.    server {  
21.        listen 80;  
22.        root /var/www/<%=@site_name%>;  
23.        server_name<%=@site_domain%>;  
24.    }
```

The <%=%> signs mark where parameters will go; we will supply `site_name` and `site_domain` later, when we use the template. Puppet will replace <%=@site_name%> with the value of the `site_name` variable.

25. Then in the `site.pp` file, we include the `nginx` module on the node:

```
26. node 'puppet-agent' {  
27.     include nginx  
28.     $site_name = 'test'  
29.     $site_domain = 'test.com'  
30.     file { '/etc/nginx/sites-enabled/test.conf':  
31.         content => template('nginx/vhost.conf.erb'),  
32.         notify => Service['nginx'],  
33.     }  
34.     ...  
35. }
```

Note that before using the template, we need to set values for the variables `site_name` and `site_domain`:

```
$site_name = 'test'
```

```
$site_domain = 'test.com'
```

Also note that when we refer to these variables in Puppet code, we use a **\$ prefix (\$site_name)**, but in the template it's an **@ prefix (@site_name)**. This is because in templates we're actually writing Ruby, not Puppet!

Now we can use the template to generate the nginx virtual host file:

```
file { '/etc/nginx/sites-enabled/test.conf':  
    content => template('nginx/vhost.conf.erb'),  
    notify => Service['nginx'],  
}
```

This looks just like any other file resource, with a content attribute. Though we can give the contents of the file as a literal string:

```
content => "Hello, world\n".
```

However, here we call the template function:

```
content => template('nginx/vhost.conf.erb'),
```

The argument to template tells Puppet where to find the template file. The path

nginx/vhost.conf.erb

Translates to

```
modules/nginx/templates/vhost.conf.erb
```

36. Run puppet:

37. `ubuntu@ip-172-31-45-62:/etc/puppet$ sudo puppet apply modules/nginx/manifests/init.pp`

38. Notice: Compiled catalog for ip-172-31-45-62.ec2.internal in environment production in 0.02 seconds

39. Notice: Finished catalog run in 0.02 seconds

40. Check the resulting virtual host file on agent node:

41. `ubuntu@puppet-agent:~$ cat /etc/nginx/sites-enabled/test.conf`

42. `server {`

43. `listen 80;`

44. `root /var/www/test;`

45. `server_nametest.com;`

46. `}`

Puppet now evaluated the template, inserted the values of any variables referenced in `<%=%` `%>` signs in `modules/nginx/templates/vhost.conf.erb`, and generated the final output as we can see in the `test.conf`.

Accessing the Vagrant environment

In order to access the Vagrant environment created, Vagrant exposes some high-level networking options for things such as forwarded ports, connecting to a public network, or creating a private network. The high-level networking is meant to define an abstraction that works across multiple providers such as VirtualBox, VMWare, etc.

In the previous chapter (Provisioning), we have a web server up and running with the ability to modify files from our host and have them automatically synced to the guest. However, accessing the web pages simply from the terminal from inside the machine is not very satisfying. In this chapter, we'll use Vagrant's networking features to give us additional options for accessing the virtual machine from our host machine.

Port Forwarding

Port forwarding allows us to specify ports on the guest machine to share via a port on the host machine. This allows us to access a port on our host machine, but actually have all the network traffic forwarded to a specific port on the guest machine, over either TCP or UDP.

Let's setup a port forwarding so that we can access Apache in our guest by configuring Vagrantfile:

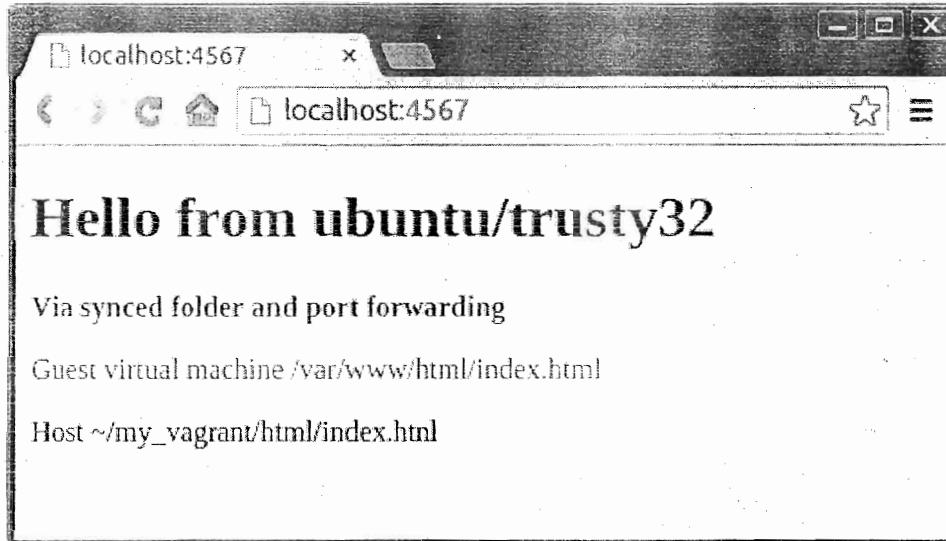
```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|  
  config.vm.box = "ubuntu/trusty32"  
  config.vm.provision :shell, path: "bootstrap.sh"  
  config.vm.network :forwarded_port, host: 4567, guest: 80  
end
```

Our guest machine is running a web server listening on port 80. The configuration makes a forwarded port mapping to port 4567 on our host machine. We can then open our browser to localhost:4567 and browse the website, while all actual network data is being sent to the guest. In short, this will allow accessing port 80 on the guest via port 4567 on the host.

Run a vagrant reload:

```
k@laptop:~/my_vagrant$ vagrant reload --provision  
...  
==>default: Preparing network interfaces based on configuration...  
default: Adapter 1: nat  
==>default: Forwarding ports...  
default: 80 => 4567 (adapter 1)  
...  
...
```

Once our virtual machine is up and running again, we can see a web page that is being served from the virtual machine that was automatically setup by Vagrant. Type in <http://127.0.0.1:4567> in our browser:



Synced folders

Using **synced folders**, Vagrant will automatically sync our files to and from the guest machine. In other words, Vagrant shares our project directory (where the **Vagrantfile** is in) to the **/vagrant** directory in our guest machine. Run **vagrant up** again and **SSH** into our machine to see:

```
$ vagrant up  
$ vagrant ssh  
vagrant@vagrant-ubuntu-trusty-32:~$ ls /vagrant  
Vagrantfile
```

The **Vagrantfile** we see inside the virtual machine is actually the same **Vagrantfile** that is on our host machine. Go ahead and touch a file to check it:

```
vagrant@vagrant-ubuntu-trusty-32:~$ touch /vagrant/test.txt  
vagrant@vagrant-ubuntu-trusty-32:~$ ls  
vagrant@vagrant-ubuntu-trusty-32:~$ exit  
logout
```

Connection to 127.0.0.1 closed.

```
k@laptop:~/my_vagrant$ ls  
test.txt Vagrantfile
```

The "test.txt" file is now on our host machine. As we can see, Vagrant kept the folders in sync. With synced folders, we can continue to use our own editor on our host machine and have the files sync into the guest machine.

Provisioning - config.vm.provision

Now, we have a virtual machine running a basic copy of Ubuntu and we can edit files from our machine and have them synced into the virtual machine. Let's now serve those files using a webserver.

Vagrant has built-in support for automated provisioning. Using Vagrant's provisioning feature, Vagrant will automatically install software when we vagrant up so that the guest machine can be repeatable created and ready-to-use.

Apache install via shell provisioner

In this section, we'll setup Apache for our basic project using a shell script, **bootstrap.sh**:

```
k@laptop:~/my_vagrant$ ls  
bootstrap.sh test.txtVagrantfile
```

The "bootstrap.sh" file looks like this:

```
k@laptop:~/my_vagrant$ ls  
#!/usr/bin/env bash  
  
apt-get update  
apt-get install -y apache2  
rm -rf /var/www  
ln -fs /vagrant /var/www
```

Next, we need to configure Vagrant to run this shell script when setting up our machine. We do this by editing the **Vagrantfile**:

The "bootstrap.sh" file looks like this:

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|  
  config.vm.box = "ubuntu/trusty32"  
  config.vm.provision :shell, path: "bootstrap.sh"  
end
```

The "provision" line tells Vagrant to use the **shell provisioner** to setup the machine, with the **bootstrap.sh** file. The file **path** is relative to the location of the **project root** (where the **Vagrantfile** is).

After the configuration, we just run **vagrant up** to create our virtual machine via automatic provision by Vagrant. We should see the output from the shell script appear in our terminal.

```
k@laptop:~/my_vagrant$ vagrant up  
Bringing machine 'default' up with 'virtualbox' provider...  
==> default: Checking if box 'ubuntu/trusty32' is up to date...  
==> default: VirtualBox VM is already running.
```

If the guest machine is already running from a previous step as in our case, we run **vagrant reload --provision**, which will quickly restart our virtual machine, skipping the initial import step.

```
k@laptop:~/my_vagrant$ vagrant reload --provision  
==> default: Attempting graceful shutdown of VM...  
==> default: Checking if box 'ubuntu/trusty32' is up to date...  
==> default: Clearing any previously set forwarded ports...
```

```
==>default: Clearing any previously set network interfaces...
==>default: Preparing network interfaces based on configuration...
default: Adapter 1: nat
==>default: Forwarding ports...
default: 22 => 2222 (adapter 1)
==>default: Booting VM...
==>default: Waiting for machine to boot. This may take a few minutes...
default: SSH address: 127.0.0.1:2222
default: SSH username: vagrant
default: SSH auth method: private key
default: Warning: Connection timeout. Retrying...
...
default: Warning: Remote connection disconnect. Retrying...
==>default: Machine booted and ready!
==>default: Checking for guest additions in VM...
==>default: Mounting shared folders...
default: /vagrant => /home/k/my_vagrant
==>default: Running provisioner: shell...
...
==>default: The following NEW packages will be installed:
==>default: apache2 apache2-bin apache2-data libapr1 libaprutil1 libaprutil1-dbd-sqlite3
==>default: libaprutil1-ldap ssl-cert
==>default: 0 upgraded, 8 newly installed, 0 to remove and 0 not upgraded.
==>default: Need to get 1,270 kB of archives.
```

```
==>default: After this operation, 5,050 kB of additional disk space will be used.  
  
==>default: Get:1 http://archive.ubuntu.com/ubuntu/ trusty/main libapr1 i386 1.5.0-1 [88.8 kB]  
  
==>default: Get:2 http://archive.ubuntu.com/ubuntu/ trusty/main libaprutil1 i386 1.5.3-1 [76.6 kB]  
  
==>default: Get:3 http://archive.ubuntu.com/ubuntu/ trusty/main libaprutil1-dbd-sqlite3 i386 1.5.3-1 [10.3 kB]  
  
==>default: Get:4 http://archive.ubuntu.com/ubuntu/ trusty/main libaprutil1-ldap i386 1.5.3-1 [8,552 B]  
  
==>default: Get:5 http://archive.ubuntu.com/ubuntu/ trusty-updates/main apache2-bin i386 2.4.7-1ubuntu4.1 [821 kB]  
  
==>default: Get:6 http://archive.ubuntu.com/ubuntu/ trusty-updates/main apache2-data all 2.4.7-1ubuntu4.1 [160 kB]  
  
==>default: Get:7 http://archive.ubuntu.com/ubuntu/ trusty-updates/main apache2 i386 2.4.7-1ubuntu4.1 [87.6 kB]  
  
==>default: Get:8 http://archive.ubuntu.com/ubuntu/ trusty/main ssl-cert all 1.0.33 [16.6 kB]  
  
==>default: dpkg-preconfigure: unable to re-open stdin: No such file or directory  
  
...  
  
==>default: Processing triggers for ureadahead (0.100.0-16) ...  
  
==>default: Processing triggers for ufw (0.34~rc-0ubuntu2) ...  
  
k@laptop:~/my_vagrant$
```

The provision flag on the reload command instructs Vagrant to run the provisioners, since usually Vagrant will only do this on the first vagrant up.

After Vagrant completes running, the web server will be up and running. However, we can't see the website from our own browser yet, but we can verify that the provisioning works by loading a file from SSH within the machine:

```
k@laptop:~/my_vagrant$ vagrant ssh  
  
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-39-generic i686)
```

```
vagrant@vagrant-ubuntu-trusty-32:~$ wget -qO- 127.0.0.1
```

This works because in the shell script above we installed Apache and setup the **defaultDocumentRoot** of Apache to point to our **/vagrant** directory, which is the default synced folder setup by Vagrant.

We can check if apache is actually running:

```
vagrant@vagrant-ubuntu-trusty-32:~$ ps -ef|grep apache2
root    2295  1  0 03:44 ?        00:00:00 /usr/sbin/apache2 -k start
www-data 2297 2295  0 03:44 ?        00:00:03 /usr/sbin/apache2 -k start
www-data 2299 2295  0 03:44 ?        00:00:03 /usr/sbin/apache2 -k start
```

we set up fully functional virtual machine for basic web development. How do we clean up our development environment?

We can **suspend**, **halt**, or **destroy** the guest machine. Each of these options have pros and cons.

1. **vagrant suspend** : We can suspend the virtual machine by **vagrant suspend** command. The current running state of the machine will be saved and stopped. When we're ready to begin working again, we can just run **vagrant up**, then it will be resumed from where we left off.
The main benefit of this method is that it is super fast, usually taking only 5 to 10 seconds to stop and start our work.
Suspending virtual machine, however, still eats up our disk space, and requires even more disk space to store all the state of the virtual machine RAM on disk.

vagrant suspend:

```
k@laptop:~/my_vagrant$ vagrant suspend  
==>default: Saving VM state and suspending execution...
```

vagrant up:

```
k@laptop:~/my_vagrant$ vagrant up  
Bringing machine 'default' up with 'virtualbox' provider...  
==>default: Checking if box 'ubuntu/trusty32' is up to date...  
==>default: Resuming suspended VM...  
==>default: Booting VM...  
==>default: Waiting for machine to boot. This may take a few minutes...  
default: SSH address: 127.0.0.1:2222  
default: SSH username: vagrant  
default: SSH auth method: private key  
default: Warning: Connection refused. Retrying...  
==>default: Machine booted and ready!
```

2. **vagrant halt** : We can halt the virtual machine using **vagrant halt** command. It will gracefully shut down the guest operating system and power down the guest machine. We can use **vagrant up** when we're ready to boot it again.

The benefit of this method is that it will cleanly shut down our machine, preserving the contents of disk, and allowing it to be cleanly started again.

The downside is that it'll take some extra time to start from a cold boot, and the guest machine still consumes disk space.

vagrant halt:

```
k@laptop:~/my_vagrant$ vagrant halt
```

```
==>default: Attempting graceful shutdown of VM...
```

vagrant up:

```
k@laptop:~/my_vagrant$ vagrant up
```

```
Bringing machine 'default' up with 'virtualbox' provider...
```

```
==>default: Checking if box 'ubuntu/trusty32' is up to date...
```

```
==>default: VirtualBox VM is already running.
```

```
k@laptop:~/my_vagrant$ vagrant halt
```

```
==>default: Attempting graceful shutdown of VM...
```

```
k@laptop:~/my_vagrant$ vagrant up
```

```
Bringing machine 'default' up with 'virtualbox' provider...
```

```
==>default: Checking if box 'ubuntu/trusty32' is up to date...
```

```
==>default: Clearing any previously set forwarded ports...
```

```
==>default: Clearing any previously set network interfaces...
```

```
==>default: Preparing network interfaces based on configuration...
```

```
default: Adapter 1: nat
```

```
==>default: Forwarding ports...
```

```
default: 80 => 4567 (adapter 1)
```

```
default: 22 => 2222 (adapter 1)
```

```
==>default: Booting VM...
```

```
==>default: Waiting for machine to boot. This may take a few minutes...
```

```
default: SSH address: 127.0.0.1:2222
```

```
default: SSH username: vagrant
```

```
default: SSH auth method: private key
```

```
default: Warning: Connection timeout. Retrying...
default: Warning: Remote connection disconnect. Retrying...
default: Machine booted and ready!
==>default: Checking for guest additions in VM...
==>default: Mounting shared folders...
default: /vagrant => /home/k/my_vagrant
==>default: Machine already provisioned. Run 'vagrant provision' or use the '--provision'
==>default: to force provisioning. Provisioners marked to run always will still run.
```

3. **vagrant destroy** : We can destroy the virtual machine by **vagrant halt** command. It will remove all traces of the guest machine from our system. It'll stop the guest machine, power it down, and remove all of the guest hard disks. Again, when we're ready to work again, just issue a **vagrant up**.

The benefit of this is that no trace is left on our machine. The disk space and RAM consumed by the guest machine is reclaimed and our host machine is left clean.

The downside is that **vagrant up** to get working again will take some extra time since it has to reimport the machine and reprovision it.

4. Rebuild - vagrant up

5. Since the Vagrant environment is already all configured via the **Vagrantfile**, we simply have to run a **vagrant up** at any time and Vagrant will recreate your work environment.

6. **\$ vagrant up**

In addition to providing a development environment box, Vagrant also makes it easy to share and collaborate on these environments. The primary feature to do this in Vagrant is called **Vagrant Share**.

Vagrant share has three main features:

1. **HTTP sharing** creates a URL that will be routed directly into our Vagrant environment.
2. **SSH sharing** allows instant SSH access to our Vagrant environment by anyone by running **vagrant connect --ssh** on the remote side.
3. **General sharing** allows anyone to access any exposed port of our Vagrant environment by running **vagrant connect** on the remote side.

4. Vagrant Cloud

5. Before being able to share our Vagrant environment, we need an account on Vagrant Cloud.

6. Once we have an account, we can log in using **vagrant login**:

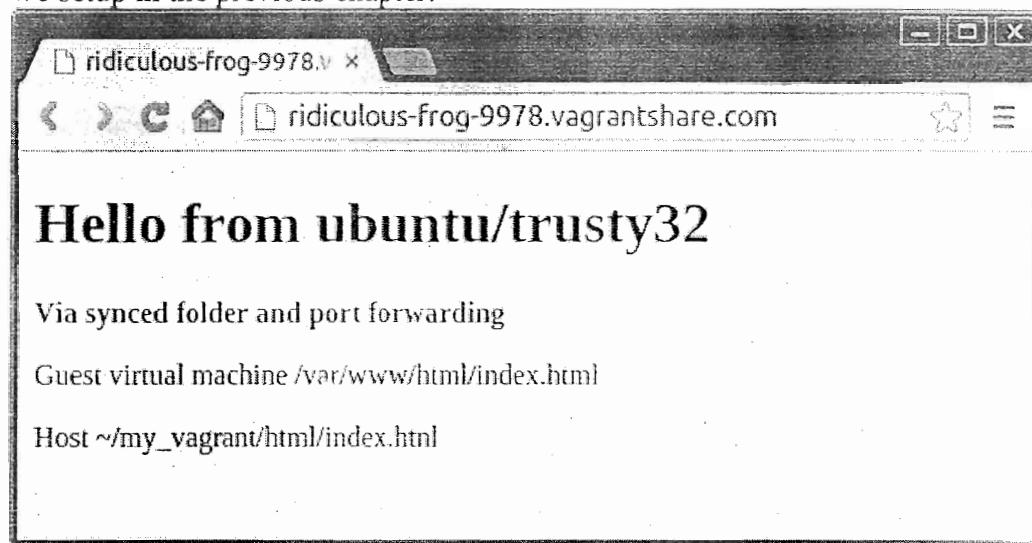
7. **\$ vagrant login**
8. In a moment we'll ask for your username and password to Vagrant Cloud.
9. After authenticating, we will store an access token locally. Your
10. login details will be transmitted over a secure connection, and are
11. never stored on disk locally.
- 12.
13. If you don't have a Vagrant Cloud account, sign up at vagrantcloud.com
- 14.
15. Username or Email: k@bogotobogo.com
16. Password (will be hidden):
17. You're now logged in!

18. k@laptop:~/my_vagrant\$

19. Now, it's share time, issue vagrant share command:

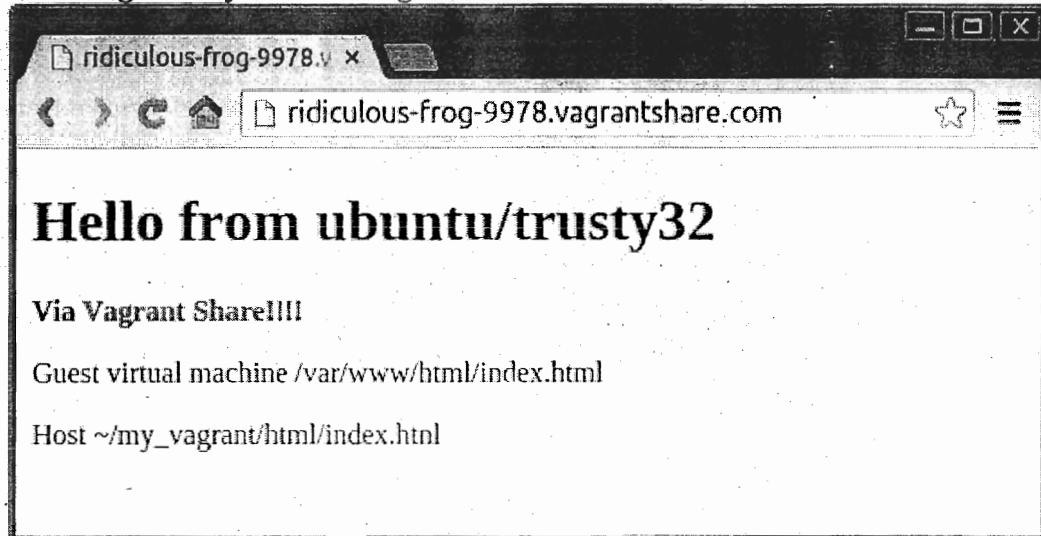
```
20. k@laptop:~/my_vagrant$ vagrant share
21. ==>default: Detecting network information for machine...
22. default: Local machine address: 127.0.0.1
23. default:
24. default: Note: With the local address (127.0.0.1), Vagrant Share can only
25. default: share any ports you have forwarded. Assign an IP or address to your
26. default: machine to expose all TCP ports. Consult the documentation
27. default: for your provider ('virtualbox') for more information.
28. default:
29. default: Local HTTP port: 4567
30. default: Local HTTPS port: disabled
31. default: Port: 2222
32. default: Port: 4567
33. ==>default: Checking authentication and authorization...
34. ==>default: Creating Vagrant Share session...
35. default: Share will be at: ridiculous-frog-9978
36. ==>default: Your Vagrant Share is running! Name: ridiculous-frog-9978
37. ==> default: URL: http://ridiculous-frog-9978.vagrantshare.com
38. ==> default:
39. ==>default: You're sharing your Vagrant machine in "restricted" mode. This
40. ==> default: means that only the ports listed above will be accessible by
41. ==>default: other users (either via the web URL or using `vagrant connect`).
```

42. Let's copy the URL ("http://ridiculous-frog-9978.vagrantshare.com") that vagrant share outputted for us and visit that in a web browser. It should load the index page we setup in the previous chapter:



43.

44. Now, modify our **index.html** file and refresh the URL. It will be updated! That URL is routing directly into our Vagrant environment:



45.

46. To end the sharing session, hit Ctrl+C in our terminal. We can refresh the URL again to verify that our environment is no longer being shared.

```
47. k@laptop:~/my_vagrant$ vagrant share
48. ==>default: Detecting network information for machine...
49. ...
50. ==>default: You're sharing your Vagrant machine in "restricted" mode. This
51. ==> default: means that only the ports listed above will be accessible by
52. ==>default: other users (either via the web URL or using 'vagrant connect').
53. ^CExiting due to interrupt.
```

You'll learn how to:

- Create and use a repository
- Start and manage a new branch
- Make changes to a file and push them to GitHub as commits
- Open and merge a pull request

What is GitHub?

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

This tutorial teaches you GitHub essentials like *repositories*, *branches*, *commits*, and *Pull Requests*. You'll create your own Hello World repository and learn GitHub's Pull Request workflow, a popular way to create and review code.

No coding necessary

To complete this tutorial, you need a [GitHub.com account](#) and Internet access. You don't need to know how to code, use the command line, or install Git (the version control software GitHub is built on).

Tip: Open this guide in a separate browser window (or tab) so you can see it while you complete the steps in the tutorial.

Step 1. Create a Repository

A **repository** is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets – anything your project needs. We recommend including a *README*, or a file with information about your project. GitHub makes it easy to add one at the same time you create your new repository. *It also offers other common options such as a license file.*

Your `hello-world` repository can be a place where you store ideas, resources, or even share and discuss things with others.

To create a new repository

1. In the upper right corner, next to your avatar or identicon, click and then select **New repository**.
2. Name your repository `hello-world`.
3. Write a short description.
4. Select **Initialize this repository with a README**.

Owner: hubot / Repository name: hello-world

Great repository names are short and memorable. Need inspiration? How about petulani-shame.

Description (optional): Just another repository

Public: Anyone can see this repository. You choose who can commit.

Private: You choose who can see and commit to this repository.

Initialize this repository with a README: This will allow you to git clone the repository immediately. Skip this step if you have already run git init locally.

Add ignore: None Add a license: None

Create repository

Click **Create repository**.

Step 2. Create a Branch

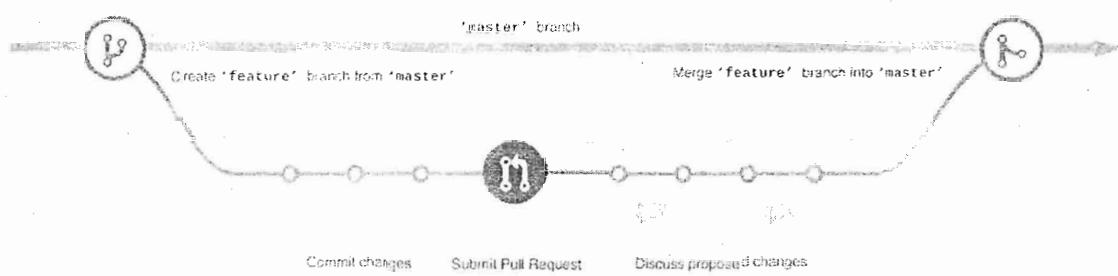
Branching is the way to work on different versions of a repository at one time.

By default your repository has one branch named `master` which is considered to be the definitive branch. We use branches to experiment and make edits before committing them to `master`.

When you create a branch off the `master` branch, you're making a copy, or snapshot, of `master` as it was at that point in time. If someone else made changes to the `master` branch while you were working on your branch, you could pull in those updates.

This diagram shows:

- The `master` branch
- A new branch called `feature` (because we're doing 'feature work' on this branch)
- The journey that `feature` takes before it's merged into `master`.



Have you ever saved different versions of a file? Something like:

story.txt
story-joe-edit.txt

story-joe-edit-reviewed.txt

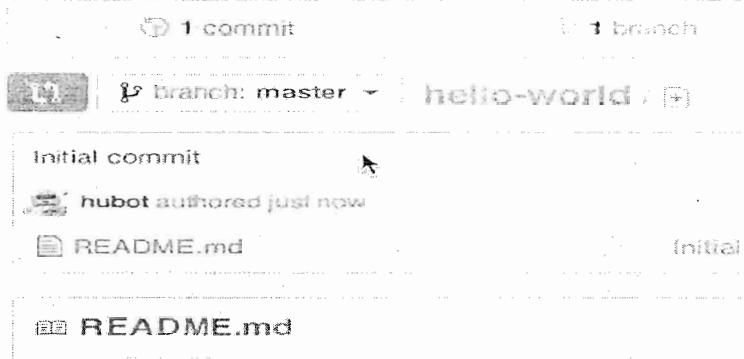
Branches accomplish similar goals in GitHub repositories.

Here at GitHub, our developers, writers, and designers use branches for keeping bug fixes and feature work separate from our `master` (production) branch. When a change is ready, they merge their branch into `master`.

To create a new branch

1. Go to your new repository `hello-world`.
2. Click the drop down at the top of the file list that says **branch: master**.
3. Type a branch name, `readme-edits`, into the new branch text box.
4. Select the blue **Create branch** box or hit “Enter” on your keyboard.

Just another repository — Edit



Now you have two branches, `master` and `readme-edits`. They look exactly the same, but not for long! Next we'll add our changes to the new branch.

Step 3. Make and commit changes

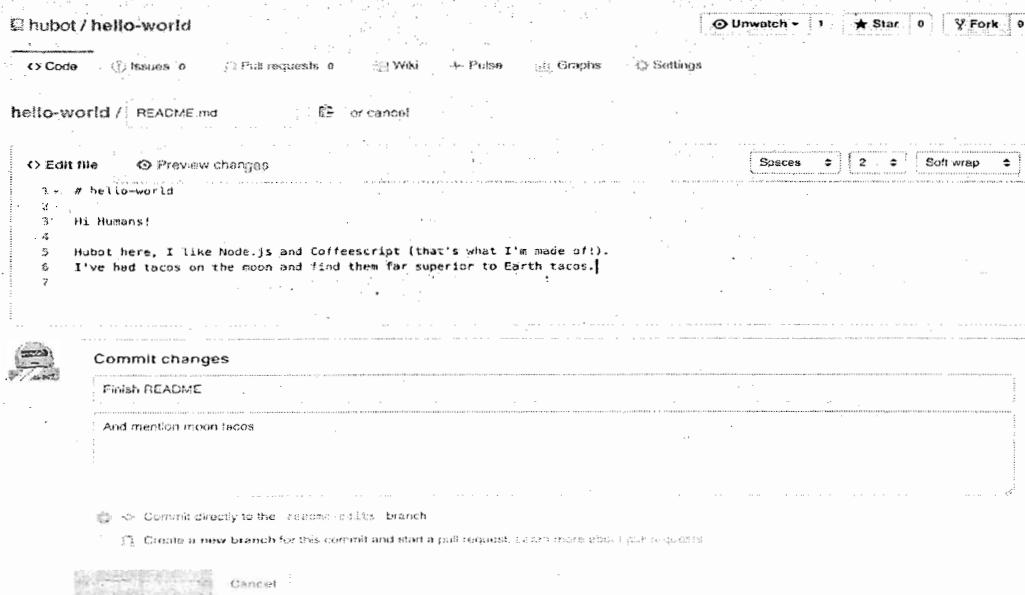
Bravo! Now, you're on the code view for your `readme-edits` branch, which is a copy of `master`. Let's make some edits.

On GitHub, saved changes are called *commits*. Each commit has an associated *commit message*, which is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you've done and why.

Make and commit changes

1. Click the `README.md` file.
2. Click the pencil icon in the upper right corner of the file view to edit.

3. In the editor, write a bit about yourself.
4. Write a commit message that describes your changes.
5. Click Commit changes button.



These changes will be made to just the README file on your `readme-edits` branch, so now this branch contains content that's different from `master`.

Step 4. Open a Pull Request

Nice edits! Now that you have changes in a branch off of `master`, you can open a *pull request*.

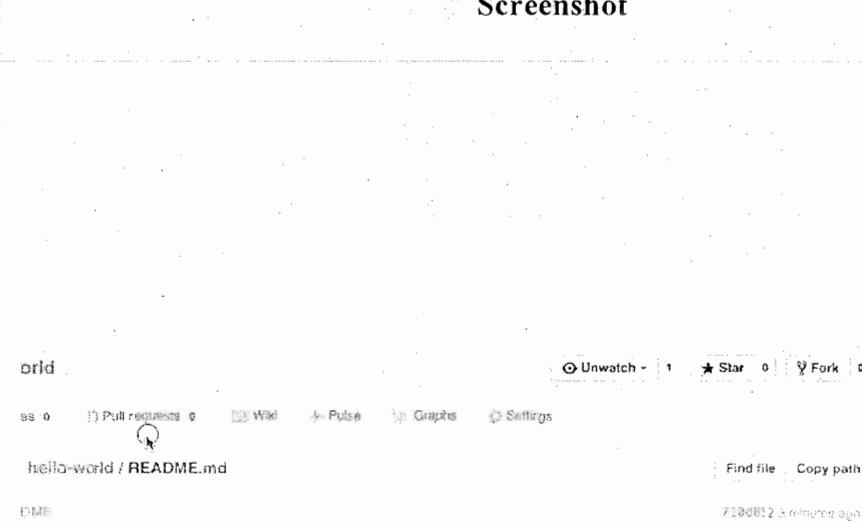
Pull Requests are the heart of collaboration on GitHub. When you open a *pull request*, you're proposing your changes and requesting that someone review and pull in your contribution and merge them into their branch. Pull requests show *diffs*, or differences, of the content from both branches. The changes, additions, and subtractions are shown in green and red.

