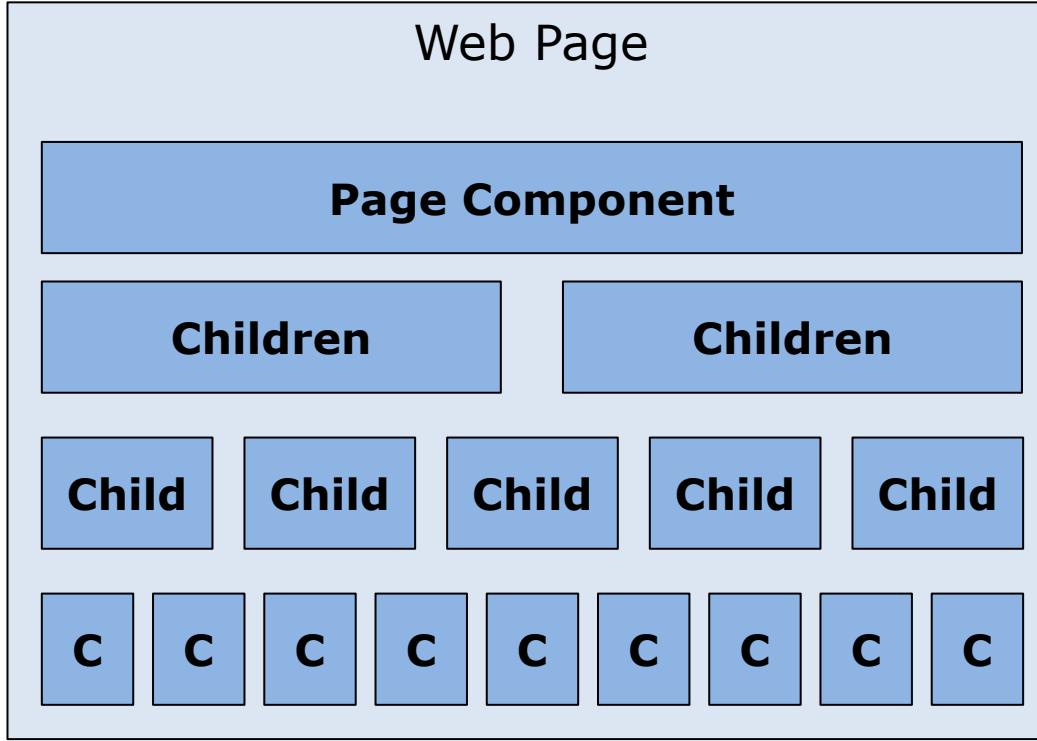
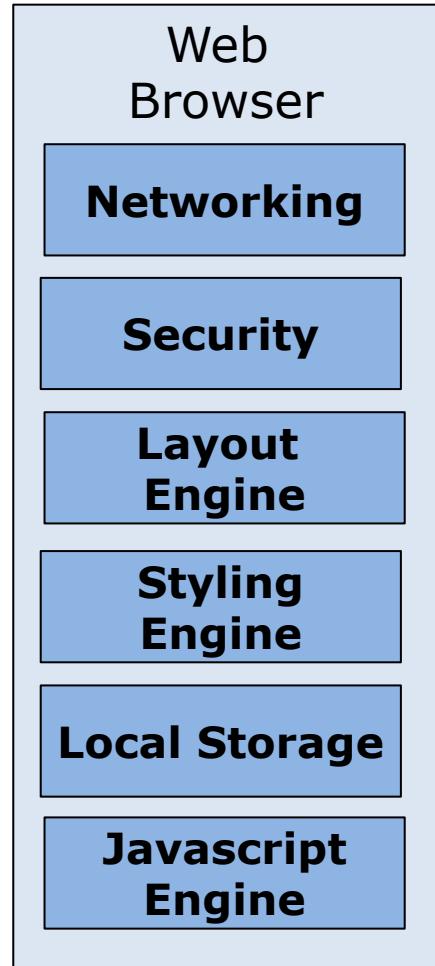


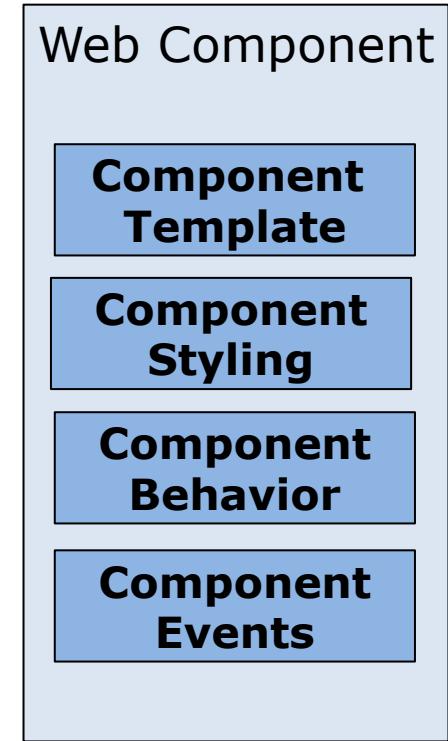
**SE 577**  
**Software Architecture**

**SPA, API, Linux, Docker &  
Kubernetes (k8s) Architecture  
Primer**

# Single Page Application Architecture - SPA



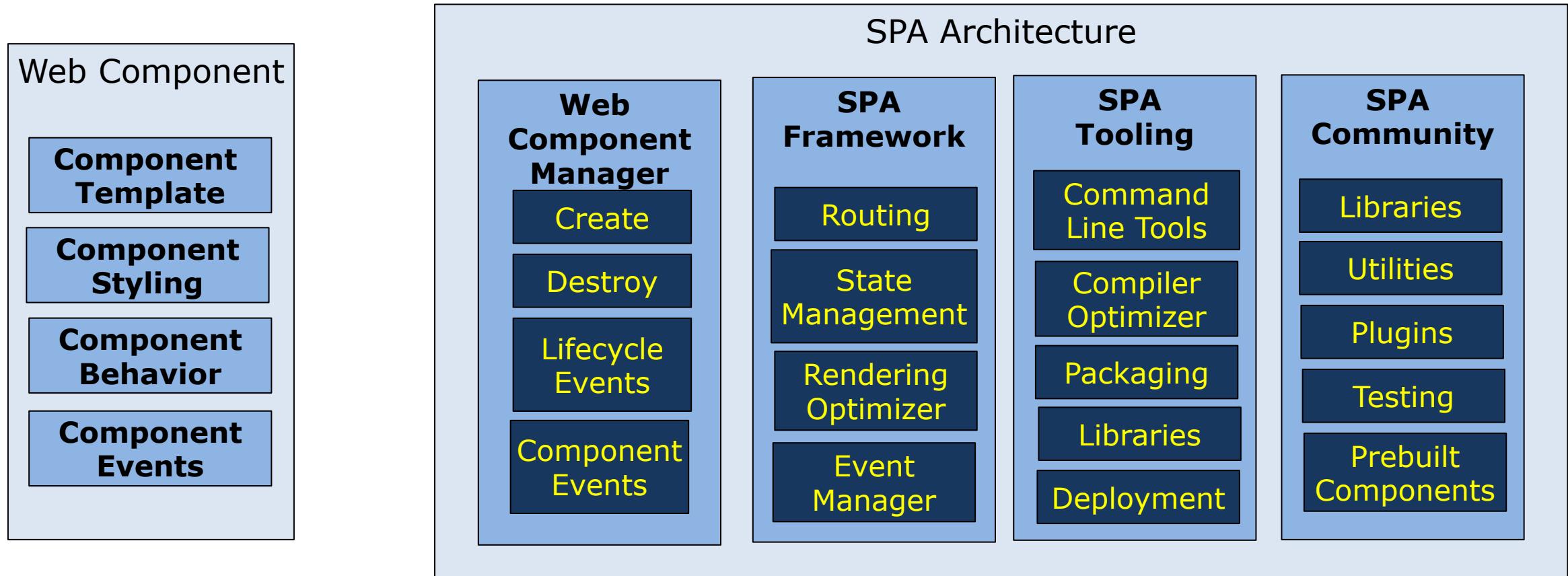
Kind of Trivial but Web Pages can be thought of as a hierarchy of components, where components nest, the leafs of the webpage are native controls such as entry fields, static text, buttons, etc



