12-Factor App Design

Our Objectives

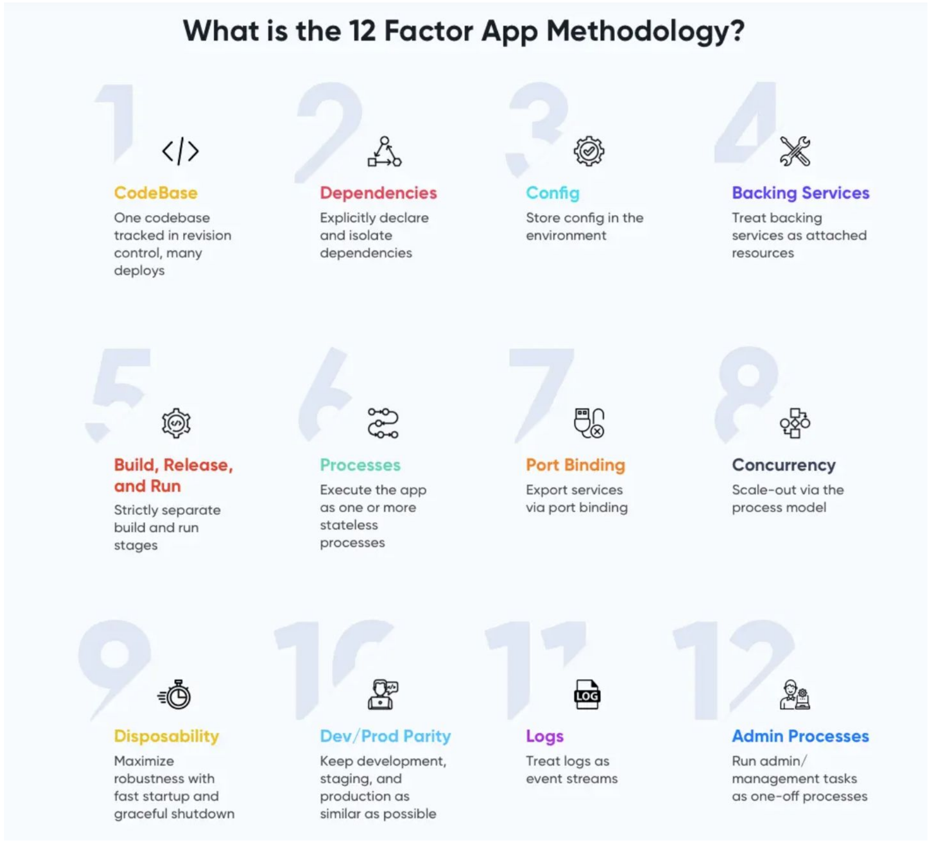
- Minimize ramp up time and cost of new developers joining the same project
- Increase portability between environments
- Eliminate the need for administration of servers and systems
- Maximize the agility of the continuous deployment process
- Reduce the amount of changes to tooling, architecture, and processes when scaling up
- Reduce time to-market for new innovations by adopting modern cloud services (e.g., AI, ML, Serverless)

Source: https://www.castsoftware.com/blog/the-12-factor-app-manifesto-assessing-your-applications

12-Factor App Methodology

It defines a set of rules to follow when building SaaS applications encouraging developers to keep the following goals in mind during development:

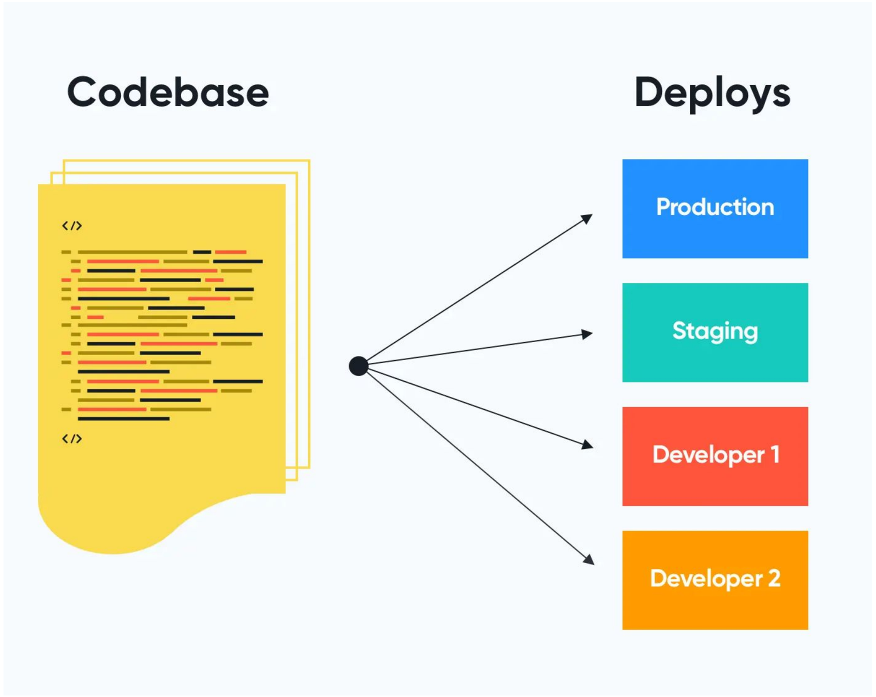
- To avoid the cost of software erosion
- To control the dynamics of organic growth of an app over time
- To Improve collaboration between developers working on same app's codebase
- To raise awareness of systemic problems that occur in modern engineering