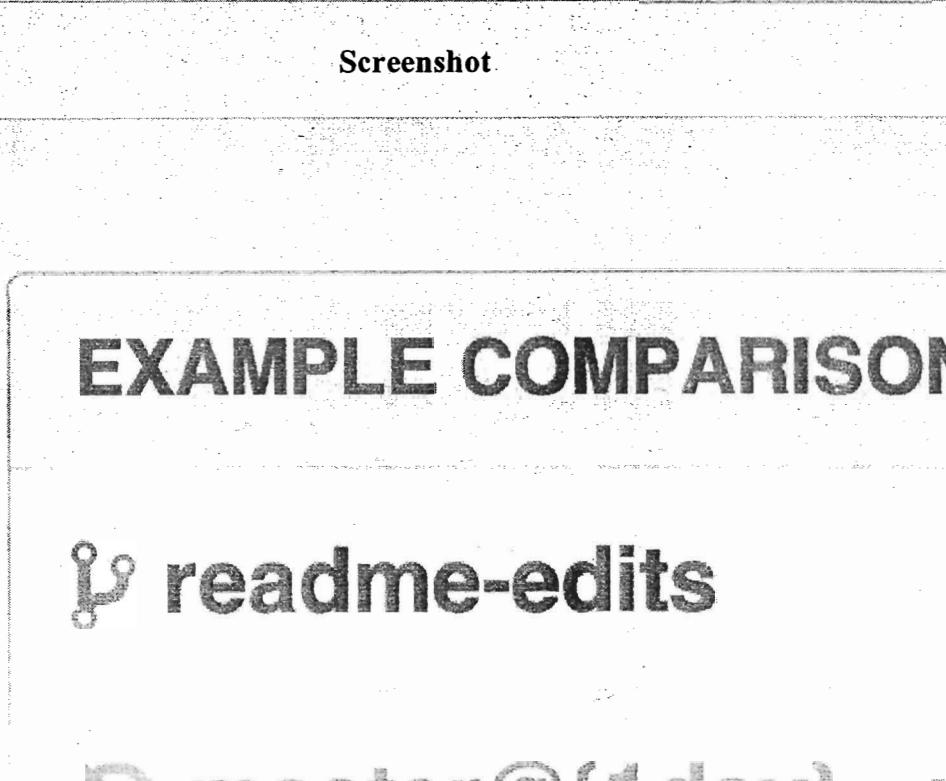
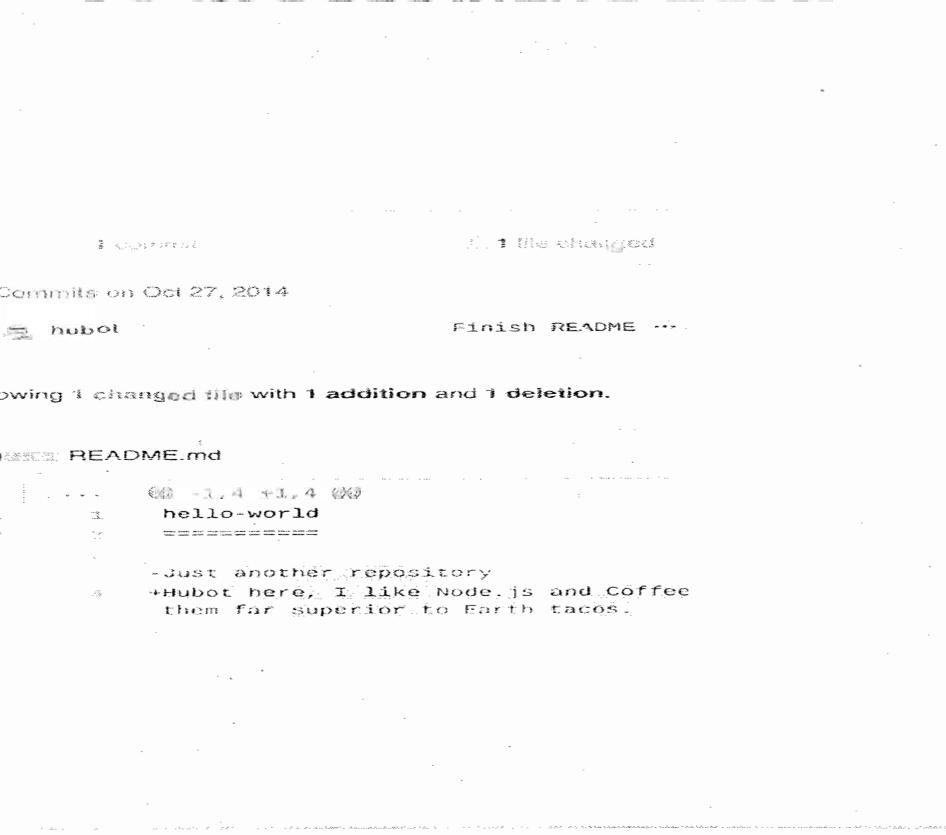
As soon as you make a commit, you can open a pull request and start a discussion, even before the code is finished.

By using GitHub's @mention system in your pull request message, you can ask for feedback from specific people or teams, whether they're down the hall or 10 time zones away.

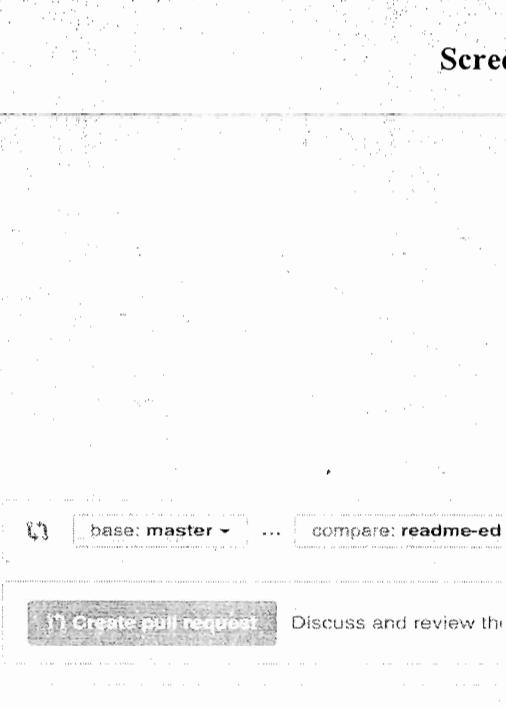
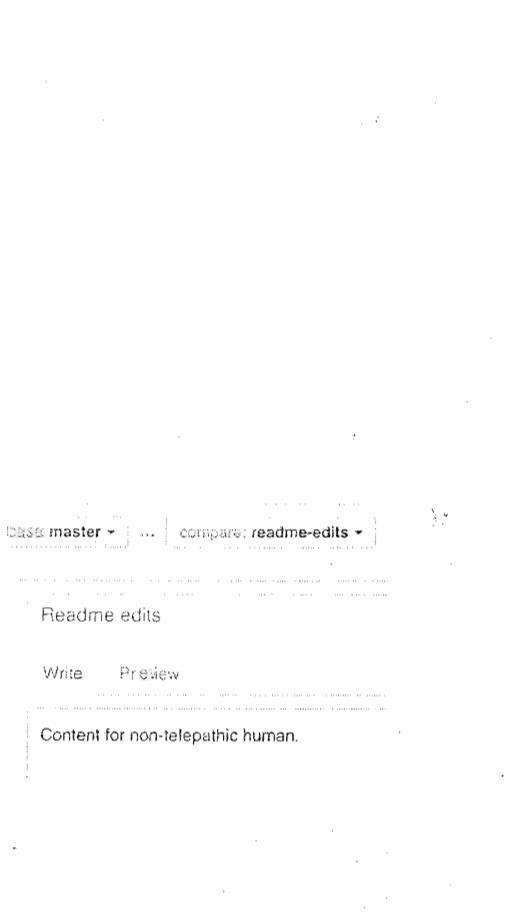
You can even open pull requests in your own repository and merge them yourself. It's a great way to learn the GitHub Flow before working on larger projects.

Open a Pull Request for changes to the README*Click on the image for a larger version*

Step	Screenshot
Click the Pull Request tab, then from the Pull Request page, click the green New pull request button.	

Step	Screenshot
Select the branch you made, readme-edits, to compare with master (the original).	
Look over your changes in the diffs on the Compare page, make sure they're what you want to submit.	

DEVOPS MATERIAL

Step	Screenshot
When you're satisfied that these are the changes you want to submit, click the big green Create Pull Request button.	
Give your pull request a title and write a brief description of your change.	

Step	Screenshot
es.	

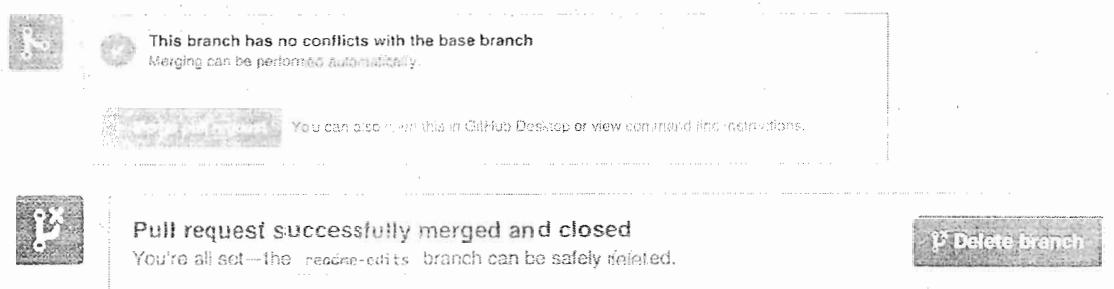
When you're done with your message, click **Create pull request!**

Tip: You can use emoji and drag and drop images and gifs onto comments and Pull Requests.

Step 5. Merge your Pull Request

In this final step, it's time to bring your changes together – merging your `readme-edits` branch into the `master` branch.

1. Click the green **Merge pull request** button to merge the changes into `master`.
2. Click **Confirm merge**.
3. Go ahead and delete the branch, since its changes have been incorporated, with the **Delete branch** button in the purple box.



Git init

Let's start with Git Bash.

First, I created a new directory C:\MyGit. Then, issue a command "git init GitProjects", and it will generate a subdirectory named **GitProject**:



MINGW32: /c/MyGit

```
admin@KHONG /c/MyGit
$ ls
admin@KHONG /c/MyGit
$ git init GitProject
Initialized empty Git repository in c:/MyGit/GitProject/.git/
admin@KHONG /c/MyGit
$ ls -la
total 0
drwxr-xr-x    3 admin   Administ .      0 Sep 13 23:55 .
drwxr-xr-x    69 admin  Administ ..     0 Sep 13 22:33 ..
drwxr-xr-x    1 admin   Administ .     0 Sep 13 23:55 GitProject
admin@KHONG /c/MyGit
$ cd GitProject/
admin@KHONG /c/MyGit/GitProject (master)
$ ls -la
total 0
drwxr-xr-x    1 admin   Administ .      0 Sep 13 23:55 .
drwxr-xr-x    3 admin  Administ ..     0 Sep 13 23:55 ..
drwxr-xr-x    1 admin   Administ .     0 Sep 13 23:55 .git
admin@KHONG /c/MyGit/GitProject (master)
$
```

The **GitProject** contains all of our necessary repository files - a Git repository skeleton. At this point, nothing in our project is tracked yet.

Note that we could have issued the "gitinit" command within the directory, "GitProject", then it would have created .git subdirectory.

Now, we may want to create some files:



```
MINGW32:c/MyGit/GitProject
cd /c/MyGit/GitProject
echo "readme.txt" >> readme.txt

cd /c/MyGit/GitProject
echo "test.txt" >> test.txt

cd /c/MyGit/GitProject
ls -la
total 1
drwxr-xr-x  1 admin   Administ  0 Sep 14 00:10 .
drwxr-xr-x  3 admin   Administ  0 Sep 13 23:55 ..
drwxr-xr-x  1 admin   Administ  0 Sep 13 23:55 .git
-rw-r--r--  1 admin   Administ 11 Sep 14 00:10 readme.txt
-rw-r--r--  1 admin   Administ  9 Sep 14 00:10 test.txt

admin@KHONG ~%
```

Using Git

Let's go into the GitProject directory where we have two files: readme.txt and test.txt.

Status

To check the status of git, we issue "git status".

The **status** is the main tool we use to determine which files are in which state is the git status command. If we run this command directly after a clone, we should see something like this:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

This means we have a clean working directory - in other words, no tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells us which branch we're on.

In our case, we get:



```
MINGW32:/c/MyGit/GitProject
admin@KHONG /c/MyGit/GitProject (master)
$ pwd
/c/MyGit/GitProject

admin@KHONG /c/MyGit/GitProject (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.txt
    test.txt

nothing added to commit but untracked files present (use "git add" to track)

admin@KHONG /c/MyGit/GitProject (master)
$
```

We can see we are on **master** branch and we have **untracked** two files (in red color). Untracked basically means that Git sees a file we didn't have in the previous snapshot (commit); Git won't include it in our commit snapshots until we explicitly tell it to do so. It does this so that we don't accidentally begin including generated binary files or other files that we did not mean to include. Let's start tracking the files.

add

If we want to version-control existing files (as opposed to an empty directory), we should probably begin tracking those files and do an initial commit. We can accomplish that with a few **git add** commands that specify the files we want to track, followed by a **commit**:

With "**git add .**", we can add all files in the folder and make them be tracked.

```
MINGW32:/c/MyGit/GitProject
$ git add .
warning: LF will be replaced by CRLF in README.txt.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.

$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  README.txt
    new file:  test.txt

$ aachin@KHONG /c/MyGit/GitProject (master)
```

Now the files are displayed in green and git tells us both of them are new and they are now **tracked and staged**. Note that the two files are before the commit and the snapshot hasn't occurred yet.

We can tell that they staged because they're under the "Changes to be committed" heading. If we commit at this point, the version of the files at the time we ran 'git add' are what will be in the historical snapshot.

We may recall that when we ran 'git init' earlier, we then ran git add (files) - that was to begin tracking files in our directory. The git add command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

If we want to remove "test.txt", we do **git rm --cached test.txt**:



```
MINGW32:/c/MyGit/GitProject
$ git rm --cached test.txt
rm 'test.txt'
admin@KHONG:/c/MyGit/GitProject (master)
$ git status
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file: README.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  test.txt

admin@KHONG:/c/MyGit/GitProject (master)
$
```

config email and name

Before we commit, we need to configure email and name:



```
MINGW32:/c/MyGit/GitProject
admin@KHONG:/c/MyGit/GitProject (master)
$ git config --global user.email "k@bogotobogo.com"
admin@KHONG:/c/MyGit/GitProject (master)
$ git config --global user.name "k"
admin@KHONG:/c/MyGit/GitProject (master)
$
```

Commit

We now want to commit:

```
MINGW32:/c/MyGit/GitProject
$ git commit -m "Initial commit"
[master (root-commit) 577920d] Initial commit
 1 file changed, 2 insertions(+)
 create mode 100644 readme.txt
```

We can check what's been done so far by issuing log:

```
MINGW32:/c/MyGit/GitProject
admin@KHONG /c/MyGit/GitProject (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

nothing added to commit but untracked files present (use "git add" to track)
admin@KHONG /c/MyGit/GitProject (master)
$ git log
commit 577920d2677bd2c478bf4828dce87d893b7711d3
Author: k <k@bogotobogo.com>
Date:   Sun Sep 14 13:11:21 2014 -0700
    initial commit
admin@KHONG /c/MyGit/GitProject (master)
$
```

Git shows the name, email, and the message for the operation.

The other file, "test.txt" is not yet being tracked, so we want add that in.

```

MINGW32:/c/MyGit/GitProject
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

nothing added to commit but untracked files present (use "git add" to track)

admin@KHONG:/c/MyGit/GitProject (master)
$ git add "test.txt"
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.
admin@KHONG:/c/MyGit/GitProject (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   test.txt

admin@KHONG:/c/MyGit/GitProject (master)
$ git commit -m "committing test.txt"
[master b284204] committing test.txt
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.
 1 file changed, 2 insertions(+)
 create mode 100644 test.txt
admin@KHONG:/c/MyGit/GitProject (master)
$
```

If we do "git log" again, it will show two commits we've made so far:

```

MINGW32:/c/MyGit/GitProject
$ git log
commit b28420400e77dfb7c52302c206da22780d24fd37
Author: k <k@bogotobogo.com>
Date:  Sun Sep 14 13:20:26 2014 -0700

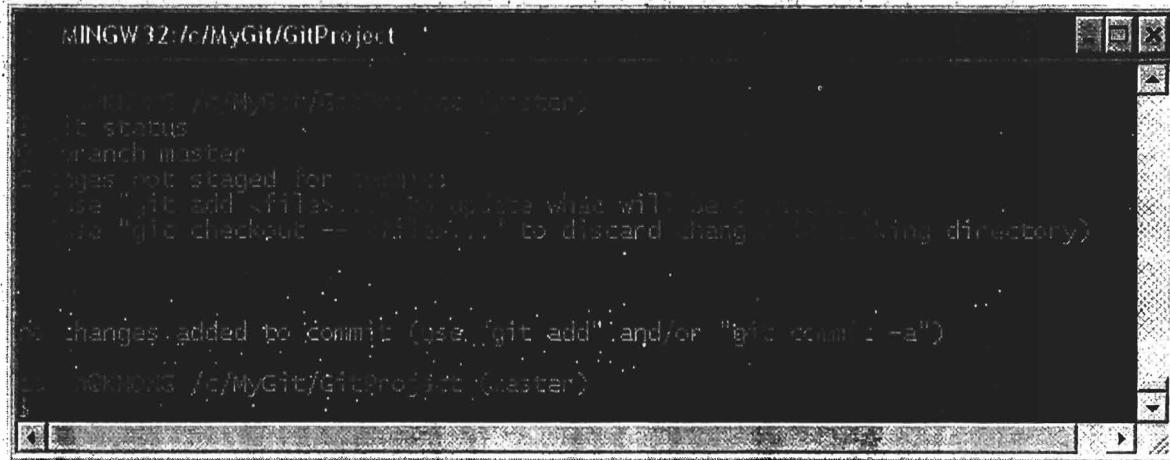
  committing test.txt

commit 577920d2677bd2c478bf4828dca87d893b7711d3
Author: k <k@bogotobogo.com>
Date:  Sun Sep 14 13:11:21 2014 -0700

  initial commit
admin@KHONG:/c/MyGit/GitProject (master)
$
```

Modify

Now, let's modify "readme.txt", and run status:



```
MINGW32:/c/MyGit/GitProject
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>" to update what will be committed)
    modified:   readme.txt

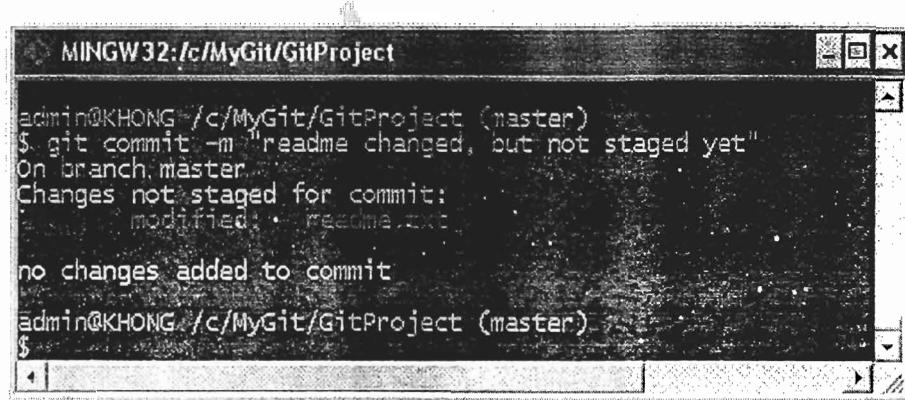
no changes added to commit (use "git add" and/or "git commit -a")
admin@KHONG:/c/MyGit/GitProject (master)
```

In this case, we changed a file that was already tracked.

If we change a previously tracked file and then run our status command again, we get something that looks like the picture above:

The "readme.txt" file appears under a section named "Changes not staged for commit" - which means that a file that is tracked has been modified in the working directory but not yet staged.

At this point, if we run **commit**, it will fail:

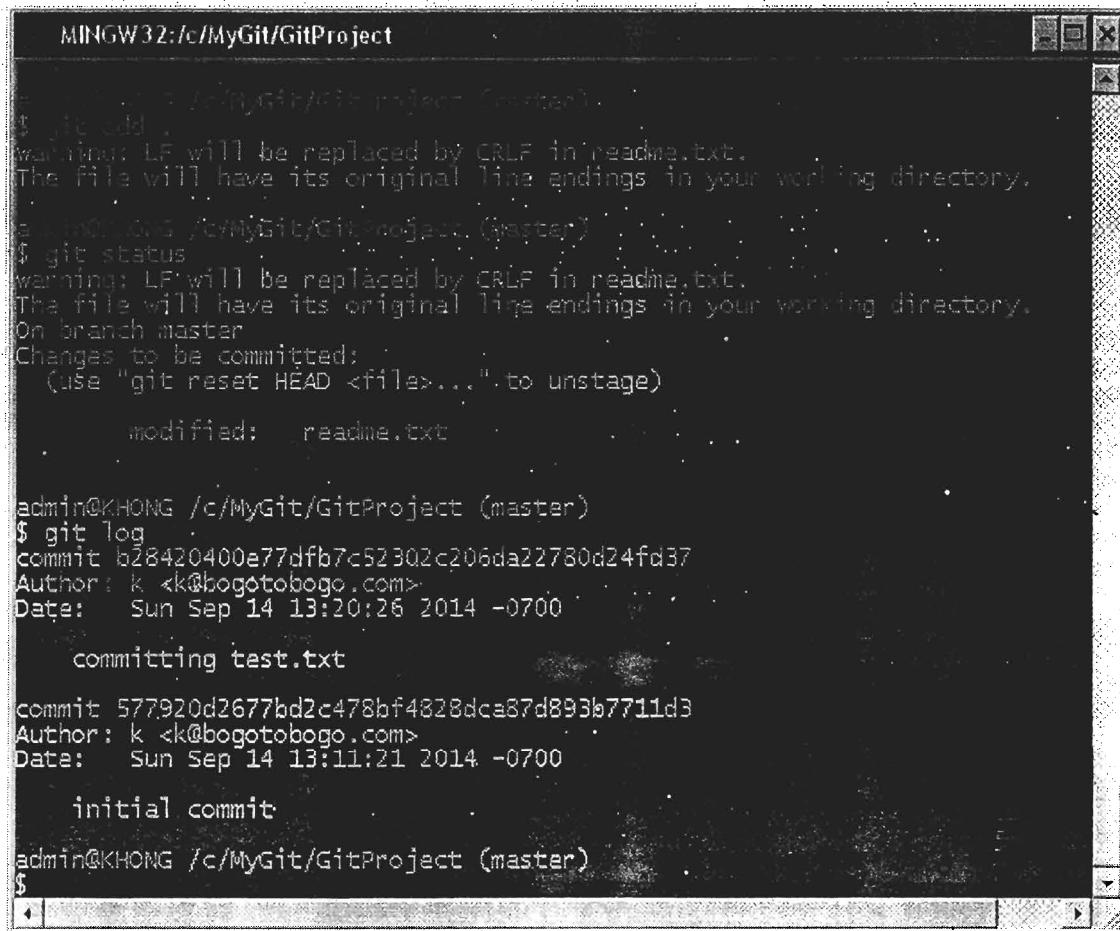


```
MINGW32:/c/MyGit/GitProject
$ git commit -m "readme changed, but not staged yet"
On branch master
Changes not staged for commit:
  (use "git add <file>" to update what will be committed)
    modified:   readme.txt

no changes added to commit
admin@KHONG:/c/MyGit/GitProject (master)
$
```

It failed because it's not at staging area. So, whenever we do commit, we should put a file at the staging area bef the commit.

In other words, to stage it, we run the **git add** command (it's a multipurpose command - we use it to begin tracking new files, to stage files, and to do other things like marking merge - conflicted files as resolved).



MINGW32:/c/MyGit/GitProject

```
$ git add README.txt
warning: LF will be replaced by CRLF in README.txt.
The file will have its original line endings in your working directory.

admin@KHONG /c/MyGit/GitProject (master)
$ git status
warning: LF will be replaced by CRLF in README.txt.
The file will have its original line endings in your working directory.
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.txt

admin@KHONG /c/MyGit/GitProject (master)
$ git log
commit b28420400e77dfb7c52302c206da22780d24fd37
Author: k <k@bogotobogo.com>
Date:   Sun Sep 14 13:20:26 2014 -0700

  committing test.txt

commit 577920d2677bd2c478bf4828dca87d893b7711d3
Author: k <k@bogotobogo.com>
Date:   Sun Sep 14 13:11:21 2014 -0700

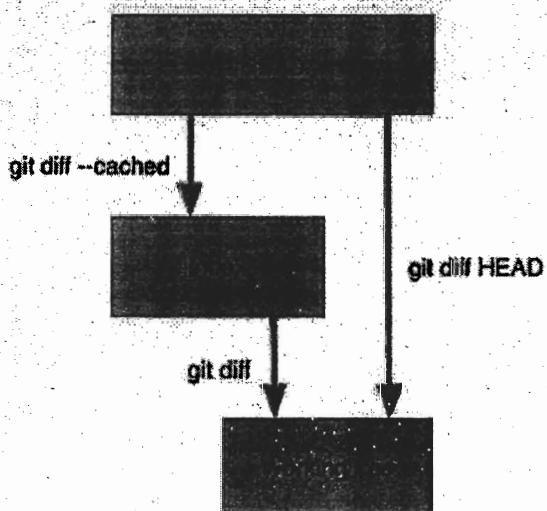
  initial commit

admin@KHONG /c/MyGit/GitProject (master)
$
```

By running **add**, we did put the file into the staging area:

Now, we can commit the changes made to the "readme.txt" file:

Git diff



Usually, since the **git status** command is too vague and does not give enough information about the change, when we want to know exactly what we changed not just which files were changed, we can use the **git diff** command. We'll probably use it most often to answer these two questions: What have we changed but not yet staged? And what have we staged that we are about to commit? Although **git status** answers those questions very generally, **gitdiff** shows us the exact lines added and removed.

Let's say we edit and stage the readme file again and then edit the readme file without staging it. If we run status command, we see something like this:

```
MINGW32:/c/MyGit/GitProjects
```

```

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file: README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

```

```
KHyuck@KHONG /c/MyGit/GitProjects (master)
```

If we want to see what we've changed but not yet staged, type **git diff** with no other arguments:

```
MINGW32:/c/MyGit/GitProjects
```

```
$ git diff
diff --git a/readme.txt b/readme.txt
index 9660950..ab3f3ad 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,4 +1,5 @@
 readme 1
 readme 2
 readme 3
+readme 4
+readme 5
@@ -1,4 +1,5 @@
 \ No newline at end of file
+readme 4
+readme 5
\ No newline at end of file
```

```
KHyuck@KHONG /c/MyGit/GitProjects (master)
```

git diff command compares what is in our working directory with what is in our staging area. The result tells us the changes we've made that we haven't yet staged.

If we want to see what we've staged that will go into our next commit, we can use **git diff --cached**. (In Git versions 1.6.1 and later, we can also use **git diff --staged**, which may be easier to remember.) This command compares our staged changes to our last commit:



```
MINGW32:/c/MyGit/GitProjects
$ git diff --cached
diff --git a/readme.txt b/readme.txt
index d19d403..050f0a8 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,3 +1,3 @@
readme 1
readme 2
readme 3
$
```

It's important to note that `git diff` by itself doesn't show all changes made since our last commit - only changes that are still **unstaged**. This can be confusing, because if we've staged all of our changes, `git diff` will give us no output as shown in the picture below:

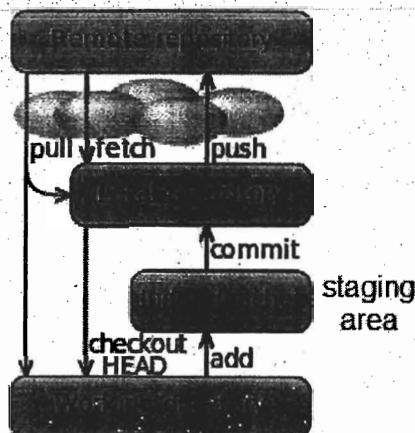


```
MINGW32:/c/MyGit/GitProjects
$ git add readme.txt
$ status
$ sh.exe": status: command not found
$
```

Git commit

Though we learned how we commit in the previous chapter, in this chapter, we'll review it again.

Now that our staging area is set up the way we want it, we can commit our changes. Remember that anything that is still **unstaged** - any files we have created or modified but we haven't run `git add` after we edited them - won't go into this commit. They will stay as modified files on our disk.



When we run **git status**, if we see that everything has been staged, then we're ready to commit our changes. The simplest way to commit is to type **git commit**:

```
$ git commit
```

The commit gives us some output about itself: which branch we committed to (master), what SHA-1 checksum the commit has (463dc4f), how many files were changed, and statistics about lines added and removed in the commit.

```

MINGW32:/c/MyGit/GitProjects
$ git add .
# Changes to be committed:
#   (use "git reset HEAD <file>" to unstage)

nothing staged for commit.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   README

KHuyck@KHONG /c/MyGit/GitProjects (master)
$ git add README
KHuyck@KHONG /c/MyGit/GitProjects (master)
$ git commit -m "commit README changes"
[master 0ef93d2] commit README changes
 1 file changed, 3 insertions(+), 1 deletion(-)
KHuyck@KHONG /c/MyGit/GitProjects (master)

```

Remember that the **commit** records the **snapshot** we set up in our **staging area**. Anything we didn't stage is still sitting there modified; we can do another commit to add it to our history. Every time we perform a **commit**, we're recording a snapshot of our project that we can revert to or compare to later.

Deleting files

Deleting files from our repo is simple, delete them, and then commit.

We have 4 files in repository: Book1, Book2, Book3, and OldBook.

```

k@laptop:~/GitDemo$ ls
Book1.rtf      Book2.rtf      Book3.rtf      OldBook.rtf

```

```
k@laptop:~/GitDemo$ git status
```

On branch master

nothing to commit, working directory clean

```
k@laptop:~/GitDemo$
```

Now, we want to remove "OldBook" which is already in our repository.

```
k@laptop:~/GitDemo$ gitrm OldBook.rtf
```

```
k@laptop:~/GitDemo$ git status
```

On branch master

Changes not staged for commit:

(use "git add/rm ..." to update what will be committed)

(use "git checkout -- ..." to distestbranchd changes in working directory)

deleted: OldBook.rtf

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

But our repository still has the 'OldBook'. So, we need to do commit the deleted file:

```
k@laptop:~/GitDemo$ git commit -am "deleted OldBook"
```

```
[master 7a4caa8] deleted OldBook
```

1 file changed, 9 deletions(-)

```
delete mode 100644 OldBook.rtf
```

Now, it's gone and the 'OldBook' became a history. So, deleting a file from a repo is straight forward: delete and commit.

Renaming files

We have two ways of renaming a file in our repository.

1. Renaming a file in a working folder.

Now we want to rename the 'Book3' to 'Appendix'.

First, we directly rename the file in our working directory:

```
k@laptop:~/GitDemo$ mv Book3.rtf Appendix.rtf  
k@laptop:~/GitDemo$ ls  
Appendix.rtf      Book1.rtf      Book2.rtf
```

If we check the git status:

```
k@laptop:~/GitDemo$ git status
```

On branch master

Changes not staged for commit:

(use "git add/rm ..." to update what will be committed)

(use "git checkout -- ..." to testbranchd changes in working directory)

deleted: Book3.rtf

Untracked files:

(use "git add ..." to include in what will be committed)

Appendix.rtf

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

The git thinks the 'Book3' is removed and 'Appendix' is added. Now we should remove 'Book3' and add 'Appendix' on repo:

```
k@laptop:~/GitDemo$ gitrm Book3.rtf  
rm 'Book3.rtf'  
k@laptop:~/GitDemo$ git add Appendix.rtf
```

Then, check the status again:

```
k@laptop:~/GitDemo$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

renamed: Book3.rtf -> Appendix.rtf

Notice that Git is smart enough to realize what we did was just renaming the file!

All we have to do now is to commit:

```
k@laptop:~/GitDemo$ git commit -m "renamed Book3 as Appendix"  
[master 8d7d7f1] renamed Book3 as Appendix  
1 file changed, 0 insertions(+), 0 deletions(-)  
rename Book3.rtf => Appendix.rtf (100%)
```

2. Using "git mv" command

This time we want to rename 'Book2' to "Introduction".

```
k@laptop:~/GitDemo$ git mv Book2.rtf Introduction.rtf
```

Now git immediately recognizes we renamed it:

```
k@laptop:~/GitDemo$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

renamed: Book2.rtf -> Introduction.rtf

All we have to do now is doing commit:

```
k@laptop:~/GitDemo$ git commit -m "Book2->Introduction"
```

```
[master e1ec728] Book2->Introduction
```

1 file changed, 0 insertions(+), 0 deletions(-)

rename Book2.rtf => Introduction.rtf (100%)

```
k@laptop:~/GitDemo$ git status
```

On branch master

nothing to commit, working directory clean

gitinit&gitconfig

We have "README" in our GitProject directory:



```
MINGW32:/c/MyGit/GitProject
$ ls -la
total 1
drwxr-xr-x  1 admin  Administ  0 Sep 14 23:36 .
drwxr-xr-x  3 admin  Administ  0 Sep 13 23:55 ..
-rw-r--r--  1 admin  Administ 21 Sep 14 23:05 README
admin@KHONG:/c/MyGit/GitProject
$
```

Then we need to issue **gitinit** command:



```
MINGW32:/c/MyGit/GitProject
admin@KHONG /c/MyGit/GitProject
$ git init
Initialized empty Git repository in c:/MyGit/GitProject/.git/
admin@KHONG /c/MyGit/GitProject (master)
$ ls -la .git
total 2
drwxr-xr-x  9 admin  Administ  0 Sep 14 23:47 .
drwxr-xr-x  1 admin  Administ  0 Sep 14 23:47 ..
-rw-r--r--  1 admin  Administ 23 Sep 14 23:47 HEAD
-rw-r--r--  1 admin  Administ 157 Sep 14 23:47 config
-rw-r--r--  1 admin  Administ 73 Sep 14 23:47 description
drwxr-xr-x 11 admin  Administ  0 Sep 14 23:47 hooks
drwxr-xr-x  3 admin  Administ  0 Sep 14 23:47 info
drwxr-xr-x  4 admin  Administ  0 Sep 14 23:47 objects
drwxr-xr-x  4 admin  Administ  0 Sep 14 23:47 refs
admin@KHONG /c/MyGit/GitProject (master)
$
```

As we can see the **gitinit** command creates an empty Git repository - basically a **.git** directory with subdirectories for **objects**, **refs/heads**, **refs/tags**, and **template** files. An initial **HEAD** file that references the **HEAD** of the **master** branch is also created.