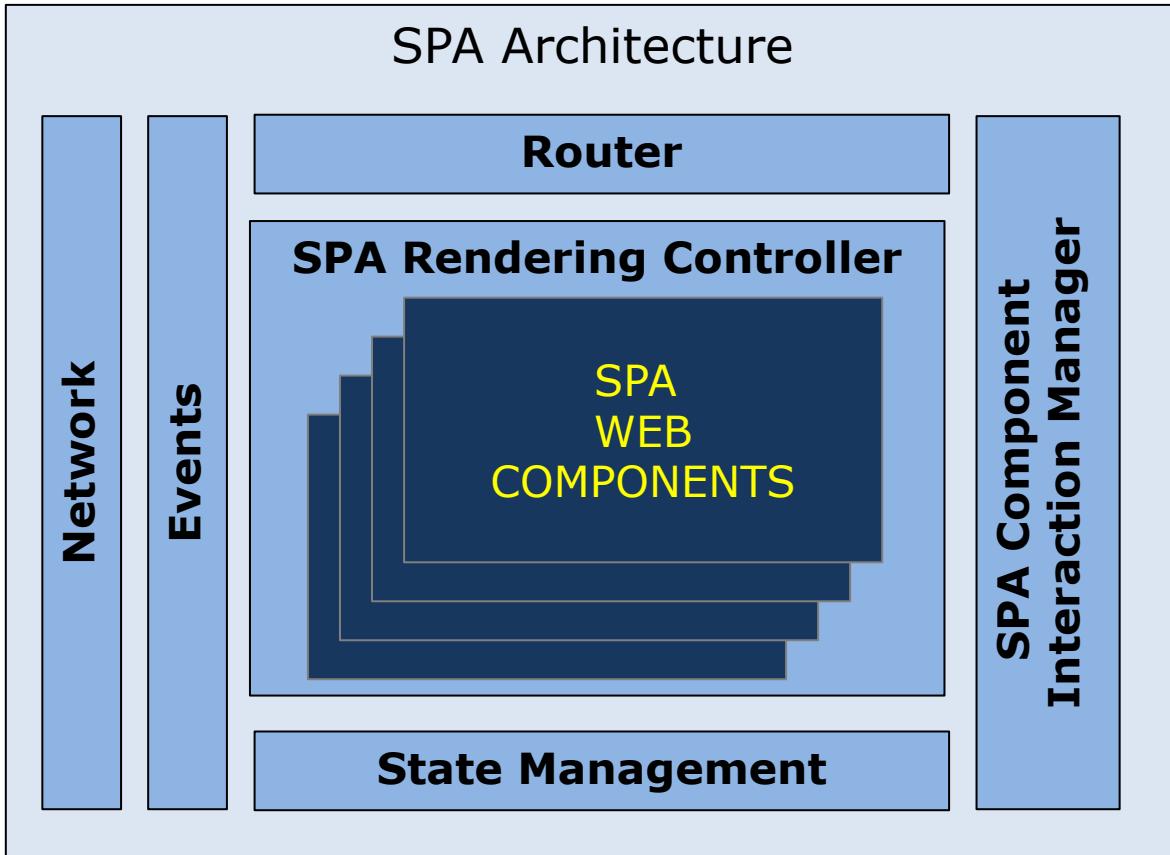
Web Components in an SPA consist of a template for layout, styling for look/feel, Javascript/TypeScript code to manage behavior, and an eventing framework to communicate with other web components

# SPA Frameworks

Building SPAs by hand can be very complicated, over the years multiple frameworks and libraries have been introduced to make this easier. Top frameworks are React, Vue, Angular, and most recently Svelte.



# SPA Frameworks



SPA Components are **event-driven**, almost all interesting interactions happen over sending and reacting to asynchronous events.

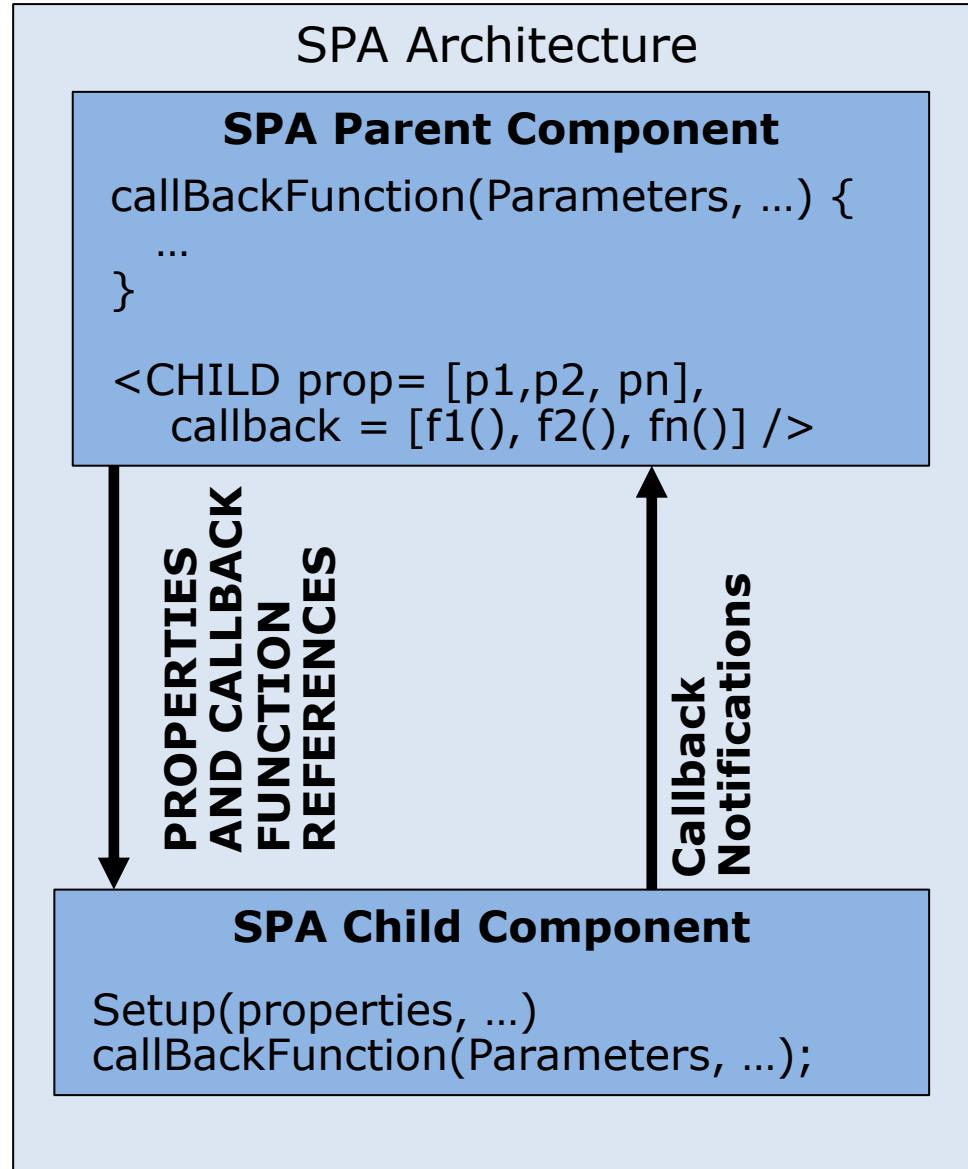
SPA architectures often have abstractions for the **network** interfaces that treat ingress and egress network interactions as events