What is the 12 Factor App Methodology?

1. **CodeBase** – One codebase tracked in revision control, many deploys
2. **Dependencies** – Explicitly declare and isolate dependencies
3. **Config** – Store config in the environment
4. **Backing Services** – Treat backing services as attached resources
5. **Build, Release, and Run** – Strictly separate build and run stages
6. **Processes** – Execute the app as one or more stateless processes
7. **Port Binding** – Export services via port binding
8. **Concurrency** – Scale-out via the process model
9. **Disposability** – Maximize robustness with fast startup and graceful shutdown
10. **Dev/Prod Parity** – Keep development, staging, and production as similar as possible
11. **Logs** – Treat logs as event streams
12. **Admin Processes** – Run admin/ management tasks as one-off processes

- Each application should have a single codebase. However, deploying to multiple environments is possible.
- If an application has multiple codebases, it violates the methodology and becomes known as a *distributed system*.
- But, each component in a distributed system is an app and can comply with the 12-Factors.

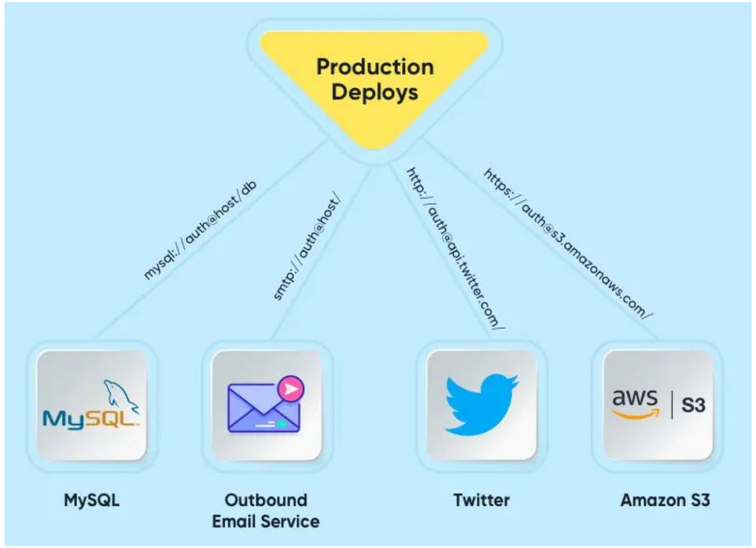


- An app must always declare all the dependencies and their correct versions explicitly.
- Apps might depend on external packages or libraries. Still, you should never think that these will be available on the target system.
- A Twelve Factor Application never depends on the system-wide packages' implicit existence.

- Store config in environment variables (env vars or env).
- The reason is env is easy to change between the deploys without having to change the code.
- These environment variables are never grouped in a 12-Factor App but are managed independently for each deploy.

Backing Services

- Backing Services means any attached services that the app consumes over the network for executing its normal operations such as MySql or S3.
- An app that complies with 12 Factor methodology makes no distinction between these services and treats all like attached resources accessed using a URL or other credentials stored in config.
- If the location or connection details of such service changes, you shouldn't need to make changes in the code. These details should be available in the config.