We may also want to do email and name configuration using **gitconfig** command:

MINGW32:/c/MyGit/GitProject

```
$ git config --global user.name "Kong getobaya.com"  
$ git config --global user.email "kong@getobaya.com"  
$ git config --global user.name "K"  
$ cd /c/MyGit/GitProject (master)
```

Add & Commit

Now we want to put our "README" to staging area and then do commit (take snapshot).

```
MINGW32:/c/MyGit/GitProject  
  
admin@KHONG /c/MyGit/GitProject (master)  
$ git add README  
warning: LF will be replaced by CRLF in README.  
The file will have its original line endings in your working directory.  
  
admin@KHONG /c/MyGit/GitProject (master)  
$ git commit -m "initial commit"  
[master (root-commit) 4aece6d] initial commit  
warning: LF will be replaced by CRLF in README.  
The file will have its original line endings in your working directory.  
1 file changed, 2 insertions(+)  
create mode 100644 README  
  
admin@KHONG /c/MyGit/GitProject (master)  
$ git status  
On branch master  
nothing to commit, working directory clean  
  
admin@KHONG /c/MyGit/GitProject (master)  
$
```

adding remote repo

Up to this point, we've been working on local repo, but now it's time to use remote repo which is GitHub. To use remote repo, we need to tell git what remote repo we're going to use via **git remote add** command:

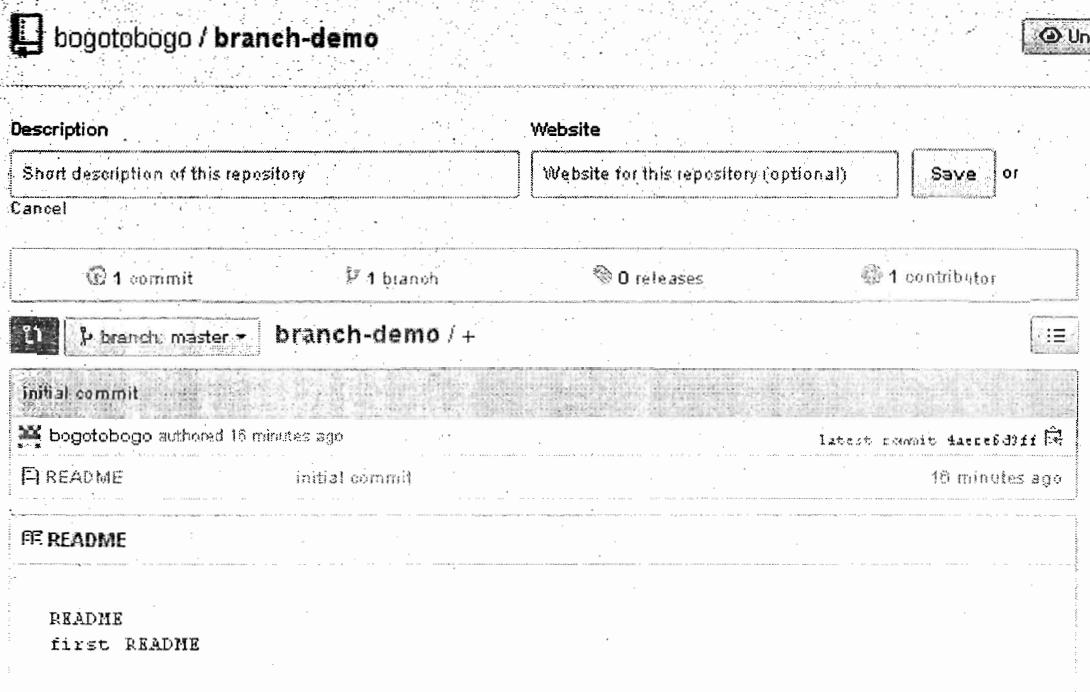
```
MINGW32:/c/MyGit/GitProject
admin@KHONG:/c/MyGit/GitProject (master)
$ git remote -v
admin@KHONG:/c/MyGit/GitProject (master)
$ git remote add origin git@github.com:bogotobogo/branch-demo.git
admin@KHONG:/c/MyGit/GitProject (master)
$ git remote -v
origin	git@github.com:bogotobogo/branch-demo.git (fetch)
origin	git@github.com:bogotobogo/branch-demo.git (push)
admin@KHONG:/c/MyGit/GitProject (master)
$
```

git push

Since we can use remote repo name as **origin**, let upload our README to our GitHub:

```
MINGW32:/c/MyGit/GitProject
admin@KHONG:/c/MyGit/GitProject (master)
$ git push origin master
Warning: Permanently added the RSA host key for IP address '192.30.252.131' to the list of known hosts.
Counting objects: 3, done.
Writing objects: 100% (3/3), 217 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:bogotobogo/branch-demo.git
 * [new branch]      master -> master
admin@KHONG:/c/MyGit/GitProject (master)
$
```

We can check whether the README file is really in GitHub:



Description

Short description of this repository

Website

Website for this repository (optional)

Save or **Cancel**

1 commit 1 branch 0 releases 1 contributor

b1 **b1** branch master → **branch-demo / +**

initial commit

bogotobogo authored 16 minutes ago

latest commit **4ace6d0ff** **fix**

README initial commit 16 minutes ago

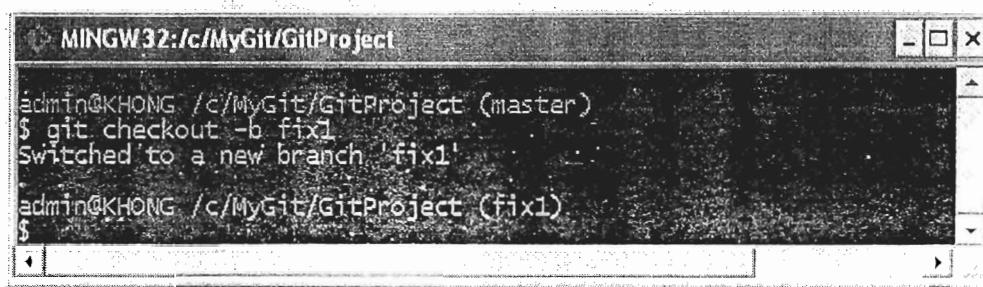
README

README
first README

branching & merging

git checkout

git checkout -b <new_branch> will create a new branch (the "-b" causes a new branch to be created) with the name of "new_branch":



```
MINGW32:/c/MyGit/GitProject
$ git checkout -b fix1
Switched to a new branch 'fix1'
$
```

We can check which branch we're now by using **git branch**:



```
MINGW32:/c/MyGit/GitProject
$ git branch
* fix1
  master
```

As expected, we're on the new branch, 'fix1'. Now, we want to modify the README like this:

```
README
```

```
first README fix1
```

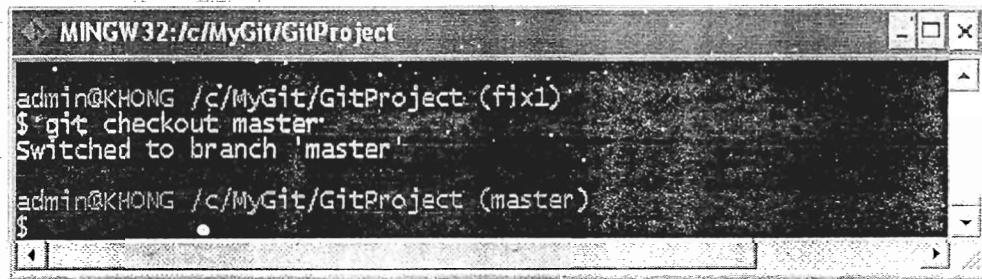
To add the change and commit:

```
$ git commit -a -m "Added branch fix1"
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

Note that we committed the change only to the branch not to the master.

Now we can go back (switch) to the master:



```
MINGW32:/c/MyGit/GitProject
$ git checkout master
Switched to branch 'master'
$
```

branchgit push

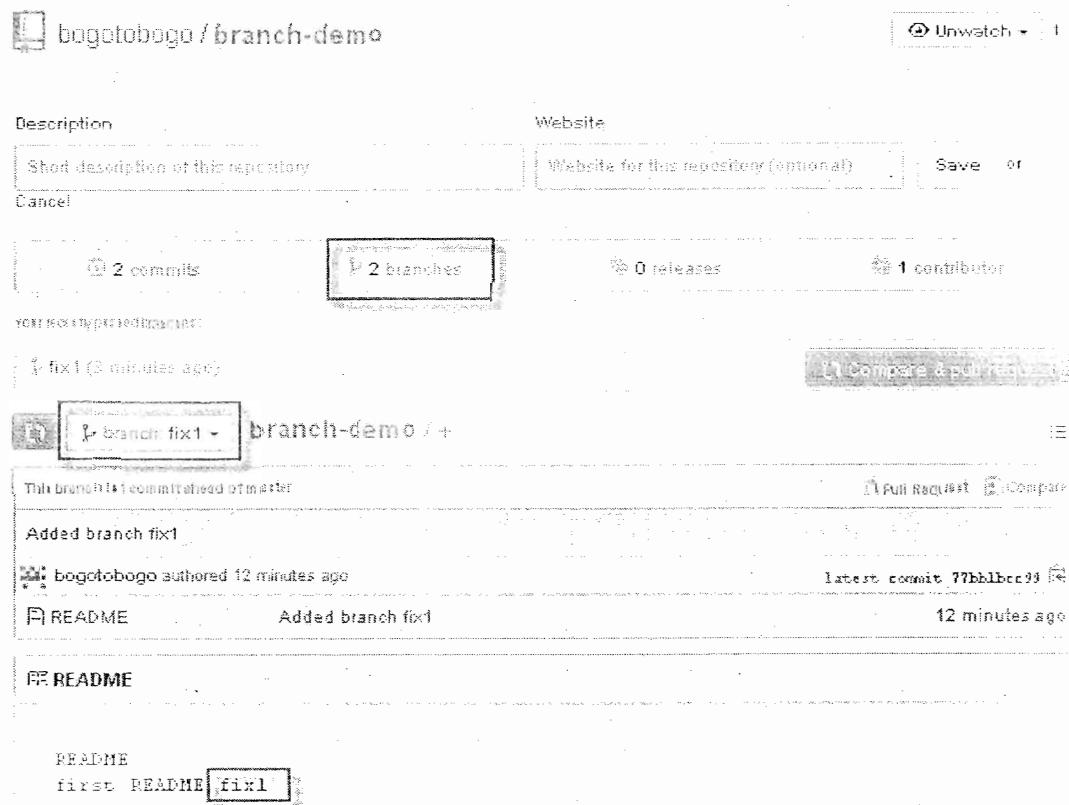
We can push our branch to GitHub:

```
MINGW32:/c/MyGit/GitProject

$ git push origin fix1
Warning: Permanently added the RSA host key for IP address '192.30.252.130' to the list of known hosts.
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), 251 bytes | 0 bytes/s, done.
Writing objects: 100% (3/3), 251 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:bogotobogo/branch-demo.git
 * [new branch]      fix1 -> fix1

admin@KHONG /c/MyGit/GitProject (master)
```

If we check our GitHub, it now has two branches and the newly added fix1 branch shows our change:



bogotobogo / branch-demo

Description

Short description of this repository

website

Save or

Cancel

2 commits

2 branches

0 releases

1 contributor

fix1 (3 minutes ago)

branch fix1 - branch-demo / +

This branch has been initialized or master

Added branch fix1

bogotobogo authored 12 minutes ago

README Added branch fix1

Latest commit 77bb1bcc99 12 minutes ago

Full Request Compare

README

README first README fix1

listing branches

To see which branch we're now on:

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch
```

```
fix1
```

```
* master
```

```
admin@JRAYA /c/MyGit/GitProject (master)
```

To see the merged branch:

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch --merged
```

```
* master
```

```
merged
```

```
admin@JRAYA /c/MyGit/GitProject (master)
```

To list unmerged branch:

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch --no-merged
```

```
fix1
```

```
admin@JRAYA /c/MyGit/GitProject (master)
```

To see all the branches and the last commit:

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch -v
```

```
fix1 77bb1bc Added branch fix1
```

```
* master 4aece6d initial commit
```

```
merged 4aece6d initial commit
```

```
admin@JRAYA /c/MyGit/GitProject (master)
```

merge

We're now on *master, and check the README:

```
README
```

```
first README
```

To merge with fix1:



MINGW32:/c/MyGit/GitProject

```
admin@KHONG /c/MyGit/GitProject (master)
$ git branch
* master
  fix1
  merged

admin@KHONG /c/MyGit/GitProject (master)
$ git merge fix1
Updating 4aaca6d..77bb1bc
Fast-forward
 README | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)

admin@KHONG /c/MyGit/GitProject (master)
$
```

Now the README file has been merged on master:

README

first README fix1

deleting branch

Let's check our branches again:

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch
```

fix1

* master

merged

The fix1 branch is there but we do not need it any more since it's been merged. So, let's remove it:

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch -d fix1
```

Deleted branch fix1 (was 77bb1bc).

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
$ git branch
```

* master

merged

```
admin@JRAYA /c/MyGit/GitProject (master)
```

If we want to delete unmerged branch, we can use "git branch -D name_of_unmerged_branch".

Deleting github branch

How we can delete a branch on our GitHub?

```
admin@JRAYA /c/MyGit/GitProject (master)
```

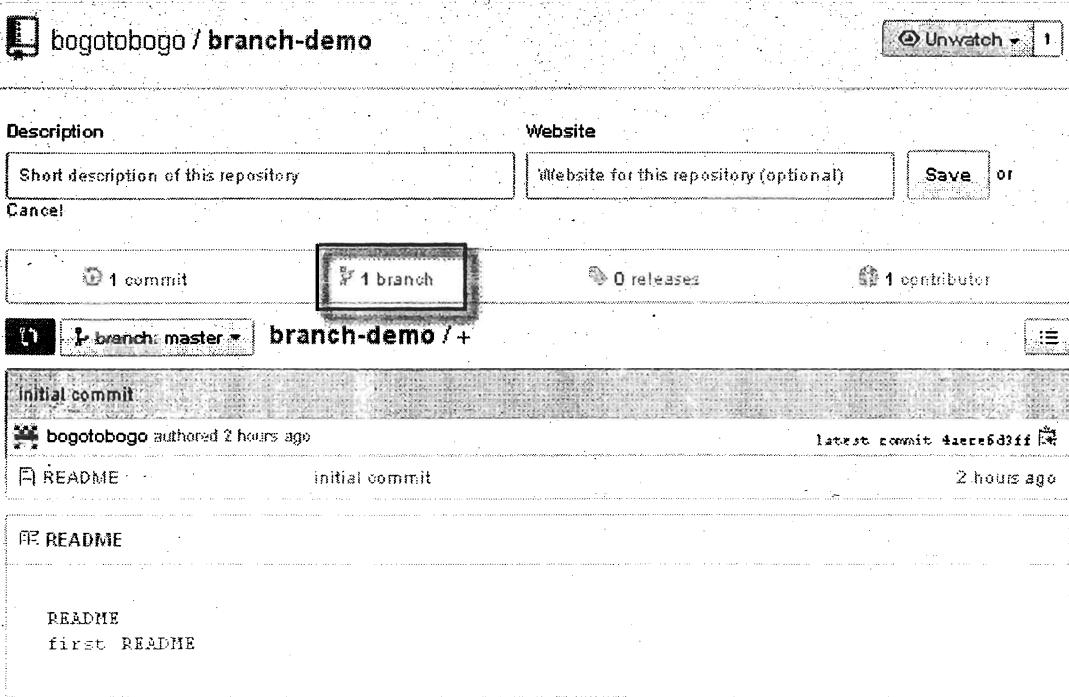
```
$ git push origin :fix1
```

To git@github.com:test/branch-demo.git

- [deleted] fix1

```
admin@JRAYA /c/MyGit/GitProject (master)
```

Let's check how our GitHub looks like:



The screenshot shows the GitHub repository settings for 'branch-demo'. It includes fields for 'Description' and 'Website', and buttons for 'Save' and 'Cancel'. Below these are statistics: 1 commit, 1 branch (which is highlighted), 0 releases, and 1 contributor. The main repository view shows a single commit from 'bogotobogo' labeled 'initial commit' made 2 hours ago. A file named 'README' is visible.

Our GitHub now has only one branch which is master since another branch fix1 has been deleted.

Now, we need to push our merge local repo to remote repo:

```
admin@JRAYA /c/MyGit/GitProject (master)
$ git push origin master
Counting objects: 5, done.
Writing objects: 100% (3/3), 254 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:test/branch-demo.git
  4aece6d..77bb1bc  master -> master
```

```
admin@JRAYA /c/MyGit/GitProject (master)
```

```
 README
```

```
 README  
 first README fix1
```

GIT AND GITHUBMERGE CONFLICTS WITH SIMPLE EXAMPLE

Creating conflicts

In the previous chapter (Fast-forward merge), the merge was very easy because we did not have any conflicts. In other words, in that case, the master branch (trunk) has no updates since we branched 'testbranch' off the 'master'. All we had to do was simply appending our new updates to the master trunk.

In this chapter, we'll create conflicts by updating our 'master' branch at the same time we update the 'testbranch' branch. So, when we try to merge the 'testbranch' branch, we'll have two different versions of 'Book1', and git has no way to figure out which one to take in.

We are currently on the 'master' branch and we have another branch called 'testbranch':

```
k@laptop:~/GitDemo$ ls
```

Appendix Book1 Introduction

```
k@laptop:~/GitDemo$ git branch
```

testbranch

* master

The log:

```
k@laptop:~/GitDemo$ git log --oneline  
  
d567a72 Merge branch 'testbranch'  
  
4552553 Added 2nd commit to the testbranch branch  
  
...  
  
b440952 initial commit
```

Check if it's clean:

```
k@laptop:~/GitDemo$ git status  
  
On branch master  
  
nothing to commit, working directory clean
```

Let's modify 'Book1' on the 'master' by adding the following as its first line:

Year 2014 - updated master branch

Of course, we need to commit it:

```
k@laptop:~/GitDemo$ git add .  
  
k@laptop:~/GitDemo$ git commit -m "updated 1st line 2014 on master branch"  
  
[master 79b0888] updated 1st line 2014 on master branch  
  
1 file changed, 1 insertion(+)
```

Updated log:

```
k@laptop:~/GitDemo$ git log --oneline
```

79b0888 updated 1st line 2014 on master branch

d567a72 Merge branch 'testbranch'

4552553 Added 2nd commit to the testbranch branch

a7f55e3 added a line at the beginning

...

k@laptop:~/GitDemo\$ git status

On branch master

nothing to commit, working directory clean

k@laptop:~/GitDemo\$ git branch

testbranch

*** master**

Then, we switch to the 'testbranch' branch.

k@laptop:~/GitDemo\$ git checkout testbranch

Switched to branch 'testbranch'

k@laptop:~/GitDemo\$ git branch

*** testbranch**

master

k@laptop:~/GitDemo\$ git status

On branch testbranch

nothing to commit, working directory clean

DEVOPS MATERIAL

Then, add the similar line as we've done on the 'master' as its first line:

Year 2014 - updated testbranch branch

Commit:

```
k@laptop:~/GitDemo$ gitadd .
```

```
k@laptop:~/GitDemo$ git commit -m "updated 1st line 2015 on testbranch branch"
```

```
[testbranch a428001] updated 1st line 2015 on testbranch branch
```

```
1 file changed, 1 insertion(+)
```

Log for the 'testbranch' branch:

```
k@laptop:~/GitDemo$ git log --oneline
```

```
a428001 updated 1st line 2015 on testbranch branch
```

```
4552553 Added 2nd commit to the testbranch branch
```

```
a7f55e3 added a line at the beginning
```

Fixing Merge conflicts

Now we have two different versions of 'Book1' that were committed to each branch, 'master' and 'testbranch'.

We're now on 'master' branch:

```
k@laptop:~/GitDemo$ git branch
```

```
testbranch
```

```
* master
```

Let's try to merge them;

```
k@laptop:~/GitDemo$ git merge testbranch
```

Auto-merging Book1

CONFLICT (content): Merge conflict in Book1

Automatic merge failed; fix conflicts and then commit the result.

```
k@laptop:~/GitDemo$
```

Basically, git is saying, "I don't know which one is the most up-to-date version. It's your job to fix it".

Note that at this merging phase, we're not on any specific branch but we're somewhere "floating land" which is not 'master' nor 'testbranch'.

But when we open the 'Book1' in our working directory, it shows something like this:

```
k@laptop:~/GitDemo$ git merge testbranch
```

Auto-merging Book1

CONFLICT (content): Merge conflict in Book1

Automatic merge failed; fix conflicts and then commit the result.

```
k@laptop:~/GitDemo$
```

Clearly, it's not the one from 'master' nor from 'testbranch'.

```
<<<<< HEAD
```

Year 2014 - updated master branch

Year 2015 - updated testbranch branch

>>>>>testbranch

It shows the difference in the two versions of 'Book1'. We're lucky because our sample is simple enough to fix manually, however, in real cases, it could become really messy, and that's the reason why we should minimize the conflicts by commit as often as possible.

Ok, let's combine the conflicts like this:

Year 2015 - updated by master &testbranch at the same time

Save it and then commit:

k@laptop:~/GitDemo\$ git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

(use "git add ..." to mark resolution)

both modified: Book1

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

```
k@laptop:~/GitDemo$ git add .
```

```
k@laptop:~/GitDemo$ git commit -m "fixed the conflict, set year = 2015"
```

```
[master 1ae8f09] fixed the conflict, set year = 2015
```

```
k@laptop:~/GitDemo$ git status
```

On branch master

nothing to commit, working directory clean

Now we resolved the conflicts. Let's look at the graph log:

```
k@laptop:~/GitDemo$ git log --graph --oneline
*   1ae8f09 fixed the conflict, set year = 2015
  \
  * a428001 updated 1st line 2015 on car branch
* | 79b0888 updated 1st line 2014 on master branch
* | d567a72 Merge branch 'car'
  \
  |
  * 4552553 Added 2nd commit to the car branch
  /
* a7f55e3 added a line at the beginning
```

Here is the reading from the graph log above:

1. "d567a72 Merge branch 'testbranch'" -

Since we branched off the 'testbranch', we did fast-forward merge in the previous chapter (Fast-forward merge)

2. "a428001 updated 1st line 2015 on testbranch branch", "79b0888 updated 1st line 2014 on master branch" -

Each branch has its own update.

3. " 1ae8f09 fixed the conflict, set year = 2015" -

Then, both branches merged.

QUICK COMMAND REFERENCE

remote means remote repository uri,
for example, <https://github.com/User/myrepo.git>

Clone & branches

1. To clone a remote git repository with depth = 1 and specifying my project name:

```
2. $ git clone --depth=1 remote myproject
```

3. To clone a remote git repository with specific branch (eg. master):

```
4. $ git clone -b master remote
```

5. To see the branches after cloned all branches (cloned w/o specifying any):

```
6. $ git branch -a
```

```
7. * master
```

```
8. remotes/origin/HEAD -> origin/master
```

```
9. remotes/origin/dev
```

```
10. remotes/origin/master
```

11. Switching branches (**checkout -b**):

```
12. $ git checkout -b dev
```

Switched to a new branch 'dev'

About remote repository

1. To setup local repo with a remote repositories.

First, we need to use **git init** and then use **git remote add remote-name remote-url**:

2. \$ git init

This will create .git directory into our current working directory. Now we want to add the remote repo:

```
$ git remote add origin https://github.com/User/repo.git
```

Here the 4th one (**origin**), we can name it whatever we want to. But **origin** is a kind of convention.

3. To see what are the remote repositories:

4. \$ git remote -v

5. origin https://github.com/User/repo.git (fetch)

6. origin https://github.com/User/repo.git (push)

If we want to use the same name, we can drop our project name which is the last argument.

7. To switch remote repositories from repo1 to repo2:

This is my current remote:

8. \$ git remote -v

9. origin https://github.com/User/repo1.git (fetch)

10. origin https://github.com/User/repo1.git (push)

Here it is:

```
$ git remote set-url origin https://github.com/repo2.git
```

Note that we overwrote (replaced with a new remote) an existing one.

If we wanted to keep the old one, we could have used **add** instead of **set-url**:

```
$ git remote add origin2 https://github.com/repo2.git
```

Then, we have two remote repos:

```
$ git remote -v
origin  https://github.com/User/repo1.git (fetch)
origin  https://github.com/User/repo1.git (push)

origin2 https://github.com/User/repo2.git (fetch)
origin2 https://github.com/User/repo2.git (push)
```

Staging

There are couple of options when we do stage files.

1. Staging only "modified (including deleted)":
2. `$ git add -u`

or

```
$ git add --update
```

If we want to do commit at one shot:

```
$ git commit -a
```

3. Staging all (modified + new + deleted):
"git add -A" is equivalent to "git add --all"

4. **\$ git add -A**

or

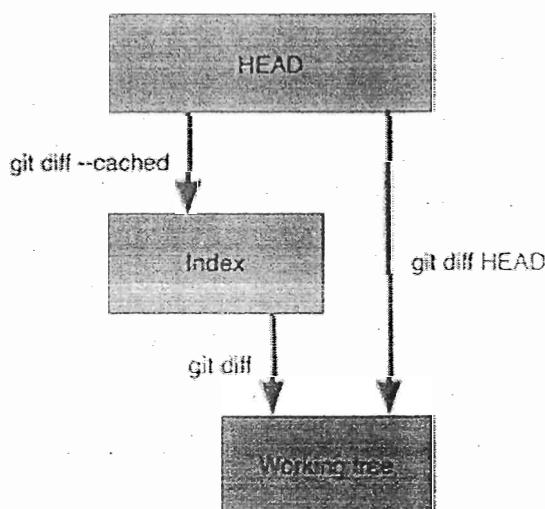
\$ git add --all

As we can see from the table below, as far as Git 2, there is not difference "git add -A" and "git add ."

	New Files	Modified Files	Deleted Files	
git add -A	✓	✓	✓	Stage All (new, modified, deleted) files
git add .	✓	✓	✓	Stage All (new, modified, deleted) files
git add --ignore-removal .	✓	✓	✗	Stage New and Modified files only
git add -u	✗	✓	✓	Stage Modified and Deleted files only

Source : Difference between "git add -A" and "git add ."

Diff



Just for demonstration purpose, I put my file (a.txt) in several stage with a text of that stage name:

1. A file on working directory (working tree) - "My Working directory file"
2. A file in staging area - "My Staged file"
3. A file already committed - "My HEAD file"
4. A file already pushed to remote - "My Remote file"

So, when we issue a diff command, we can see the diff more clearly.

1. After we staged a file, and want to see the diff of the file from the one that's already in the HEAD (local repo):

2. **\$ git diff --staged a.txt**
3. **-My HEAD file**
4. **+My Staged file**

5. Worked on a file that's on our working directory (or working tree), and want to see the diff from the one already staged:

6. **\$ git diff a.txt**
7. **-My Staged file**
8. **+My Working directory file**

9. To see the diff of my file on working tree with the one already in the HEAD

10. **\$ git diff HEAD a.txt**
11. **-My HEAD file**
12. **+My Working directory file**

13. To see the diff of my file on working tree with the remote one.

To be more clear, we may want to know how the remote looks like:

14. **\$ git remote -v**

15. origin https://github.com/User/repo.git (fetch)
16. origin https://github.com/User/repo.git (push)

Now, let's do diff:

```
$ git diff origin/master:a.txt a.txt
```

-My remote file

+My Working directory file

We can switch the positions

```
$ git diff HEAD:a.txt origin/master:a.txt
```

-My HEAD file

+My remote file

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository frequently.

Each check-in is then verified by an automated build, allowing teams to detect problems early so that we can detect errors quickly, and locate them more easily.

Jenkins is an open source tool to perform continuous integration: monitor a version control system and to start a build system.

Jenkins monitors the whole build process and provides reports and notifications.

What is Jenkins?

Jenkins is an award-winning application that monitors executions of repeated jobs, such as building a software project or jobs run by cron. Among those things, current Jenkins focuses on the following two jobs:

1. Building/testing software projects continuously, just like CruiseControl or DamageControl. In a nutshell, Jenkins provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases the productivity.
2. Monitoring executions of externally-run jobs, such as cron jobs and procmail jobs, even those that are run on a remote machine. For example, with cron, all you receive is regular e-mails that capture the output, and it is up to you to look at them diligently and notice when it broke. Jenkins keeps those outputs and makes it easy for you to notice when something is wrong.

Adding Jenkins service to GitHub

Let's go back to GitHub, under "Settings"=>"Webhooks& Services"=>"Add services"=>"Add Jenkins (Git plugin)". Now, we're going to use the Git plugin, so what we want to do is to paste in the URL for our Jenkins server (<http://<jenkinsserverip>/>), and we'll make it active.



Install Notes

- Requires Git Plugin v1.1.18, released 2012-04-27, and the "Poll SCM" build trigger needs to be enabled. (Though you can have it poll very infrequently, I recommend something like 0 */3 * * *)
- "Jenkins Url" is the base URL of your Jenkins server. For example, <http://ci.jenkins-ci.org/>. We will hit `/git/notifyCommit` under this URL. (See the Git plugin wiki page for more details.)

Details

Jenkins is a popular continuous integration server.

If you're using the standard Jenkins Git plugin to poll & check out your repository, you can quickly and easily switch to a push model using this service.

It will send a request to your Jenkins instance telling it about the repositories and branches that changed. Jenkins will then poll the repository and build if needed. See push notification from repository on the Jenkins wiki for information.

Jenkins url

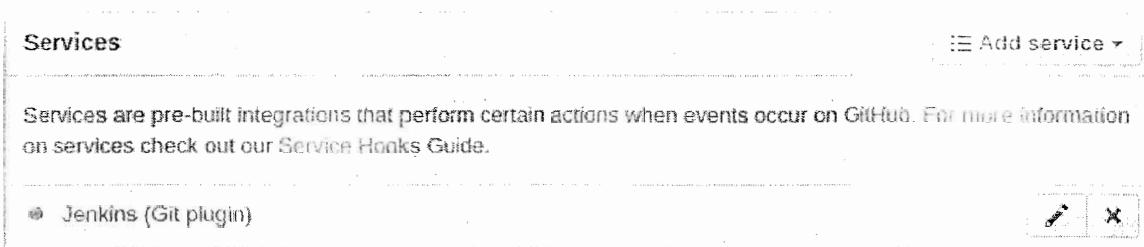
Active
 We will run this service whenever an event is triggered.

Add service

Let's take a look at the "Install Notes":

Requires Git Plugin v1.1.18, released 2012-04-27, and the "Poll SCM" build trigger needs to be enabled. (Though you can have it poll very infrequently, I recommend something like 0 */3 * * *)

Click on "Add service".



Services

Add service ▾

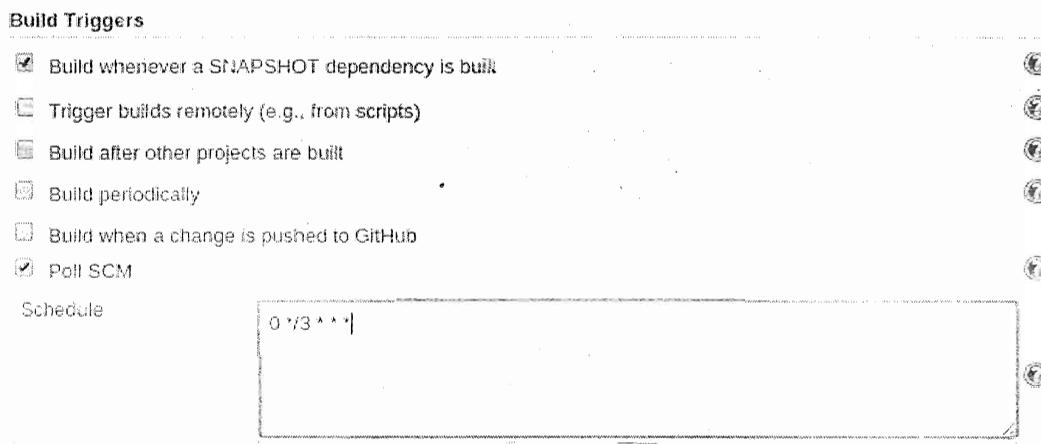
Services are pre-built integrations that perform certain actions when events occur on GitHub. For more information on services check out our Service Hooks Guide.

Jenkins (Git plugin)

Now we have Github configured to notify our Jenkins server everytime the changes pushed into the repository.