**State management** for SPAs (discussed soon) is often facilitated via variations to the flux pattern. This pattern features state being managed in an immutable store, and uses events to query, mutate or react to changes in the store

User interface updates are handled entirely locally, they can be full page swaps, which are driven by the **router**, or they can be incremental page updates based on reacting to user behavior (clicking a button, pressing a key, etc)

SPA Web Components, via an **interaction manager**, can communicate with each other via sending events, and reacting to callbacks.

# SPA Component Interaction Best Practices



Remember, SPAs create hierarchies of components

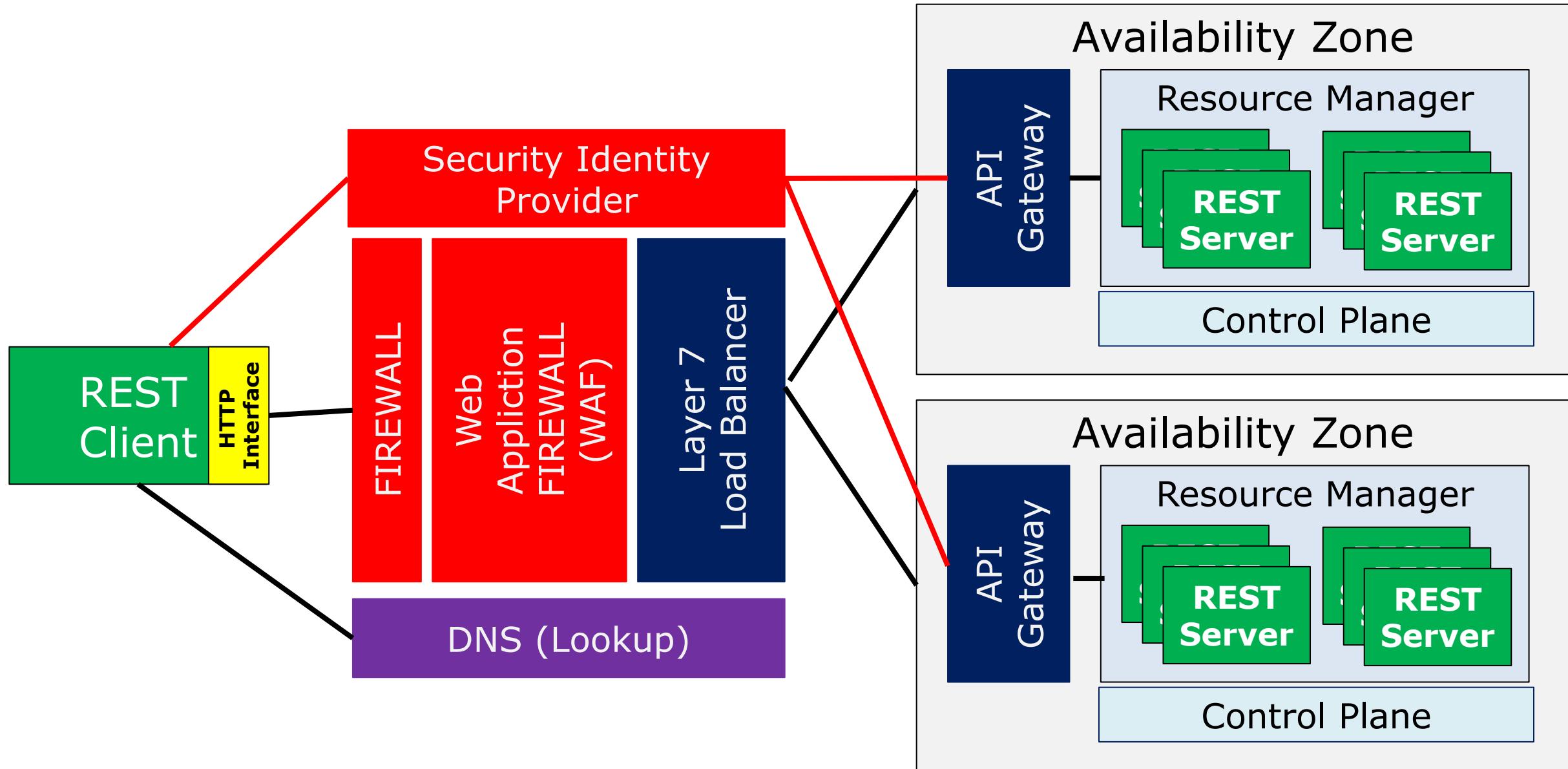
There are strict parent/child relationships

Best practice, when a parent creates a child, it passes all properties that that child needs. These take the form of:

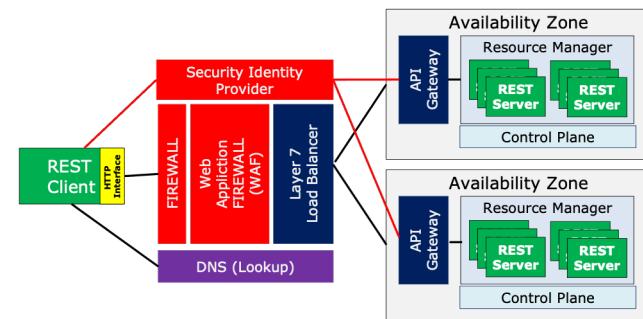
- Properties, or props – think of these as a web component constructor, the parent passes data to the child for initialization
- Callback functions – as the child component runs, sometimes it needs to notify the parent about interesting things, such as state changes or anything else of interest. The parent passes a callback function to the child, that the child can use to interact with the parent