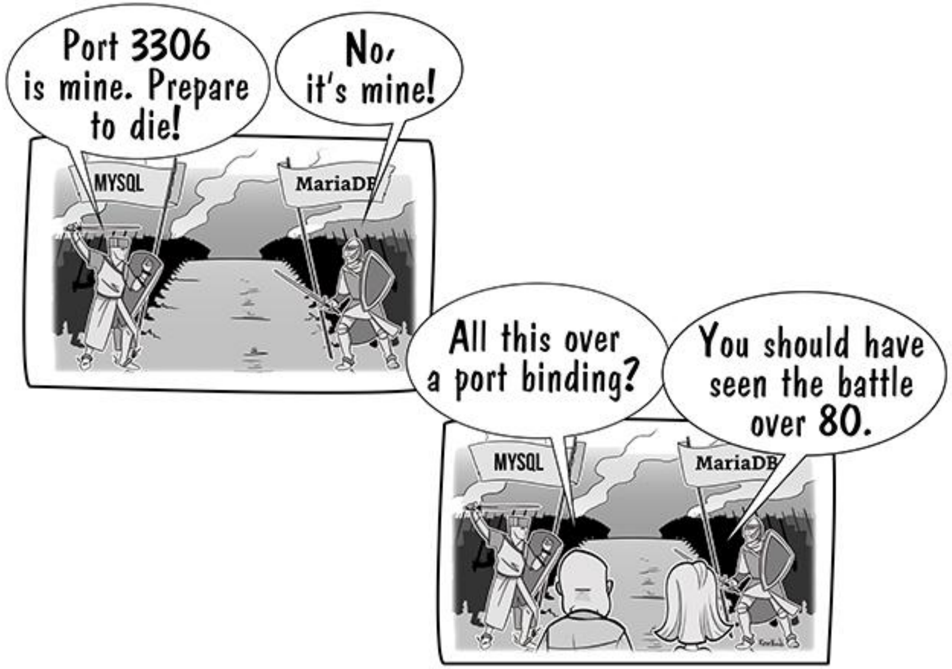


- These three stages should be separated
    - Build: It converts the code repo into an executable bundle of code called build, along with fetching vendor dependencies.
    - Release: It takes the build and combines it with the current config of deploy. Therefore, this stage gives us build and config ready for execution.
    - Run: It runs the app in an execution environment.
- This separation can be done using many modern tools, making maintaining the entire system as easy as possible.

- 12 Factor processes are stateless and share nothing.
- Any data that is required time and again must be stored in a stateful backing service.
- Apps never expects that anything cached will be there in the future for new requests.

7 Port Binding

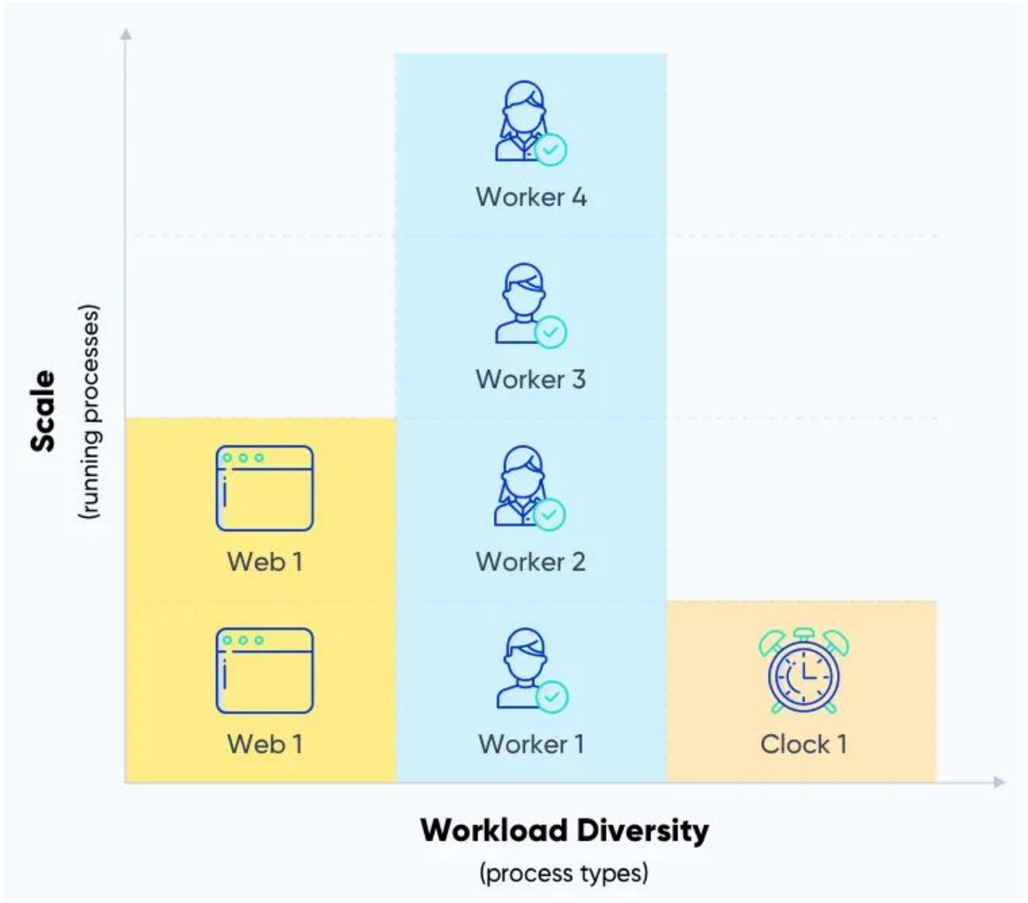
- Apps act as a standalone service and don't require runtime injection of a webserver in an execution environment to make a web-facing service.
- Apps are self-contained and don't require any running or existing app server for the execution. The web application exports HTTP as a service by binding to a port and listens to coming in requests.


