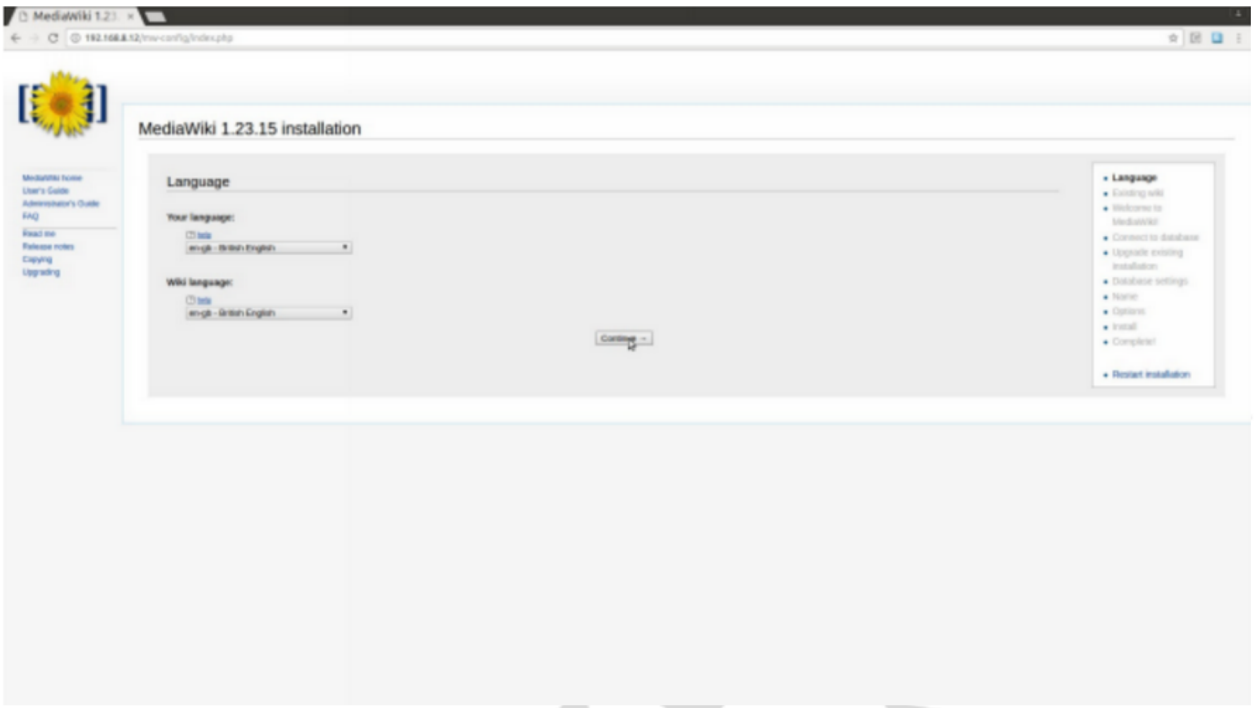
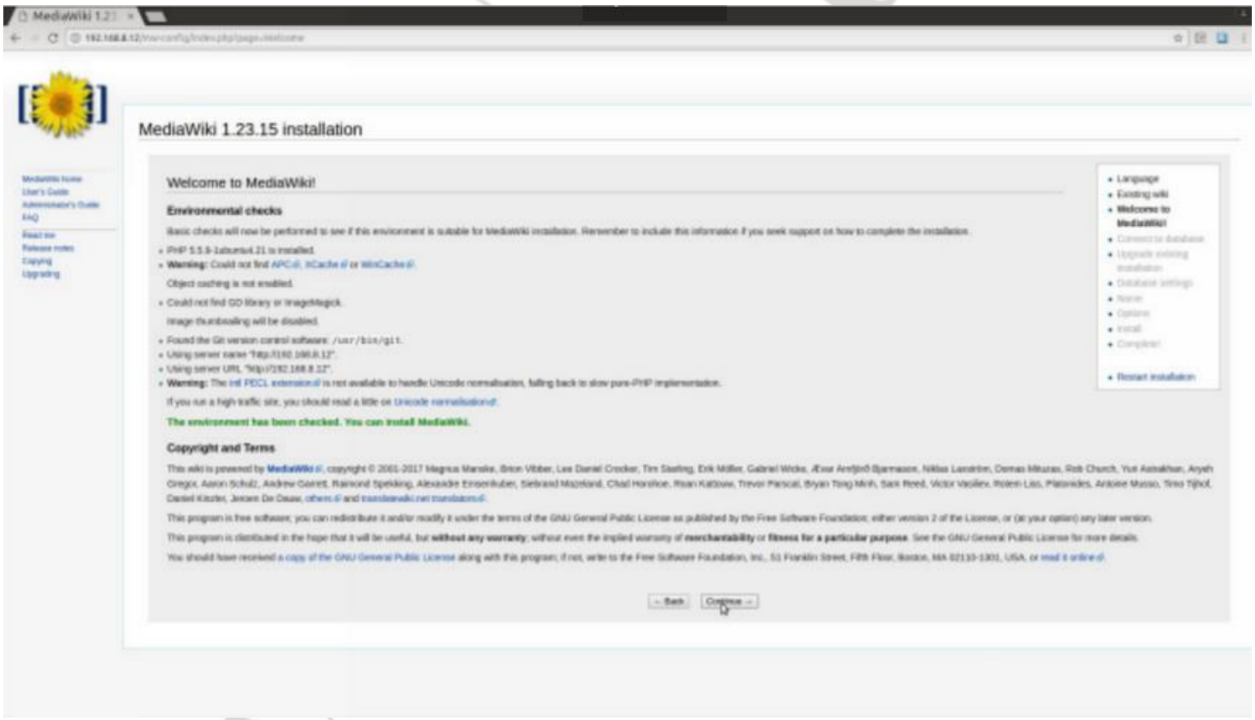
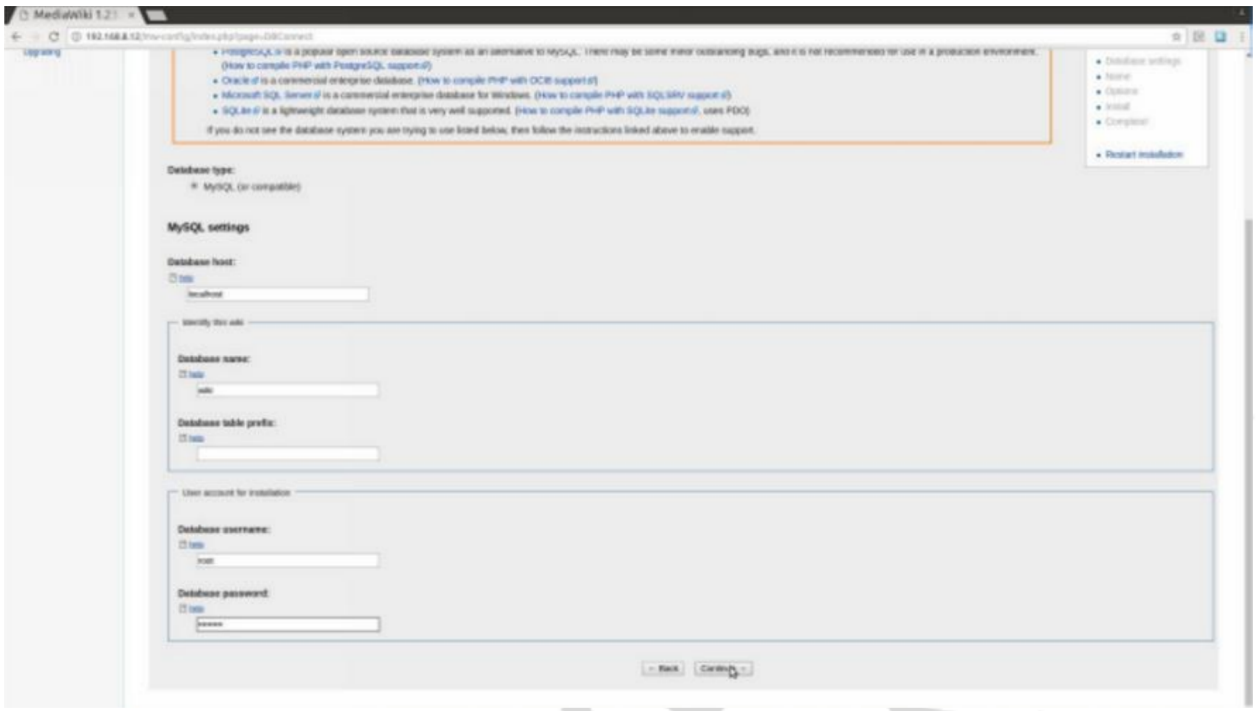
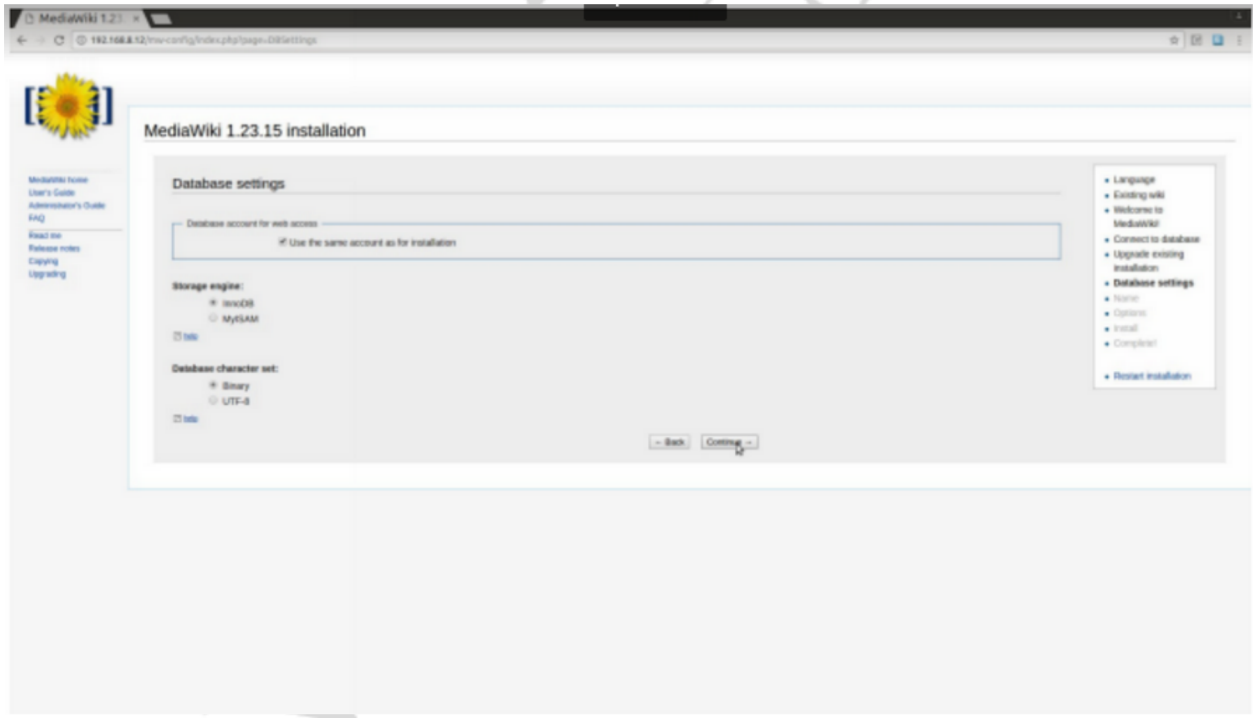