Configure Jenkins

Now we need to go over the Jenkins and do configure a bit. Let's go to the project and click on "Configure". We need to enable the poll SCM build option:



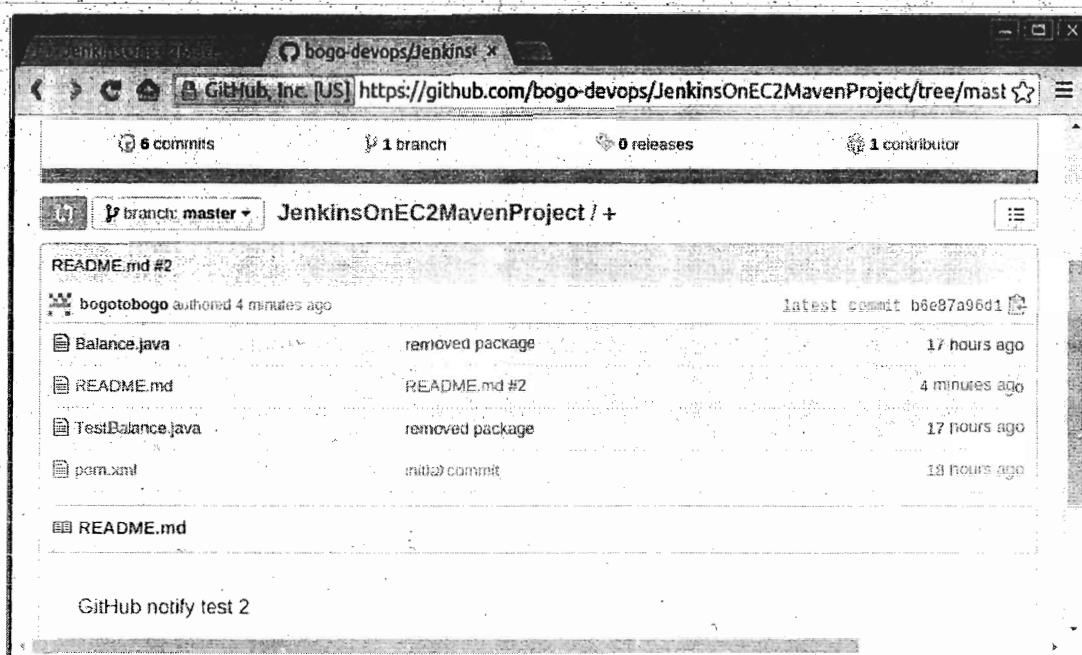
We set polling every 3 hour.

Click on "Save"

Push changes to GitHub

Let's try pushing updates to GitHub and make sure that it automatically notifies Jenkins and it starts to build for us.

We made a new README.md, and pushed to GitHub:



Right after the push, the "Build History" looks like this:

Build History	
#6	Nov 16, 2014 11:39:20 PM
#5	Nov 16, 2014 11:04:50 PM
#4	Nov 16, 2014 11:03:25 PM
#3	Nov 16, 2014 10:51:10 PM
#2	Nov 16, 2014 10:47:55 PM
#1	Nov 16, 2014 5:28:53 PM

We can see our Jenkins server got notification from GitHub repository.

Installing Jenkins on Ubuntu

We'll install the Jenkins on Ubuntu 14.04 system.

Let's use Debian package repository of Jenkins to automate installation and upgrade. First add the key to our system:

```
$ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
```

Then add the following entry in our `/etc/apt/sources.list`:

```
deb http://pkg.jenkins-ci.org/debian-stable binary/
```

Or one liner:

```
$ sudosh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

We need to update our local package index, and then finally install Jenkins:

```
$ sudo apt-get update
```

```
$ sudo apt-get install jenkins
```

Now we have `/etc/init.d/jenkins` start script which starts Jenkins automatically at boot time. In other words, Jenkins will be launched as a daemon up on start.

What does this package do?

1. Jenkins will be launched as a daemon up on start (`/etc/init.d/jenkins`).
2. The **jenkins** user is created to run this service.
3. Log file will be placed in `/var/log/jenkins/jenkins.log`. Check this file if troubleshooting Jenkins. `/etc/default/jenkins` will capture configuration parameters for the launch like e.g **JENKINS_HOME**.

4. By default, Jenkins listen on port **8080**. Access this port with browser to start configuration.

If we start it locally, we can see it running under **http://localhost:8080/** URL.

Just follow the instructions.

Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

/var/lib/jenkins/secrets/initialAdminPassword

Please copy the password from either location and paste it below.

Administrator password

.....

Continue

Getting Started

Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

Install suggested plugins

Install plugins the Jenkins community finds most useful

Select plugins to install

Select and install plugins most suitable for your needs.

Jenkins 2.9

Select "Suggested Plugins":

Getting Started

Folders Plugin

✓ Timestamper	✓ Workspace Cleanup Plugin	✓ Ant Plugin	✓ Gradle plugin
✓ Pipeline	✓ GitHub Organization Folder Plugin	✓ Pipeline: Stage View Plugin	✓ Git plugin
Subversion Plug-in	SSH Slaves plugin	✓ Matrix Authorization Strategy Plugin	✓ PAM Authentication plugin
✓ LDAP Plugin	Email Extension Plugin	✓ Mailer Plugin	

Jenkins 2.9

Sign In

--> Pipeline: Stage View
 ** Pipeline: Racine
 Pipeline
 ** Javadoc Plugin
 ** GitHub API Plugin
 Jenkins Git plugin
 ** GitHub plugin
 ** GitHub Branch Source Plugin
 GitHub Organization Folder Plugin
 Pipeline: Stage View Plugin
 Jenkins Git plugin
 ** MapDB API Plugin
 Jenkins Subversion Plug-in
 >> - required dependency

Getting Started

Create First Admin User

Username:

Password:

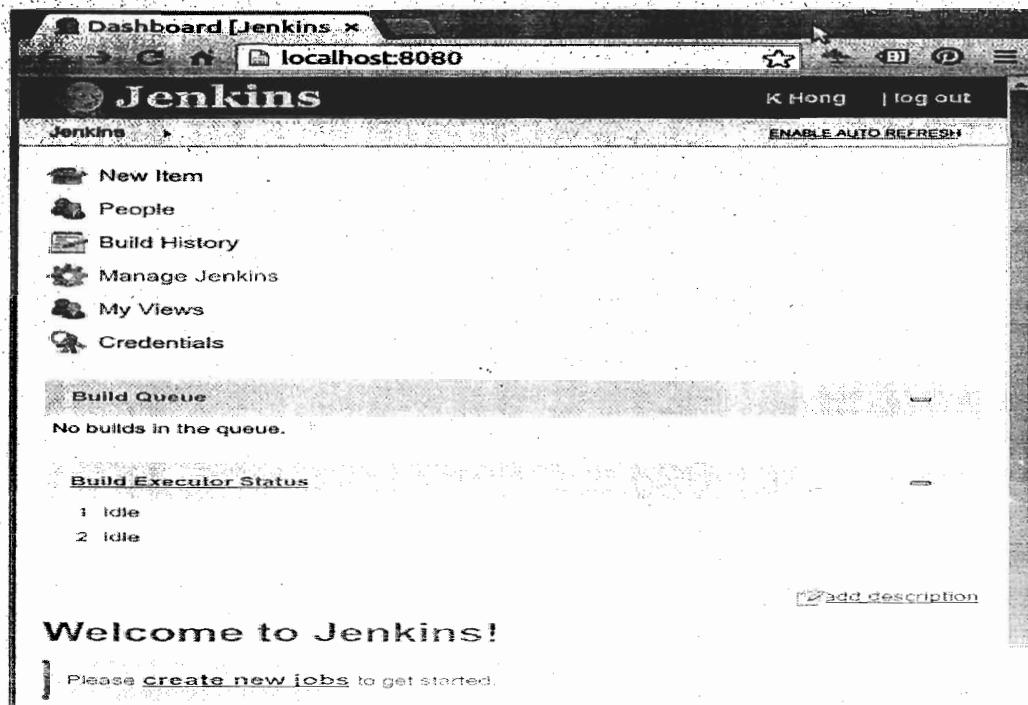
Confirm password:

Full name:

E-mail address:

Jenkins 2.5

Create account **Save and finish**



Java

At this point, most likely we already have java installed since Jenkins is a Java web application. We need at least the Java Runtime Environment, or JRE to run it. So, let's check if we have it:

```
$ java -version
```

```
openjdk version "1.8.0_91"
```

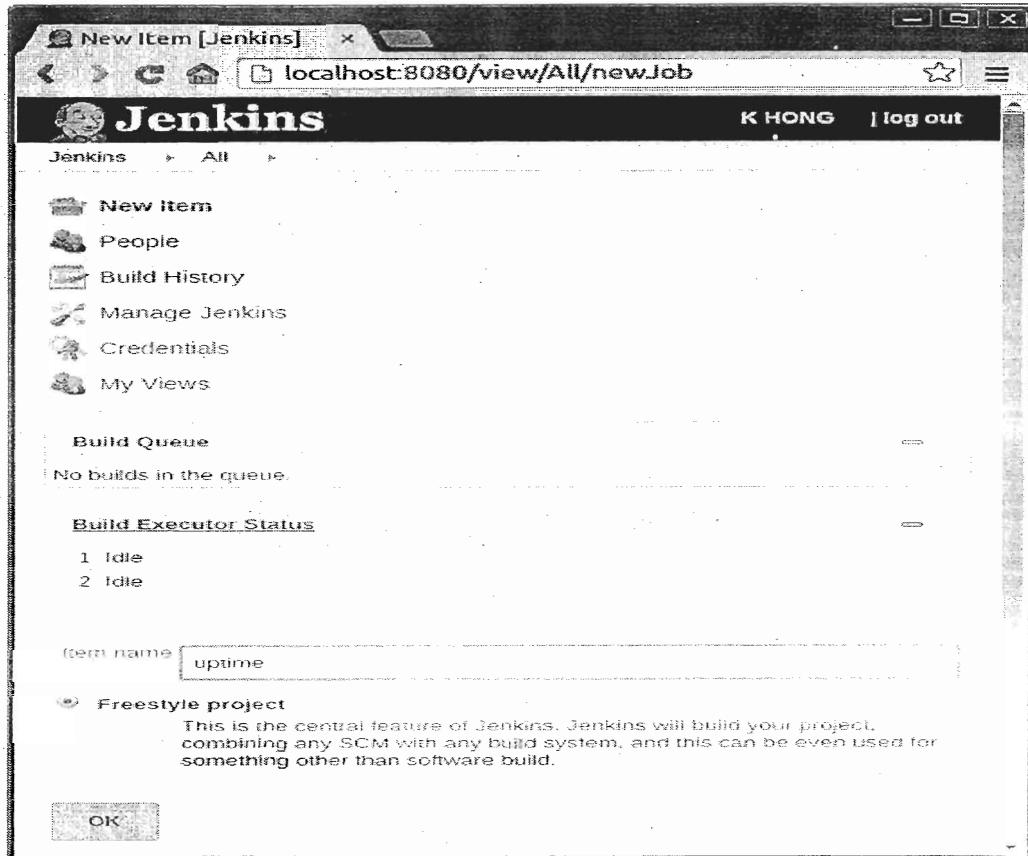
```
OpenJDK Runtime Environment (build 1.8.0_91-8u91-b14-0ubuntu4~14.04-b14)
```

```
OpenJDK 64-Bit Server VM (build 25.91-b14, mixed mode)
```

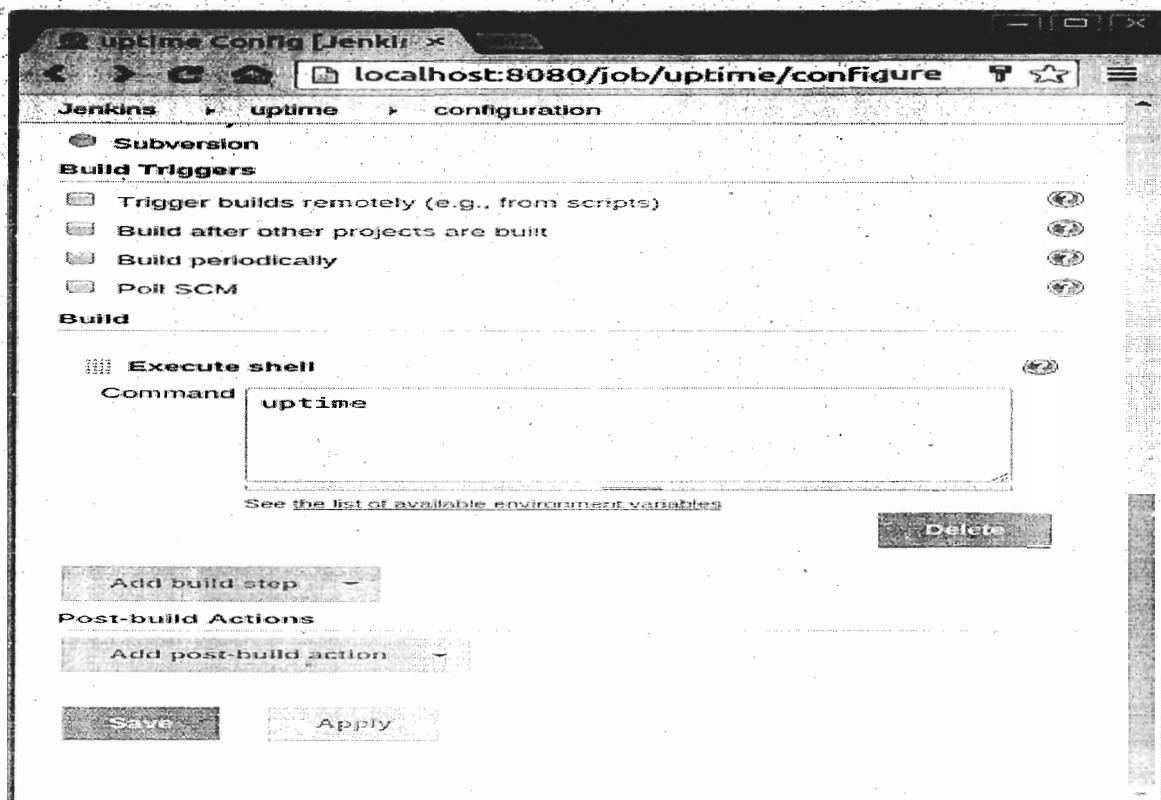
Note : we'll be using Jenkins mostly to build a Java application using Maven. Maven is a widely-used build tool in the Java world, with many powerful features such as declarative dependency management, convention over configuration, and a large range of plugins. For our build, we will also be using recent versions of the Java Development Kit (JDK) and Maven.

Adding a job

Now that everything is up and running, it's time to create our first job. Click the New Job link:

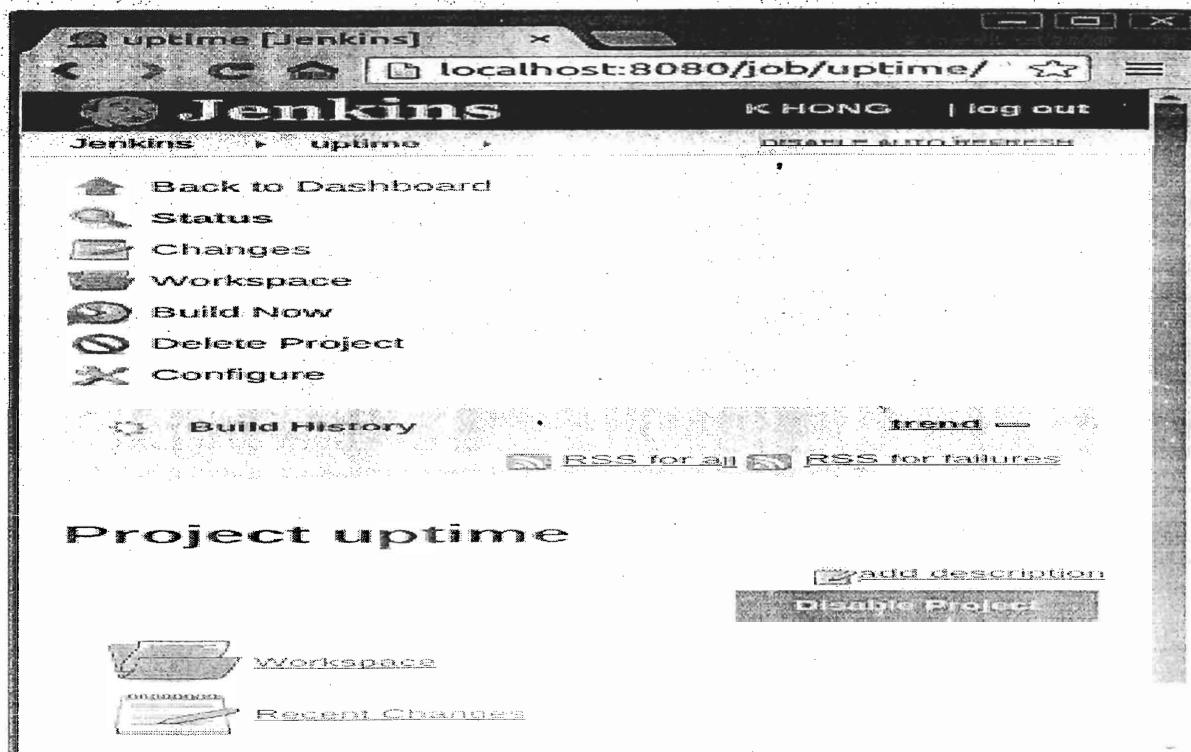


Press Ok and on the next page, add a Execute shell build step:

DEVOPS MATERIAL

Click the 'Save' button at the bottom of the screen.

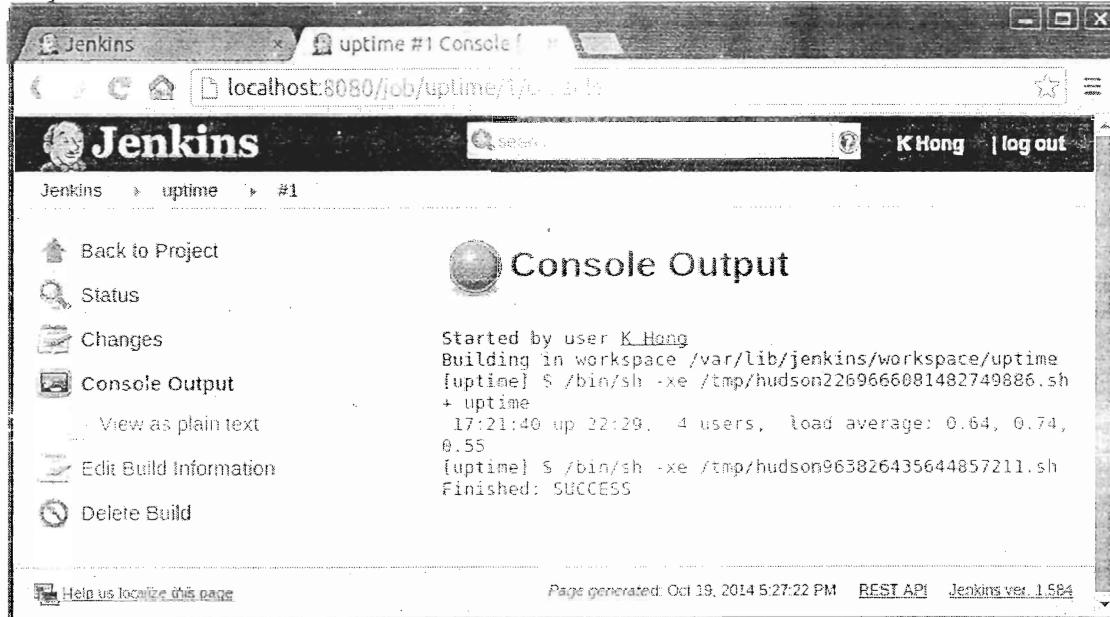
Build



Click the 'Build Now' button, then we can see the build show up in the build history block.

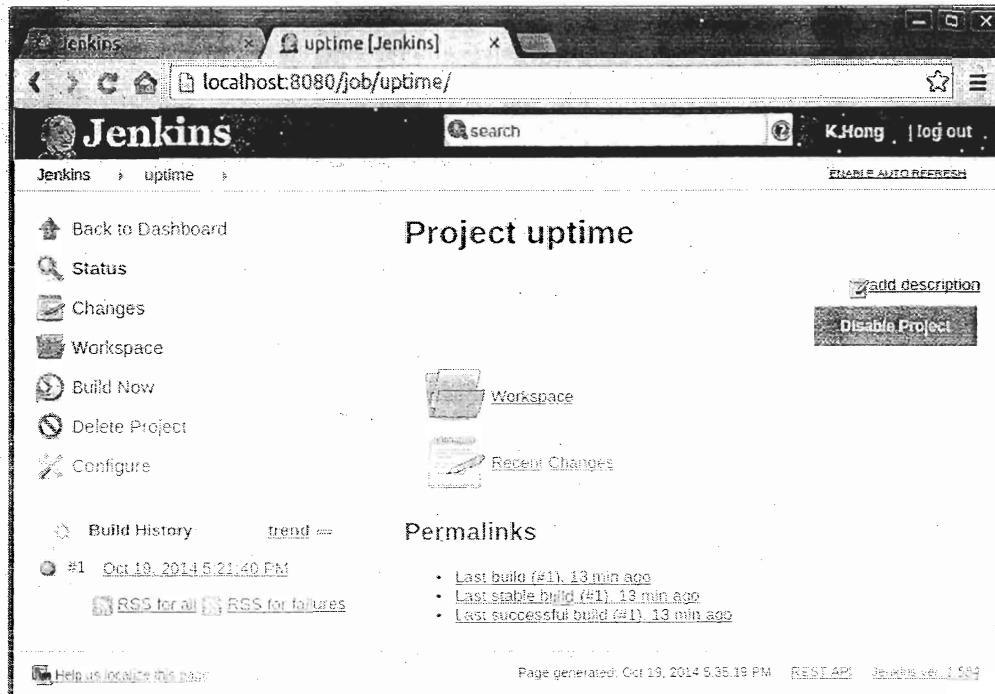
Click the blue sphere for the console

output:



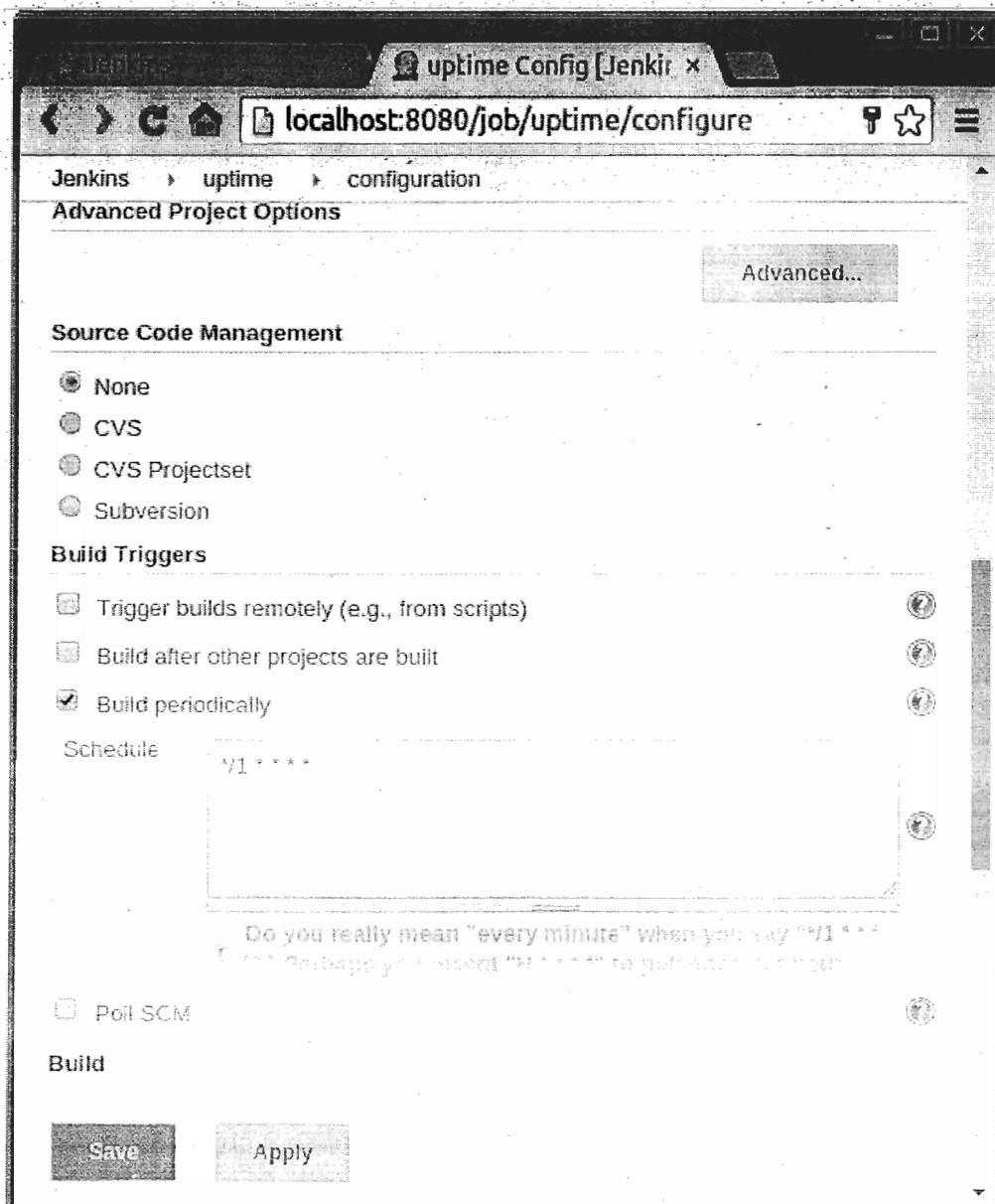
Scheduling jobs

Jenkins can run our job either on-demand or at a scheduled time. Since we've set up the basic build, it's time to configure a build schedule. First, click the 'Back to Project' link to return to the job overview:



The screenshot shows the Jenkins interface for the 'uptime' project. The left sidebar contains links like 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', and 'Configure'. The main content area is titled 'Project uptime' and includes sections for 'Workspace' and 'Recent Changes'. On the right, there are buttons for 'Add description' and 'Disable Project'. Below these are 'Permalinks' and a list of recent builds. The 'Configure' link in the sidebar is highlighted with a red box.

Press the 'Configure' button which leads us to the configuration page. Now look for 'Build Triggers' and check 'Build periodically':



Now this input field accepts the 'cron' syntax. We'll set it to `0/1 * * * *`, starting it every minute. When the job is saved, the scheduler will start the job at its designated times. We can check the job scheduled every minute indeed ran as expected.

SETTING UP MASTER AND SLAVE NODES ON EC2

In this tutorial, we'll setup 1 master and 2 slaves on EC2 Ubuntu.

DEVOPS MATERIAL

Then, we're going to run two jobs on Jenkins master and see how the loads are distributed across server/slave nodes.

This tutorial is based on the following 2 references:

Installing Jenkins on Ubuntu.

Setting Up Jenkins Continuous Integration.

Create instances

We need to create 3 instances: one for master and two for slave nodes:

Number of instances

The Security Group looks like this:

Type	Protocol	Port Range	Source
HTTP	TCP	80	Anywhere ▾ 0.0.0.0/0
SSH	TCP	22	Anywhere ▾ 0.0.0.0/0
All ICMP	ICMP	0 - 65535	Anywhere ▾ 0.0.0.0/0

Here are our nodes just created

Name
Jenkins-Master
Jenkins-Node1
Jenkins-Node2

SSH key : copy master's public key to slave nodes

To enable talks between our master and slave nodes, we need to copy public key (`id_rsa.pub`) of the master into slave's `authorized_keys`.

First, generate the key on Master node (52.53.240.42):

```
$ ssh-keygen -t rsa
```

We need to copy master's `~/.ssh/id_rsa.pub` and put it into our slave nodes' `authorized_keys`.

So, on the slave node 1, issue the following command:

```
$ echo "ssh-rsa  
AAAAAB3NzaC1yc2EAAAQABAAQCxhy2b6q/nX+iMY4jyymwCEBKPPGbeG+x0Oh  
VgoribLfDvE9ilDnzgRNnQ0deezqpiEpNQZTMn4/4kpFjHwJPMLnKTbwRW/2gu/h1GMtbYJu  
EzpgkezBVJCfGbOX6S+J6AYcvKDVKwOXOSPh/hezTg23Qe+Jw5ljT4O+D5halfYG2NPVF09  
8eYBpoKjm1P4uByB9+BGPMJ7avjzhv4WS5ZNTxLPVQPKX0Np7NXAju3dp6RxYHomAOO  
R3H90VLvc7p9IuQUv5NnjM2i1da/0B6EeUAdgB0VwsSdqNxF98QQtlIsqQogb3eoERyZDEsOr  
XVDTNyoBV59BECEK0TlTOe7T ubuntu@ip-172-31-20-240" >> ~/.ssh/authorized_keys
```

Do the same on slave node 2.

Jenkins install on the master server

```
$ wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -  
$ sudosh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable binary/ >  
/etc/apt/sources.list.d/jenkins.list'  
$ sudo apt-get update  
$ sudo apt-get upgrade  
$ sudo apt-get install jenkins
```

Run Jenkins:

```
$ sudo service jenkins start
```

Install jre on slave nodes

SSH into slave nodes, upgrade the packages, and install a Java Runtime Environment:

```
$ sudo apt-get update  
$ sudo apt-get upgrade  
$ sudo apt-get install default-jre
```

That's the only package Jenkins needs for slaves by default.

Setting up an Nginx Proxy for port 80 -> 8080

The Jenkins default port 8080 is not opened in our security group. So, we need to setup Nginx to proxy port 80 to 8080 so that you can keep Jenkins on 8080.

Install Nginx:

```
$ sudo apt-get install nginx
```

Remove default configuration:

```
$ cd /etc/nginx/sites-available  
$ sudo rm default ..sites-enabled/default
```

Our Nginx proxy conf file (`/etc/nginx/sites-available/jenkins`) looks like this:

```
upstream app_server {  
    server 127.0.0.1:8080 fail_timeout=0;
```

```
}

server {
listen 80;
listen [::]:80 default ipv6only=on;
server_name ec2-52-53-240-42.us-west-1.compute.amazonaws.com;

location / {
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header Host $http_host;
proxy_redirect off;

if (!-f $request_filename) {
proxy_pass http://app_server;
break;
}
}
}
```

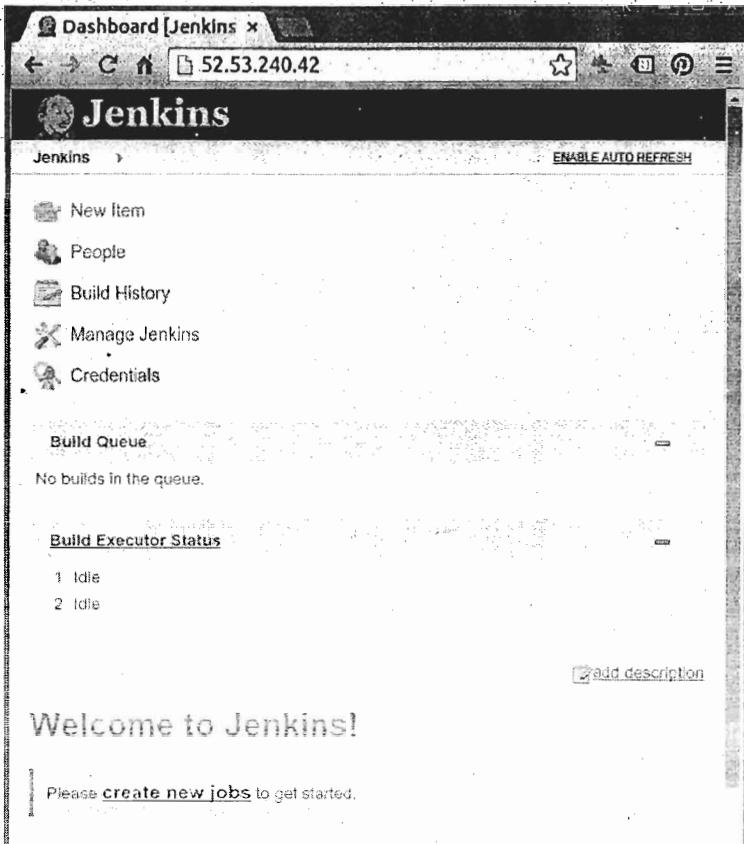
Link the configuration from sites-available to sites-enabled:

```
$ sudo ln -s /etc/nginx/sites-available/jenkins /etc/nginx/sites-enabled/
```

Then, restart Nginx:

```
$ sudo service nginx restart
```

Open up browser. Jenkins is now available on port 80:



Click on "Build Executor Status":

The screenshot shows the Jenkins 'Nodes' page at the URL 52.53.240.42/computer/. The page title is 'Nodes [Jenkins]'. It includes a 'Back to Dashboard' link, 'Manage Jenkins' link, and 'New Node' button. A 'Build Queue' section indicates 'No builds in the queue.' Below it is a 'Build Executor Status' section showing two idle executors. A table lists the single master node: Name (master), Architecture (Linux (amd64)), Clock Difference (In sync), Free Disk Space (6.11 GB), and Free Swap (76 ms). A 'Refresh status' button is at the bottom.

S	Name	Architecture	Clock Difference	Free Disk Space	Free Swap
	master	Linux (amd64)	In sync	6.11 GB	76 ms
	Data obtained		79 ms		

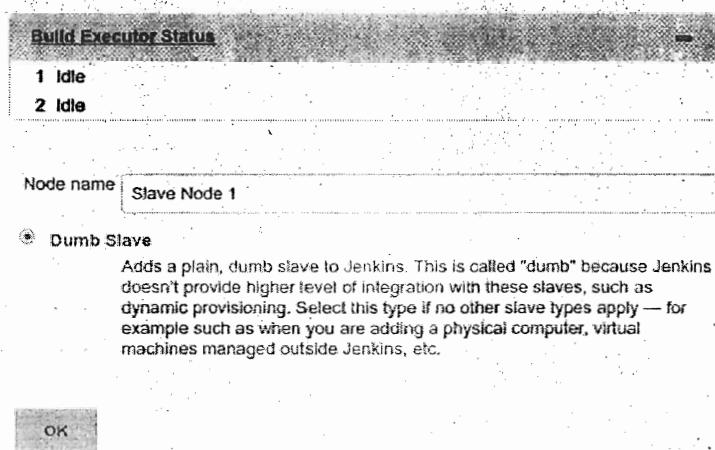
We only see the master node at this point.

Let's add new nodes for our slaves.

Slave node setup

Click New Node:

DEVOPS MATERIAL



Hit OK:

Type in "Remote root directory" and make sure to use private ip for the Host.

Jenkins

52.53.240.42/computer/createtem

Jenkins	Nodes
Name	Slave Node 1
Description	
# of executors	1
Remote root directory	/home/ubuntu
Labels	
Usage	Utilize this node as much as possible
Launch method	Launch slave agents on Unix machines via SSH
Host	172.31.20.238
Credentials	Add Advanced
Availability	Keep this slave on-line as much as possible
Node Properties	
Environment variables	
Tool Locations	
Save	

Click "Add" button on Credentials:

Type in "Username" and copy the private key (`~/.ssh/id_rsa`) of the master server:

 Add Credentials

Kind	SSH Username with private key
Scope	Global
Username	ubuntu
Description	
Private Key	<input type="radio"/> Enter directly <input checked="" type="radio"/> From a file on Jenkins master <input type="radio"/> From the Jenkins master ~/.ssh <pre>-----BEGIN RSA PRIVATE KEY----- MIIEowIBAAKCAQEAQJ...Z+vuCSBhKTDBZ86k7aFBropyXewTAbuoIhbBWme -----END RSA PRIVATE KEY-----</pre>
Advanced...	
<input type="button" value="Add"/> <input type="button" value="Cancel"/>	

Hit "Add":

 Jenkins

52.53.240.42/computer/createnode

Jenkins

- Back to Dashboard
- Manage Jenkins
- New Node
- Configure
- Build Custom
- No builds in the queue.
- Build Executor Status

Nodes

Name	Slave Node 1
Description	
# of executors	1
Remote root directory	/home/ubuntu
Labels	
Usage	Utilize this node as much as possible
Launch method	Launch slave agents on Unix machines via SSH
Host	172.31.20.238
Credentials	ubuntu  
Advanced...	
Availability	Keep this slave on-line as much as possible

Node Properties

- Environment variables
- Tool Locations

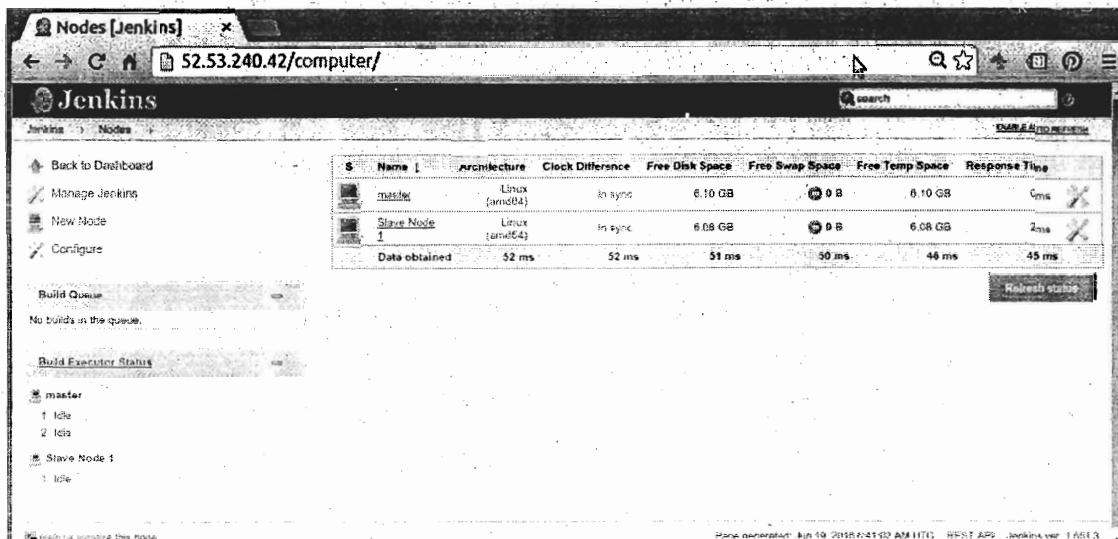
Save

[Help us localize this page](#)

Page generated: Jun 19, 2016 5:24:23 AM UTC REST API Jenkins ver. 1.651.3

Click "Save"

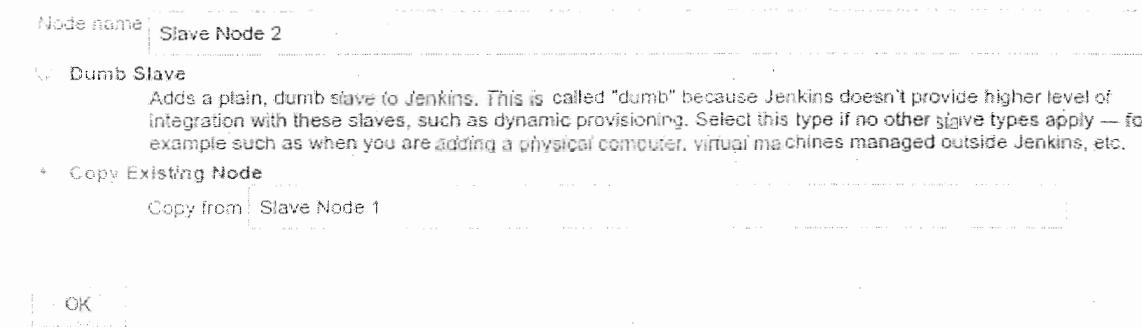
Now we can see our Slave Node #1 is up and connected:



The screenshot shows the Jenkins 'Nodes' page. On the left sidebar, there are links for 'Back to Dashboard', 'Manage Jenkins', 'New Node', and 'Configure'. Under 'Build Queue', it says 'No builds in the queue.' Below that is 'Build Executor Status' with entries for 'master' (1 idle, 2 total) and 'Slave Node 1' (1 idle). The main table lists nodes: 'master' (Linux (amd64), In sync, 6.10 GB free disk space, 0 B free swap space, 6.10 GB free temp space, 0ms response time) and 'Slave Node 1' (Linux (amd64), In sync, 6.08 GB free disk space, 0 B free swap space, 6.08 GB free temp space, 2ms response time). A 'Data obtained' row shows 52 ms, 52 ms, 51 ms, 50 ms, 48 ms, and 45 ms respectively. A 'Refresh status' button is at the bottom right.

Click "Save"

Do the same to Slave Node #2. This time we'll use "Copy Existing Node":



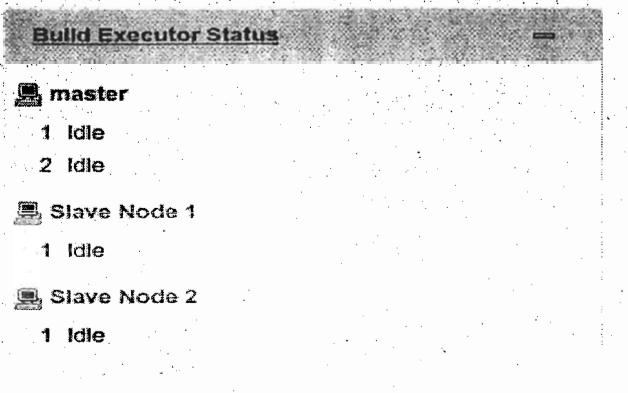
The dialog box has 'Node name' set to 'Slave Node 2'. Under 'Type', 'Dumb Slave' is selected with the note: 'Adds a plain, dumb slave to Jenkins. This is called "dumb" because Jenkins doesn't provide higher level of integration with these slaves, such as dynamic provisioning. Select this type if no other slave types apply — for example such as when you are adding a physical computer, virtual machines managed outside Jenkins, etc.' The 'Copy Existing Node' option is also visible. The 'Copy from' field is set to 'Slave Node 1'. At the bottom are 'OK' and 'Cancel' buttons.

Make sure type in private ip address of Slave Node #2.

Now we have one master and two slave nodes are linked up and running:

S	Name	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
1	DELL	Linux (amd64)	In sync	6.10 GB	0 B	6.10 GB	0ms
2	Slave Node 1	Linux (amd64)	In sync	6.08 GB	0 B	6.08 GB	26ms
3	Slave Node 2	Linux (amd64)	In sync	6.08 GB	0 B	6.08 GB	25ms
Data obtained		0.2 sec	0.2 sec	0.2 sec	0.19 sec	0.19 sec	0.19 sec

[Refresh status](#)



Running jobs

Go to Jenkins Dashboard and create two new jobs.

Item name: job_1

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

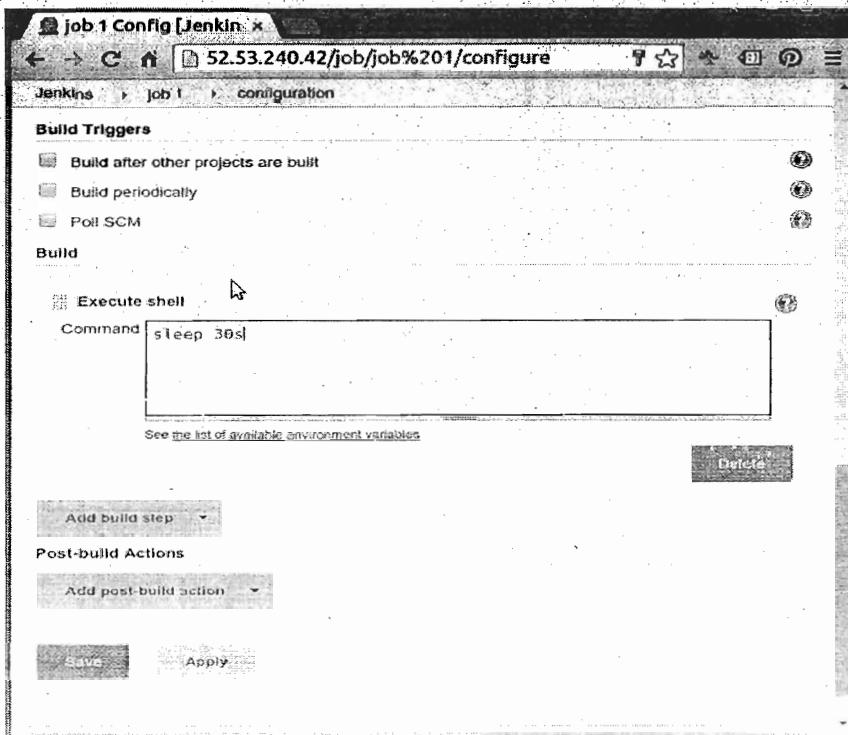
Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

OK

We're going to run a very simple job: sleep 30 seconds:



Click "Save".

Create another one for job 2 doing the same thing: sleep 30s.

Now the two jobs are running: one on master the other one on slave #2:

Dashboard [Jenkins]

← → C ⌂ 52.53.240.42

Jenkins

search

Jenkins

-  New Item
-  People
-  Build History
-  Manage Jenkins
-  Credentials

All +

S	W	Name	Last Succ
		job 1	N/A
		job 2	N/A

Icon: S M L Legend: RSC

Build Queue

No builds in the queue.

Build Executor Status

master

- 1 Idle
- 2 job 2  #1 

Slave Node 1

- 1 Idle

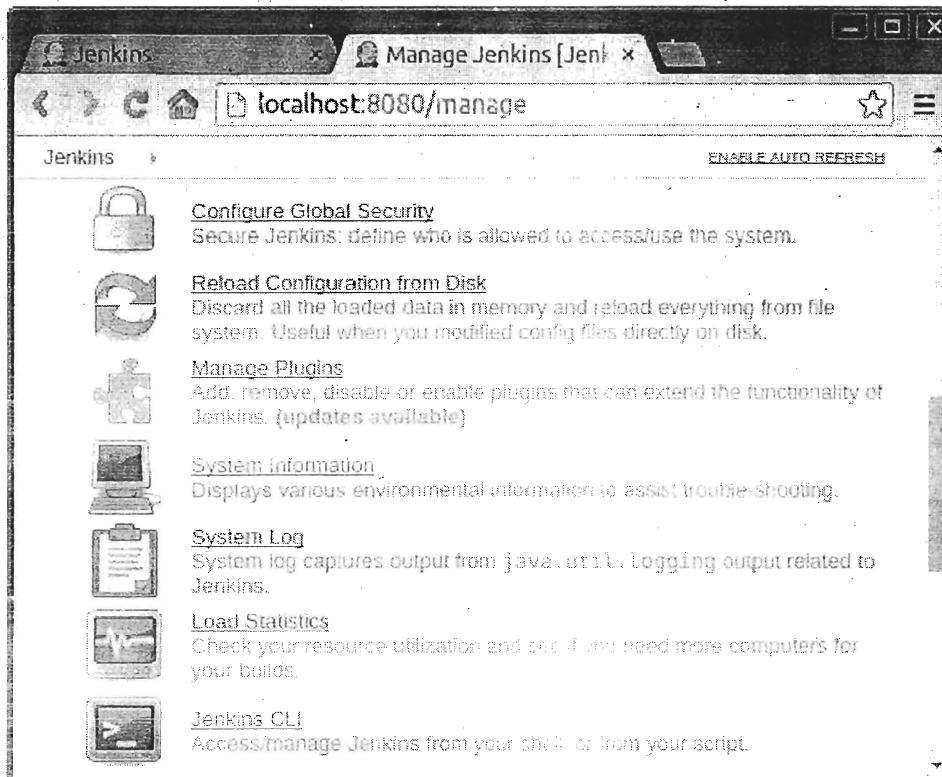
Slave Node 2

- 1 job 1  #1 

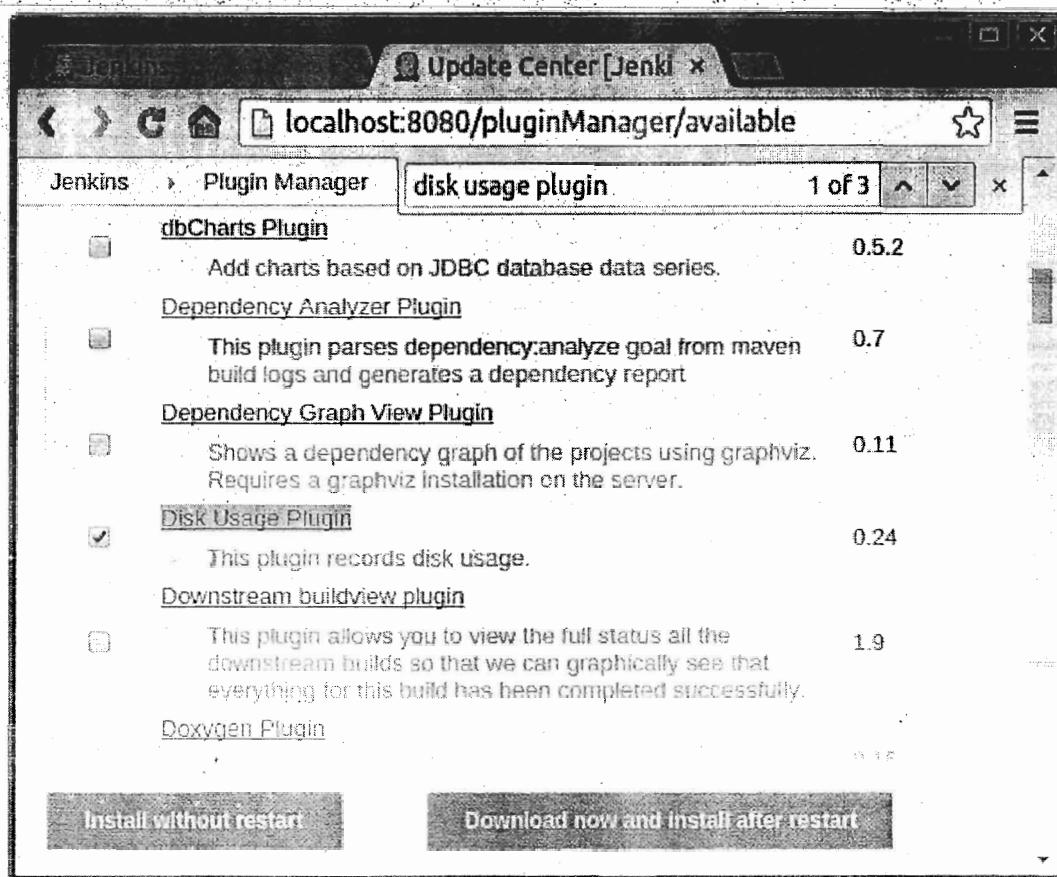
If we run 3 jobs, there will be no idling slave node.

Manage Plugins

Among the plugins that are available for Jenkins, in this chapter, we're going to install a plugin that keeps track of the disk space used by different builds and jobs. To get started, we need to go to 'Manage Jenkins' again, and click on 'Manage Plugins':



Look for the 'Disk Usage Plugin' (we may want to use Filter search to find this), select it, and then click on the 'Install without the restart' button at the bottom of our screen:

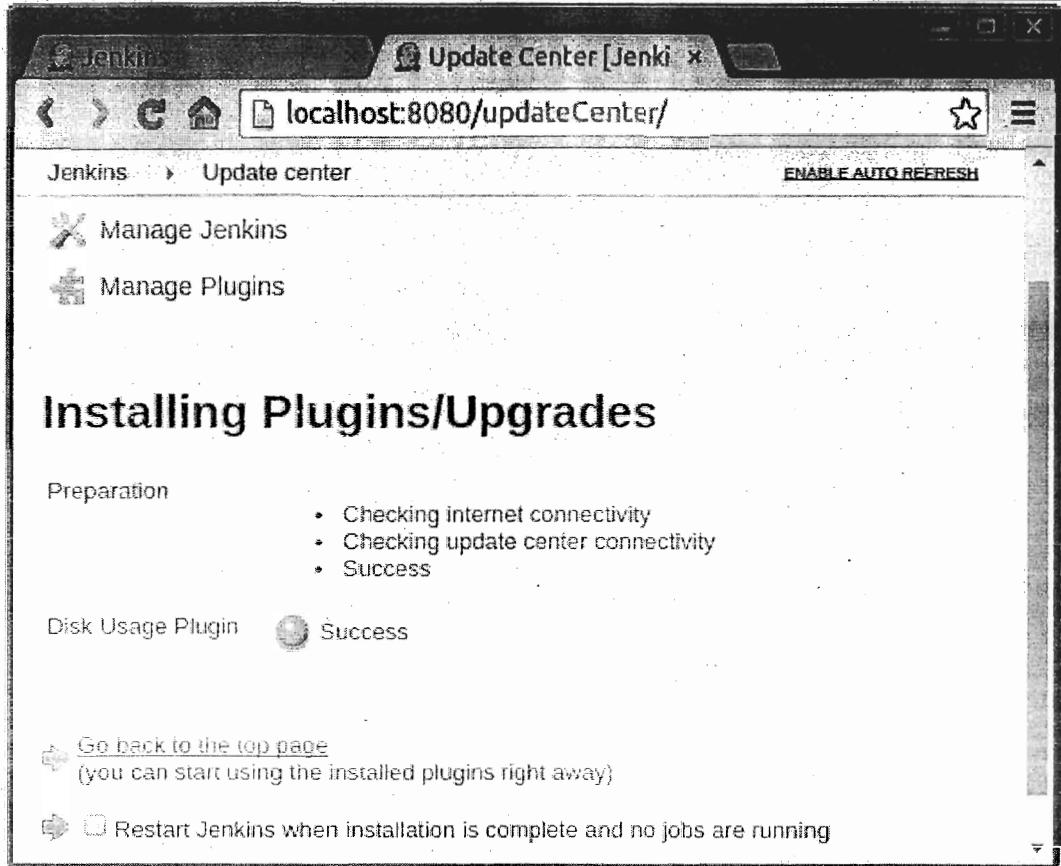


The screenshot shows the Jenkins Update Center interface. The URL in the address bar is `localhost:8080/pluginManager/available`. The search bar at the top contains the text "disk usage plugin". Below the search bar, it says "1 of 3". A list of available plugins is displayed:

- dbCharts Plugin** (version 0.5.2): Add charts based on JDBC database data series.
- Dependency Analyzer Plugin** (version 0.7): This plugin parses dependency:analyze goal from maven build logs and generates a dependency report.
- Dependency Graph View Plugin** (version 0.11): Shows a dependency graph of the projects using graphviz. Requires a graphviz installation on the server.
- Disk Usage Plugin** (version 0.24): This plugin records disk usage. (This plugin is selected, indicated by a checked checkbox icon.)
- Downstream buildview plugin** (version 1.9): This plugin allows you to view the full status all the downstream builds so that we can graphically see that everything for this build has been completed successfully.
- Doxygen Plugin**

At the bottom of the page are two buttons: "Install without restart" and "Download now and install after restart".

Just wait until the plugin has been installed and it's ready for use.



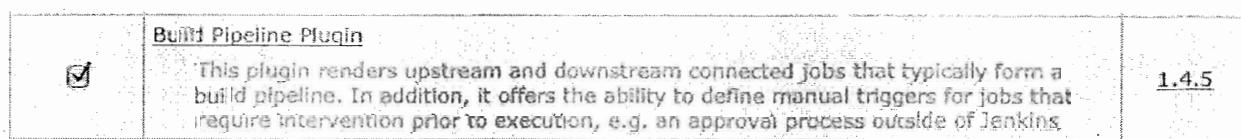
The screenshot shows the Jenkins Update Center interface at localhost:8080/updateCenter/. The main content area displays the heading "Installing Plugins/Upgrades". Under "Preparation", there is a bulleted list: "Checking internet connectivity", "Checking update center connectivity", and "Success". Below this, the "Disk Usage Plugin" is listed with a green circular icon and the word "Success". At the bottom left, there is a link to "Go back to the top page" with the note "(you can start using the installed plugins right away)". At the bottom right, there is a checkbox labeled "Restart Jenkins when installation is complete and no jobs are running".

While this is the most simplest plugin with no settings, there are other plugins, such as the SSH plugin, that require rather complicated configuration after installing before we can use.

Installing Build Pipeline Plugin

The "Manage Jenkins" page has a "Manage Plugins" link on it, which takes us a list of all the available plugins for our Jenkins installation.

To install the build pipeline plugin, simply put a tick in the checkbox next to "build pipeline plugin"



and click either one of the following buttons which are located in the plugin page.

[Install without restart](#)
[Download now and install after restart](#)

Once installed, we should see the following in the Installed Plugins page under Plugin Manager:

Updates	Available	Installed	Advanced		
Enabled	Name	Version	Previously installed version	Pinned	Uninstall
	Build Pipeline Plugin	1.4.5			Uninstall

Creating a Pipeline

To create a pipeline in Jenkins, we need to create the build jobs. Each pipeline section represents one build job. Then, we have to then tell each build job about the downstream build which is must trigger, using the **build other projects** option.

Creating a Pipeline View

The Build Pipeline plugin does not have any global configuration options, instead it adds additional functionality to the views available within Jenkins. If no view has been created, then our system will be using the default "All" view. This view is read only. So, in order to get started with this plugin, we first need to create a view.

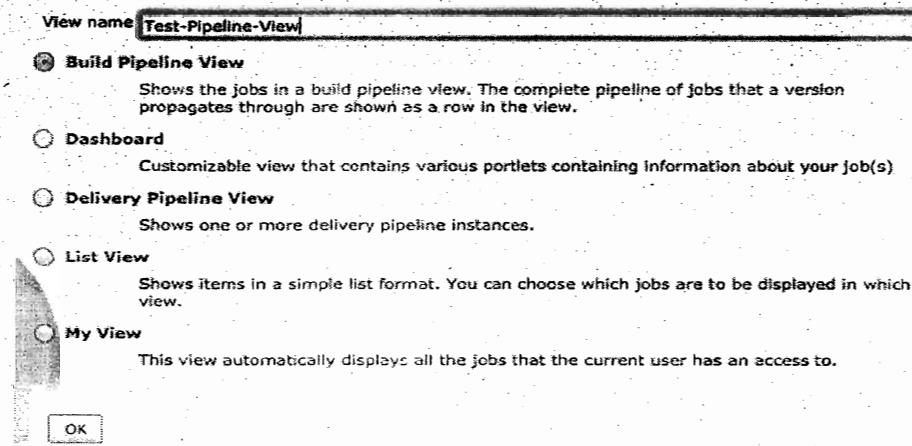
1. Jenkins has one built in view type, List View:

View name:

List View
Shows items in a simple list format. You can choose which jobs are to be displayed in which view.

My View
This view automatically displays all the jobs that the current user has an access to.

2. From Jenkins instance's root page, there is a tab called "+" at the end of all the tabs. Click on that tab to create a new view.
Give the view a name, and select the type of view we want to create.
View types are an extension point that other plugins can contribute.



We now have a new pipeline! The next thing to do is kick it off and see it in action:

Job	Status	Date	Duration
Build-1-1 (#5)	Building	Feb 5, 2015 6:14:19 PM	0 ms
Build-1-2 (#5)	Building	Feb 5, 2015 6:14:25 PM	73 ms
Build-2-1 (#3)	Building	Feb 5, 2015 6:14:25 PM	64 ms
Build-2-2 (#4)	Building	Feb 5, 2015 6:14:35 PM	49 ms
Build-1-3 (#4)	Building	Feb 5, 2015 6:14:35 PM	0,14 sec
Build-1-4 (#4)	Building	Feb 5, 2015 6:14:35 PM	20 ms
#5 Build-1-1	Success	Feb 5, 2015 6:14:19 PM	0 ms
#5 Build-1-2	Success	Feb 5, 2015 6:14:19 PM	0 ms
#3 Build-2-1	Success	Feb 5, 2015 6:14:25 PM	0 ms
#4 Build-2-2	Success	Feb 5, 2015 6:14:35 PM	0 ms
#4 Build-1-3	Success	Feb 5, 2015 6:14:35 PM	0 ms
#4 Build-1-4	Success	Feb 5, 2015 6:14:35 PM	0 ms
#4 Build-2-3	Success	Feb 5, 2015 6:14:35 PM	0 ms
#4 Build-2-4	Success	Feb 5, 2015 6:14:35 PM	0 ms

Build graph-view plugin

This plugin visualize builds relations as a graph.

DEVOPS MATERIAL

Jenkins > Build-Pipeline-Testing > configuration

- [Back to Dashboard](#)
- [Status](#)
- [Changes](#)
- [Workspace](#)
- [Build Now](#)
- [Delete Project](#)
- [Configure](#)

Build History

#	Build Number
#5	Feb 5, 2015 6:14:19 PM
#4	
#3	
#2	
#1	

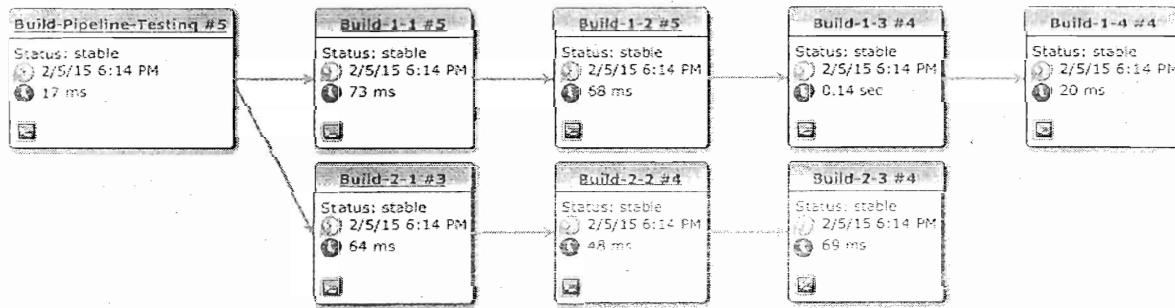
[Changes](#)
[Console Output](#)
[Edit Build Information](#)
[Delete Build](#)

(trend) —

RSS for all

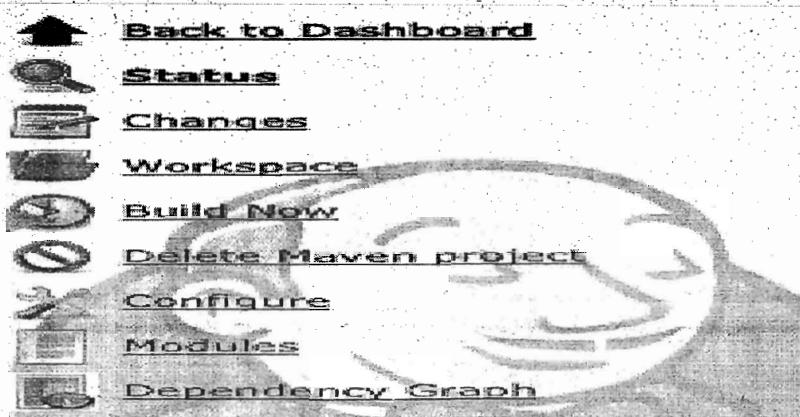
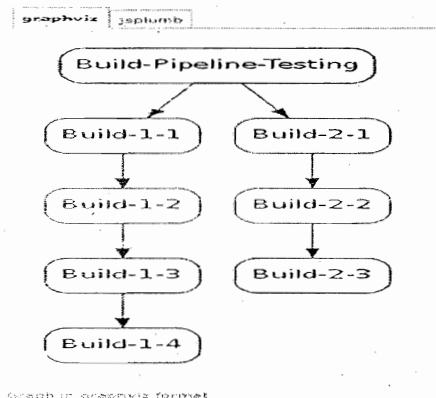
RSS for failures

Build Graph

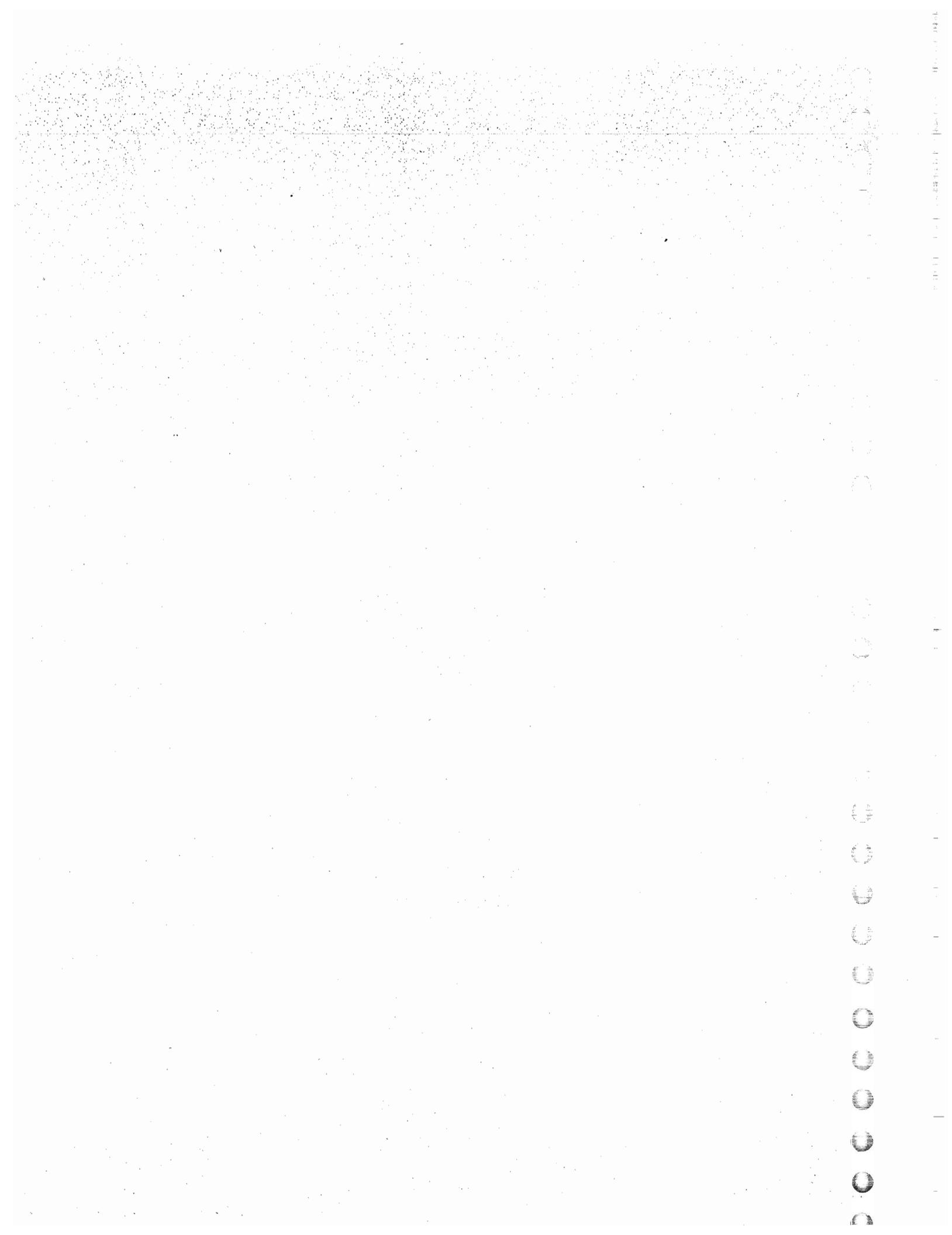


Dependency Graph Viewer Plugin

Dependency Graph Viewer Plugin shows a dependency graph of the projects. It uses dot (from graphviz) for drawing.

**Dependency Graph**

Download in graphviz format
Edit this graph



Container run

1. **docker create** creates a container but does not start it.
2. **docker rename** allows the container to be renamed.
3. **docker run** creates and starts a container in one operation.

```
$ docker run -it ubuntu-ssh-k /bin/bash
```

i: interactive, t: with tty

4. **dockerrm** deletes a container.
5. **docker update** updates a container's resource limits.

Container info

1. **dockerp s** shows running containers.

```
$ dockerp s
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
e2481c1bad5dubuntu-ssh-k:latest	"/bin/bash"		10 hours ago	Up 10 hours
	hopeful_carson			

2. **docker logs** gets logs from container. (You can use a custom log driver, but logs is only available for json-file and journald in 1.10)
3. **docker inspect** looks at all the info on a container (including IP address).