Port 3306 is mine. Prepare to die!
No, it's mine!
MYSQL
MariaDB
All this over a port binding?
You should have seen the battle over 80.
MYSQL
MariaDB

BY: ROELBOB ON DECEMBER 28, 2020

Concurrency

- Deploy more copies of your application instead of making your app larger.
- Support horizontal scaling of an app instead of vertical scaling.



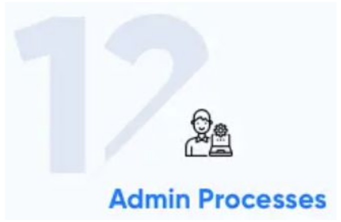
An applications processes are disposable, which means:

- These can start and end at a moment's notice
- Are robust against sudden failure or app crash
- Can shut down gracefully

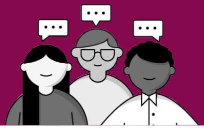
Development, staging, and production of an app should be as similar as possible to ensure that anyone can understand and release it and is designed for continuous deployment by keeping the following gaps as minimum as possible:

- The Time Gap: A developer can write a code and deploy it hours or just a few minutes later.
- The Personnel Gap: Programmers or owners of the code should be closely involved in deploying it.
- The Tool Gap: The tools used for development and production should be as similar as possible.

- Treat log entries as event streams that are routed to a separate service for analysis and archival.
- The app logs will be written as standard outputs, but the execution environment will take care of its storage, capture, and archival.



- Apps should run management or admin tasks in an identical environment as the app’s regular and long-running processes.



Discussion:

12-factor App Rules

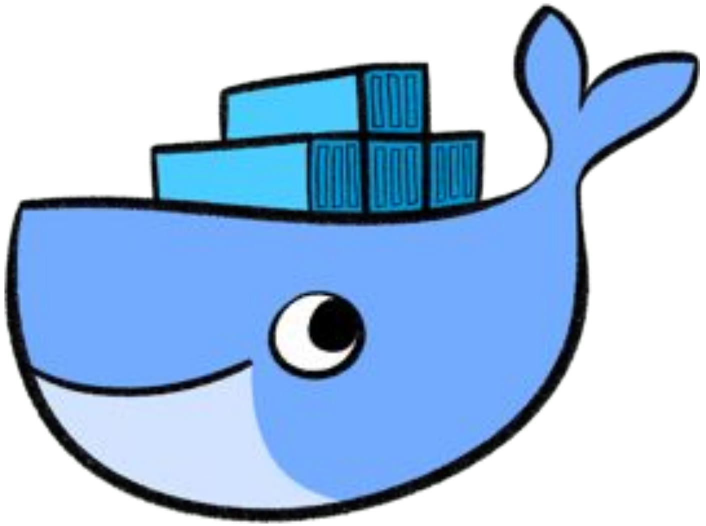
Why would we use these rules?

- Adhering to the rules of the 12 Factor app help to design and maintain robust and architecture.
- It promotes tooling that lends itself towards fulfilling all cloud based requirements and reduces the risks of showing up bugs in a specific environment.
- Apps can grow and shrink in response to demand, saving you infrastructure costs.
- Credentials or any other confidential information should not be in code repo but in the application's environment. This ensures security and also enforces segregation of duties.

Intro to Docker

Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one component. The container becomes the unit for distributing and testing your application.



The Problems Docker Solves

Let's tell a little story:

In the olden days of Web 2.0 (basically, the early 2000s), there was a huge difference between the machine that a developer used to build an app and what was used to deploy that app. It took days, if not weeks, to "provision" or build a computer with the right hardware and software to deploy an app. More often than not, an app that was built on a developer's machine did not work properly in production. These issues could range from the dependencies not connecting, to a different operating system, to versioning problems... or worse.

Or, to put it another way:

One New Way

Things started getting a bit better when virtual machines were introduced.

In computing, a virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide the functionality of a physical computer. VMs can keep applications on the same hardware completely separate.