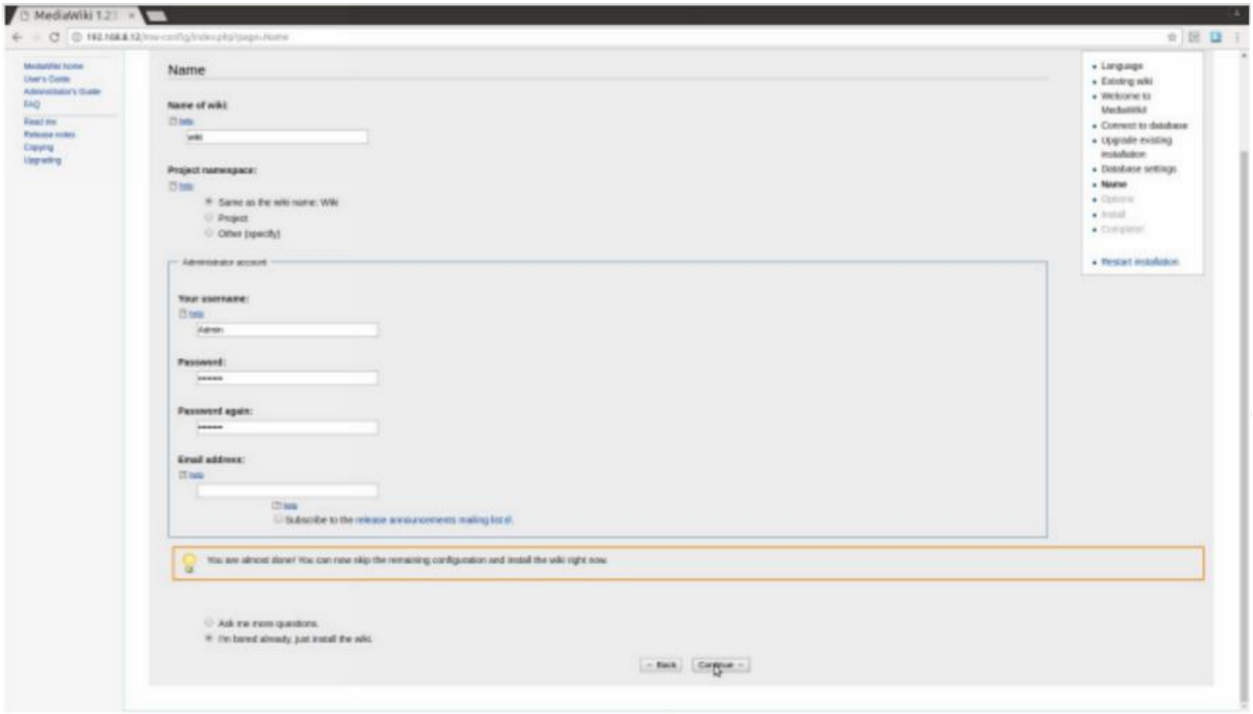
Enter database password that we have set in our mysql module (‘training’).
- Set Mediawiki website user & password, use password as “training”.





