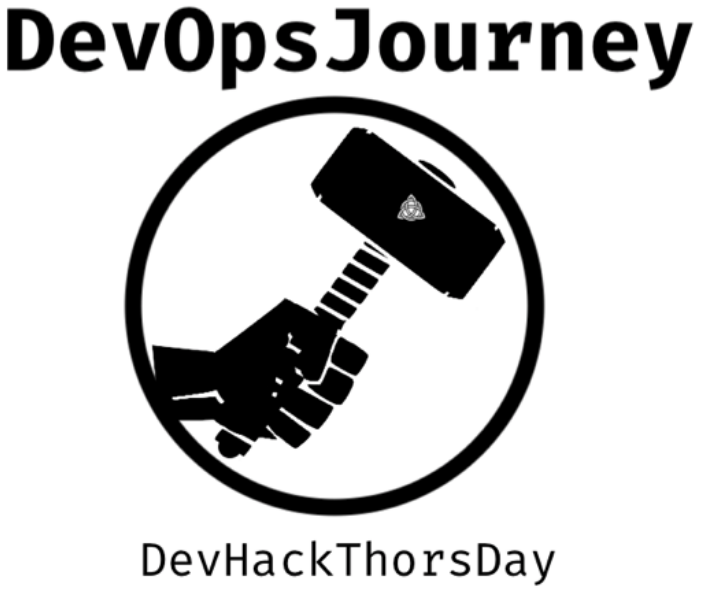
DevOps Journey

jimmy mg lim & many other contributors



DEVOPS Journey

On the summer of 2002, as a group of software engineers were out at a pub enjoying the sun, one of them receives a message from a colleague saying their website was down. This cannot be happening as the same thoughts ran through in the minds of each of the engineers. The spike was expected and precautions were done including adding an additional fixed line, fail-over servers and the lot. The group scrambled to the office started to connect to the servers. No response. Are we hacked? whats happened?

Here one of the engineers, moved quickly straight into the server room, where interestingly enough, the machines were powered off. A quick look around, a monumental discovery was revealed, the cleaner unknowingly unplugged the mains which lead to a chain of disasters.

So what can we learn from this? Even the most thoughtout failover prevention have infact a chance of failure and its only on the rear mirror that we can anticipate this event. The most mundane things can happen and cause a monumental structure to collapse, and in this case, it was not even the cause of a mis-designed architecture.

And what does this story tell us about DevOps? nothing really.

shocked? moving on, lets dig a bit deeper, before we move into DevOps definition. Think about the scenario and how a group of software engineers resolved the issue. its not about role, its not about area or design of architecture. its about how quickly a group a people resolved it irregardless of title. This is the core of the idea that will echo thru eons; fast feedback loop, shared knowledge of the entire stack, and a simple design. Just as Etsy encourages the use of boring technology. Look beyond the hype and use simple battle tested toolings that everyone understands well. Ultimately, its the business we are trying to run, not an ego of adopting the trendy technologies. trend sucks.

DevOps is a set of software development practices that combines software development (Dev) and information technology operations (Ops) to shorten the systems development life cycle while delivering features, fixes, and updates frequently in close alignment with business objectives. Different disciplines collaborate, making quality everyone’s job.

next question. who is this book meant for? if you are in technology, developing software or use a form of technology tool to help your business, its you. It does not cover everything in detail, for the details, there are a few

good recommended books in appendices.

The goals of DevOps span the entire delivery pipeline including :

- improved deployment frequency
- faster time to market
- lower failure rate of new releases
- shortened lead time between fixes
- faster mean time to recovery (in the event of a new release crashing or otherwise disabling the current system)

hope that gives an idea; stay cool and read on. from a great man in our era; he with the muscles bigger than arnold, quotes - “from the power of gray skull, you have the power..”

chapter 0 - getting started from zero

skip this chapter if you already have a nice setup with your laptop, and development environment. You may have come from various backgrounds, front end developer, backend developer, sys admin, freelancer design or even an accountant wanting to pick up some skills on development. This mob book will be covering topics on a wide spectrum of software development methods and technologies under the sun; so we will have you covered.

A few things to get yourself embarked on this journey.