VMs work because they reduce conflicts among software components and minimize competition for hardware resources. However, they're bulky (each machine requires its own operating system), difficult to maintain, and use up a lot of storage and resources.

The BETTER New Way

While VMs are still used by many companies, other developers kept trying to make things better.

Enter containers.

In contrast to VMs, containers isolate applications' execution environments *but* share the underlying operating system kernel. They use far fewer resources than VMs and start up almost immediately. What's more, they can be packed far more densely on the same hardware and spun up and down en masse with far less effort and overhead.

Knowledge check: *Does anyone know what we mean by an operating system “kernal”?*

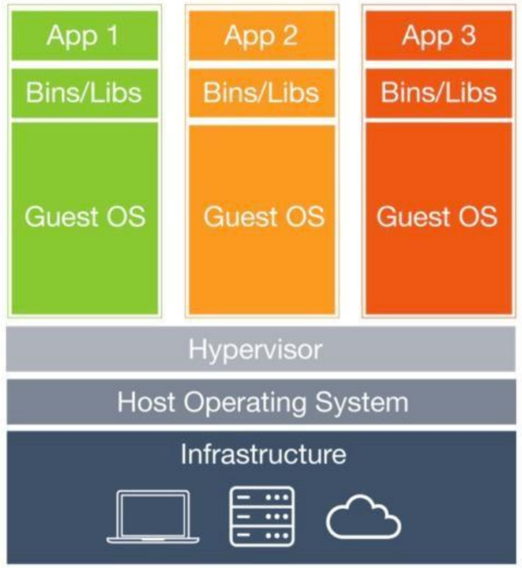
Group Exercise:

Game: Spot the Differences

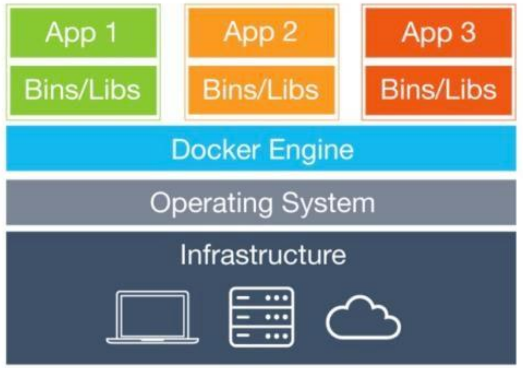
5 mins

Let's play that game where you spot the differences between the two images. What's different in a container? Why do you think that's important?

Virtual Machine



Container



How’s Docker Work

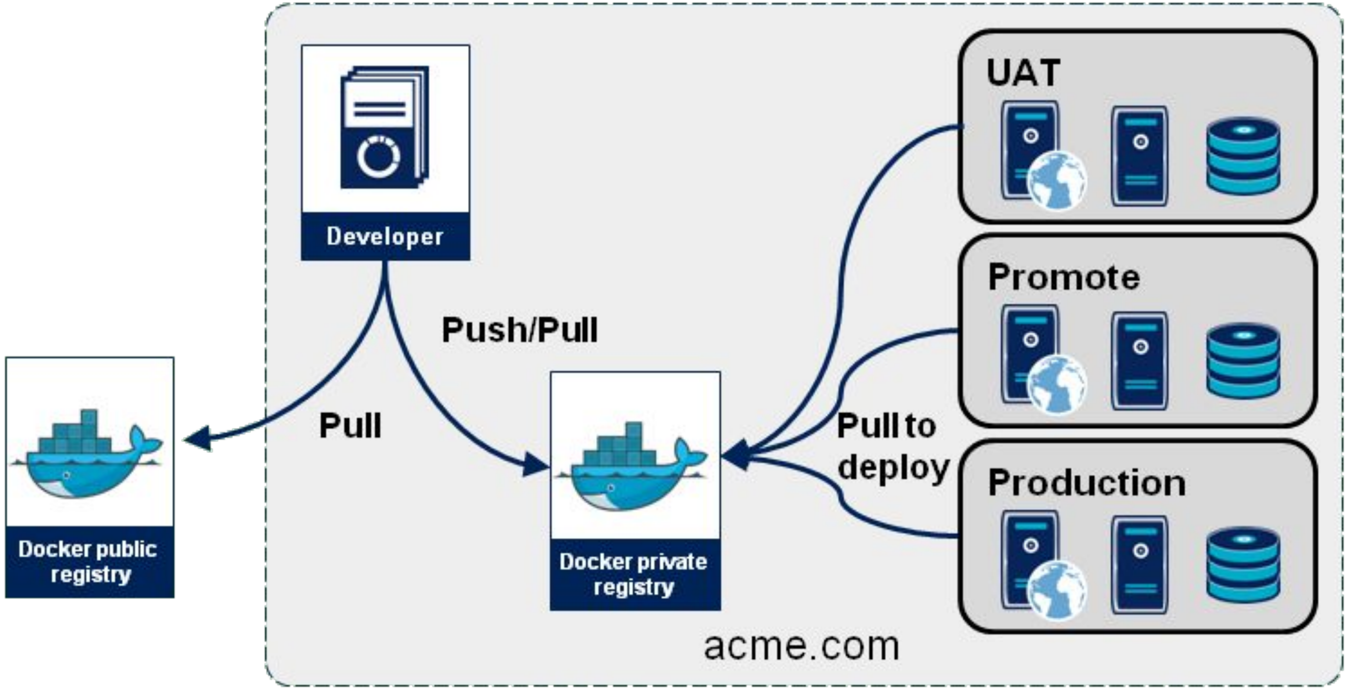
Docker is an open-source platform (the most popular one) for building applications using containers.