-After the setup, it will download a file named ‘LocalSettings.php’.
LocalSettings.php file has to be uploaded back to mediawiki site in /var/www/html/LocalSettings.php location. We can do that manually but we will use template to push this file through our puppet code.



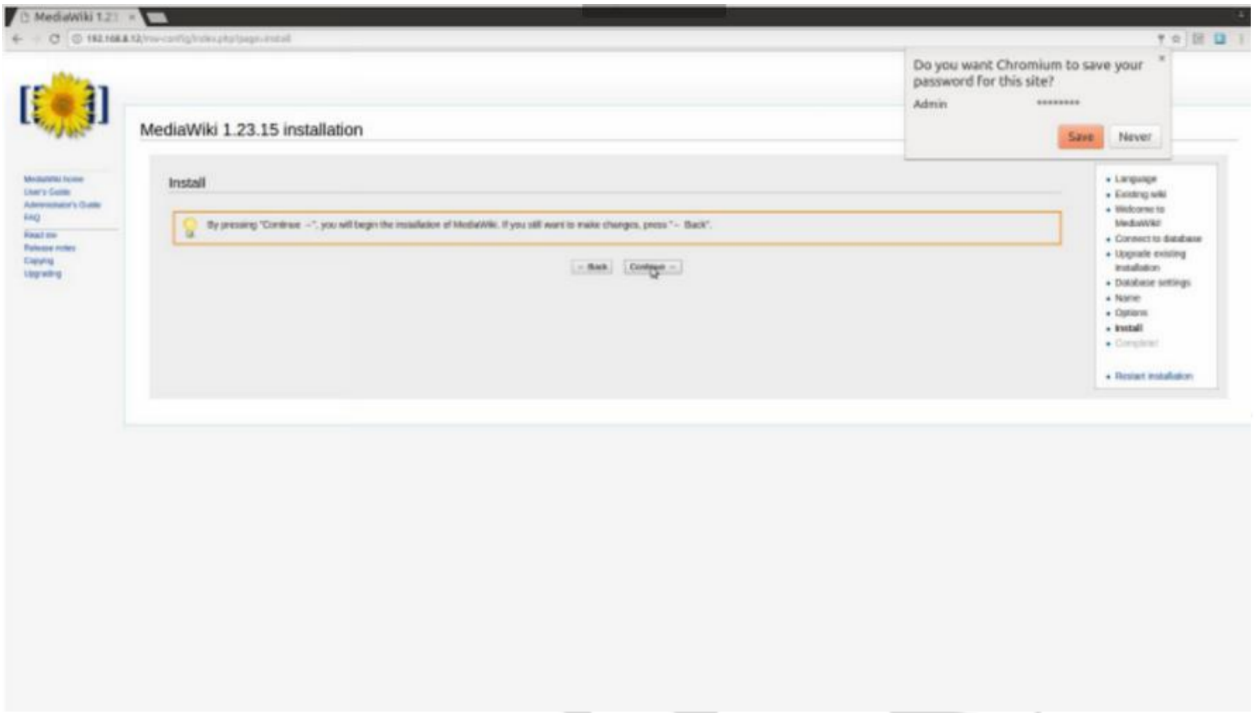


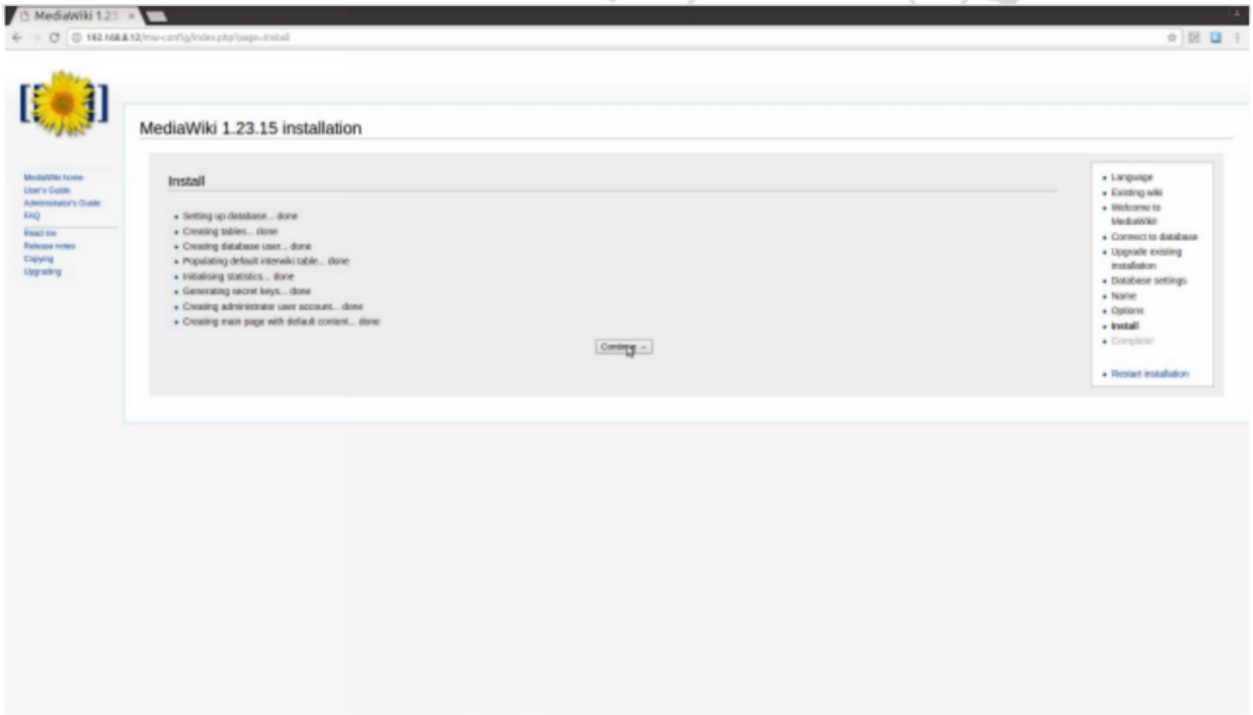
6.Templates.