- sign up for meetups on technology often; be it Ruby language, cloud initiative or AWS summits; sign up!
- start a github account or heroku account
- familiarise yourself with a basic coding language; ruby, python are good picks to get you started quickly
- sign up to stackoverflow where there is a large range of tech Q and As
- make your laptop your temple; clear it up and start building it as your personalised tool for development

system setup

for your day to day development, either use a linux flavor or use macosx (which is bsd a variant of unix); windows is possible with powershell or cygwin, which allows bash shell on the machine. recommend to stick to linux or unix flavoured machines; it makes more sense.

if you are on mac, use iterm2 (https://iterm2.com/), its much better than the default terminal. On linux, it is recommended to use one of the debian flavors such as ubuntu, linux mint (https://linuxmint.com/) or even debian itself. For each of these, i recommend using MATE (https://mate-desktop.org/) as your desktop due to its lightweight no frills stable interface (https://linuxmint.com/). The alternatives are cinnamon or lxde (https://lxde.org/). Others such as kde or kubuntu (ubuntu bundled kde) are heavier on resources and will run slower even if they look fancier.

as a mac user, your system already has

- ruby
- python
- default terminal

it is recommended to use a package manager such as homebrew to improve your system application management. this in the long run will keep your

system clear from bloat. generally it is recommended to use mac as its the best of both worlds,

- (1) you get a default linux base system having you to be familar with system structures
- (2) there are quite alot of great developers on macs so your support on getting better quality software is there
- (3) there are quite a number of mac users so when you develop or have issues, its quite easy to get help online from forums

IDEs, are integrated development environments. over my career, swapping one over another, ides are overrated. get a good editor to start with, notepad/brackets/nano/atom/vim. they are all relatively good and i have ordered them by difficulty; you might want to personally progress thru them from left to right. Vim is outright a monster to deal with but once you are there, you will love it; just like marmite. some of the greatest devs are on vim, it has been there since 1991 and still very active; of course for a reason. its in the top 5 of editors, uber great and not going to be replaced probably in the next 100 years. Just like that katana passed from generation to generation in ancient times.

development habits

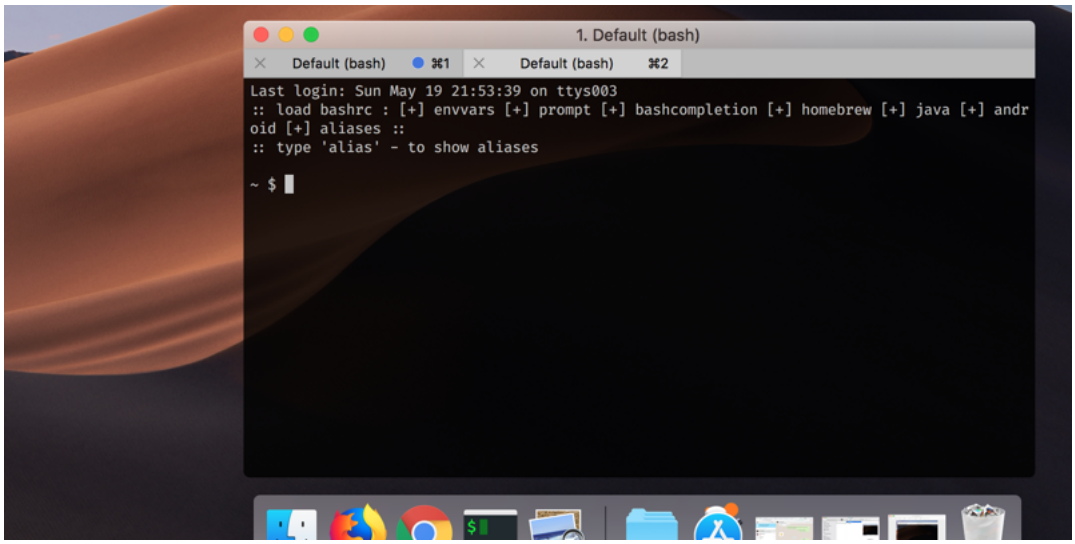
google, youtube, stackoverflow and github are your friends. google or stackoverflow your questions to help you progress in the long run. you are encouraged to setup accounts in both stackoverflow/github (free) and start contributing by answering questions or raising issues. this will help you in your journey in software development.

try to be part of a development team or group of power users. they often impart or share good habits to allow you to progress very rapidly. join meetups (like mentioned above) to further your skill and widen your perspective on an area.

Youtube?? yes, youtube. not for surfing games, gossips but yeah, search for tutorials on programming and development. there are tons of good tooling introduction and great instructors there often releasing videos for free.

finally be an openminded **opinionated** developer; :D learn all the good tools where you can, and be religious on a few good ones. keep learning, keep bettering. the difficulty here is **knowing** which are the good ones. one smart person in our era left us with a good proverb; stay hungry stay

foolish. we boldly top that by adding : stay naive. whaaadup...hold the friggin hammer and be brave.



chapter 1 - docker

Docker is a collection of inter-operating services that employs operating-system-level virtualization to facilitate development and delivery of software inside standardised software packages called containers. The software that hosts and manages the containers is called docker engine. Its first started in 2013 and is developed by docker, inc.