**SPAs are very strict on flow of direction, properties flow down, callbacks / events flow up**

# Reference Architecture for REST

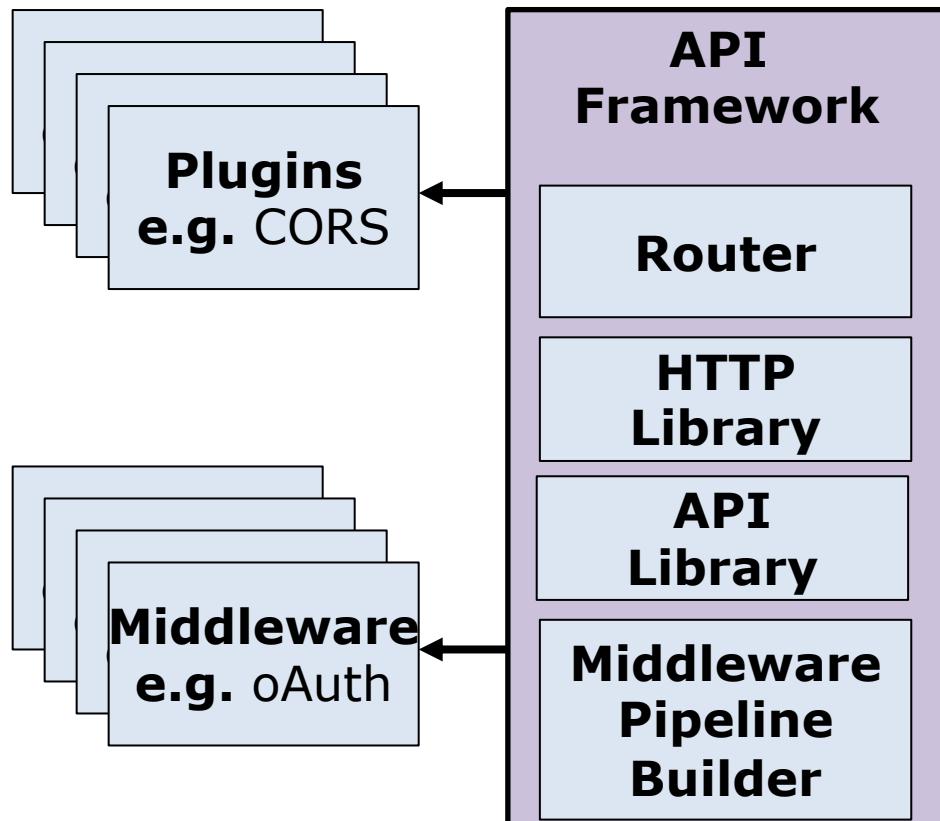


# REST Architecture Components



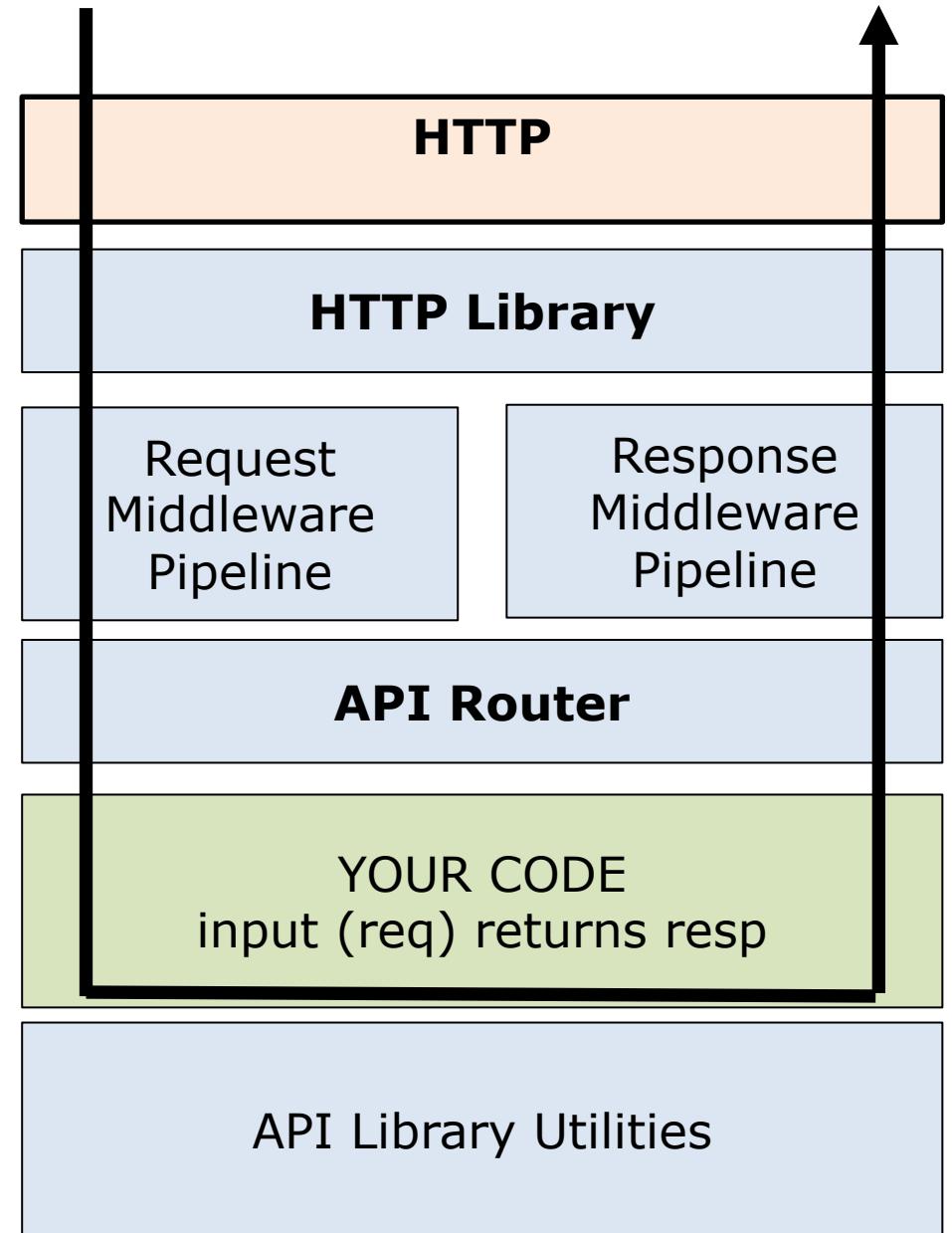
Component	Description
REST Client	Component that makes API calls and receives responses from the REST Server
Firewall	Layer 2/3 firewall that deals with IP addressing, IP Routing and Port Filtering
WAF	Layer 7 (e.g, HTTP aware) firewall that supports protecting against HTTP attacks – SQL Injection, Bots, etc
L7 Load Balancer	Load balancer that distributes load and ensures health of downstream components – eg., the API Gateway. It works at Layer 7 so its optimized for HTTP protocol
Identity Provider (IdP)	A RESTful security endpoint that handles authentication and authorization requests. Typically they issue tokens that can be used to assert trust
DNS	Location services for REST
API Gateway	An intelligent proxy that can load balance REST services, act as a security enforcement point, apply policies around traffic management and shaping, etc
Resource Manager/ Control Plane	An intelligent runtime that supervises running services. It can react to traffic congestion, errors, etc and can scale up/down running instances as needed
REST Server	Component that receives and processes REST Client calls