6.1 Setting Variables in Template

- Open localsettings.php file.
- Replace php variables values with puppet variables.

```
$wgSitename = "<%= wikisitename%>";
$wgMetaNamespace = "<%= wikimetanamespace%>";
$wgServer = "<%= wikiserver%>";
$wgDBserver = "<%= wikidbserver%>";
$wgDBname = "<%= wikidbname%>";
$wgDBuser = "<%= wikidbuser%>";
$wgDBpassword = "<%= wikidbpassword%>";
$wgUpgradeKey = "<%= wikiugradekey%>";
```

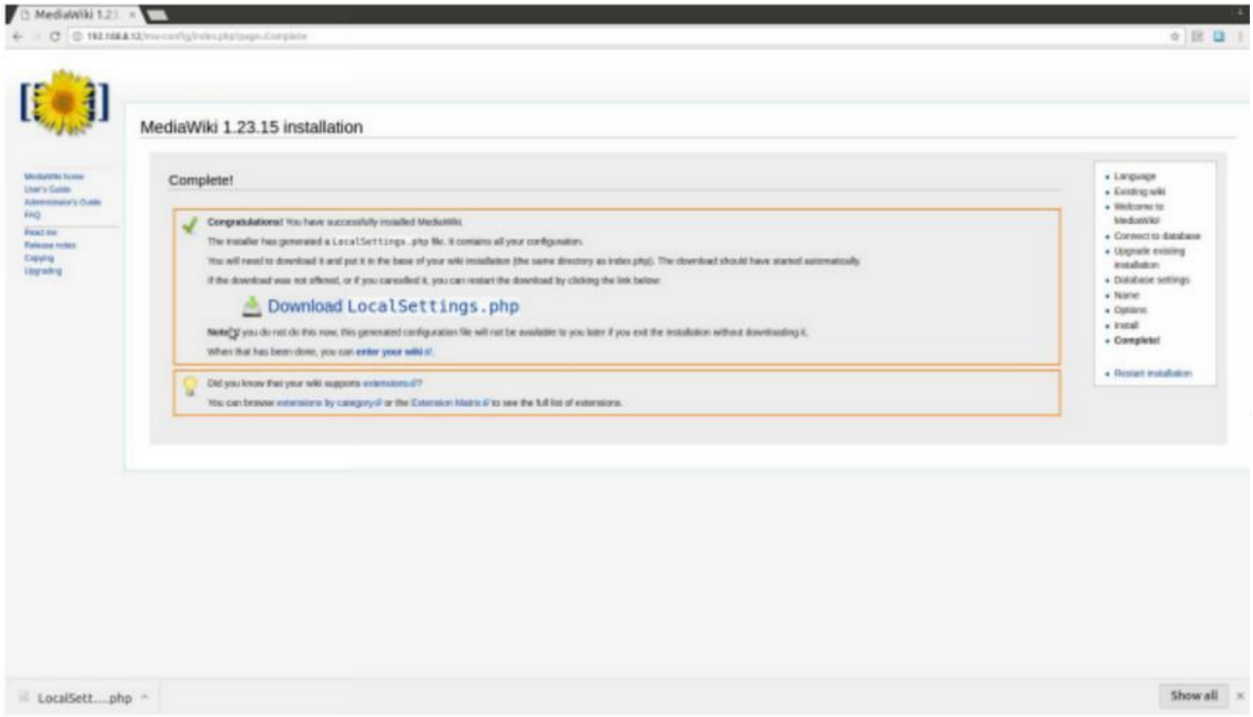




-
Login to puppetmaster & create template file.
- Create template directory.
mkdir /etc/puppet/environments/production/modules/mediawiki/templates.
- Create LocalSettings.erb template file and paste content of LocalSettings.php we just edited.
vi /etc/puppet/environments/production/modules/mediawiki/templates/LocalSettings.erb

6.2 Declaring template.

- Add template definition in Mediawiki Module.



vi /etc/puppet/environments/production/modules/mediawiki/manifests/init.pp

```
class { 'firewall': }
firewall { '000 allow http access':
port => '80',
Proto => 'tcp',
action => 'accept',
}
file {'LocalSettings.php':
path => '/var/www/html/LocalSettings.php',
ensure => 'file',
content => template('mediawiki/LocalSettings.erb'),
}
```

- Define variables used in template file.

vi /etc/puppet/environments/production/manifests/nodes.pp

- Assign variable values for wiki.

```
node 'wiki'{
$wikisitename = 'wiki'
$wikimetanamespace = 'Wiki'
$wikiserver = "http://192.168.8.12"
$wikidbserver = 'localhost'
$wikidbname = 'wiki'
$wikidbuser = 'root'
$wikidbpassword = 'training'
$wikiugradekey = 'puppet'
class {'linux':}
class {'mediawiki':}
}
```

- Pull the latest change from wiki.

puppet agent --verbose –onetime --no-daemonize
- Go and check the mediawiki website from browser.

7. Hiera

Hiera facilitates to place all the node specific variables in a file other than manifests files. This makes easier to work on the variables and the manifest files can be used for other projects without any modification in them.

Here we are removing all the variables from the manifests file and placing those node specific variables in differently located files.

vim /etc/puppet/environments/production/modules/mediawiki/manifests/init.pp

```
class mediawiki {
$wikimetanamespace = hiera('mediawiki::wikimetanamespace')
$wikisitename = hiera('mediawiki::wikisitename')
$wikiserver = hiera('mediawiki::wikiserver')
$wikidbserver = hiera('mediawiki::wikidbserver')
$wikidbname = hiera('mediawiki::wikidbname')
$wikidbuser = hiera('mediawiki::wikidbuser')
$wikidbpassword = hiera('mediawiki::wikidbpassword')
$wikiupgradekey = hiera('mediawiki::wikiupgradekey')

$phpmysql = $osfamily ? {
'redhat' => 'php-mysql',
'debian' => 'php5-mysql',
default => 'php-mysql',
      }
}
```

- Remove variables from nodes.pp file.
- Create Hiera configuration file.

vi /etc/puppet/hiera.yaml

```
:backends:
- yaml
:yaml:
:datadir:
:hierarchy:
- "%{clientcert}"
- wikidefault
```

- Add variables for wiki node in /var/lib/hiera/wiki.yaml

vi /var/lib/hiera/wiki.yaml

```
---
mediawiki::wikisitename: wiki
mediawiki::wikisitenamespace: Wiki
mediawiki::wikiserver: http://192.168.1.11
mediawiki::wikidbname: wiki
```

- Add variables for wikitest node in /var/lib/hiera/wikitest.yaml
---
mediawiki::wikisitename: wikitest
mediawiki::wikisitenamespace: Wikitest
mediawiki::wikiserver: http://192.168.1.12
mediawiki::wikidbname: wikitest

- Add common variables in /var/lib/hiera/wikidefault.yaml
---
mediawiki::wikidbserver: localhost
mediawiki::wikidbuser: root
mediawiki::wikidbpassword: training
mediawiki::wikiupgradekey: puppet

- Execute puppet agent to test it.