To get an IP address of a running container:

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' $(docker ps -q)
```

172.17.0.5

4. **docker events** gets events from container.

5. **docker port** shows public facing port of container.
6. **docker top** shows running processes in container.
7. **docker stats** shows containers' resource usage statistics.
8. **docker diff** shows changed files in the container's FS.

Container start/stop

1. **docker start** starts a container so it is running.
2. **docker stop** stops a running container.
3. **docker restart** stops and starts a container.
4. **docker pause** pauses a running container, "freezing" it in place.
5. **docker unpause** will unpause a running container.
6. **docker wait** blocks until running container stops.
7. **docker kill** sends a SIGKILL to a running container.
8. **docker attach** will connect to a running container.

Image create/remove

1. **docker images** : shows all images
2. **docker import** : creates an image from a tarball.
3. **docker build** : creates image from Dockerfile.
4. **docker commit** : creates image from a container, pausing it temporarily if it is running.

```
$ docker commit b9b2ee7004fe ubuntu-ssh-k
```

where the 'b9b2ee7004fe' is the container-id and 'ubuntu-ssh-k' is the new name of the image.

5. **docker rmi** : removes an image
6. **docker load** : loads an image from a tar archive as STDIN, including images and tags

7. **docker save** : saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions

Container info

Getting Docker Container's IP Address from host machine:

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' $(docker ps -q)
```

```
172.17.0.2
```

```
172.17.0.3
```

Image info

1. **docker history** shows history of image.
2. **docker tag** tags an image to a name (local or registry).

Network create/remove/info/connection

1. **docker network create**
2. **docker network rm**
3. **docker network ls**
4. **docker network inspect**
5. **docker network connect**
6. **docker network disconnect**

Docker Repo

1. **docker login** to login to a registry.
2. **docker logout** to logout from a registry.
3. **docker search** searches registry for image.

```
$ docker search mysql
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
mysql	MySQL is a widely used, open-source relati...	2797	[OK]	
mysql/mysql-server	Optimized MySQL Server Docker images. Crea...	182		[OK]

4. **docker pull** pulls an image from registry to local machine.

```
$ docker pull ubuntu
```

```
latest: Pulling from ubuntu
```

```
20ee58809289: Pull complete
```

```
f905badeb558: Pull complete
```

```
119df6bf2a3a: Pull complete
```

```
94d6eea646bc: Pull complete
```

```
bb4eabee84bf: Pull complete
```

```
Digest: sha256:85af8b61adffea165e84e47e0034923ec237754a208501fce5dbeecbb197062c
```

```
Status: Downloaded newer image for ubuntu:latest
```

Docker images can consist of multiple layers. In the example above, the image consists of five layers

5. **docker push** pushes an image to the registry from local machine.

Introduction

We create Docker containers using [base] images. An image can be basic, with nothing but the operating-system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch.

When we build images with docker, each action taken (i.e. a command executed such as **apt-get install**) forms a new layer on top of the previous one. These base images then can be used to create new containers.

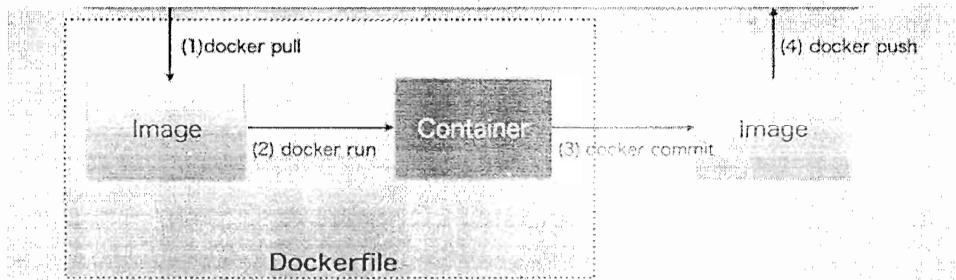


Image source: Docker

In this chapter, we're going to learn how to automate this process via instructions in **Dockerfiles**. A Dockerfile is a text document that contains all the commands we would normally execute manually in order to build a Docker image. By calling `docker build` from our terminal, we can have Docker build our image executing the instructions successively step-by-step, layer-by-layer, automatically from a source (base) image.

The docker project offers higher-level tools which work together, built on top of some Linux kernel features. The goal is to help developers and system administrators port applications with all of their dependencies included, and get them running across systems and machines headache free.

Docker achieves this by creating safe, LXC (Linux Containers) based environments for applications called docker containers. These containers are created using docker images, which can be built either by executing commands manually or automatically through Dockerfiles.

Dockerfile

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

Dockerfiles begin with defining an image FROM which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

Sample Dockerfile

Here is our Dockerfile we're going to playing with in this chapter. We'll run instructions from this file step by step by uncommenting and commenting each line.

```
FROM debian:latest

MAINTAINER devops@bogotobogo.com


# 1 - RUN

RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils
```

```
RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop
```

```
RUN apt-get clean
```

```
# 2 - CMD
```

```
CMD ["htop"]
```

```
CMD ["ls", "-l"]
```

```
# 3 - WORKDIR and ENV
```

```
WORKDIR /root
```

```
ENV DZ version1
```

```
# 4 - ADD
```

```
ADD run.sh /root/run.sh
```

```
CMD ["/root/run.sh"]
```

```
# 5 - ENTRYPOINT (vs CMD)
```

```
ENTRYPOINT ["/root/run.sh"]
```

```
CMD ["arg1"]
```

Dockerfile format

Here is the format of the Dockerfile:

Comment**INSTRUCTION arguments**

The Instruction is not case-sensitive, however convention is for them to be UPPERCASE in order to distinguish them from arguments more easily.

Docker runs the instructions in a Dockerfile in order. The first instruction must be FROM in order to specify the Base Image from which we are building.

Docker will treat lines that begin with # as a comment.

FROM

The FROM instruction sets the Base Image for subsequent instructions. As such, a valid Dockerfile must have FROM as its first instruction. The image can be any valid image. But it is especially easier to start by pulling an image from the Public Repositories.

MAINTAINER

MAINTAINER <name>

The MAINTAINER instruction allows us to set the Author field of the generated images.

Docker build & context

To build an image from a source repository, we create a description file called **Dockerfile** at the root of our repository. This file will describe the steps to assemble the image.

Then we call `docker build` with the path of our source repository as the argument:

```
Usage: docker build [OPTIONS] PATH | URL | -
```

It will build a new image from the source code at PATH.

Here is an example using the current directory ("..") as its path:

```
$ sudo docker build ..
```

The path to the source repository defines where to find the context of the build. The build is run by the Docker daemon, not by the CLI, so the whole context must be transferred to the daemon. The Docker CLI reports "Sending build context to Docker daemon" when the context is sent to the daemon.

Note: Avoid using root directory, `/`, as the root of the source repository. The `docker build` command will use whatever directory contains the **Dockerfile** as the build context(including all of its subdirectories). The build context will be sent to the Docker daemon before building the image, which means if we use `/` as the source repository, the entire contents of our hard drive will get sent to the daemon (and thus to the machine running the daemon). We probably don't want that. In most cases, it's best to put each **Dockerfile** in an empty directory, and then add only the files needed for building that **Dockerfile** to that directory. To further speed up the build, we can exclude files and directories by adding a `.dockerignore` file to the same directory.

DEVOPS MATERIAL

The Docker daemon will run our steps one-by-one, committing the result to a new image if necessary, before finally outputting the ID of our new image. The Docker daemon will automatically clean up the context we sent.

Note that each instruction is run independently, and causes a new image to be created.

Whenever possible, Docker will re-use the intermediate images, accelerating docker build significantly.

```
$ docker build --help
```

```
Usage: docker build [OPTIONS] PATH | URL | -
```

```
Build a new image from the source code at PATH
```

```
-t, --tag="": Repository name (and optionally a tag) to be applied to the resulting image in case of success
```

Base image

Let's download our base image:

```
$ docker pull debian:latest
```

```
debian:latest: The image you are pulling has been verified
```

```
511136ea3c5a: Pull complete
```

```
f10807909bc5: Pull complete
```

```
f6fab3b798be: Pull complete
```

```
Status: Downloaded newer image for debian:latest
```

```
k@laptop:~/Documents/demo$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
debian	latest	f6fab3b798be	2 weeks ago

In our local working directory, we have only one file, Dockerfile:

```
k@laptop:~/Documents/demo$ ls  
Dockerfile
```

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

docker build 'FROM'

Let's look at the syntax of docker build command:

```
$ docker build --help
```

```
Usage: docker build [OPTIONS] PATH | URL | -
```

```
Build a new image from the source code at PATH
```

```
-t, --tag=""           Repository name (and optionally a tag) to be  
applied to the resulting image in case of success
```

Dockerfiles begin with defining an image **FROM** which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

```
FROM debian:latest  
  
MAINTAINER devops@bogotobogo.com
```

Let's run docker build command with the two-line Dockerfile:

```
k@laptop:~/Documents/demo$ docker build -t bogodevops/demo:v1 .  
  
Sending build context to Docker daemon 2.56 kB  
  
Sending build context to Docker daemon  
  
Step 0 : FROM debian:latest  
  
--> f6fab3b798be  
  
Step 1 : MAINTAINER k@bogotobogo.com  
  
--> Running in 4181b54ab22e  
  
--> 511bcbdd59ba  
  
Removing intermediate container 4181b54ab22e  
  
Successfully built 511bcbdd59ba
```

Now if we list the images:

```
k@laptop:~/Documents/demo$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
bogodevops/demo	v1	511bcbdd59ba	About a minute ago
debian	latest	f6fab3b798be	2 weeks ago 85.1 MB

Note that the path to the source repository defines where to find the context of the build. The build is run by the Docker daemon, not by the CLI, so the whole context must be transferred to the daemon. The Docker CLI reports "Sending build context to Docker daemon" when the context (2.56kB) is sent to the daemon as shown in the output:

```
Sending build context to Docker daemon 2.56 kB
```

If we send big chuck to the daemon, it will take longer to copy things. For example, if we send duplicate device files(/dev/zero) with dd:

```
k@laptop:~/Documents/demo$ dd if=/dev/zero of=testimage bs=4096 count=8192
```

```
8192+0 records in
```

```
8192+0 records out
```

```
33554432 bytes (34 MB) copied, 0.118561 s, 283 MB/s
```

```
k@laptop:~/Documents/demo$ ls
```

```
Dockerfile testimage
```

```
k@laptop:~/Documents/demo$ docker build -t bogodevops/demo:v1
```

```
Sending build context to Docker daemon 33.56 MB
```

Sending build context to Docker daemon

Step 0 : FROM debian:latest

---> f6fab3b798be

Step 1 : MAINTAINER k@bogotobogo.com

---> Using cache

---> 511bcbdd59ba

Successfully built 511bcbdd59ba

Note that the size has been increased from 2.56kb to 33.56MB. That's why the Docker document gives us a Warning like this:

"Warning Avoid using your root directory, /, as the root of the source repository:
The docker build command will use whatever directory contains the Dockerfile as the build context (including all of its subdirectories). The build context will be sent to the Docker daemon before building the image, which means if you use / as the source repository, the entire contents of your hard drive will get sent to the daemon (and thus to the machine running the daemon). You probably don't want that."

Or like this:

"Warning: Do not use your root directory, /, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.

So, we should be aware of the context of our build directory!

Again:

"The build is run by the Docker daemon, not by the CLI. The first thing a build process does is send the entire context (recursively) to the daemon. In most cases, it's best to start with an empty directory as **context** and keep your **Dockerfile** in that directory. Add only the files needed for building the **Dockerfile**"

Also note the the difference in the Step 1 of the two cases:

In the first instance of docker build:

```
Step 1 : MAINTAINER k@bogotobogo.com
```

```
---> Running in 4181b54ab22e
```

```
---> 511bcbdd59ba
```

But in the second run, Docker used cache:

```
Step 1 : MAINTAINER k@bogotobogo.com
```

```
---> Using cache
```

```
---> 511bcbdd59ba
```

When we build a Docker image, it's using a Dockerfile, and every instruction in the Dockerfile is run inside of a container. If that returns successfully, then that container is stored as a new image.

In our case, in Step 0, we created 'f6fab3b798be' which is a hash identifier, and Step 1, we created '511bcbdd59ba' hash.

Note that in our 2nd run (the 'docker run' with 'dd'), the hash is the same. What does this mean? If the instructions in our Dockerfile are the same, Docker uses the cache:

```
k@laptop:~/Documents/demo$ docker images -a
```

REPOSITORY	TAG	IMAGE ID	CREATED
VIRTUAL SIZE			
bogodevops/demo	v1	511bcbdd59ba	57 minutes
ago	85.1 MB		
debian	latest	f6fab3b798be	2 weeks ago
85.1 MB			
<none><none>	f10807909bc5	2 weeks ago	85.1 MB

<none><none>

511136ea3c5a

17 months ago

0 B

So, every step along the way, we create a new image. As it succeeds, we'll build a new layer on top of the previous one as we read in an instruction. As this caching allows us to build other environment similar to the previous image without rebuilding from every steps involved.

Dockerfile - RUN

This section is from <http://docs.docker.com/reference/builder/>.

RUN has 2 forms:

```
RUN <command> (the command is run in a shell - /bin/sh -c - shell form)
```

```
RUN ["executable", "param1", "param2"] (exec form)
```

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

Layering RUN instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

The exec form makes it possible to avoid shell string munging, and to RUN commands using a base image that does not contain /bin/sh.

1. **Note:** To use a different shell, other than '/bin/sh', use the exec form passing in the desired shell. For example, RUN ["/bin/bash", "-c", "echo hello"].
2. **Note:** The exec form is parsed as a JSON array, which means that you must use double-quotes ("") around words not single-quotes ('').

3. **Note:** Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, RUN ["echo", "\$HOME"] will not do variable substitution on \$HOME. If you want shell processing then either use the shell form or execute a shell directly, for example: RUN ["sh", "-c", "echo", "\$HOME"].

Dockerfile 'RUN' sample

Here is our Dockerfile we're going to playing with in this chapter. We'll run instructions from this file step by step by uncommenting and commenting each line.

```
FROM debian:latest
MAINTAINER devops@bogotobogo.com
```

```
# 1 - RUN
RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils
RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop
RUN apt-get clean
```

We have three instructions for RUN, and each of these instruction will create a new container, and at the completion of each instruction, it will become an image.

The following environment setting is to block any terminal output caused by some errors:

```
DEBIAN_FRONTEND=noninteractive
```

The 2nd instruction, htop is to monitor processes in linux system. Then, we removes all packages from the package cache using apt-get clean.

Let's run docker build with v2 instead of v1:

```
$ docker build -t bogodevops/demo:v2 .
Sending build context to Docker daemon 33.56 MB
Sending build context to Docker daemon
Step 0 : FROM debian:latest
--> f6fab3b798be
Step 1 : MAINTAINER k@bogotobogo.com
--> Using cache
--> 511bcbdd59ba
Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils
--> Running in 10ffa5b21a27
...
Setting up apt-utils (0.9.7.9+deb7u6) ...
--> e6e2c03b8efc
Removing intermediate container 10ffa5b21a27
Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop
--> Running in 2fe900ff207c
...
Setting up htop (1.0.1-1) ...
--> fac6e3168cfe
Removing intermediate container 2fe900ff207c
```

DEVOPS MATERIAL

Step 4 : RUN apt-get clean

---> Running in 990373d72cc9

---> 327d400a953c

Removing intermediate container 990373d72cc9

Successfully built 327d400a953c

Listing images:

```
k@laptop:~/Documents/demo$ docker images -a
REPOSITORY      TAG          IMAGE ID       CREATED        VIRTUAL SIZE
bogodevops/demo  v2          327d400a953c   7 minutes ago  96.16 MB
<none><none>      fac6e3168cf   7 minutes ago  96.16 MB
<none><none>      e6e2c03b8efc  7 minutes ago  95.12 MB
bogodevops/demo  v1          511bcbdd59ba   2 hours ago   85.1 MB
debian           latest       f6fab3b798be   2 weeks ago   85.1 MB
<none><none>      f10807909bc5  2 weeks ago   85.1 MB
<none><none>      511136ea3c5a  17 months ago  0 B
```

As we discussed in the previous chapter, if we run this again, it will be completed much faster thanks to caching:

```
k@laptop:~/Documents/demo$ docker build -t bogodevops/demo:v2 .
```

Sending build context to Docker daemon 33.56 MB

Sending build context to Docker daemon

Step 0 : FROM debian:latest

---> f6fab3b798be

Step 1 : MAINTAINER k@bogotobogo.com

---> Using cache

---> 511bcbdd59ba

Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils

---> Using cache

---> e6e2c03b8efc

Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop

---> Using cache

---> fac6e3168cf

Step 4 : RUN apt-get clean

---> Using cache

---> 327d400a953c

Successfully built 327d400a953c

docker run - launching container

```
k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo:v2 /bin/bash
```

root@cf6430ffba1b:# exit

exit

If we drop the :v2 tag in the command:

```
k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo /bin/bash
```

Unable to find image 'bogodevops/demo' locally

Pulling repository bogodevops/demo

2014/11/24 18:55:36 Error: image bogodevops/demo not found

So, to make it work, we need to build default as latest:

```
k@laptop:~/Documents/demo$ docker build -t bogodevops/demo .
```

Now, if look at the images:

```
k@laptop:~/Documents/demo$ docker images -a
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
bogodevops/demo	v2	327d400a953c	32 minutes ago	96.16 MB
bogodevops/demo	latest	327d400a953c	32 minutes ago	96.16 MB

we have a new bogodevops/demo image tagged as 'latest'. So, from now on, we can execute docker run without the 'tag' since it'll look for 'latest' tag by default:

```
k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo /bin/bash
```

```
root@88d48b65ebd7:/#
```

Now, we're in our Docker container for Debian, and htop has been installed.

```
root@88d48b65ebd7:/# htop
```

```
htop.png
```

```
root@88d48b65ebd7:/# exit
```

```
exit
```

We should not see any container:

```
k@laptop:~/Documents/demo$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS

```
POROS NAMES
```

No containers are hanging around!

Dockerfile - CMD

This section is from <http://docs.docker.com/reference/builder/>.

CMD has 3 forms:

CMD ["executable","param1","param2"] (exec form, this is the preferred form)

1. CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
2. CMD command param1 param2 (shell form)

There can only be one CMD instruction in a Dockerfile. If we list more than one CMD then only the last CMD will take effect.

The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case we must specify an ENTRYPOINT instruction as well.

1. **Note:** If CMD is used to provide default arguments for the ENTRYPOINT instruction, both the CMD and ENTRYPOINT instructions should be specified with the JSON array format.
2. **Note:** The exec form is parsed as a JSON array, which means that you must use double-quotes ("") around words not single-quotes ('').
3. **Note:** Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, CMD ["echo", "\$HOME"] will not do variable substitution on \$HOME. If you want shell processing then either use the shell form or execute a shell directly, for example: CMD ["sh", "-c", "echo", "\$HOME"].

4.Dockerfile 'CMD' sample

5. Here is our Dockerfile that we're going to play with in this chapter. We'll run instructions from this file step by step by uncommenting and commenting each line.

```
6. FROM debian:latest
7. MAINTAINER devops@bogotobogo.com
8.
9. # 1 - RUN
10. RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get
    install -yq apt-utils
11. RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop
12. RUN apt-get clean
13.
14. # 2 - CMD
15. CMD ["htop"]
```

16. We have one instruction for CMD, and at the completion of the CMD, it will become an image.

```
17. k@laptop:~/Documents/demo$ docker build -t bogodevops/demo .
18. Sending build context to Docker daemon 33.56 MB
19. Sending build context to Docker daemon
20. Step 0 : FROM debian:latest
21. ---> f6fab3b798be
22. Step 1 : MAINTAINER k@bogotobogo.com
23. ---> Using cache
24. ---> 511bcbdd59ba
25. Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive
    apt-get install -yq apt-utils
26. ---> Using cache
27. ---> e6e2c03b8efc
```

```

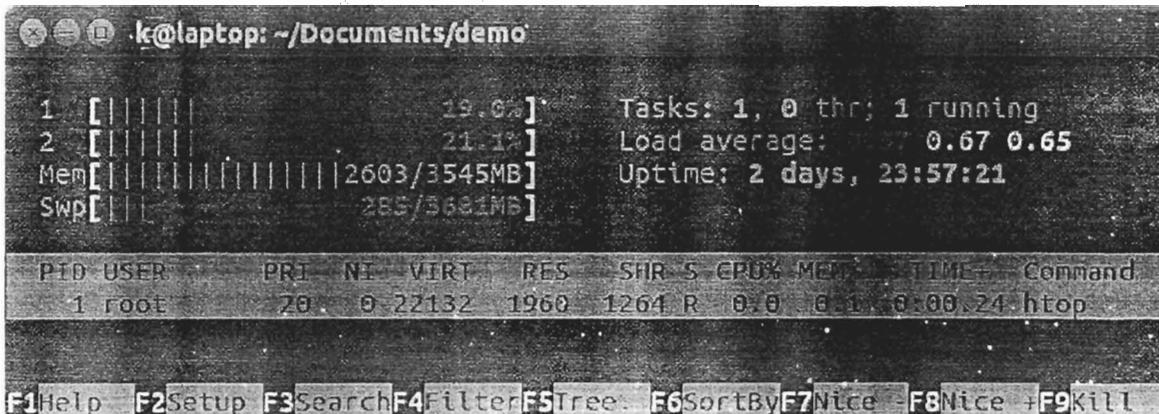
28. Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq
   htop
29. ---> Using cache
30. ---> fac6e3168cfe
31. Step 4 : RUN apt-get clean
32. ---> Using cache
33. ---> 358b5cc4b9fa
34. Step 5 : CMD htop
35. ---> Running in d31a73253846
36. ---> b64547129d16
37. Removing intermediate container d31a73253846
38. Successfully built b64547129d16

```

But unlike in the previous chapter where we ran htop explicitly within the container, this time, it becomes a default environment.

So, even though we issue docker run without passing in any command, we have htop running automatically when the container is created:

```
39. k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo
```



The screenshot shows a terminal window with the title bar 'k@laptop: ~/Documents/demo'. The window displays system statistics and a process list. At the top, it shows CPU usage (Tasks: 1, 0 thr; 1 running), load average (0.67 0.65), uptime (2 days, 23:57:21), memory usage (Mem: 2603/3545MB), and swap usage (Swp: 285/3621MB). Below this is a table of processes:

PID	USER	PRI	VIRT	RES	SIR	CPU%	MEM%	TIME+	Command
1	root	20	0	22132	1960	1264 R	0.0	0:1:0.00	24 htop

At the bottom of the terminal, there is a footer with various keyboard shortcuts: F1 Help, F2 Setup, F3 Search, F4 Filter, F5 Tree, F6 SortBy, F7 Nice, F8 Nice +, and F9 Kill.

We get the htop as soon as we're in the container. It's given us as an environment.

If we pass in /bin/bash, then we'll have bash instead of htop:

```
40. k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo  
/bin/bash  
41. root@00e40007ed7d:/# exit  
42. exit  
  
43.
```

Before we start new thing, we need to remove 'testimage' in our directory:

```
44. k@laptop:~/Documents/demo$ ls  
45. Dockerfile testimage  
46. k@laptop:~/Documents/demo$ rm testimage
```

Then, let's switch our CMD instruction to CMD ["ls", "l"]. Here is our new Dockerfile:

```
47. FROM debian:latest  
48. MAINTAINER k@bogotobogo.com  
49.  
50. # 1 - RUN  
51. RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get  
    install -yq apt-utils  
52. RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop  
53. RUN apt-get clean  
54.  
55. # 2 - CMD  
56. #CMD ["htop"]  
57. CMD ["ls", "l"]
```

Build a new image with the new CMD ["ls", "l"]:

```
58. k@laptop:~/Documents/demo$ docker build -t bogodevops/demo .  
59. Sending build context to Docker daemon 2.56 kB  
60. Sending build context to Docker daemon  
61. Step 0 : FROM debian:latest  
62. ---> f6fab3b798be  
63. Step 1 : MAINTAINER k@bogotobogo.com  
64. ---> Using cache  
65. ---> 511bcbdd59ba  
66. Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive  
    apt-get install -yq apt-utils
```

```

67. ---> Using cache
68. ---> e6e2c03b8efc
69. Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq
    htop
70. ---> Using cache
71. ---> fac6e3168cf8
72. Step 4 : RUN apt-get clean
73. ---> Using cache
74. ---> 358b5cc4b9fa
75. Step 5 : CMD ls -l
76. ---> Running in 717df1a3baa2
77. ---> d2f3de97b6ef
78. Removing intermediate container 717df1a3baa2
79. Successfully built d2f3de97b6ef

```

If we go in our container, it will automatically gives the output from `ls -a`:

```

80. k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo
81. total 68
82. drwxr-xr-x 2 root root 4096 Nov  5 21:37 bin
83. drwxr-xr-x 2 root root 4096 Sep 21 18:17 boot
84. drwxr-xr-x 4 root root  360 Nov 25 05:25 dev
85. drwxr-xr-x 32 root root 4096 Nov 25 05:25 etc
86. drwxr-xr-x 2 root root 4096 Sep 21 18:17 home
87. drwxr-xr-x 8 root root 4096 Nov 25 02:27 lib
88. drwxr-xr-x 2 root root 4096 Nov  5 21:33 lib64
89. drwxr-xr-x 2 root root 4096 Nov  5 21:31 media
90. drwxr-xr-x 2 root root 4096 Sep 21 18:17 mnt
91. drwxr-xr-x 2 root root 4096 Nov  5 21:31 opt
92. dr-xr-xr-x 253 root root   0 Nov 25 05:25 proc
93. drwx----- 2 root root 4096 Nov  5 21:31 root
94. drwxr-xr-x 5 root root 4096 Nov  5 21:37 run
95. drwxr-xr-x 2 root root 4096 Nov  5 21:37 sbin
96. drwxr-xr-x 2 root root 4096 Jun 10 2012 selinux
97. drwxr-xr-x 2 root root 4096 Nov  5 21:31 srv
98. dr-xr-xr-x 13 root root   0 Nov 25 05:25 sys
99. drwxrwxrwt 2 root root 4096 Nov  5 21:37 tmp
100. drwxr-xr-x 16 root root 4096 Nov 25 02:27 usr
101. drwxr-xr-x 17 root root 4096 Nov 25 02:27 var

```

Dockerfile - WORKDIR & ENV

This section is from <http://docs.docker.com/reference/builder/>.

`WORKDIR /path/to/workdir`

The WORKDIR instruction sets the working directory for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

WORKDIR & ENV - sample

Here is our updated Dockerfile:

```
FROM debian:latest

MAINTAINER k@bogotobogo.com


# 1 - RUN

RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils

RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop

RUN apt-get clean


# 2 - CMD

#CMD ["htop"]

#CMD ["ls", "-l"]


# 3 - WORKDIR and ENV

WORKDIR /root

ENV DZ version1
```

Let's build the image:

```
$ docker build -t bogodevops/demo .
```

```
Sending build context to Docker daemon 2.56 kB

Sending build context to Docker daemon

Step 0 : FROM debian:latest

--> f6fab3b798be

Step 1 : MAINTAINER k@bogotobogo.com

--> Using cache

--> 511bcbdd59ba

Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get
install -yq apt-utils

--> Using cache

--> e6e2c03b8efc

Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop

--> Using cache

--> fac6e3168cf8

Step 4 : RUN apt-get clean

--> Using cache

--> 358b5cc4b9fa

Step 5 : WORKDIR /root

--> Running in 2ce95d5fedel

--> a205c4badd68

Removing intermediate container 2ce95d5fedel

Step 6 : ENV DZ version1

--> Running in 6ac629a3506b

--> 6f9de0a5099f

Removing intermediate container 6ac629a3506b
```

```
Successfully built 6f9de0a5099f
```

```
$
```

Here we're using repository name (tag) for the image, and the dot('.') indicates our Dockerfile is in local directory.

What images do we have now?

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
VIRTUAL SIZE			
bogodevops/demo	latest	6f9de0a5099f	About a minute ago
		96.16 MB	
<none>	d2f3de97b6ef		About an hour ago
		96.16 MB	
<none>	e171cd1dd9e7		About an hour ago
		96.16 MB	
<none>	b64547129d16		About an hour ago
		96.16 MB	
bogodevops/demo	v2	358b5cc4b9fa	2 hours ago
		96.16 MB	
bogodevops/demo	v1	511bcbdd59ba	7 hours ago
		85.1 MB	
debian	latest	f6fab3b798be	2 weeks ago
		85.1 MB	

Note the images tagged with <none>. These are the images which had no tag, and left behind when a new image is tagged as 'latest'.

Now we're going to run a new container and run bash inside of it:

```
$ docker run -it --rm bogodevops/demo /bin/bash
```

We can check the WORKDIR and ENV settings in our Dockerfile:

```
root@52a10702207c:~# pwd
/root
root@52a10702207c:~# echo $DZ
version1
root@52a10702207c:~# exit
exit
```

OK. We've got what we expected.

Dockerfile - ADD

```
ADD <src>... <dest>
```

The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resource may be specified but if they are files or directories then they must be relative to the source directory that is being built (the context of the build).

Each <src> may contain wildcards and matching will be done using Go's `filepath.Match` rules. For most command line uses this should act as expected, for example:

```
ADD hom* /mydir/      # adds all files starting with "hom"  
ADD hom?.txt /mydir/   # ? is replaced with any single character
```

The <dest> is the absolute path to which the source will be copied inside the destination container.

Here is our new Dockerfile:

```
FROM debian:latest  
  
MAINTAINER k@bogotobogo.com  
  
# 1 - RUN  
  
RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils  
  
RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop  
  
RUN apt-get clean  
  
# 2 - CMD  
  
#CMD ["htop"]  
#CMD ["ls", "-l"]  
  
# 3 - WORKDIR and ENV  
  
WORKDIR /root  
  
ENV DZ version1  
  
# 4 - ADD  
  
ADD run.sh /root/run.sh
```

```
CMD ["./run.sh"]
```

The run.sh should be referencing current working directory in our local machine.

Here is the run.sh script:

```
#!/bin/sh

echo "The current directory : $(pwd)"

echo "The DZ variable : $DZ"

echo "There are $# arguments: $@"
```

We should build the image:

```
$ docker build -t bogodevops/demo .