# API Architecture



Most API Frameworks/Libraries  
Use a Repository Style Architecture

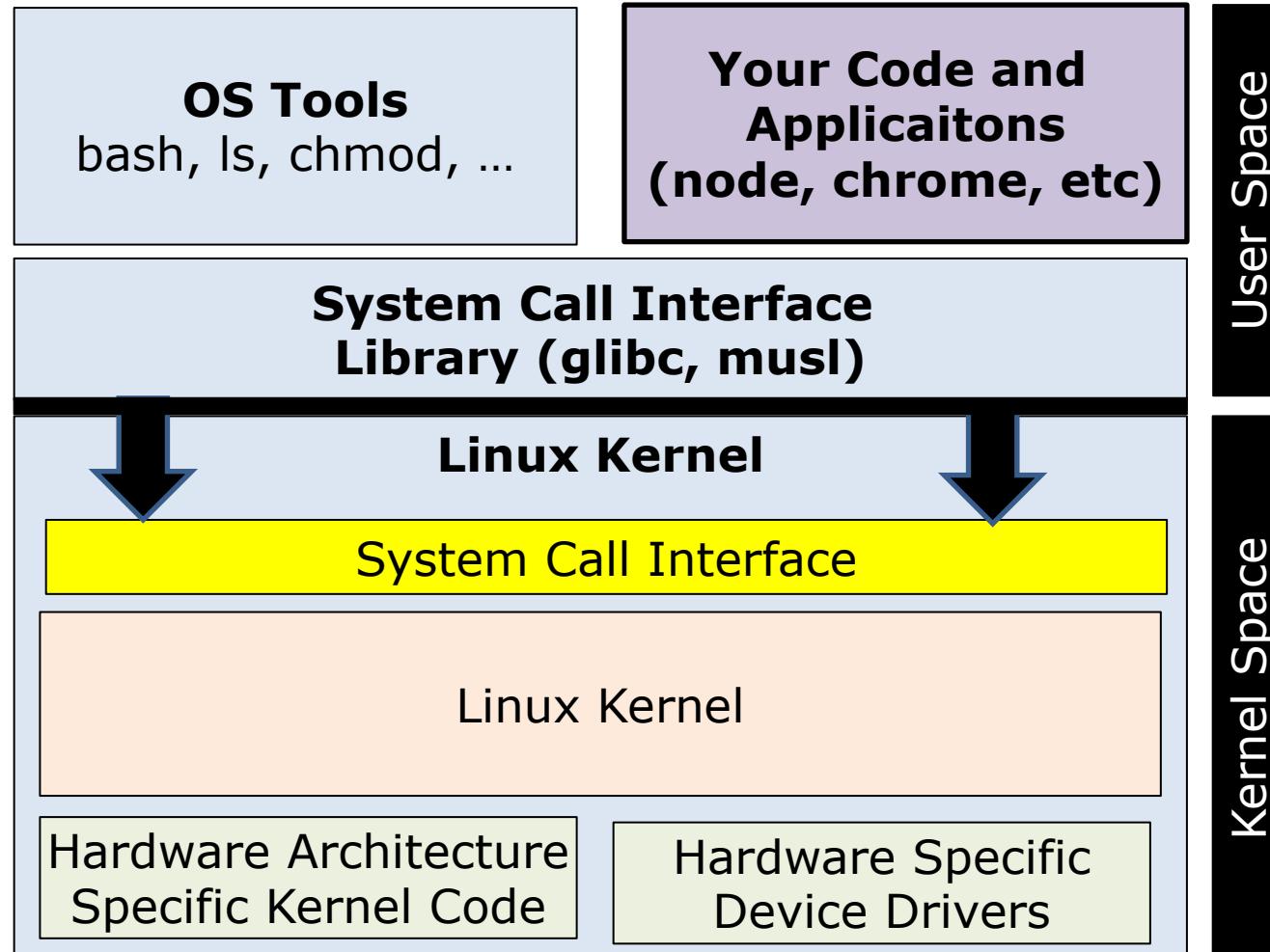
We have used several in this course  
Koa, Fastify, GoLang Gin, Ktor, etc



# REST Frameworks We Looked At, or I used in my web services REPO – you can google them

Language	Framework
Javascript/Typescript	Fastify
Javascript/Typescript	Koa
Go	Gin
Kotlin	Ktor
Rust	actix

# Linux Architecture



Linux is a layered architecture, the kernel sits close to the hardware and even has some hardware specific code (e.g., x-64 vs ARM).

The code we run, tools, custom code, etc sits at the top layer

User code leverages the system call interface to bridge between user space and kernel space

When you pick a Linux distribution, they basically provide the stuff in the blue boxes

Note the actual linux architecture is significantly more complex, but this should suffice for our needs

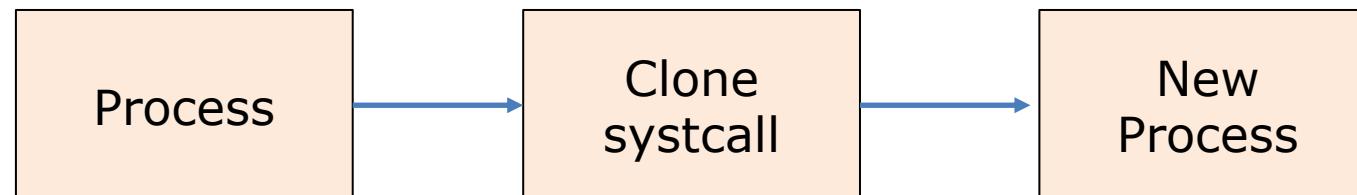
# Linux Namespaces

When linux boots, it synthesizes process Id (pid) 1 - its called the `init` process, often implemented by the `systemd` process. This process is used to spawn all other processes in the system.

One way linux can create a child process is via the `clone` syscall. By default the new child process runs in the same execution context as the parent. **Processes in the same execution context share a common linux namespace.**