Summary:

- ✓ Puppet is an open source Configuration Management tool that has majorly adopted by the huge organisations in implementing DevOps architecture for automation.
- ✓ We can easily manage and automate infrastructure through puppet master-client model, a better architecture compared to using scripts.
- ✓ Puppet has already proven its capability in DevOps life cycles and has been continuing its saga in automation technologies.
- ✓ Puppet follows basic Ruby Language. But we can build puppet code in manifests in simple syntaxes on the puppet master.
- ✓ Puppet provisions us to define multiple nodes in the main manifest files through the use of resources. Resources are small pieces of software and can bring desired states in nodes.
- ✓ Puppet agents pull their catalog from puppet master to make configuration changes. Catalog is the collection of resources defined to that specific node written in main manifests files.
- ✓ For making puppet code more readable, it can be written in manifests files in modules directory (other than main manifests files that is under /etc/puppet). These manifests files can be invoked from the main manifests file.
- ✓ The functions include, class can invoke the puppet code in modules manifests
- ✓ There is a provision in Puppet to create environments. We can make puppet code and can make it better when dealing with different groups of nodes.
- ✓ Puppet with its code features like variables, conditionals (if statements, selectors, cases) makes it much closer to automation process. These features make puppet to act dynamically based on nodes status.
- ✓ One more feature of Puppet Hiera - a key/value lookup tool for configuration data, built to make Puppet better and let you set node-specific data without repeating the values.

XIV. Dockers

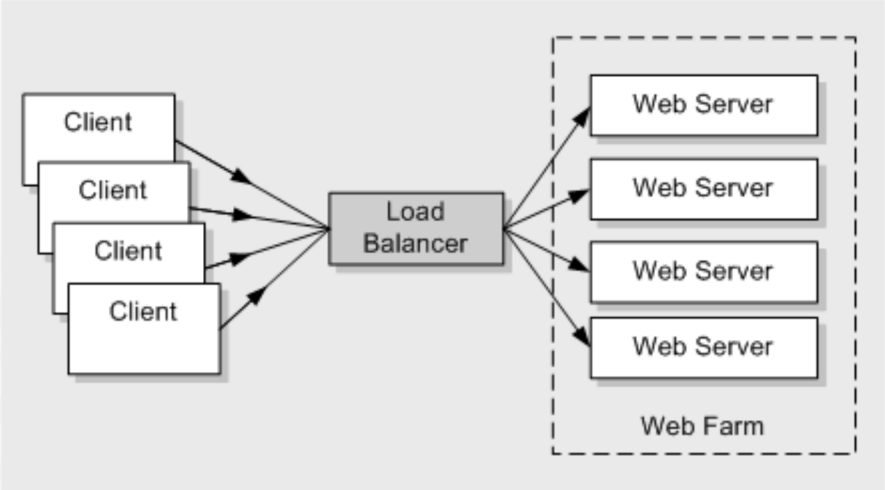
1. Applications Era

In today's world, we are all surrounded by apps and websites. We use our smartphones and computers to browse around internet and use all the web services through our mobile apps or browsers. All these millions of web based data is coming somewhere far from some computers which would be located in some datacenter. We generally call them servers; these servers could be those physical machines that we see racked up in a datacenter with all those flashing lights and cables.

If we take some examples like Amazon, Google, Netflix, Goibibo etc, all these businesses are running on applications or we can say their applications are their business. This makes a very important point that we cannot separate their business with their application.

Application needs compute resource to run and that comes from the server where they hosted their application. In olden days when we did not have any virtualization or cloud computing we use to run them directly on a physical server.

So, if I want to host an application on 10 webservers, I need ten physical servers under load balancer serving the web traffic.

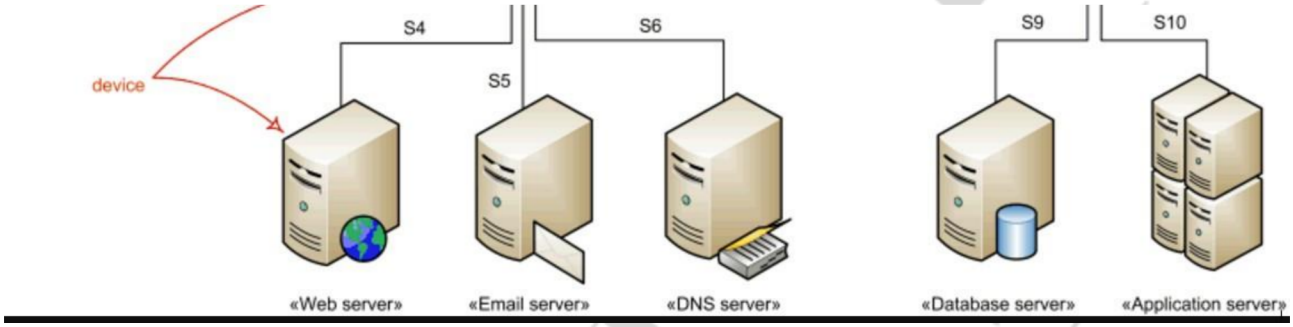


These servers are very expensive and we need to do lot of maintenance for them.

- We need to procure a server. A process where we place an order for the purchase.
- There is Capital expenditure or CapEx required.
- There is Operational expenditure (OpEx), like cooling, power, admins to maintain that server farm.

So, if I want to increase the capacity and add more servers I need to spend money and time on above mentioned process. This is very common as business starts from very small user base and then users/consumers traffic increases if business is doing well.

We deploy one application per server because we want our applications to be isolated. For example, if we need web app, db app and few backend apps.
We may end up having multiple physical system each running a single instance of that app.



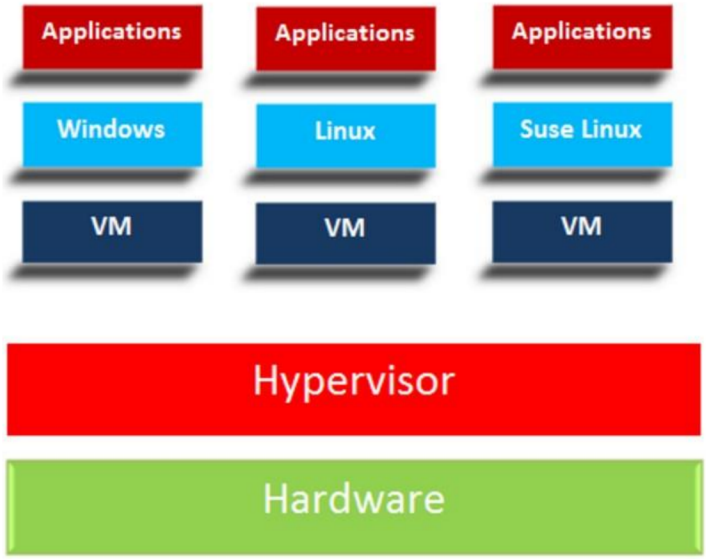
So, every time we need a new app to run we buy servers, install OS and setup our app on that.
And most of the time nobody knew the performance requirements of the new application!
This meant IT had to make guesses when choosing the model and size of servers to buy.

As a result, IT did the only reasonable thing - it bought big fast servers with lots of resiliency. After all, the last thing anyone wanted - including the business - was under-powered servers.