The docker containers are isolated from each other and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines such as vagrant on virtualbox. Containers are created from images that specify their precise contents and are often created by combining and modifying standard images downloaded from public repositories.

priming

here are some docker commands to get you started. give them a spin in your terminal and check docker website to see more about other commands, such as volume, compose etc.

```
docker ps -a
# to show all running docker processes

docker images
# to see all local images

docker images purge
# to remove older version of images. such as called dangling images

docker build -t [name] .
# to build a container based on current dockerfile and tag it

docker run -it myruby
# to run load image (myruby) and run container application it interactively

docker run -d -p 3000:3000 --name myproc myruby
# to run image (myruby) as a daemon

docker run -d -p 8080:80 --name myproc myruby
# map from 8080 (host) to 80 (container)
```

```
docker exec -it [containerid] sh -c "echo a && echo b"
# to run commands in a running container

docker exec -it [containerid] bash
# to enter the interactive shell of the running container

docker rmi [image id]
# to remove docker images from system. example with image id 4c0d231f6a74

docker rm [container id]
# to remove running/stopped containers

docker system prune [-a]
# to remove all danging containers (which are stopped) and networks. optional -
a for all

docker container prune
# to remove all danging containers (which are stopped)

docker volume prune
# to clean up unused volumes
```

challenge one

fulfill the following requirements :

- on your local machine install docker
- create a dockerfile based off alpine linux
- install nginx as a web server in the container
- spin up and share the container on public docker hub (https://hub.docker.com/)
- have the final image under 30 mb

hints / pitfalls :

- try adding multiple runs one by one to ensure that the commands work as expected
- it is possible to run an empty container and execute commands to see how the container is responding
- use docker logs to see whats wrong when a container dies randomly

suggested solutions :

- JamesMannion (https://github.com/mannion007/devopsjourney/tree/master/src/assignment1)
- LiamPetch (https://github.com/liampetch/devopsjourney/tree/master/src/Challenge1)

challenge two

fulfill the following requirements :

- include an engine (such as phpfpm/ruby-puma/java-tomcat/other) based off your public nginx container
- show a page displaying basic system information
- ensure you are able to run the entire stack within a single container (hint look at tini/s6/dumbinit)

suggested solutions :

- LiamPetch (https://github.com/liampetch/devopsjourney/tree/master/src/Challenge2)

challenge three

fulfill the following requirements :

- include spin up a db container and use volume to mount this database (either mariadb/postgres/couchdb)

- use compose methods to spin up both your web server container and db container in the same network
- initiate a basic write and read from the webserver container to the db container; a basic helloworld is suffice
- (additional challenge) : use custom networking to achieve this by not using docker compose

challenge four

fulfill the following requirements :

- spin up docker swarm using 4-5 nodes - you can use virtualbox driver to create multiple machines (https://docs.docker.com/get-started/part4/)
- run your stack in the docker swarm ensuring that the web service can perform the same read and writes
- test your docker swarm web serve uptime by randomly killing of 2 nodes

challenge five

fulfill the following requirements :

- deploy either (a) ghost blog, or (b) wordpress, or (c) jekyllrb
- ensure that your build can serve as a development environment; i.e. reasonable syncing speed between host and containers

chapter 2 - pipelines

In todays world, you will come across pipelines, containers and maybe rabbits. for this lets focus on rabbits, i mean pipelines. pipelines are essentially steps. just like going to 20 steps in your lego manual to get to the end result of a robin hood treehouse. the pipeline itself is an instruction set.

The goal of pipelines is to ensure a few points :

- verbosity in building
- able to repeat
- able to rebuild
- ease of reviewing steps and issues

Why do we need pipelines? ever heard of the common term “it works on my machine”. pipelines are just to ensure that we have less of that and make sure a common way to get to the goal; in this case, deploying to production with expected quality.

today, there are ways to build pipelines from basic to complex including

- stepped makefiles with cron jobs
- jenkins
- concourse
- gitlab
- travis
- circle

The key in pipelines is to ensure that the correct build happens with the right dependencies. As your application grows in functionality, it no doubt will grow in dependencies such as libraries, framework components, runtime buildtime environments and distribution targets.

the key steps in a basic pipeline are as follows

- prime environment
- test src in unit tests
- build
- test in integration tests
- test in performance / integrity tests
- deploy

Canary deployments; have you heard of canary and blue green. on canary deployments, the word canary refers to an old method where miners who are

working deep in the mines, would bring with them a canary bird. as the oxygen levels in the mines may not be suffice for miners, the canary bird itself serves as a warning for miners. if the canarys suffocate, it would mean for the miners to quickly leave the area. canary deployments make use of the word and therefore means a sub release or release to a small group of users to see if the new feature works as intended.

chapter 3 - testing

The history of unit testing can be traced back to the early days of software development. In the 1970s, a number of researchers began to develop methods for testing individual units of code. One of the earliest methods was called "white box testing", which involves testing the code itself, without considering the external environment.