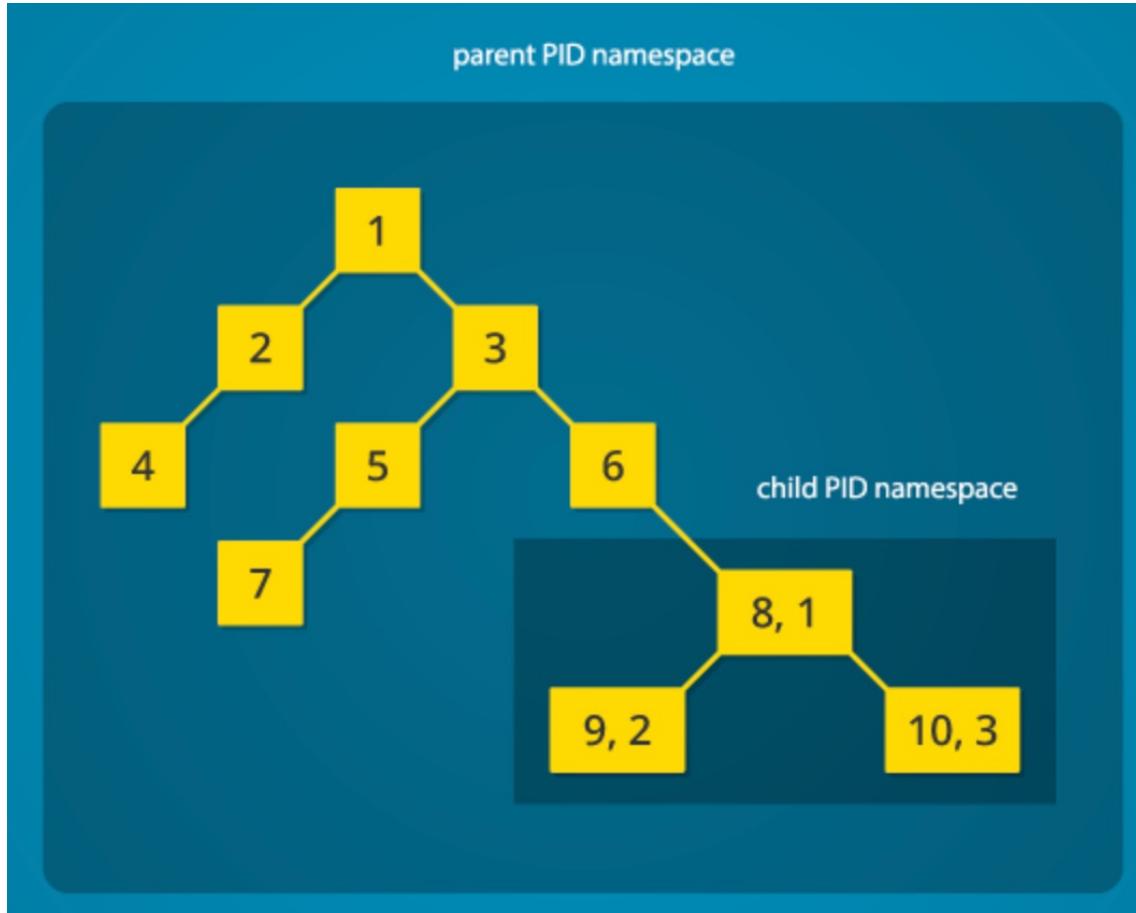
The `clone` system call also provides options to allow the child to be put in separate namespaces.

Linux currently supports 7 different namespaces – CGroup, IPC, Network, Mount (filesystem), PID, Time, User (user permissions and groups), UTS



**Processes that have attributes that run in a common namespace are not isolated, processes with different namespaces are**

# Linux Namespaces – Example, cloning a child process into a new process/pid name space



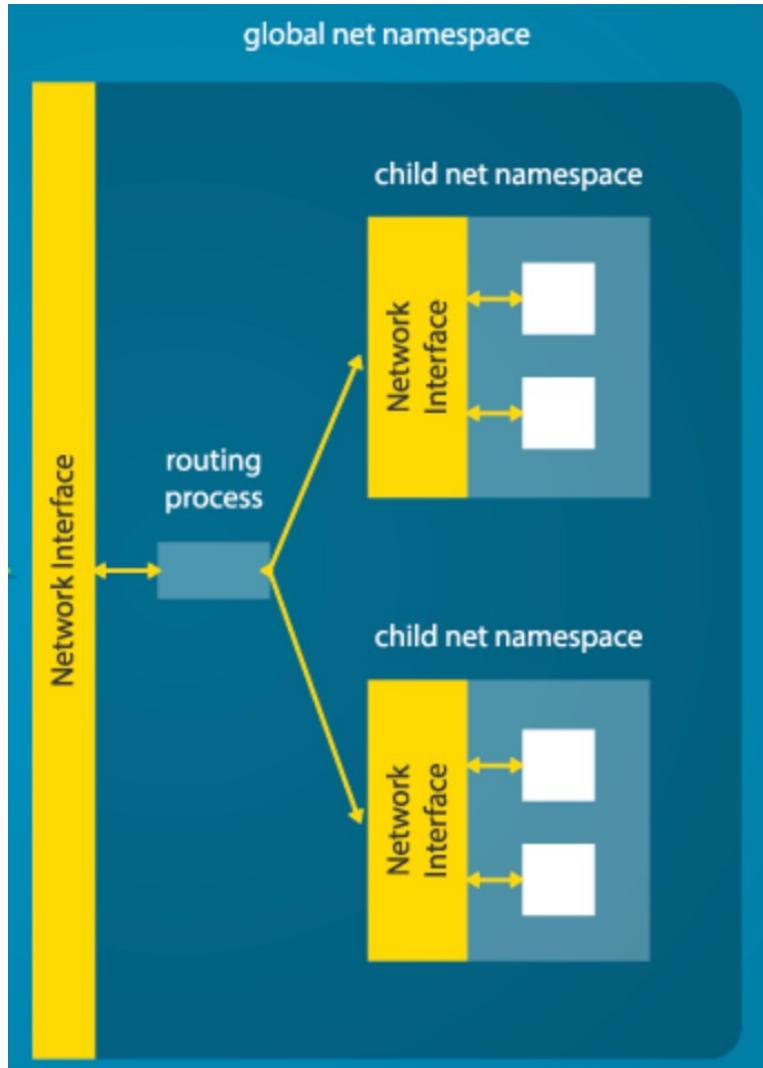
Ultimately every process is given a unique id in the guest operating system. Thus if you listed all processes in the system you would see PID 1-10.

However look at pid 6 it cloned PID8, and PID8 cloned both PID9 and PID10

The cool thing is that if you asked PID 8 what its process ID is it would come back as 1; Same for pid9 and 10, they would come back as 2 and 3.

**Thus everything in the child namespace can see each other, but it cannot see anything in the parent namespace.**

# Linux Namespaces – Example, cloning a child process into a new network namespace



Ultimately every process is given a unique id in the guest operating system. Thus if you listed all processes in the system you would see PID 1-10.

However look at pid 6 it cloned PID8, and PID8 cloned both PID9 and PID10

The cool thing is that if you asked PID 8 what its process ID is it would come back as 1; Same for pid9 and 10, they would come back as 2 and 3.

**Namespaces can see each other, but it cannot see anything in the parent**

# Linux Namespaces – allow for logical isolation in linux

## Namespace types

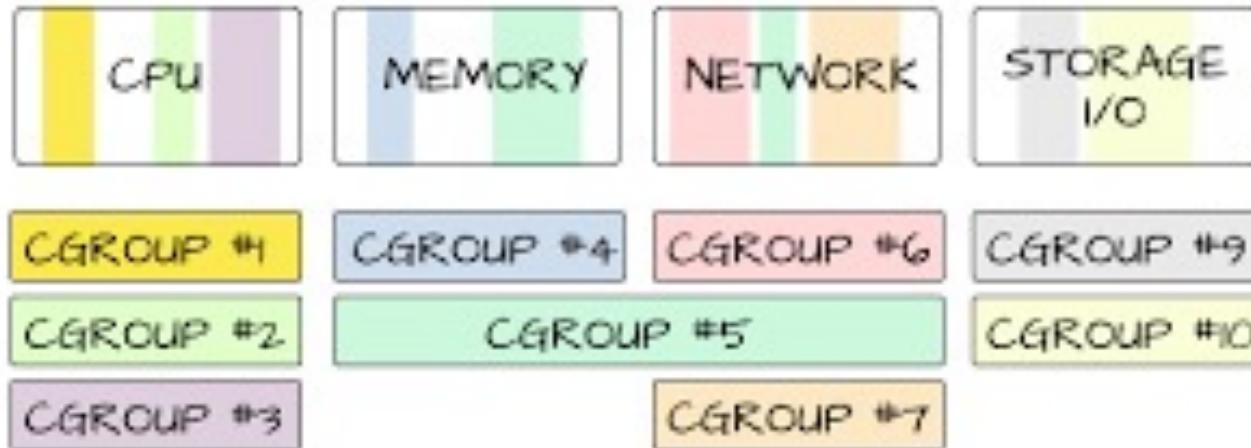
The following table shows the namespace types available on Linux. The second column of the table shows the flag value that is used to specify the namespace type in various APIs. The third column identifies the manual page that provides details on the namespace type. The last column is a summary of the resources that are isolated by the namespace type.