However, Docker isn't a completely new technology. Many of the components and principles existed previously. Docker is written in Go and takes advantage of several features of the Linux kernel to deliver its functionality, including namespaces and cgroups (more on those in the Additional Resources section).

The ultimate goal of Docker is to mirror our dev environment with our production environment. This is mostly useful for your back-end but can be applied to your front-end applications as well! It's cool to run Docker locally, but its real benefit comes into play while running it on production machines.

How's Docker Work?

Take a look at this *very* simplified version of how Docker works:



The Main Components of Docker

OK, yes, the short answer is "containers," but those don't just grow on trees. Where do containers come from, you might ask? Let's find out!

The High Level

```
Dockerfile --> Image --> Container --> Docker Engine --> *chef kiss emoji*
```

The Details

- Dockerfile: Instructions to create an image (more on those later).
    - Specifies the OS, languages, etc.
    - Explains what the container will do when it's run.
    - You can create your own or use premade ones.
- Image: A blueprint to build a container.
    - A portable file system that may contain files for the OS, framework files, and the files for an app.
    - Tells each component what to do and when.
    - Containers can share image layers, making them very efficient.

The Details (cont’d)

- Container: You already know this! An isolated instance created from an image, running the app on whatever OS you have.
- Docker Engine: Integrates with the OS to run the Docker containers; sits directly on top of the OS. It's very fast and inexpensive.
  - Docker Daemon: The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. It does the heavy lifting and is a part of the Docker engine.
  - Docker Client (CLI): The CLI uses Docker APIs to control or interact with the Docker daemon through scripting or direct CLI commands.

The Details (cont'd)

- Docker Desktop: A native application for MacOS and Windows machines that provides an easy-to-use development environment for building, shipping, and running dockerized apps.

In this course, we'll also be using Docker Compose, a tool for simplifying the process of developing and testing multi-container applications.

Group Exercise:

Learning by Teaching

15 mins

Teaching a new concept you just learned can help you to develop a better understanding of that concept and also helps with knowledge retention. In your group, have each person choose one of the following topics. Then spend 5-10 minutes to research that topic individually.

- Docker image
- Container
- Dockerfile
- Docker Hub

When everyone is ready, present what you learned to others in your group.