In the 1980s, unit testing became more popular as developers began to realize the importance of early defect detection. A number of unit testing frameworks were developed during this time, including SUnit for Smalltalk and JUnit for Java.

In the 1990s, unit testing became even more widely adopted as part of the development of agile methodologies such as Extreme Programming (XP). XP emphasizes the importance of testing throughout the development process, and unit testing is a key part of this.

Today, unit testing is a standard practice in software development. It is used by developers of all levels of experience and in all types of software projects.

Here are some of the key figures in the history of unit testing:

Glenford Myers: Myers is considered the father of unit testing. He published the book "The Art of Software Testing" in 1979, which introduced the concept of white box testing.

Kent Beck: Beck is a software engineer who is known for developing the Extreme Programming (XP) methodology. He also created the JUnit unit testing framework for Java.

Erich Gamma: Gamma is a software engineer who is known for co-authoring the book "Design Patterns: Elements of Reusable Object-Oriented Software". He also co-developed the JUnit unit testing framework.

What is Unit Testing?

Unit testing is a software testing method by which individual units of source code, such as functions, modules, and classes, are tested to determine whether they meet their design specifications. Unit testing is often automated using frameworks that allow developers to write tests that can be run repeatedly and consistently.

Why is Unit Testing Important?

Unit testing is important for a number of reasons, including:

it helps to improve the quality of software by catching errors early in the development process. it helps to prevent defects from being introduced into the code later on. it helps to ensure that the code is testable and maintainable. it can help to identify potential performance bottlenecks. it can help to improve the code coverage, which is the percentage of the code that is actually tested. it can help to document the code by providing examples of how it should be used.

How to Write Unit Tests in Python

There are a few different ways to write unit tests in Python. One way is to use the unittest framework. The unittest framework is included with the standard Python library.

To write a unit test using the unittest framework, you need to create a class that inherits from unittest.TestCase. Then, you can write methods in the class that test your code.

Here's an example of a unit test written in Python using the unittest framework:

```
import unittest


class TestFactorial(unittest.TestCase):
  """Tests the factorial function."""

  def test_factorial_zero(self):
    self.assertEqual(factorial(0), 1)

  def test_factorial_one(self):
    self.assertEqual(factorial(1), 1)

  def test_factorial_two(self):
    self.assertEqual(factorial(2), 2)
```

```
  def test_factorial_three(self):
    self.assertEqual(factorial(3), 6)

  def test_factorial_four(self):
    self.assertEqual(factorial(4), 24)
```

This unit test tests the factorial() function by calling it with different values and verifying that the results are correct.

The assertEqual() method is used to assert that two values are equal. If the values are not equal, the unit test will fail.

Other assertions that can be used in unit tests include:

assertNotEqual(): Asserts that two values are not equal. assertTrue(): Asserts that a value is True. assertFalse(): Asserts that a value is False. assertRaises(): Asserts that a call to a function raises an exception.

Running Unit Tests in Python

Once you have written your unit tests, you can run them using the unittest framework. To do this, you can use the unittest.main() function. For example:

```
unittest.main()
```

This will run all of the unit tests in the current file.

You can also run specific unit tests by passing their names to the unittest.main() function. For example:

```
unittest.main(['test_factorial_zero', 'test_factorial_one'])
```

This will run the test_factorial_zero() and test_factorial_one() unit tests.

Other Unit Testing Frameworks

In addition to the unittest framework, there are a few other unit testing frameworks available for Python. Some of these frameworks offer additional features, like the ability to run tests in parallel or generate reports on the results of the tests.

Some of the other popular unit testing frameworks for Python include:

Pytest Nose pytest-bdd Robot Framework

Unit testing is an important part of software development. It can help you find bugs early, make sure your code is testable and maintainable, and

improve the overall quality of your software.

Here are some tips for writing good unit tests:

- test small units of code.
- test for expected and unexpected behavior.
- use assertions to verify the results of your tests.
- write clear and concise test cases.
- run your unit tests frequently.

misc topics

- compliance
- architectures
- monitoring and reporting
- dev tooling