Sending build context to Docker daemon 3.584 kB

Sending build context to Docker daemon

Step 0 : FROM debian:latest
--> f6fab3b798be

Step 1 : MAINTAINER k@bogotobogo.com
--> Using cache
--> 511bcbdd59ba

Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get
install -yq apt-utils
--> Using cache
--> e6e2c03b8efc
```

Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop

---> Using cache

---> fac6e3168cf8

Step 4 : RUN apt-get clean

---> Using cache

---> 358b5cc4b9fa

Step 5 : WORKDIR /root

---> Using cache

---> a205c4badd68

Step 6 : ENV DZ version1

---> Using cache

---> 6f9de0a5099f

Step 7 : ADD run.sh /root/run.sh

---> b4a525cd8f8c

Removing intermediate container 81cad1be4425d

Step 8 : CMD ./run.sh

---> Running in 7f9dad902cff

---> b17ff9ebc8f8

Removing intermediate container 7f9dad902cff

Successfully built b17ff9ebc8f8

Then, run a container with no command:

```
$ docker run -it --rm bogodevops/demo
```

The current directory : /root

```
The DZ variable : version1
```

```
There are 0 arguments:
```

If we add a command to docker run, we get this:

```
$ docker run -it --rm bogodevops/demo ./run.sh Hello bogotobogo
```

```
The current directory : /root
```

```
The DZ variable : version1
```

```
There are 2 arguments: Hello bogotobogo
```

Dockerfile - ENTRYPOINT

ENTRYPOINT has two forms:

1. ENTRYPOINT ["executable", "param1", "param2"] (the preferred exec form)
2. ENTRYPOINT command param1 param2 (shell form)

An ENTRYPOINT allows us to configure a container that will run as an executable.

For example, the following will start nginx with its default content, listening on port 80:

```
docker run -i -t --rm -p 80:80 nginx
```

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD. This allows arguments to be passed to the entry point, i.e., docker run <image> -d will pass the -d argument to the entry point. We can override the ENTRYPOINT instruction using the docker run --entrypoint flag.

Here is our updated Dockerfile which includes ENTRYPOINT:

```
FROM debian:latest

MAINTAINER k@bogotobogo.com

# 1 - RUN

RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -yq apt-utils

RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop

RUN apt-get clean

# 2 - CMD

#CMD ["htop"]

#CMD ["ls", "-l"]

# 3 - WORKDIR and ENV

WORKDIR /root

ENV DZ version1

# 4 - ADD

ADD run.sh /root/run.sh

#CMD ["/root/run.sh"]

# 5 - ENTRYPOINT (vs CMD)
```

```
ENTRYPOINT ["../run.sh"]
```

```
CMD ["arg1"]
```

Build our image again:

```
$ docker build -t bogodevops/demo .  
Sending build context to Docker daemon 3.584 kB  
Sending build context to Docker daemon  
Step 0 : FROM debian:latest  
--> f6fab3b798be  
Step 1 : MAINTAINER k@bogotobogo.com  
--> Using cache  
--> 511bcbdd59ba  
Step 2 : RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get  
install -yq apt-utils  
--> Using cache  
--> e6e2c03b8efc  
Step 3 : RUN DEBIAN_FRONTEND=noninteractive apt-get install -yq htop  
--> Using cache  
--> fac6e3168cf8  
Step 4 : RUN apt-get clean  
--> Using cache  
--> 358b5cc4b9fa  
Step 5 : WORKDIR /root  
--> Using cache
```

---> a205c4badd68

Step 6 : ENV DZ version1

---> Using cache

---> 6f9de0a5099f

Step 7 : ADD run.sh /root/run.sh

---> Using cache

---> c7ecd3c5437e

Step 8 : ENTRYPOINT ./run.sh

---> Running in 2f84e971ba97

---> 9c6bcba955d9

Removing intermediate container 2f84e971ba97

Step 9 : CMD arg1

---> Running in 42b50c05e9f8

---> ff6f9d2ad977

Removing intermediate container 42b50c05e9f8

Successfully built ff6f9d2ad977

Container run without any argument:

```
$ docker run -it --rm bogodevops/demo
```

The current directory : /root

The DZ variable : version1

There are 1 arguments: arg1

It still runs run.sh shell. If we pass in something like /bin/bash:

```
$ docker run -it --rm bogodevops/demo /bin/bash  
  
The current directory : /root  
  
The DZ variable : version1  
  
There are 1 arguments: /bin/bash
```

Still it runs run.sh file while /bin/bash was passed in as an argument.

It can be used multiple times in the one Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction. For example:

```
WORKDIR /a  
WORKDIR b  
WORKDIR c  
RUN pwd
```

The output of the final pwd command in this Dockerfile would be /a/b/c.

The WORKDIR instruction can resolve environment variables previously set using ENV. We can only use environment variables explicitly set in the Dockerfile. For example:

```
ENV DIRPATH /path
```

```
WORKDIR $DIRPATH/$DIRNAME
```

The output of the final pwd command in this Dockerfile would be /path/\$DIRNAME.

ENV <key><value>

The ENV instruction sets the environment variable <key> to the value <value>. This value will be passed to all future RUN instructions. This is functionally equivalent to prefixing the command with <key>=<value>.

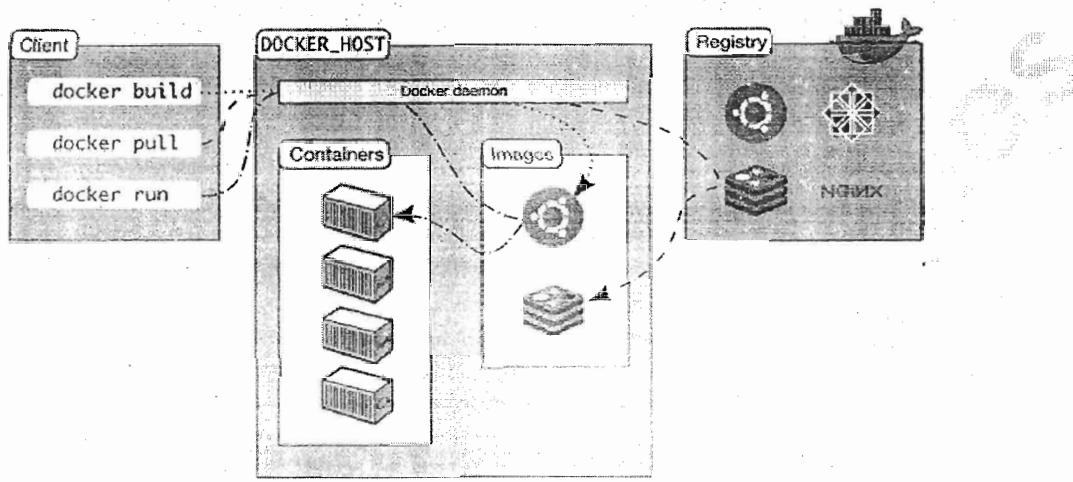
The environment variables set using ENV will persist when a container is run from the resulting image. We can view the values using docker inspect, and change them using docker run --env <key>=<value>.

Note: One example where this can cause unexpected consequences, is setting ENV DEBIAN_FRONTEND noninteractive. Which will persist when the container is run interactively; for example: docker run -t -i image bash.

Introduction

Docker images are used to create Docker containers.

Docker images are the build component of Docker.



Picture credit : Understand the architecture

We create Docker containers using [base] images. An image can be basic, with nothing but the operating-system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch.

When we build images with docker, each action taken (i.e. a command executed such as **apt-get install**) forms a new layer on top of the previous one. These base images then can be used to create new containers.

Docker registries hold images.

These are public or private stores from which we can upload or download images.

The public Docker registry is provided with the Docker Hub.

What happens to an image at "docker run"?

We use "docker run" command from client to tell the Docker daemon to run a container, for example:

```
$ docker run -it ubuntu:latest /bin/bash
```

Basically, it's a container "launch" command.

At **docker**, a Docker client is launched and at **run** subcommand, a new container will be launched.

The new container will be built from **ubuntu** base image with "latest" tag.

Here *i*: interactive, : terminal.

As described in the official document, here are the things happening under the hood:

1. **Pulls the ubuntu image:**

Docker checks for the presence of the ubuntu image and, if it doesn't exist locally on the host, then Docker downloads it from Docker Hub.

If the image already exists, then Docker uses it for the new container.

2. **Creates a new container:**

Once Docker has the image, it uses it to create a container.

3. **Allocates a filesystem and mounts a read-write layer:**

The container is created in the file system and a read-write layer is added to the image.

4. **Allocates a network and sets up an IP address:**

Creates a network interface that allows the Docker container to talk to the local host.

5. Executes a process that we specify:

Runs our application.

Search images - docker search

•**docker search** searches registry for image.

```
$ docker search ubuntu
```

NAME	STARS	OFFICIAL	AUTOMATED	DESCRIPTION	4416	[OK]
ubuntu	Ubuntu	Ubuntu	Ubuntu	Ubuntu is a Debian-based Linux operating s...	4416	[OK]
ubuntu-upstart	for ...	65	[OK]	Upstart is an event-based replacement		

Download images - docker pull

docker pull pulls an image from registry to local machine.

```
$ docker pull ubuntu
latest: Pulling from ubuntu
20ee58809289: Pull complete
f905badeb558: Pull complete
119df6bf2a3a: Pull complete
94d6eea646bc: Pull complete
```

```
bb4eabee84bf: Pull complete
Digest:
sha256:85af8b61adffea165e84e47e0034923ec237754a208501fce5dbeecbb197062c
Status: Downloaded newer image for ubuntu:latest
```

Docker images can consist of multiple layers.

In the example above, the image consists of five layers (20ee58809289,...,bb4eabee84bf).

We can use a tag to specify what to download. For example, 'latest' for tag:

```
$ docker pull ubuntu:latest
```

Listing images - docker images

To list the images on the host:

docker images				
REPOSITORY	CREATED	VIRTUAL SIZE	TAG	IMAGE ID
ubuntu			latest	
bb4eabee84bf	2 weeks ago	124.8 MB		
ubuntu			16.04	
bb4eabee84bf	2 weeks ago	124.8 MB		
centos			7	
2a332da70fd1	9 weeks ago	196.8 MB		
ubuntu			trusty	
9bc953763843	10 weeks ago	188 MB		
debian			latest	
cea663c8c811	10 weeks ago	125.1 MB		
centos			latest	
ce20c473cd8a	9 months ago	172.3 MB		

Running container - docker run

Now we want to launch a container:

```
$ docker run -it ubuntu:latest /bin/bash
```

We're logged in as a root:

```
root@859d4a27d4c8:/# whoami  
root
```

Check the OS:

```
root@859d4a27d4c8:/# cat /etc/*release  
DISTRIB_ID=Ubuntu  
DISTRIB_RELEASE=16.04
```

Let's check what processes are currently running:

```
root@859d4a27d4c8:/# top
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+
COMMAND										
1	root	20	0	18232	2032	1556	S	0.0	0.1	0:00.09
bash										
14	root	20	0	36628	1704	1272	R	0.0	0.0	0:00.02
top										

As we can see there are only two processes, and they are isolated ones from the host processes.

Getting out of a container without stopping it

"Ctrl + P + Q" will do the trick:

```
root@859d4a27d4c8:/#
```

```
k@laptop:~$
```

From the PID, we can check the "top" on the host is using different space from the "top" on the docker container:

```
k@laptop:~$ ps aux | grep top
```

```
root      14512  0.0  0.0  36628  1704 pts/17    S+   22:01   0:00 top
```

Now, we may want to get back to our container again via "attach". To do that, we need to know the "CONTAINER ID":

```
k@laptop:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
859d4a27d4c8ubuntu:latest		"/bin/bash"	5 minutes ago
Up 5 minutes		cranky_swartz	

Let's do attach:

```
k@laptop:~$ docker attach 859d4a27d4c8
root@859d4a27d4c8:/#
```

Stops a container

We can stop the container and get out of it via "Ctrl + D":

```
root@859d4a27d4c8:/# exit  
  
k@laptop:~$
```

If we issue "docker ps" command again, we see the container is not running anymore:

```
k@laptop:~$ docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES  
859d4a27d4c8        ubuntu:latest       "/bin/bash"        31 minutes ago   Exited (0) 4 minutes ago   cranky_swartz  
k@laptop:~$
```

We can list all containers including old ones that stopped running using "docker ps -a":

```
k@laptop:~$ docker ps -a  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES  
COMMAND  
PORTS  
NAMES  
859d4a27d4c8        ubuntu:latest       "/bin/bash"        31 minutes ago   Exited (0) 4 minutes ago   cranky_swartz  
...  
k@laptop:~$
```

Running a container in a background

To run a container in background, we use "docker start" command:

```
k@laptop:~$ docker start cranky_swartz
cranky_swartz
k@laptop:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
859d4a27d4c8 ago	ubuntu:latest	"/bin/bash"	36 minutes
	Up 4 seconds	cranky_swartz	

Check what processes are running inside a container

We can check what processes are running inside a container using "docker top":

```
k@laptop:~$ docker top cranky_swartz
```

UID	PID	PPID	C
STIME	TTY	TIME	CMD
root	15680	1559	0
22:33	pts/17	00:00:00	/bin/bash

Stop a container running in background

To stop a container running in background, we use "docker stop" command:

```
k@laptop:~$ docker stopcranky_swartz
```

What's in the system

What's in our system related to the Docker containers?

```
root@laptop:/# tree /var/lib/docker/containers
|-- 859d4a27d4c883db39e68590f1d1d2c340f8775a94fe721eba85111d3b79c1fe
  |-- config.json
  |-- hostconfig.json
  |-- hostname
  |-- hosts
  |-- resolv.conf
  |-- resolv.conf.hash
```

Note that there are no binary images for the container which makes docker consumes much less space compare to the other virtual tools.

Detach mode run

We can run a container in a "detach" mode, and later we can attach to it:

Note we need to use "-it" so that we can do something with the container after attaching. Also, we used "--name" to give our own name to the container.



DEVOPS MATERIAL

```
k@laptop:~$ docker run -d -it --name=yaong ubuntu:16.04 /bin/bash  
6bbba8e00d68a9b9c38bd7fdbd807dae01b9329d3b5ecd7ad2918305743bf5ea
```

Ok, it's started, and we can check it:

k@laptop:~\$ docker ps				
CONTAINER ID	IMAGE	COMMAND	CREATED	
STATUS	PORTS	NAMES		
6bbba8e00d68	ubuntu:16.04	"/bin/bash"	About a	
minute ago	Up About a minute			yaong

To stop it:

```
k@laptop:~$ docker stop 6bbbba8e00d68
```

```
k@laptop:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	

```
k@laptop:~$
```

Install on Ubuntu 14.04

Docker is an open-source project that makes creating and managing Linux containers really easy. Containers are like extremely lightweight VMs - they allow code to run in isolation from other containers but safely share the machine's resources, all without the overhead of a hypervisor.

In this article, we'll install Docker using Docker-managed release packages as well as Ubuntu managed packages. Using Docker-managed release packages ensures us get the latest release of Docker.

Docker requires a 64-bit installation regardless of our Ubuntu version.

Ubuntu Trusty comes with a 3.13.0 Linux kernel, and a `docker.io` package which installs Docker 0.9.1 and all its prerequisites from Ubuntu's repository.

Ubuntu (and Debian) contain a much older KDE3/GNOME2 package called `docker`, so the package and the executable are called `docker.io`.

To install Docker properly, our kernel must be 3.10 at minimum. The latest 3.10 minor version or a newer maintained version are also acceptable. That's because Kernels older than 3.10 lack some of the features required to run Docker containers. These older versions are known to have bugs which cause data loss and frequently panic under certain conditions.

We can check our current kernel version:

```
$ uname -r
```

```
3.13.0-40-generic
```

So, we met the Prerequisites!

Let's start.

First, update our package manager. Downloads the package lists from the repositories and "updates" them to get information on the newest versions of packages and their dependencies by synchronizing the package index files fetched from `/etc/apt/sources.list`.

```
$ sudo apt-get update
```

Install using Ubuntu-managed packages

To install Docker, we need to use :

```
$ sudo apt-get install docker.io
```

To make the shell easier to use, we need to create a symlink since /usr/local/bin is for normal user programs not managed by the distribution package manager. The following command overwrites the link (/usr/local/bin/docker):

```
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

To enable tab-completion of Docker commands in BASH, either restart BASH or:

```
$ source /etc/bash_completion.d/docker.io
```

Or

```
$ source /etc/bash_completion.d/docker
```

To check if Docker is running:

```
$ ps aux | grep docker
```

```
root 8725 0.0 0.2 362736 9352 ? Ssl 10:05 0.00 /usr/bin/docker.io -d
```

If we want to run docker as root user, we should add a user (in my case, 'k') to the docker group:

```
$ sudo usermod -aG docker k
```

```
$ id k
```

```
uid=1000(k) gid=1000(k) groups=1000(k),4(adm),24(cdrom),27(sudo),30(dip),33(www-data),46(plugdev),108(lpadmin),124(sambashare),1005(svn),131(docker)
```

Install using Docker-managed release packages

We may want to uninstall the Docker if it's already there:

```
$ sudo apt-get remove docker.io
```

Then, get the Docker package, and install it from the Docker script:

```
$ curl -sSL https://get.docker.com/ | sh
```

or

```
$ curl -sSL -qo- https://get.docker.com/ | sh
```

The vertical line pipe character before the **sh** pipes the output from standard output(**-o**) to the terminal to the **sh** command which says to execute the file that was downloaded in the terminal as a program, assuming that the file that was downloaded first has its permissions set to allow executing the file as a program.

Let's run daemon:

```
$ sudo docker -d
```

To verify docker is installed correctly:

```
$ sudo docker run hello-world
```

This command downloads a test image and runs it in a container as we can see from the output:

```
$ sudo docker run hello-world
```

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

535020c3e8ad: Pull complete

af340544ed62: Already exists

library/hello-world:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.

Digest: sha256:d5fb996e6562438f7ea5389d7da867fe58e04d581810e230df4cc073271ea52

Status: Downloaded newer image for hello-world:latest

Hello from Docker.

⊕ This message shows that your installation appears to be working correctly.

Checking Docker version

To see which version of Docker is installed:

```
$ docker -v
```

```
Docker version 1.8.1, build d12ea79
```

Install latest Docker from Ubuntu Package

To install the latest version, we need to add the Docker repository key to our local keychain:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys  
36A1D7869245C8950F966E92D8576A8BA88D21E9
```

```
[sudo] password for k:
```

```
Executing: gpg --ignore-time-conflict --no-options --no-default-keyring --homedir  
/tmp/tmp.0gMbMUENwN --no-auto-check-trustdb --trust-model always --keyring  
/etc/apt/trusted.gpg --primary-keyring /etc/apt/trusted.gpg --keyring  
/etc/apt/trusted.gpg.d/nagiosinc-ppa.gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys  
36A1D7869245C8950F966E92D8576A8BA88D21E9
```

```
gpg: requesting key A88D21E9 from hkp server keyserver.ubuntu.com
```

```
gpg: key A88D21E9: "Docker Release Tool (releasedocker) " not changed
```

```
gpg: Total number processed: 1
```

```
gpg: unchanged: 1
```

Add the Docker repository to our apt sources list:

```
$ sudosh -c "echo deb http://get.docker.io/ubuntu docker main > /etc/apt/sources.list.d/docker.list"
```

```
$ cat /etc/apt/sources.list.d/docker.list  
  
deb http://get.docker.io/ubuntu docker main
```

Then, update and install the lxc-docker package:

```
$ sudo apt-get update  
  
$ sudo apt-get install lxc-docker  
  
$ sudo ln -sf /usr/bin/docker /usr/local/bin/docker  
  
$ docker -v  
  
Docker version 1.9.1, build a34a1d5
```

We can check if our Docker is installed by running the following command as well:

```
$ docker info  
  
Containers: 12  
  
Images: 77  
  
Server Version: 1.9.1  
  
Storage Driver: aufs  
  
Root Dir: /var/lib/docker/aufs  
  
Backing Filesystem: extfs  
  
Dirs: 101  
  
Dirperm1 Supported: false  
  
Execution Driver: native-0.2  
  
Logging Driver: json-file  
  
Kernel Version: 3.13.0-40-generic
```

Operating System: Ubuntu 14.04.4 LTS

...

To verify that everything has worked as expected, we can check if Docker downloads the ubuntu image, and then start bash in a container:

```
$ sudo docker run -i -t ubuntu /bin/bash  
...  
Status: Downloaded newer image for ubuntu:latest  
# exit  
exit  
k@laptop:/etc/apt/sources.list.d$
```

We were able to start bash in a container. And we can check if the image for ubuntu is there:

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	latest	5506de2b643b	4 weeks ago	199.3 MB

permission denied for docker.sock

We may get the following message when we use docker without sudo:

```
$ docker info
```

```
2014/11/21 19:59:03 Get http://var/run/docker.sock/v1.15/info: dial unix /var/run/docker.sock:  
permission denied
```

That's because my user name "k" does not belong "docker" group!

So, let's resolve the issue. First, we need to check if there is docker group:

```
k@laptop:~$ cat /etc/group
```

```
root:x:0:
```

```
daemon:x:1:
```

```
bin:x:2:
```

```
sys:x:3:
```

```
adm:x:4:syslog,k
```

```
...
```

```
docker:x:131:
```

```
k@laptop:~$
```

Ok, docker is one of the names in the group. So, I need to add my username "k" to the docker group:

```
k@laptop:~$ sudo adduser k docker
```

```
[sudo] password for k:
```

```
Adding user k to group docker
```

Or:

```
$ sudo usermod -aG docker k
```

Restart the Docker daemon:

```
k@laptop:~$ sudo service docker restart
```

Now, there is no more permission error when we issue docker info without sudo:

```
k@laptop:~$ docker info
```

Containers: 1

Images: 7

Storage Driver: aufs

Root Dir: /var/lib/docker/aufs

Dirs: 9

Execution Driver: native-0.2

Kernel Version: 3.13.0-35-generic

Operating System: Ubuntu 14.04.1 LTS

WARNING: No swap limit support

Note: In my case, since I added myself to the group, I had to log out and log back in.

To check the groups I belong:

```
k@laptop:~$ groups
```

```
kadmcdromsudo dip plugdevlpadminsambashare docker svn
```

Why we get permission denied for docker.sock

This section is from:

<https://docs.docker.com/installation/ubuntu/#giving-non-root-access>.

Giving non-root access

The docker daemon always runs as the **root** user, and since Docker version 0.5.2, the docker daemon binds to a **Unix socket** instead of a **TCP port**. By default that Unix socket is owned by the user root, and so, by default, you can access it with **sudo**.

Starting in version 0.5.3, if you (or your Docker installer) create a Unix group called **docker** and add users to it, then the docker daemon will make the ownership of the Unix socket read/writable by the **docker group** when the daemon starts. The docker daemon must always run as the root user, but if you run the docker client as a user in the docker group then you don't need to add sudo to all the client commands. From Docker 0.9.0 you can use the **-G** flag to specify an alternative group.

Warning: The docker group (or the group specified with the **-G** flag) is root-equivalent; see Docker Daemon Attack Surface for details.

Example:

```
# Add the docker group if it doesn't already exist.
```

```
$ sudogroupadd docker
```

```
# Add the connected user "${USER}" to the docker group. # Change the user name to  
match your preferred user. # You may have to logout and log back in again for # this to  
take effect.
```

```
$ sudo gpasswd -a ${USER} docker
```

```
# Restart the Docker daemon. # If you are in Ubuntu 14.04, use docker.io instead of docker
```

```
$ sudo service docker restart
```

chkconfig or sysv-rc-conf

The **chkconfig** utility is a command-line tool that allows us to specify in which **runlevel** to start a selected service, as well as to list all available services along with their current setting.

On CentOS, we can simply turn it on to start it automatically:

```
$ sudo chkconfig docker on
```

On Ubuntu > 14, the equivalent to chkconfig is **sysv-rc-conf**. So, we use this instead of chkconfig for Run-level configuration. The **sysv-rc-conf** an easily communicate and managing with **/etc/rc{runlevel}.d/** symlinks.

See the **runlevel** table below:

ID	Name
0	Halt
1	Single-user mode
2	Multiuser mode
3	Multiuser mode with networking
4	Not used/user-definable
5	Start the system normally with appropriate display manager (with GUI)
6	Reboot

We may want to install the **sysv-rc-conf** package using apt-get command, and then run **sysv-rc-conf**:

```
$ sudo apt-get install sysv-rc-conf
```

```
$ sudsosysv-rc-conf --list
```

```
acpid
```

```
anacron
```

```
apache2 0:off 1:off 2:on 3:on 4:on 5:on 6:off
```

```
...
```

```
docker
```

```
docker.io
```

```
...
```

```
vmware-works 0:off 2:on 3:on 4:on 6:off
```

```
x11-common S:on
```

```
$ sudsosysv-rc-conf docker on
```

```
$ sudo sysv-rc-conf --list  
...  
docker    2:on   3:on    4:on    5:on  
docker.io  
...  
...
```

docker run -v

Though we executed docker run commands with various argument combinations in the previous chapter (More on docker run command (docker run -it, docker run --rm, etc.)), docker run was not doing useful operations.

So, in this chapter, we'll learn more about docker run commands that do more useful things.

We're going to run docker run command with -v argument:

```
k@laptop:~$ docker run -help
```

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

-v, --volume=[] Bind mount a volume (e.g., from the host: **-v /host:/container**, from Docker: **-v /container**)

--volumes-from=[] Mount volumes from the specified container(s)

Let's do it:

```
k@laptop:~$ docker run -it --rm -v /home/k/myDocker:/k busyboxsh
```

```
/ # cd k
```

```
/k # ls
```

```
/k # touch bogotobogo.txt
```

```
/k # exit
```

```
k@laptop:~$ cd /home/k/myDocker
```

```
k@laptop:~/myDocker$ ls
```

```
bogotobogo.txt
```

```
k@laptop:~/myDocker$ ls -la
```

```
total 8
```

```
drwxrwxr-x 2 k k 4096 Nov 22 12:16 .
```

```
drwxr-xr-x 89 k k 4096 Nov 22 12:15 ..
```

```
-rw-r--r-- 1 root root 0 Nov 22 12:16 bogotobogo.txt
```

Here in the argument, we're binding a folder in our local machine (`/home/k/myDocker`) with the folder in Docker container (`k`) so that they can share files:

```
-v /home/k/myDocker:/k busybox
```

As we can see from the output, the two folders are sharing a file that was created in the container. Also note that the permissions on the file, "bogotobogo.txt". It was created with "root" user permission. A special care should be given so as not to the permission things not to be messed up. There is a way to work around it:

```
k@laptop:~/myDocker$ sudorm bogotobogo.txt
```

```
[sudo] password for k:
```

```
k@laptop:~/myDocker$ id k
```

```
uid=1000(k) gid=1000(k)  
groups=1000(k),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare),1005  
(svn),131(docker)
```

```
k@laptop:~/myDocker$ docker run -it --rm -v /home/k/myDocker:/k -u 1000:1000 busyboxsh
```

```
/ $ cd k
```

```
/k $ touch bogotobogo.txt
```

```
/k $ ls
```

```
bogotobogo.txt
```

```
/k $ exit
```

```
k@laptop:~/myDocker$ ls
```

```
bogotobogo.txt
```

```
k@laptop:~/myDocker$ ls -la
```

```
total 8
```

```
drwxrwxr-x 2 k k 4096 Nov 22 12:28 .
```

```
drwxr-xr-x 89 k k 4096 Nov 22 12:15 ..
```

```
-rw-r--r-- 1 k k 0 Nov 22 12:28 bogotobogo.txt
```

This way we can keep the ownership remains the same. Since this may cause another problem, we need to be very careful when we mount the volume to the container.

docker run with port argument

In this section, we'll learn how to use port argument, -p, with Nginx web server.

```
k@laptop:~$ docker run -help
```

Usage: **docker run [OPTIONS] IMAGE [COMMAND] [ARG...]**

Run a command in a new container

-d, --detach=false Detached mode: run the container in the background and print the new container ID

-P, --publish-all=false Publish all exposed ports to the host interface

-p, --publish=[] Publish a container's port to the host

format: ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | containerPort

(use 'docker port' to see the actual mapping)

We'll map container's port 80 to the host.

```
k@laptop:~/myDocker$ docker run -d -p 80 nginx
```

Unable to find image 'nginx' locally

nginx:latest: The image you are pulling has been verified

f10807909bc5: Pull complete

f6fab3b798be: Pull complete

d21beea329f5: Pull complete

04499cf33a0e: Pull complete

34806d38e48d: Pull complete

4cae2a7ca6bb: Pull complete

23f7e46a4bbc: Pull complete

9dfd3384699f: Pull complete

475220486d0e: Pull complete

30bb1926e17f: Pull complete

ef45dc12127b: Pull complete

e426f6ef897e: Pull complete

511136ea3c5a: Already exists

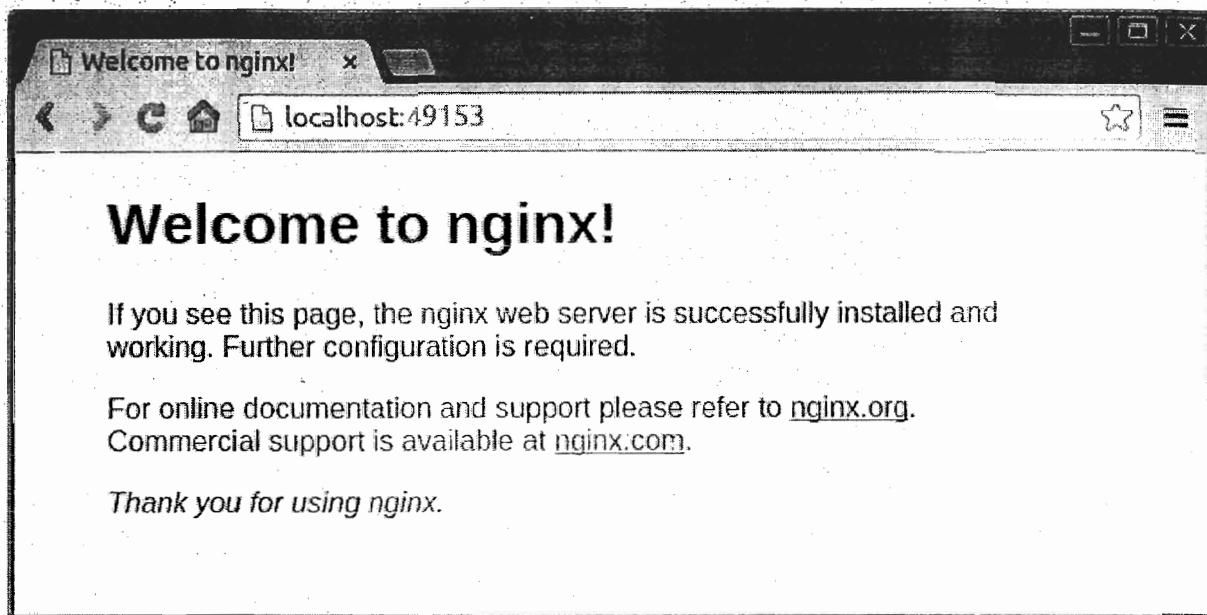
Status: Downloaded newer image for nginx:latest

72780def6c7ea4e14e497810722d297a8f4f8157009ea122e9345b76b0bab822

k@laptop:~/myDocker\$ docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
72780def6c7e	nginx:latest	"nginx -g 'daemon off;'	7 seconds ago	Up 5 seconds
0.0.0.0:49153->80/tcp	condescending_elion			443/tcp

We did docker run in detached mode (-d) meaning making it running in background. As we can see from the PORT column in the output docker ps command, the Nginx on Docker container mapped port 80 of Nginx to 49153 port of host. So the port 49153 on local machine will go to port 80 of Docker Nginx.



If we want to specify the exact port on host, suppose 8099, we can do it:

```
k@laptop:~/myDocker$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
72780def6c7e	nginx:latest	/nginx -g 'daemon off;'	23 minutes ago	Up 23 minutes
443/tcp, 0.0.0.0:49153->80/tcp	descending_elion			

Stop the container and remove it:

```
k@laptop:~/myDocker$ docker stop 72780def6c7e
```

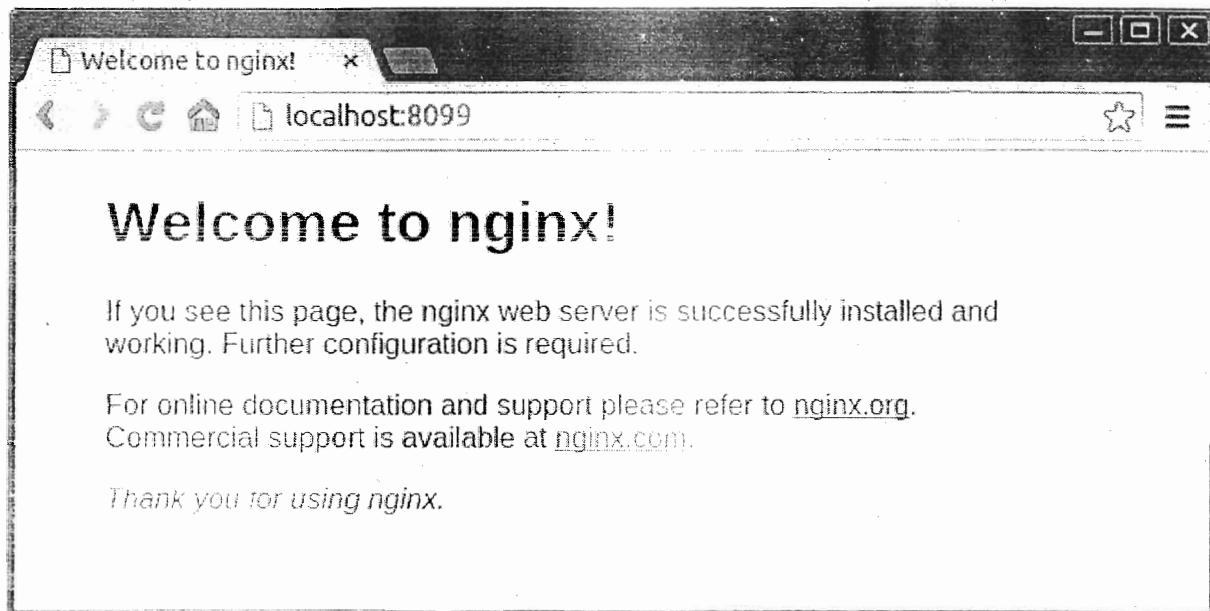
```
72780def6c7e
```

```
k@laptop:~/myDocker$ docker rm 72780def6c7e
```

```
72780def6c7e
```

Then, specify the host port number (8099) we want to use:

```
k@laptop:~/myDocker$ docker run -p 8099:80 -d nginx
5444b242bf8d1f01229e11e0838ce11e918df03a038540b5c2dd66ec52023f08
k@laptop:~/myDocker$
```



We can always check how the ports are mapped:

```
k@laptop:~/myDocker$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
5444b242bf8d	nginx:latest	"nginx -g 'daemon off;'	5 minutes ago	Up 5 minutes
443/tcp, 0.0.0.0:8099->80/tcp	mad_ptolemy			

docker run -e : passing environment variable

Occasionally, we may need to pass in environment variable to docker run with -e argument.

```
k@laptop:~/myDocker$ docker run -it --rm -e DOCK_VAR=BOGOTOBOGO busyboxsh
/ # echo $DOCK_VAR
BOGOTOBOGO
/ # exit
```

Passing in environment variables is useful when we deal with MySQL or password etc.

The 2nd sample : docker run -v

Suppose we have a file written in go, but we do not have the compiler. So, we decided to use Docker's go image, run the container, and compile it. Since we can share a file between host and container, after the compile, we get the executable on our host machine.

Here is our go file:

```
k@laptop:~/golang$ cat HelloWorld.go
package main
import "fmt"

func main() {
```

```
fmt.Println("Hello World!");  
}
```

docker run:

```
k@laptop:~/golang$ docker run -it --rm -v $(pwd):/go -u 1000:1000 golang:latest go build -o HelloWorld.out
```

Unable to find image 'golang:latest' locally

latest: Pulling from golang

```
902b87aaaec9: Pull complete
```

...
golang:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.