Most part of the time these physical server compute resource will be under-utilized as low as 5-10% of their potential capacity. A tragic waste of company capital and resources.

2. Virtualization Revolution.

VMware gave the world the virtual machine and everything changed after that. Now we could run multiple applications isolated in separate OS but in the same physical server.
In the virtualization chapter, we discussed the benefits and features of virtualization, The Hypervisor Architecture.



3. Problems with Hypervisor Architecture.

Now we know that every VM has its own OS, which is a problem. OS needs fair amount of resources like CPU, Memory, Storage etc. We also maintain OS licences and nurse them regularly like patching, upgrades, config changes. We wanted to host an application but collected good amount of fats over our infra, we are wasting OpEx and CapEx here. Think about shipping a vm from one place to other place, this sounds a great idea that if we could bundle everything in a vm image and ship it so the other person doesn't need to setup vm from scratch can directly run the vm from image. We did it in Vagrant chapter where we download preinstalled vm and have just run it.

But these images are heavy and bulky as they contain OS with the app. Booting them is a slow process. So being portable it's not convenient to ship the vm every time.

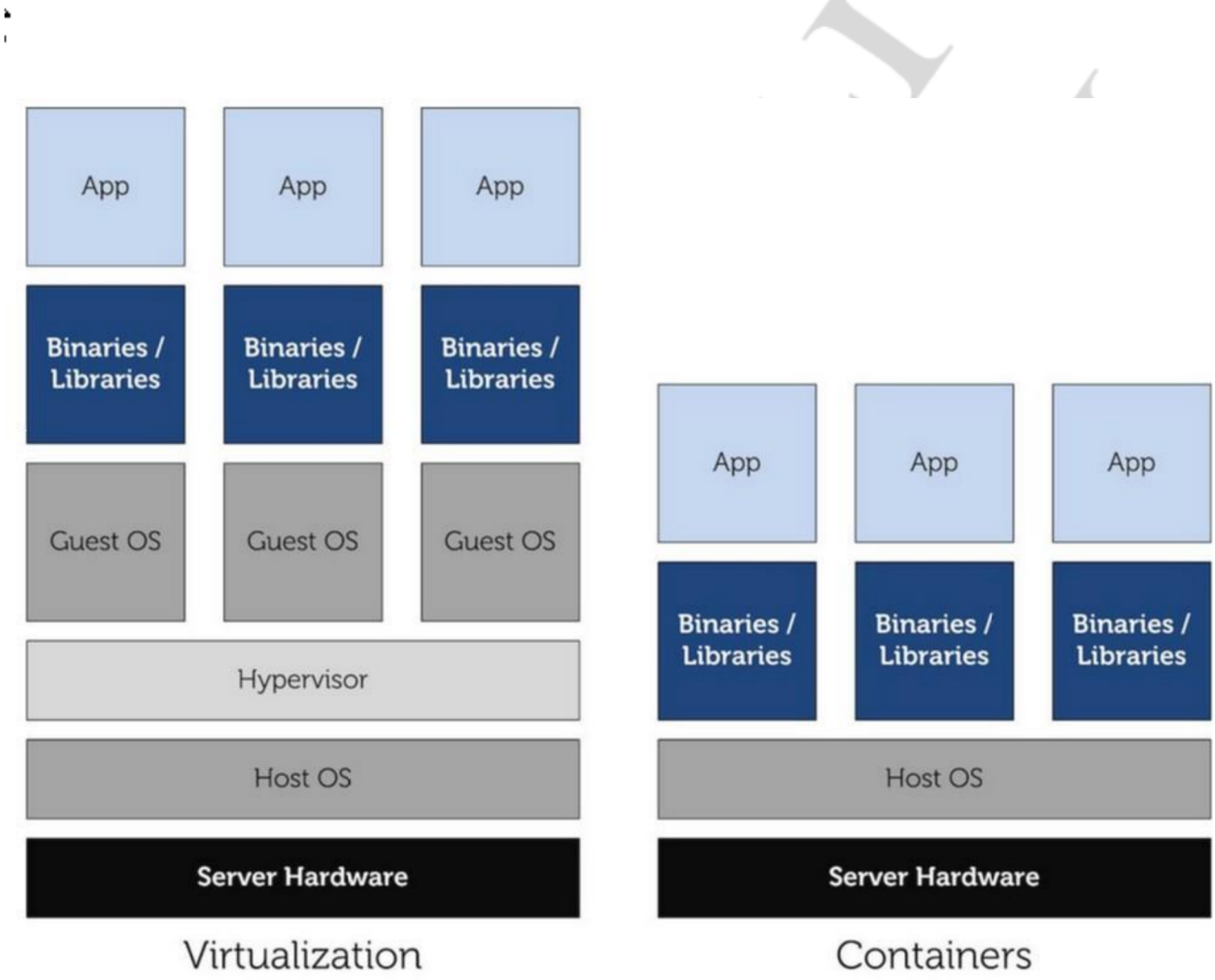
Shipping an application bundled with all the dependencies/libraries in an image **without OS.** Hmm, sounds like we solved a big problem there. That's what containers are.

Think about setting up an application in a vm or physical machine. We need OS setup, dependencies, application deployed and some config changes in the OS. We follow a series of steps to setup all these like setting up a LAMP stack. If we could bundle all these into one container and ship it, then admins don't need to do any setup on the target, all we need to do is pull a container image and run it.

4. Containers.

If virtual machines are hardware virtualization then containers are OS virtualization. We don't need a real OS in the container to install our application. Applications inside the containers are dependent on Host OS kernel where its running. So, if I have hosted java application like inside the container it will use all the java libraries and config files from container data, but for compute resource its relied

on the Host OS kernel. Containers are like other processes that run in an Operating System but its isolated, its processes, files, libraries, configurations are contained within the boundaries of the container. Containers have their own process tree and networking also. Every container will have an IP address and port on which the application inside container is running. This may sound like a virtual machine but it's not, remember VM has its own OS and containers does not.



Containers are very lightweight as it just has the libraries and application. So that means less compute resource is utilized and that means more free space to run more container's. So, in terms of resources also we are saving CapEx & OpEx.

Containers is not a modern technology, it was around us in different forms and technologies. But Docker has brought it to a whole new level when it comes to building, shipping and managing containers.

5. Dockers

Docker, Inc. started its life as a platform as a service (PaaS) provider called dotCloud. Behind the scenes, the dotCloud platform leveraged Linux contain-ers. To help them create and manage these

containers they built an internal tool that they nick-named “Docker”. And that’s how Docker was born!