Namespace	Flag	Page	Isolates
Cgroup	<code>CLONE_NEWCGROUP</code>	<code>cgroup_namespaces(7)</code>	Cgroup root directory
IPC	<code>CLONE_NEWIPC</code>	<code>ipc_namespaces(7)</code>	System V IPC, POSIX message queues
Network	<code>CLONE_NEWWNET</code>	<code>network_namespaces(7)</code>	Network devices, stacks, ports, etc.
Mount	<code>CLONE_NEWNS</code>	<code>mount_namespaces(7)</code>	Mount points
PID	<code>CLONE_NEWPID</code>	<code>pid_namespaces(7)</code>	Process IDs
Time	<code>CLONE_NEWTIME</code>	<code>time_namespaces(7)</code>	Boot and monotonic clocks
User	<code>CLONE_NEWUSER</code>	<code>user_namespaces(7)</code>	User and group IDs
UTS	<code>CLONE_NEWUTS</code>	<code>uts_namespaces(7)</code>	Hostname and NIS domain name

If you clone a child process into a new namespace for all namespace types, it will for all practical purposes be fully isolated from the parent.

Also, if the isolated process creates new processes it will inherit by default the same namespace as the parent thus allowing process groups to be isolated

# Linux Cgroups – managing resource limits on collections of processes - /sys/fs/cgroup/...



Cgroups allow resource limits to be attached to a collection of processes.

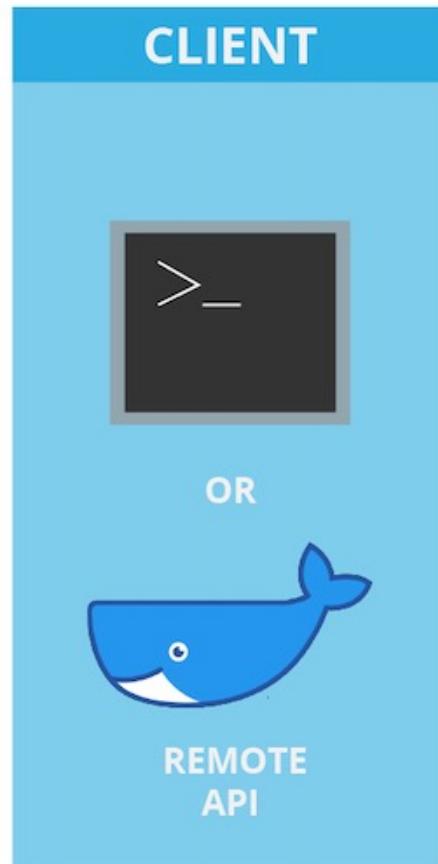
For example you might want to restrict how much network and file I/O can be used and also cap CPU at ½ of a core

```
resources:  
  requests:  
    memory: "64Mi"  
    cpu: "250m"  
  limits:  
    memory: "128Mi"  
    cpu: "500m"
```

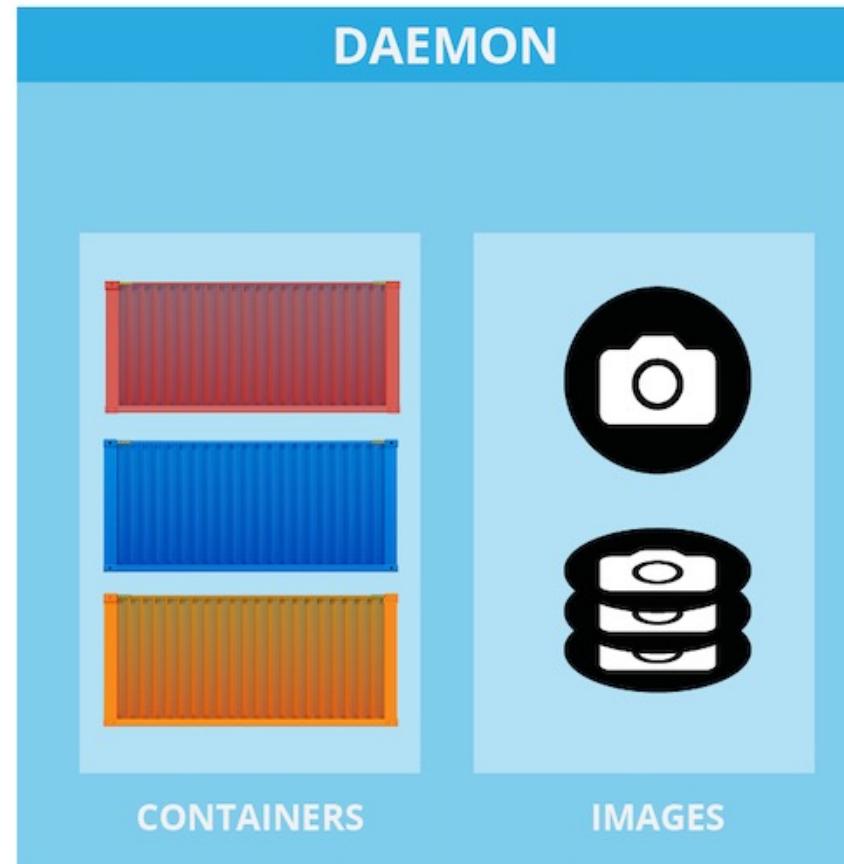
In this example we are requesting an initial allocation of 64Mb of memory and 250 millicores (1/4 of a physical core), and allowing it to burst up to the limit of 128Mb of memory and 500 millicores (1/2 of a physical core)

# Docker

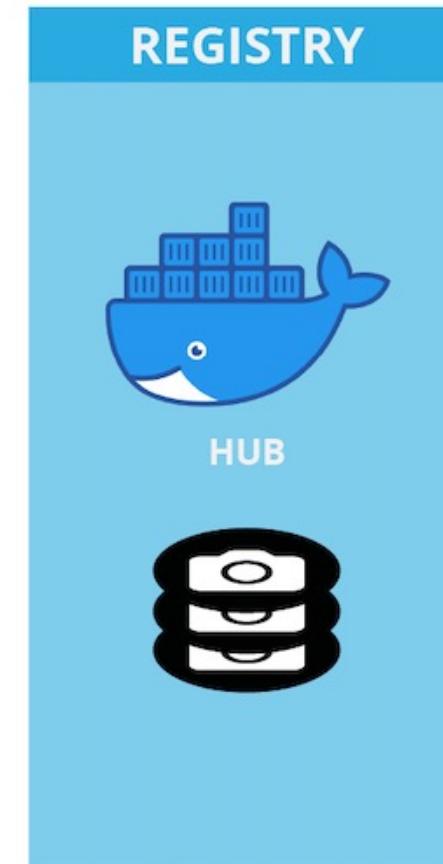
Now that we know how linux can isolate process groups (namespaces) and resources (cgroups) we can better understand the docker architecture