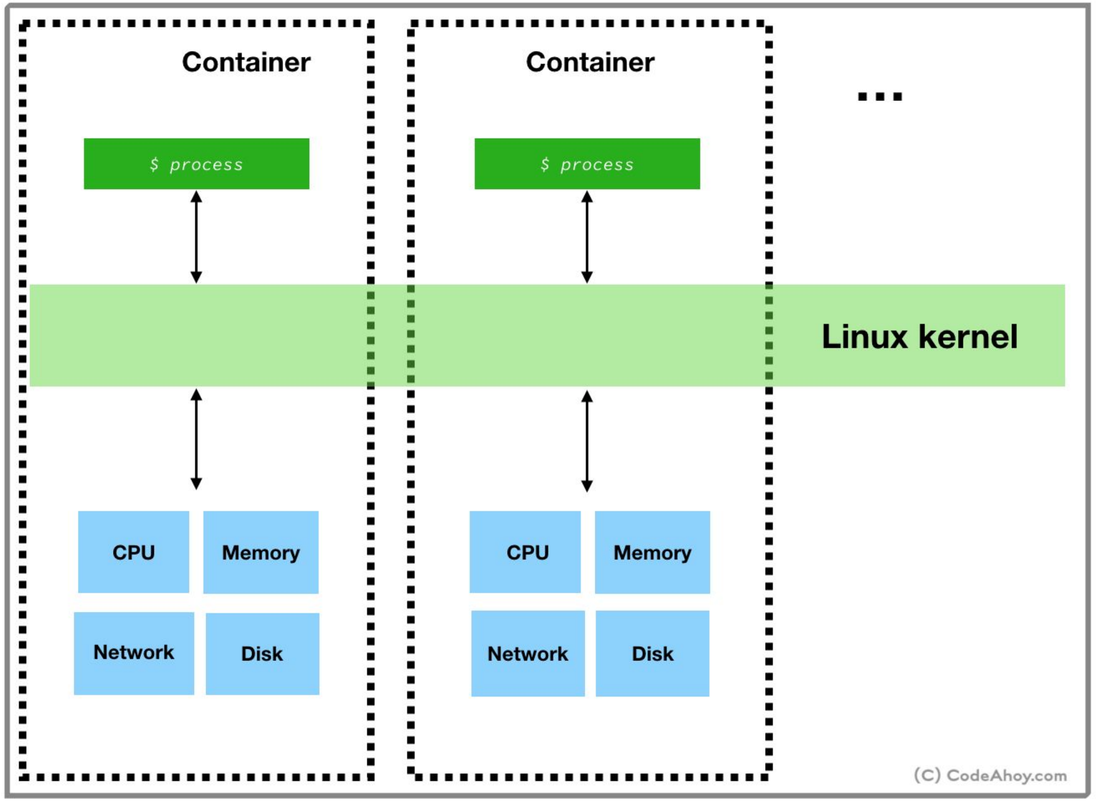
What is a Container?

Operating systems have something called a kernel. This kernel is a running software process that governs access to all the programs, different programs like Chrome, terminal etc. A kernel has complete control over everything in the system.

Lets say Chrome sends a system call to the kernel to read information from the hard drive. And the kernel based on where the request is coming from and what it needs redirects the request to the appropriate section of the hard drive. This entire process of segmenting a resource based on the process that is asking for it is called namespacing.

What is a Container?

This is what a container is based on. A container is not a physical construct that exists inside your computer. Instead it is a process or set of processes that have a grouping of resources specifically assigned to it. Resources like memory, RAM, CPU, network.



How is an image related to a Container?

As mentioned earlier, a Docker image is nothing more than a filesystem snapshot, a set of directories and files. For instance, in an image of Chrome, there will be files that Chrome needs to run and a specific startup command that actually runs Chrome. The kernel will isolate a section of the hard drive and make it available to just the running container.

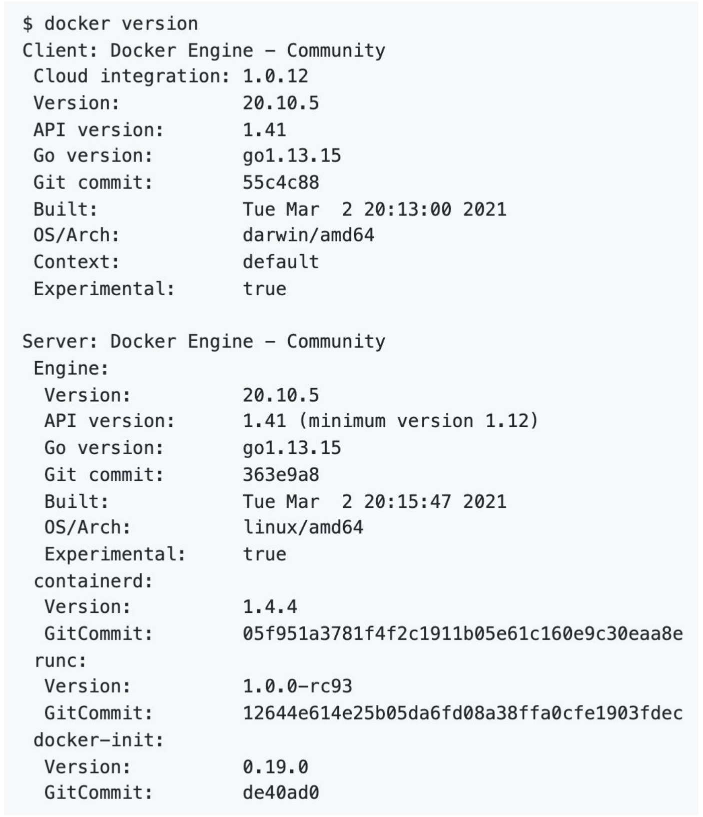
So when you run a container from an image, the filesystem snapshot of that image is placed into a section of the hard drive that can only be accessed by the running container. In our case, when the startup command executes it will start Chrome and only utilize the resources that are segmented specifically for this container.