```
Digest: sha256:2d94c130703e17f679141f392ce545673397fd51e9395fcefe965005610368bb
```

Status: Downloaded newer image for golang:latest

```
k@laptop:~/golang$ ls  
HelloWorld.go HelloWorld.out
```

Now, we have an executable **HelloWorld.out** on our local host machine. Let's run it:

```
k@laptop:~/golang$ ./HelloWorld.out  
Hello World!
```

Note that we were able to compile and run the **go** file even though **go** is not installed on our machine!

Note also that the **owner:group** is **k:k** because we used **1000:1000** in the **docker runcommand**:

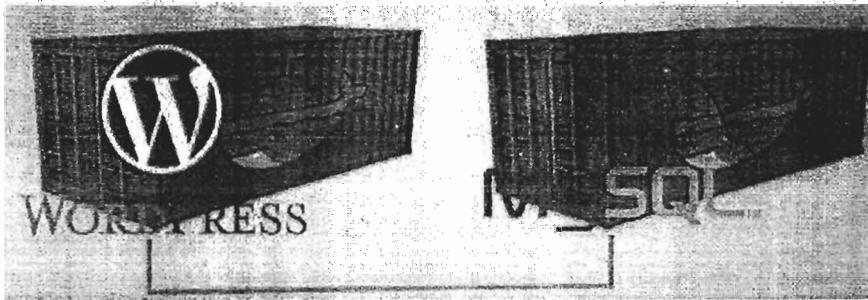
```
k@laptop:~/golang$ ls -la
total 2320
drwxrwxr-x  2 k k  4096 Aug 19 23:39 .
drwxr-xr-x 129 k k  4096 Aug 19 23:19 ..
-rw-rw-r--  1 k k   75 Aug 19 23:21 HelloWorld.go
-rwxr-xr-x  1 k k 2361088 Aug 19 23:39 HelloWorld.out
```

How to create one container that can access a service on another?

In this article, we'll show how Docker does it. We'll launch WordPress site using Docker.

In this article, we'll create 3 different Docker images:

1. One image for WordPress itself.
2. It's going to be linked to MySQL container.
3. Then, it will be storing data. We call it "Data Container". Unlike the WordPress and MySQL containers, Data container can be run only once, and does not have to be running. It's enough to just define the volume and share the volume with MySQL. Then, the WordPress will be linked to the volume so that it can have an access to MySQL database's port, locally instead of remotely.



Where Docker stores its files?

Let's look inside **/var/lib/docker** folder in Ubuntu:

```
$ sudo ls /var/lib/docker
aufs      execdriver      init      repositories-aufs trust  volumes
containers graph linkgraph.dbtmp      vfs.
```

When Docker starts an image and create a container that has a volume defined, it will store those in a UUID named directory inside two directories (**volumes** and **vsf**). Now the two directories are empty since we haven't defined any volume:

```
k@laptop:~$ sudo ls /var/lib/docker/volumes
k@laptop:~$ sudo ls /var/lib/docker/vfs/dir
```

Creating a volume

We're going to create a volume using tiny linux distro called busybox.

Docker system should maintain the consistency of the data store we're creating now in **/var/lib/mysql** directory.

For reference, here are the **docker run** args we're going to use to create a volume:

1. **-v**

Bind mount a volume (e.g., from the host: -v /host:/container, from Docker: -v /container)

2. **--name**

Assign a name to the container

3. **-d**

Detached mode: run the container in the background and print the new container ID.

Then, run docker. The following command just runs a container and exits as we can see from the output of **docker ps -a**:

```
k@laptop:~$ docker run -v /var/lib/mysql --name=my_datastore -d busybox echo "My Datastore"
```

```
k@laptop:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
6a7e05a1e120	busybox	"echo 'My Datastore'"	3 minutes ago	Exited (0) 3 minutes ago
	my_datastore			

However, by running once, it has created a volume and this is the key: data container does not have to be running. If we inspect **my_datastore**:

```
k@laptop:~$ docker inspect my_datastore
```

```
[{"Id": "6a7e05a1e120", "Created": "2017-05-10T11:44:10.000Z", "Status": "Exited (0) 3 minutes ago", "Image": "busybox", "Name": "my_datastore", "Ports": [{"HostPort": "0", "ContainerPort": "0"}], "Config": {"Cmd": ["echo", "'My Datastore'"], "Image": "busybox"}, "HostConfig": {"Binds": ["/var/lib/mysql"], "Links": null, "PortBindings": {}, "CgroupParent": null, "Dns": null, "DnsSearch": null, "ExtraHosts": null, "IpcMode": "private", "LinksAsAliases": false, "LogConfig": {"Config": {"MaxLineSize": 1048576, "MaxLogSize": 1048576, "Label": null}, "Type": "json-file"}, "Memory": null, "MemoryReservation": null, "OomKillDisable": false, "PidMode": "private", "UTSMode": null}, "Mounts": [{"ContainerPath": "/var/lib/mysql", "HostPath": "/var/lib/docker/volumes/7ec328ed9c22b7190e9998d13d3dc49d8d8249e336fd84667c5bf7fe724b70bc/data", "Type": "bind"}], "NetworkSettings": {"Bridge": "bridge", "GlobalIPv6": false, "LinkLocalIPv6Address": null, "LinkLocalIPv6PrefixLen": null, "MacAddress": "02:42:AC:11:00:02", "NetworkMode": "bridge", "Ports": {}, "SecondaryIPv4Cidrs": null, "SecondaryIPv6Cidrs": null, "SecondaryPorts": null, "Server": "0.0.0.0", "ServerPort": 0, "TLSConfig": null}, "Volumes": {}, "VolumesRW": {""/var/lib/mysql": true}}]
```

```
...
}
}
]
k@laptop:~$
```

```
vfs
  dir
  volumes
    7ec328ed9c22b7190c9998d13d3dc49d8d8249e336fd84667c5bf7fc724b70bc
```

So, from now on, whenever anything write to will be stored in this area of local host system!

MySQL container

Now, we need to start MySQL container (OFFICIAL REPOSITORY mysql).

```
k@laptop:~$ docker run --name my_mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw --
volumes-from my_datastore -d mysql
```

```
...
Status: Downloaded newer image for mysql:latest
```

```
9173d3457bc8ee440fbf5df30d30db8102ede04d4e344c9593a1421057f99786
```

Note that we added **my_mysql** for the container name, passed environment variable **MYSQL_ROOT_PASSWORD**, and most importantly, added our datastore for **--volumes-from**, and running it in detached mode.

```
k@laptop:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
9173d3457be8	mysql	/entrypoint.sh mysq	2 minutes ago	Up 2 minutes
3306/tcpmy_mysql				

Now, we have MySQL up and running.

If we inspect **my_mysql** container, we can see it's using the volume we specified:

```
k@laptop:~$ docker inspect my_mysql
...
{
  "Volumes": {
    "/var/lib/mysql": "/var/lib/docker/volumes/7ec328ed9c22b7190c9998d13d3de49d8d8249e336fd84667c5bf7fc724b70bc/_data"
  },
  "VolumesRW": {
    "/var/lib/mysql": true
  }
}
```

Container for WordPress site

Note our WordPress container should have a link to our MySQL container!

Let's look into the inspection in the previous section, and we can locate the "ExposedPorts":

```
k@laptop:~$ docker inspect my_mysql
...
```

```
"ExposedPorts": {
    "3306/tcp": {}
},
...
}
```

We'll link to this port to WordPress so that it's available for WordPress internally:

```
k@laptop:~$ docker run --link=my_mysql:mysql -p 8888:80 -d wordpress
```

Here, we used **--link** to add link to WordPress container in the form of **container-name:alias**, specified the ports: **hostPort:containerPort**, and run it in detached mode as before.

Now, we have two containers running:

```
k@laptop:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
9b149061e09b	wordpress	"/entrypoint.sh apac	6 seconds ago	Up 5 seconds
0.0.0.0:8888->80/tcp	jovial_noyce			

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
9173d3457bc8	mysql	"/entrypoint.sh mysq	30 minutes ago	Up 30 minutes
3306/tcp	my_mysql			

Actually, we have three containers: two are running, and another holds the volume for datastore:

```
k@laptop:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			

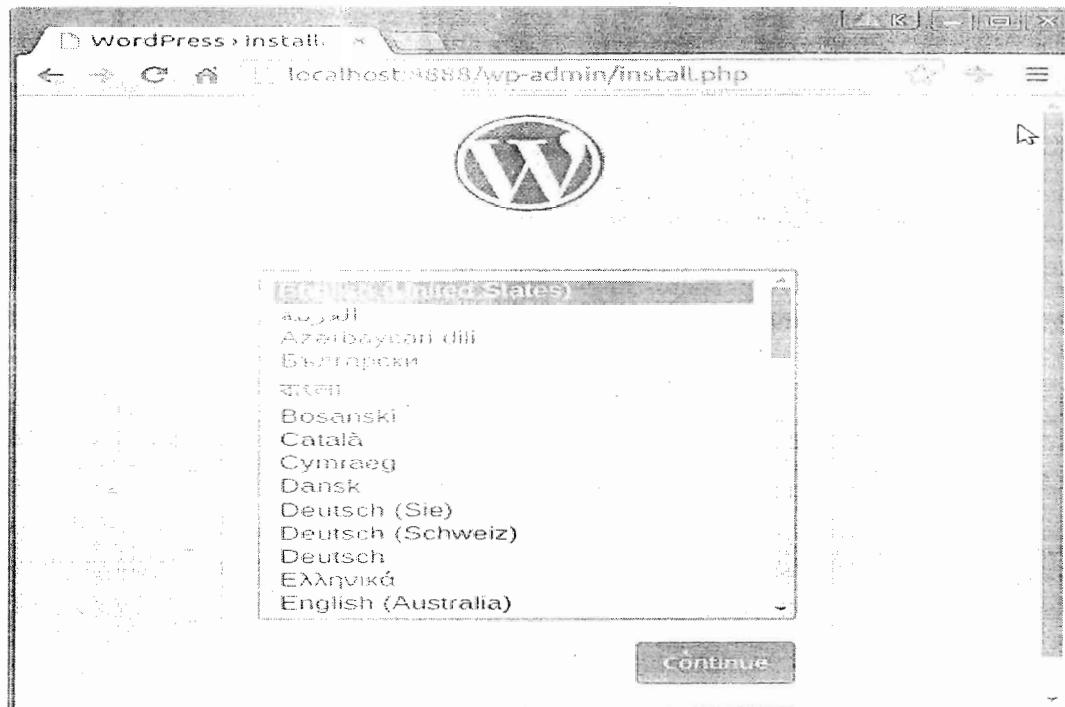
```
9b149061e09b    wordpress      "/entrypoint.sh apac  2 minutes ago   Up 2 minutes
0.0.0.0:8888->80/tcpjovial_noyce

9173d3457bc8    mysql        "/entrypoint.sh mysq  32 minutes ago   Up 32 minutes
3306/tcpmy_mysql

6a7e05a1e120    busybox      "echo 'My Datastore'  About an hour ago  Exited (0) About an
hour ago          my_datastore
```

Now, we're able to access the instance from the host without the container's IP, since we used standard port mappings.

Then, access it via <http://localhost:8888> or <http://host-ip:8888> in a browser:

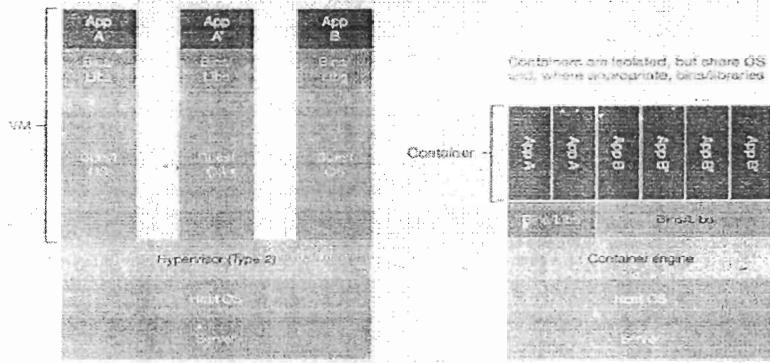


Docker container vs Virtual machine

Docker is an open source application deployment container that evolved from the LinuX Containers (LXC) used for the past decade. LXC's allow different applications to share operating system (OS) kernel, CPU, and RAM.

Docker allow us to run an **application and its dependencies in resource-isolated processes**.

Virtual machines on a Type 2 hypervisor versus application containerization with a shared OS



Credit: Open source application containers

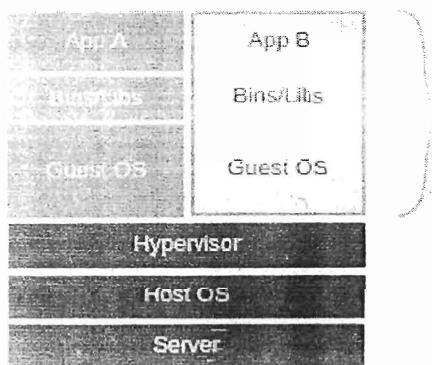
"The VM model blends **an application, a full guest OS, and disk emulation**. In contrast, the container model uses just the application's dependencies and runs them directly on a host OS. Containers do not launch a separate OS for each application, but share the host kernel while maintaining the isolation of resources and processes where required".

"The fact that a container does not run its own OS instance reduces dramatically the overhead associated with starting and running instances. Startup time can typically be reduced from 30 seconds (or more) to one-tenth of a second. The number of containers running on a typical server can reach dozens or even hundreds. The same server, in contrast, might support 10 to 15 VMs".

"In Docker, applications and their dependencies, such as binaries and libraries, all become part of a base working image".

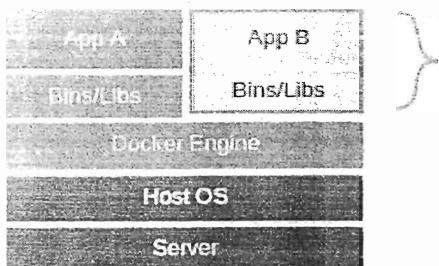
The isolation from OS kernel provided by containers is less robust than that of real virtual machines, which have independent kernels and run on top of a hypervisor. However, sharing the kernel allows containers to run faster and offers management features which are not easy with VMs.

The picture is from <https://www.docker.com/whatisdocker/> - How is this different from Virtual Machines?.



Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

Container apps

Containers make our app shareable.

DEVOPS MATERIAL

1. All the needs of our app is defined in a text file (**Dockerfile**)
2. A sample of Dockerfile - the following 4 lines construct the whole environment that's production ready:

```

3. FROM ubuntu:14.04
4. RUN apt-get install -y redis-server
5. EXPOSE 6379
6. ENTRYPOINT ["/usr/bin/redis-server"]

```

7. Containers contain everything our app needs:

1. Binaries
2. Libraries
3. File system

8. Containers use the following items from the host:

1. Networks
2. Kernel

Type 1 / Type2 Hypervisor

Type 1 hypervisor

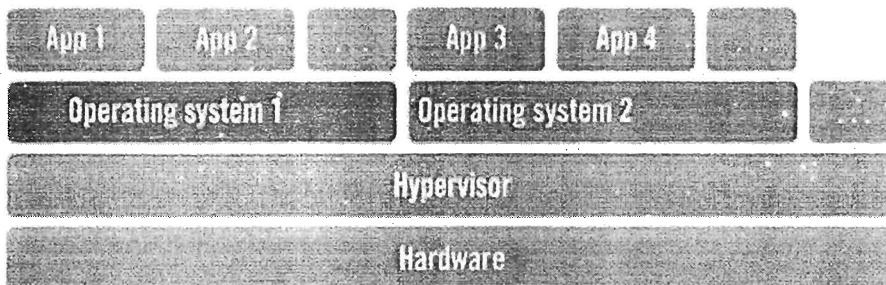


Figure 2. A Type 1 or bare-metal hypervisor sits directly on the host hardware.

Type-1 bare-metal hypervisors

DEVOPS MATERIAL

1. Xen
2. VMware ESX/ESXi
3. Microsoft Hyper-V

Type 2 hypervisor

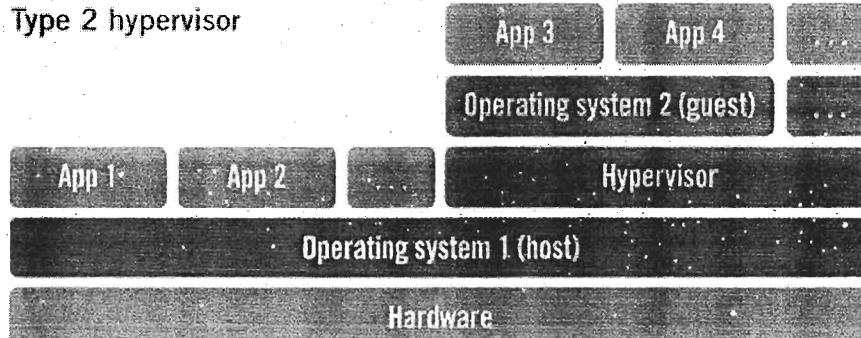
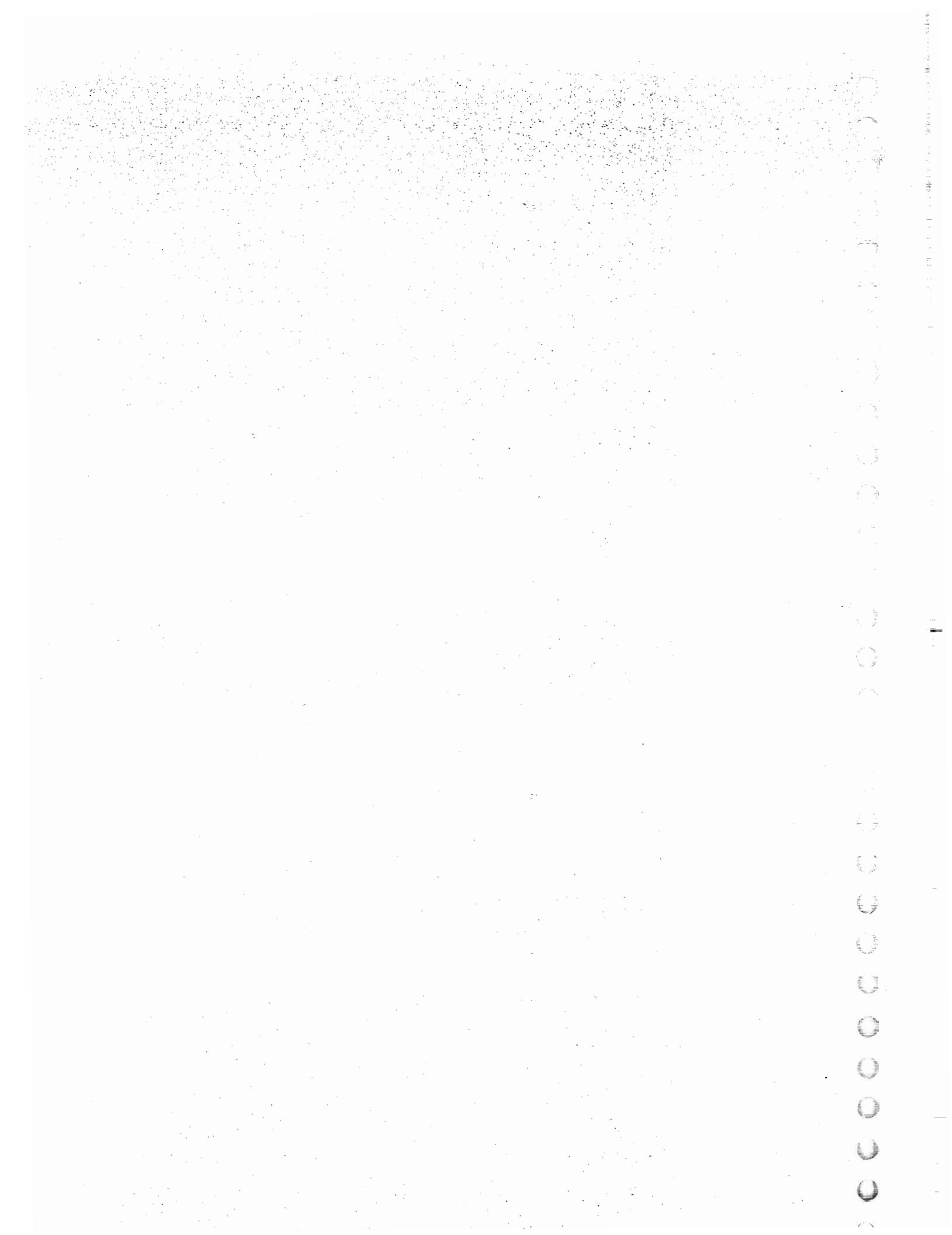


Figure 1. A Type 2 hypervisor runs as an application on a host operating system.

Type-2 hosted hypervisors

1. VirtualBox
2. VMware Workstation/Player



DEVOPS Interview Questions

Q. What is GIT?

GIT is a distributed version control system and source code management (SCM) system with an emphasis to handle small and large projects with speed and efficiency.

Q. What is a repository in GIT?

A repository contains a directory named .git, where git keeps all of its metadata for the repository. The content of the .git directory are private to git.

Q. What is the command you can use to write a commit message?

The command that is used to write a commit message is “git commit –a”. The –a on the command line instructs git to commit the new content of all tracked files that have been modified. You can use “git add<file>” before git commit –a if new files need to be committed for the first time.

Q. What is the difference between GIT and SVN?

The difference between GIT and SVN is

Git is less preferred for handling extremely large files or frequently changing binary files while SVN can handle multiple projects stored in the same repository.

GIT does not support ‘commits’ across multiple branches or tags. Subversion allows the creation of folders at any location in the repository layout.

Gits are unchangeable, while Subversion allows committers to treat a tag as a branch and to create multiple revisions under a tag root.

Q. What are the advantages of using GIT?

Data redundancy and replication

High availability

Only one.git directory per repository

Superior disk utilization and network performance

Collaboration friendly

Any sort of projects can use GIT

Q. What language is used in GIT?

GIT is fast, and ‘C’ language makes this possible by reducing the overhead of runtimes associated with higher languages.

Q. What is the function of ‘GIT PUSH’ in GIT?

‘GIT PUSH’ updates remote refs along with associated objects.

Q. Why GIT better than Subversion?

GIT is an open source version control system; it will allow you to run ‘versions’ of a project, which show the changes that were made to the code overtime also it allows you keep the backtrack if necessary and undo those changes. Multiple developers can checkout, and upload changes and each change can then be attributed to a specific developer.

Q. What is “Staging Area” or “Index” in GIT?

DEVOPS Interview Questions

Before completing the commits, it can be formatted and reviewed in an intermediate area known as ‘Staging Area’ or ‘Index’.

Q. What is GIT stash?

GIT stash takes the current state of the working directory and index and puts in on the stack for later and gives you back a clean working directory. So in case if you are in the middle of something and need to jump over to the other job, and at the same time you don’t want to lose your current edits then you can use GIT stash.

Q. What is GIT stash drop?

When you are done with the stashed item or want to remove it from the list, run the git ‘stash drop’ command. It will remove the last added stash item by default, and it can also remove a specific item if you include as an argument.

Q. How will you know in GIT if a branch has been already merged into master?

Git branch—merged lists the branches that have been merged into the current branch

Git branch—no merged lists the branches that have not been merged

Q. Is the function of git clone?

The git clone command creates a copy of an existing Git repository. To get the copy of a central repository, ‘cloning’ is the most common way used by programmers.

Q. What is the function of ‘git config’?

The ‘git config’ command is a convenient way to set configuration options for your Git installation. Behaviour of a repository, user info, preferences etc. can be defined through this command.

Q. What does commit object contain?

A set of files, representing the state of a project at a given point of time

Reference to parent commit objects

An SHA1 name, a 40 character string that uniquely identifies the commit object.

Q. How can you create a repository in Git?

In Git, to create a repository, create a directory for the project if it does not exist, and then run command “git init”. By running this command .git directory will be created in the project directory, the directory does not need to be empty.

Q. What is ‘head’ in git and how many heads can be created in a repository?

A ‘head’ is simply a reference to a commit object. In every repository, there is a default head referred as “Master”. A repository can contain any number of heads.

Q. What is the purpose of branching in GIT?

DEVOPS Interview Questions

The purpose of branching in GIT is that you can create your own branch and jump between those branches. It will allow you to go to your previous work keeping your recent work intact.

Q. What is the common branching pattern in GIT?

The common way of creating branch in GIT is to maintain one as “Main”

branch and create another branch to implement new features. This pattern is particularly useful when there are multiple developers working on a single project.

Q. How can you bring a new feature in the main branch?

To bring a new feature in the main branch, you can use a command “git merge” or “git pull command”.

Q. What is a ‘conflict’ in git?

A ‘conflict’ arises when the commit that has to be merged has some change in one place, and the current commit also has a change at the same place. Git will not be able to predict which change should take precedence.

Q. How can conflict in git resolved?

To resolve the conflict in git; edit the files to fix the conflicting changes and then add the resolved files by running “git add” after that to commit the repaired merge, run “git commit”. Git remembers that you are in the middle of a merger, so it sets the parents of the commit correctly.

Q. To delete a branch what is the command that is used?

Once your development branch is merged into the main branch, you don’t need development branch. To delete a branch use, the command “git branch -d [head]”.

Q. What is another option for merging in git?

“Rebasing” is an alternative to merging in git.

Q. What is the syntax for “Rebasing” in Git?

The syntax used for rebase is “git rebase [new-commit] “

Q. What is the difference between ‘git remote’ and ‘git clone’?

‘git remote add’ just creates an entry in your git config that specifies a name for a particular URL. While, ‘git clone’ creates a new git repository by copying an existing one located at the URI.

Q. What is GIT version control?

With the help of GIT version control, you can track the history of a collection of files and includes the functionality to revert the collection of files to another version. Each version captures a snapshot of

DEVOPS Interview Questions

the file system at a certain point of time. A collection of files and their complete history are stored in a repository.

Q. Mention some of the best graphical GIT client for LINUX?

Some of the best GIT client for LINUX is

- Git Cola
- Git-g
- Smart git
- Giggle
- Git GUI
- qGit

Q. What is Subgit? Why to use Subgit?

'Subgit' is a tool for a smooth, stress-free SVN to Git migration. Subgit is a solution for a company - wide migration from SVN to Git that is:

- a) It is much better than git-svn
- b) No requirement to change the infrastructure that is already placed
- c) Allows to use all git and all sub-version features
- d) Provides genuine stress –free migration experience.

Q. What is the function of 'git diff' in git?

'git diff' shows the changes between commits, commit and working tree etc.

Q. What is 'git status' is used for?

As 'Git Status' shows you the difference between the working directory and the index, it is helpful in understanding a git more comprehensively.

Q. What is the difference between the 'git diff' and 'git status'?

'git diff' is similar to 'git status', but it shows the differences between various commits and also between the working directory and index.

Q. What is the function of 'git checkout' in git?

A 'git checkout' command is used to update directories or specific files in your working tree with those from another branch without merging it in the whole branch.

DEVOPS Interview Questions

Q. What is the function of ‘git rm’?

To remove the file from the staging area and also off your disk ‘git rm’ is used.

Q. What is the function of ‘git stash apply’?

When you want to continue working where you have left your work, ‘git stash apply’ command is used to bring back the saved changes onto the working directory.

Q. What is the use of ‘git log’?

To find specific commits in your project history- by author, date, content or history ‘git log’ is used.

Q. What is ‘git add’ is used for?

‘git add’ adds file changes in your existing directory to your index.

Q. What is the function of ‘git reset’?

The function of ‘Git Reset’ is to reset your index as well as the working directory to the state of your last commit.

Q. What is git ls-tree?

‘git ls-tree’ represents a tree object including the mode and the name of each item and the SHA-1 value of the blob or the tree.

Q. How git instaweb is used?

‘Git Instaweb’ automatically directs a web browser and runs webserver with an interface into your local repository.

Q. What does ‘hooks’ consist of in git?

This directory consists of Shell scripts which are activated after running the corresponding Git commands. For example, git will try to execute the post-commit script after you run a commit.

Q. Explain what is commit message?

Commit message is a feature of git which appears when you commit a change. Git provides you a text editor where you can enter the modifications made in commits.

Q. How can you fix a broken commit?

To fix any broken commit, you will use the command “git commit—amend”. By running this command, you can fix the broken commit message in the editor.

DEVOPS Interview Questions

Q. Why is it advisable to create an additional commit rather than amending an existing commit?

There are couple of reason

The amend operation will destroy the state that was previously saved in a commit. If it's just the commit message being changed then that's not an issue. But if the contents are being amended then chances of eliminating something important remains more.

Abusing "git commit- amend" can cause a small commit to grow and acquire unrelated changes.

Q. What is 'bare repository' in GIT?

To co-ordinate with the distributed development and developers team, especially when you are working on a project from multiple computers 'Bare Repository' is used. A bare repository comprises of a version history of your code.

Q. How do you revert a commit that has already been pushed and made public?

One or more commits can be reverted through the use of git revert. This command, in essence, creates a new commit with patches that cancel out the changes introduced in specific commits. In case the commit that needs to be reverted has already been published or changing the repository history is not an option, git revert can be used to revert commits. Running the following command will revert the last two commits:

```
git revert HEAD~2..HEAD
```

Alternatively, one can always checkout the state of a particular commit from the past, and commit it anew.

Q. How do you squash last N commits into a single commit?

Squashing multiple commits into a single commit will overwrite history, and should be done with caution. However, this is useful when working in feature branches. To squash the last N commits of the current branch, run the following command (with {N} replaced with the number of commits that you want to squash):

```
git rebase -i HEAD~{N}
```

Upon running this command, an editor will open with a list of these N commit messages, one per line. Each of these lines will begin with the word "pick". Replacing "pick" with "squash" or "s" will tell Git to combine the commit with the commit before it. To combine all N commits into one, set every commit in the list to be squash except the first one. Upon exiting the editor, and if no conflict arises, git rebase will allow you to create a new commit message for the new combined commit.

Q. How do you find a list of files that has changed in a particular commit?

```
git diff-tree -r {hash}
```

Given the commit hash, this will list all the files that were changed or added in that commit. The `-r` flag makes the command list individual files, rather than collapsing them into root directory names only.

The output will also include some extra information, which can be easily suppressed by including a couple of flags:

```
git diff-tree --no-commit-id --name-only -r {hash}
```

Here `--no-commit-id` will suppress the commit hashes from appearing in the output, and `--name-only` will only print the file names, instead of their paths.

Q: How do you setup a script to run every time a repository receives new commits through push?

To configure a script to run every time a repository receives new commits through push, one needs to define either a pre-receive, update, or a post-receive hook depending on when exactly the script needs to be triggered.

Pre-receive hook in the destination repository is invoked when commits are pushed to it. Any script bound to this hook will be executed before any references are updated. This is a useful hook to run scripts that help enforce development policies.

Update hook works in a similar manner to pre-receive hook, and is also triggered before any updates are actually made. However, the update hook is called once for every commit that has been pushed to the destination repository.

Finally, post-receive hook in the repository is invoked after the updates have been accepted into the destination repository. This is an ideal place to configure simple deployment scripts, invoke services, continuous integration systems, dispatch notification emails to repository maintainers, etc.

Hooks are local to every Git repository and are not versioned. Scripts can either be created within the hooks directory inside the ".git" directory, or they can be created elsewhere and links to those scripts can be placed within the directory.

Q: What is git bisect? How can you use it to determine the source of a (regression) bug?

Git provides a rather efficient mechanism to find bad commits. Instead of making the user try out every single commit to find out the first one that introduced some particular issue into the code, git bisect allows the user to perform a sort of binary search on the entire history of a repository.

By issuing the command `git bisect start`, the repository enters bisect mode. After this, all you have to do is identify a bad and a good commit:

```
git bisect bad # marks the current version as bad
```

`git bisect good {hash or tag} # marks the given hash or tag as good, ideally of some earlier commit`

Once this is done, Git will then have a range of commits that it needs to explore. At every step, it will checkout a certain commit from this range, and require you to identify it as good or bad. After which the range will be effectively halved, and the whole search will require a lot less number of steps than the actual number of commits involved in the range. Once the first bad commit has been found, or the bisect mode needs to be ended, the following command can be used to exit the mode and reset the bisection state:

`git bisect reset`

Q. What are the different ways you can refer to a commit?

In Git each commit is given a unique hash. These hashes can be used to identify the corresponding commits in various scenarios (such as while trying to checkout a particular state of the code using the `git checkout {hash}` command).

Additionally, Git also maintains a number of aliases to certain commits, known as refs. Also, every tag that you create in the repository effectively becomes a ref (and that is exactly why you can use tags instead of commit hashes in various git commands). Git also maintains a number of special aliases that change based on the state of the repository, such as HEAD, FETCH_HEAD, MERGE_HEAD, etc.

Git also allows commits to be referred as relative to one another. For example, HEAD~1 refers to the commit parent to HEAD, HEAD~2 refers to the grandparent of HEAD, and so on. In case of merge commits, where the commit has two parents, ^ can be used to select one of the two parents, e.g. HEAD^2 can be used to follow the second parent.

And finally, refspecs. These are used to map local and remote branches together. However, these can be used to refer to commits that reside on remote branches allowing one to control and manipulate them from a local Git environment.

Q. What is git rebase and how can it be used to resolve conflicts in a feature branch before merge?

In simple words, git rebase allows one to move the first commit of a branch to a new starting location. For example, if a feature branch was created from master, and since then the master branch has received new commits, git rebase can be used to move the feature branch to the tip of master. The command effectively will replay the changes made in the feature branch at the tip of master, allowing conflicts to be resolved in the process. When done with care, this will allow the feature branch to be merged into master with relative ease and sometimes as a simple fast-forward operation.

Q. How do you configure a Git repository to run code sanity checking tools right before making commits, and preventing them if the test fails?

This can be done with a simple script bound to the pre-commit hook of the repository. The pre-commit hook is triggered right before a commit is made, even before you are required to enter a commit message. In this script one can run other tools, such as linters and perform sanity checks on the changes being committed into the repository. For example, the following script:

DEVOPS Interview Questions

```
#!/bin/sh

files=$(git diff --cached --name-only --diff-filter=ACM | grep '.go$')

if [ -z files ]; then
    exit 0
fi

unfmtd=$(gofmt -l $files)

if [ -z unfmtd ]; then
    exit 0
fi

echo "Some .go files are not fmt'd"

exit 1
```

... checks to see if any .go file that is about to be committed needs to be passed through the standard Go source code formatting tool gofmt. By exiting with a non-zero status, the script effectively prevents the commit from being applied to the repository.