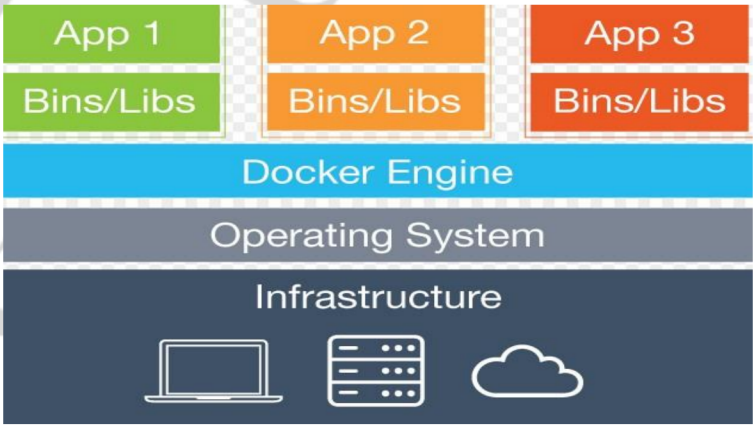
In 2013 the dotCloud PaaS business was struggling and the company needed a new lease of life. To help with this they hired Ben Golub as new CEO, rebranded the company as “Docker, Inc.”, got rid of the dotCloud PaaS platform, and started a new journey with a mission to bring to Docker and containers to the world.

Docker relies on Linux kernel features, such as namespaces and cgroups, to ensure resource isolation and to package an application along with its dependencies. This packaging of the dependencies enables an application to run as expected across different Linux operating systems. It’s this portability that’s piqued the interest of developers and systems administrators alike.

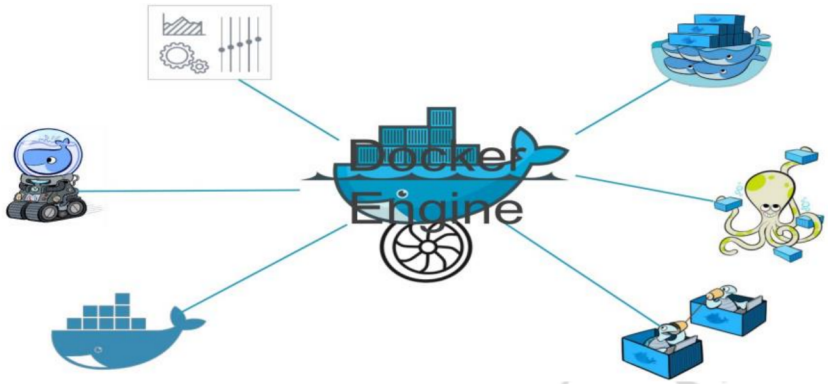
But when somebody says “Docker” they can be referring to any of at least three things:

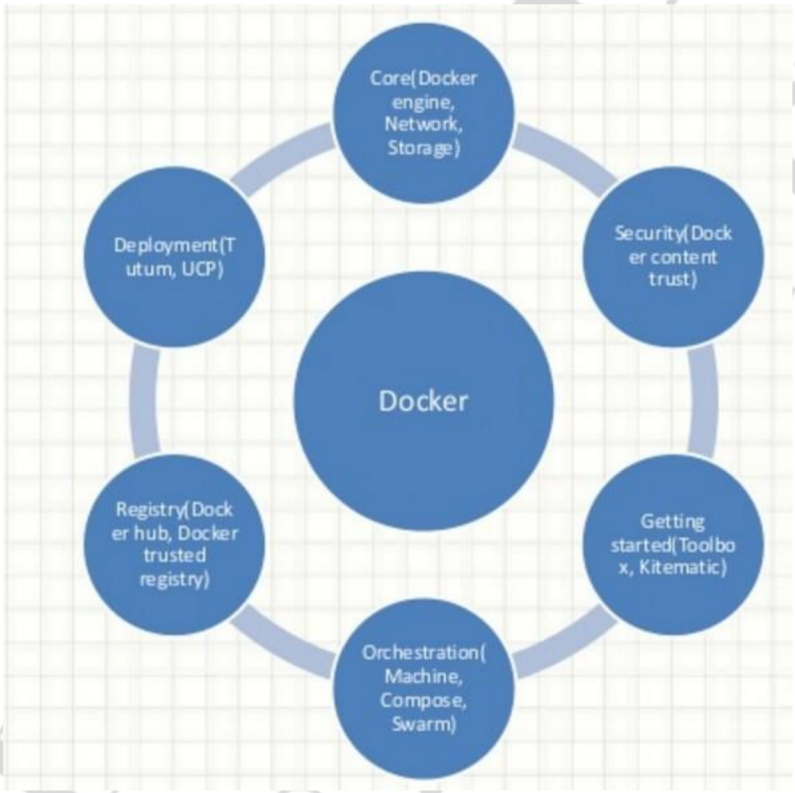
Docker, Inc. the company
Docker the container runtime and orchestration technology
3. Docker the open source project

When most of the people talk about Docker they generally refer to the Docker Engine.
Docker engine runs and orchestrate containers. As of now we can think docker engine like a hypervisor. The same way as hypervisor technology that runs virtual machines, the Docker Engine is the core container runtime that runs containers.



There are so many Docker technologies that gets integrated with the docker engine to automate, orchestrate or manage docker containers.





6. Installing Docker.

Docker can be installed on Windows, Mac and Linux OS.

We will install docker on Ubuntu 16.04 server in this tutorial.

Docker can be installed directly from Ubuntu repositories but that may not be the latest version of Docker engine. To install the latest and greatest version, we will install it from official Docker repository.

Uninstall old versions

Older versions of Docker were called `docker` or `docker-engine`. If these are installed, uninstall them:

```
$ sudo apt-get remove docker docker-engine
```

It's OK if `apt-get` reports that none of these packages are installed.

Add GPG key for Docker repository.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Add the Docker repository

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Update the package

```
sudo apt-get update
```

Install Docker:

```
sudo apt-get install docker-ce -y
```

Docker should now be installed, the daemon started, and the process enabled to start on boot. Check that it's running:

```
sudo systemctl status docker
```

Docker commands can be executed by root user or by providing sudo.
We can run docker command with normal user as well, to do it we need to add the user into the docker group.

```
sudo usermod -aG docker <username>
```

You need to logout and login to reflect the changes.

Run Docker command to check if it's working.

```
docker --version
```

When you install docker engine you get two components.

- Docker Client
- Docker Engine

7. Docker Engine's Big Picture

Let's quickly feel and taste the docker engine before we dive deep into it.

Broadly there are two areas where we operate in docker engines.

- Docker Images
- Docker containers

Images

As of now you can think images as vagrant boxes. It's very much different from the vm images but it will feel as same initially. Vagrant boxes are stopped state of a VM and Images and stopped state of containers.

Run docker images command.

```
$ docker images
```

This command will list the downloaded images on your machine, so you won't see anything now in the output. We need to download some images, in docker world we call it **Pulling** an image.