Interesting Note: Only the Linux operating system uses a kernel and namespacing. So how do Windows and macOS machines use Docker then? A Linux virtual machine is installed as part of the Docker installation. As long as Docker is running it also runs a Linux VM. All of your containers will be created inside of this VM and it hosts a Linux kernel that limits access and segments resources for the different containers

How is an image related to a Container?

Run docker version on the terminal.

You'll see the OS as Linux.



```
$ docker version
Client: Docker Engine - Community
 Cloud integration: 1.0.12
 Version:           20.10.5
 API version:       1.41
 Go version:        go1.13.15
 Git commit:        55c4c88
 Built:             Tue Mar  2 20:13:00 2021
 OS/Arch:           darwin/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.5
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       363e9a8
  Built:            Tue Mar  2 20:15:47 2021
  OS/Arch:          linux/amd64
  Experimental:     true
 containerd:
  Version:          1.4.4
  GitCommit:        05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc:
  Version:          1.0.0-rc93
  GitCommit:        12644e614e25b05da6fd08a38ffa0cfe1903fdec
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

Group Exercise:

Debate and Defend

20 mins

Let’s split half the room into "Pro Docker" and half the room into "No Docker".

Take the next 5–10 mins in your group to do some research to support your argument. Look for:

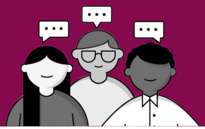
- The benefits/drawbacks of Docker.
- Types of apps or businesses for which Docker is best suited.
- Examples of businesses that use (or don't use) Docker.

Then, we'll host the Great Docker Debate and hear from both sides!

Conclusion

Before we get to installing and using Docker, let's take a moment to recap.

1. What's a container?
2. How does Docker work?
3. Why is Docker so popular right now?
4. What is Docker Hub?



Discussion:

Conclusion

5 mins

Before we get to installing and using Docker, let's take a moment to recap.

1. What's a container?
2. How does Docker work?
3. Why is Docker so popular right now?
4. What is Docker Hub?

Docker Installation

Guided Walk-Through:

Docker Hub

Docker Hub is a repository of free public images that can be downloaded and run. According to Docker, it is the world's largest library and community for container images. You can access and create an account on Docker Hub here.

Hello World

Let's run our first docker command. Open terminal, make sure docker-daemon is running.

Hello World - github

Discussion:

Conclusion

5 mins

Before we get to Docker commands, let's take a moment to recap.

1. What is Docker Hub?
2. How to use Docker Desktop?

Power of Docker

Guided Walk-Through:

Power of Docker

Before learning about Docker commands let’s look at the life of a container.

Follow along here:

[Container Lifecycle - repo]()

How to Dockerize

Dockerfile

1. So far we have used an existing image to run a container. But we haven't talked about how to create an image or like people say dockerize our app.
2. According to docs, a Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. It is used by docker to build images automatically by reading instructions from it.
3. You can specify details like, the OS, languages, etc. Explain what the container will do when it's run. You can create your own or use premade ones.
4. It's basically a set of instructions to set up the container. You can include steps such as copying source code into the container and terminal commands to run for when we are ready to start the application.

Dockerfile example

```
FROM tomcat:8.0-alpine

LABEL maintainer="tristan.hall@generalassemb.ly"

ADD JavaWebApp.war /usr/local/tomcat/webapps/

EXPOSE 8080

CMD ["catalina.sh", "run"]
```

Dockerfile is named as just `Dockerfile`.
It is case-sensitive and without any extension.

Dockerfile

```
FROM tomcat:8.0-alpine

LABEL maintainer="tristan.hall@generalassemb.ly"

ADD JavaWebApp.war /usr/local/tomcat/webapps/

EXPOSE 8080

CMD ["catalina.sh", "run"]
```

Lets go over what these instructions mean:

- The `FROM` instruction initializes a new build stage and sets the base image for subsequent instructions. So in this case we are using tomcat image.
- The `LABEL` instruction sets the Author field of the generated images. You could use any key-value pair in labels.
- The `ADD` instruction copies new files, directories or remote file URLs from and adds them to the filesystem of the image at the path .
- The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime.
- The `CMD` instruction specifies what to run when the container (not the image) is run. In our case, Tomcat server is started by running the shell script that starts the web container. There can only be one `CMD` instruction in a `Dockerfile`.

Guided Walk-Through: Dockerfile

There are lot of other instructions you can have in a Dockerfile. We will go over some of them as needed.

Let's see how we can start a simple node.js application in a docker container from scratch.