So where does it pull the image from? Again, same analogy as vagrant boxes. We download the vagrant boxes from vagrant cloud, docker images are downloaded from **Docker Registries**, the most famous docker registry is DockerHub. There are other registries as well, from google, redhat etc.

```
$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
bd97b43c27e3: Pull complete
6960dc1aba18: Pull complete
2b61829b0db5: Pull complete
1f88dc826b14: Pull complete
73b3859b1e43: Pull complete
Digest: sha256:ea1d854d38be82f54d39efe2c67000bed1b03348bcc2f3dc094f260855dff368
Status: Downloaded newer image for ubuntu:latest
```

Run the docker images command again to see the ubuntu:latest image you just pulled.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              7b9b13f7b9c0        2 weeks ago         118MB
```

We'll get into the details of where the image is stored and what's inside of it in the next chapter. For now, it's enough to understand that it contains enough of an operating system (OS), as well as all the code to run whatever application it's designed for. The ubuntu image that we've pulled has a stripped-down version of the Ubuntu Linux OS including a few of the common Ubuntu utilities.

Containers

Now that we have an image pulled locally on our Docker host, we can use the docker run command to launch a container from it.

```
imran@DevOps:~$ docker run -it ubuntu:latest /bin/bash

root@b8765d3a67a9:/#
```

Look closely at the output form the command above. You should notice that your shell prompt has changed. This is because your shell is now attached to the shell of the new container - you are literally inside of the new container!

Let's examine that docker run command.

- docker run tells the Docker daemon to start a new container.
- The -it flags tell the daemon to make the container interactive and to attach our current shell to the shell of the container.
- Next, the command tells Docker that we want the container to be based on the ubuntu:latest image.
- We tell it to run the /bin/bash process inside the container.

Run the following ps command from inside of the container to list all running processes.

```
root@b8765d3a67a9:/# ps -elf

F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD

4 S root         1     0  0  80   0 -  4560 wait   19:46 ?        00:00:00 /bin/bash

0 R root        10     1  0  80   0 -  8606 -      19:50 ?        00:00:00 ps -elf
```

As you can see from the output of the ps command, there are only two processes running inside of the container:

- PID 1. This is the/bin/bash process that we told the container to run with the docker run command.
- PID 10. This is the ps -elf process that we ran to list the running processes.

The presence of the ps -elf process in the output above could be a bit misleading as it is a short-lived process that dies as soon as the ps command exits. This means that the only long-running process inside of the container is the /bin/bash process.

Press Ctrl-PQ to exit the container. This will land you back in the shell of your Docker host. You can verify this by looking at your shell prompt.

In a previous step you pressed Ctrl-PQ to exit your shell from the container. Doing this from inside of a container will exit you form the container without killing it. You can see all the running containers on your system using the docker ps command.

```
imran@DevOps:~$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES

b8765d3a67a9        ubuntu:latest       "/bin/bash"         6 minutes ago       Up 6 minutes
inspiring_heyrovsky
```

The output above shows a single running container. This is the container that you created earlier. The presence of your container in this output

proves that it’s still running. You can also see that it was created 6 minutes ago and has been running for 6 minutes.

Attaching to running containers

You can attach your shell to running containers with the docker exec command. As the container from the previous steps are still running let’s connect back to it.

Note: The example below references a container called “inspiring_heyrovsky”. The name of your container will be different, so remember to substitute “inspiring_heyrovsky” with the name or ID of the container running on your Docker host.

```
imran@DevOps:~$ docker exec -it inspiring_heyrovsky /bin/bash

root@b8765d3a67a9:/#
```

Notice that your shell prompt has changed again. You are back inside the container.

The format of the docker exec command is:
docker exec -options <container-name or container-id> <command>.

In our example, we used the -it options to attach our shell to the container’s shell. We referenced the container by name and told it to run the bash shell.

Exit the container again by pressing Ctrl-PQ.

Your shell prompt should be back to your Docker host.

Run the docker ps command again to verify that your container is still running.

```
imran@DevOps:~$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES

b8765d3a67a9        ubuntu:latest       "/bin/bash"         13 minutes ago      Up 13 minutes
inspiring_heyrovsky
```

Stop the container and kill it using the docker stop and docker rm commands.

```
imran@DevOps:~$ docker stop inspiring_heyrovsky

inspiring_heyrovsky
```

```
imran@DevOps:~$ docker rm inspiring_heyrovsky

inspiring_heyrovsky
```

Verify that the container was successfully deleted by running another docker ps command.

```
imran@DevOps:~$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES

imran@DevOps:~$
```

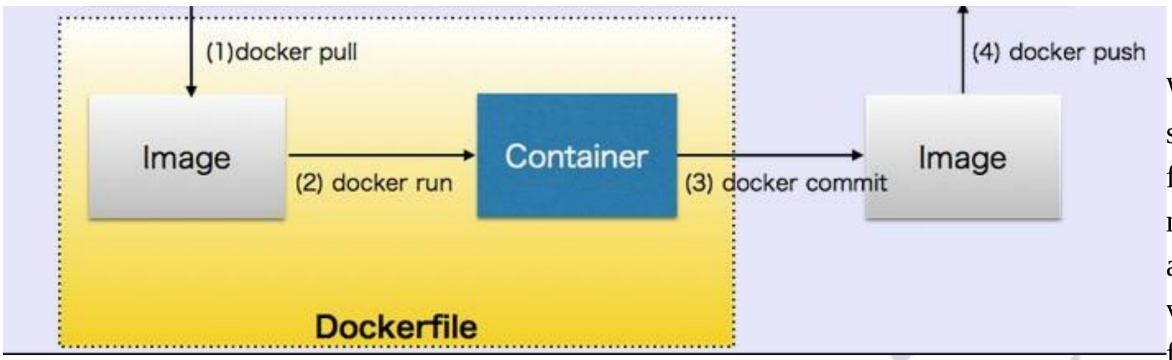
Now you would have got the taste of docker images and containers. We pulled an image, executed a container, stopped and removed it. In next section, we will dig more in detail of images and then containers.

8. Images

We have seen very basic stuff of docker images, now we will do a deep dive into docker images.

Images are built and distributed like software. As we have seen in Continuous Integration chapter that there should be a process of Build & Release a software, we must do the same if we are releasing Images.

We have mentioned earlier Images are stopped state of container so you can stop a container and create a new image from it.



We see from above figure that we pull an image, run a container, customize it as per our requiretment, commit the container into an image and then ship it.

However, once you've started a container form an image, the two constructs become dependent on each other and you cannot delete the image until the last container using it has been stopped and destroyed. Attempting to delete an image without stopping and destroying all containers using it will result in errors.

Images that we are shipping should be light weight and should only contain the files and libraries that required to run the application inside it. For example, if we are shipping a java application, it should contain only java libraries, application server like tomcat and files to run our app and not anything extra.

Pulling Images.

```
imran@DevOps:~$ docker pull node:latest
latest: Pulling from library/node
ef0380f84d05: Pull complete
24c170465c65: Pull complete
4f38f9d5c3c0: Pull complete
4125326b53d8: Pull complete
a1468c06f443: Pull complete
cb113e1791ca: Pull complete
519ce9303f95: Pull complete
f15f31d0549a: Pull complete
Digest: sha256:1d496e5c8e692dfabeb1cc8a18f01e2b501111f32c3d08e94e5402daeceb94e6
Status: Downloaded newer image for node:latest
```

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
node                latest              f3068bc71556        27 hours ago        667MB
ubuntu              latest              7b9b13f7b9c0        2 weeks ago         118MB
```