Managing Docker



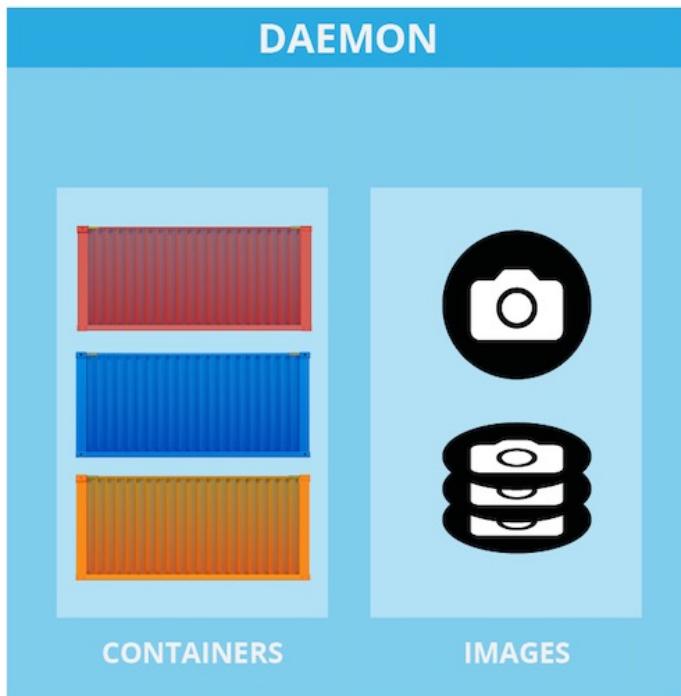
Running Containers



Getting Containers

# Docker Daemon and Linux

On windows and Mac the docker daemon is required and runs a customized, container optimized version of linux in a hypervisor. On linux machines the daemon does not use a hypervisor because it has access to the linux kernel



General Purpose Linux Distributions



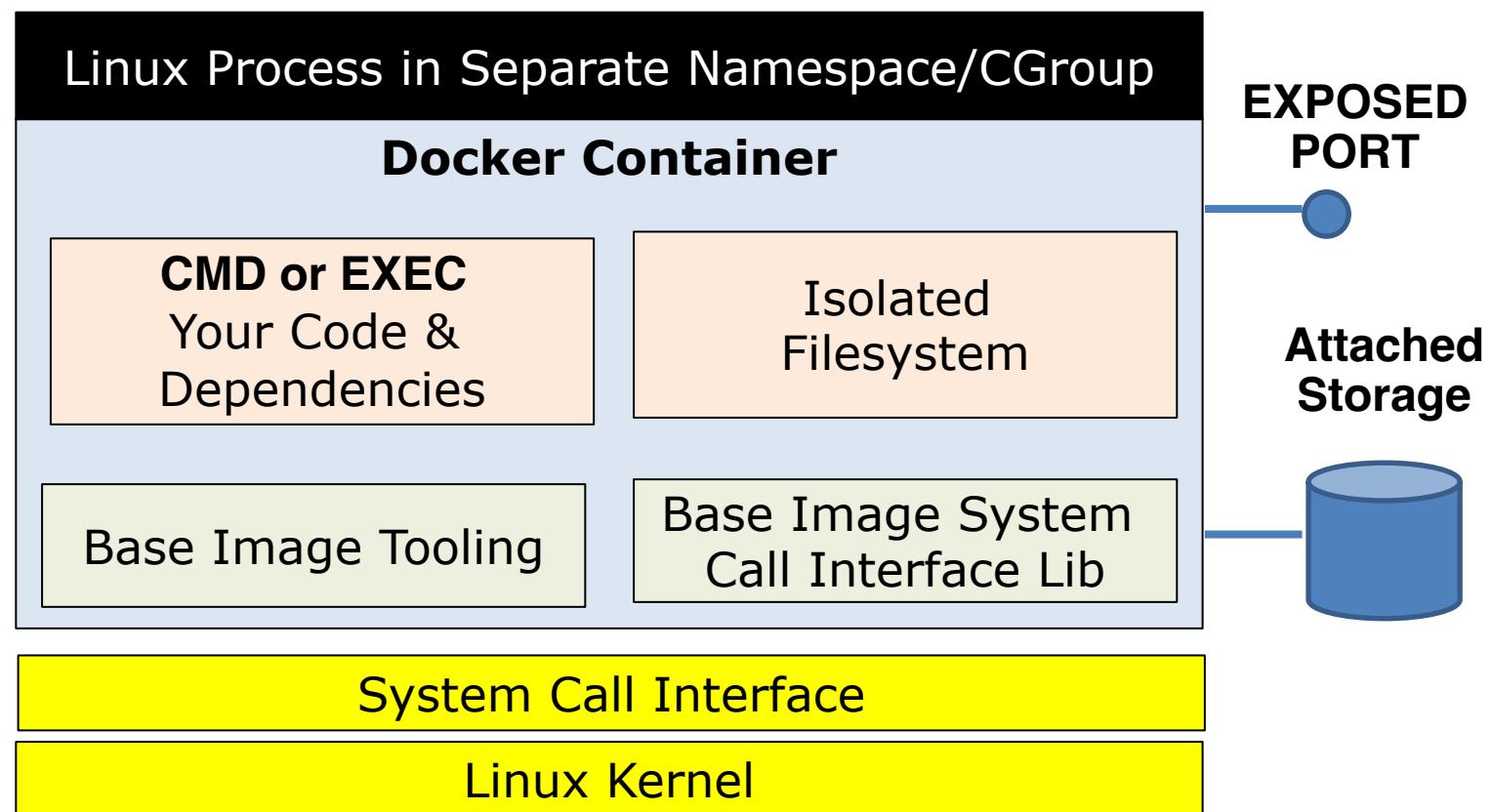
Container Optimized Linux Distributions

# The docker container

The docker container is a portable package containing everything needed to run your workload. It adheres to the open container initiative specification  
<https://opencontainers.org/>



The shipping container transformed logistics



# Building Docker Containers

```
FROM nginx:alpine

COPY --chown=nginx:nginx ..../dist/spa /usr/share/nginx/html
RUN chmod g+rwx /var/cache/nginx /var/run /var/log/nginx

RUN sed -i.bak 's/listen\(.*\)80;/listen 9080;/' /etc/nginx/conf.d/default.conf

USER nginx

EXPOSE 9080
```

## Docker Container

Create Ephemeral Writable Layer

Expose port 9080

Set the user to nginx

Run script to set default port for nginx to 9080

Copy SPA code into /usr/share/nginx/html + change owner to nginx

Alpine-Based OS and Tools with Nginx Installed

# Building Docker Containers

```
docker build -t architecting-software/se577-demo-app -f Dockerfile .
```

## Docker Container

Create Ephemeral Writable Layer

Expose port 9080

Set the user to nginx

Run script to set default port for nginx to 9080

Copy SPA code into /usr/share/nginx/html + change owner to nginx

Alpine-Based OS and Tools with Nginx Installed

```
docker build --squash -t architecting-software/se577-demo-app -f Dockerfile .
```

## Docker Container

Create Ephemeral Writable Layer

Have nginx running running and exposing port 9080, with our code in the /usr/share/nginx/html directory with file ownership of nginx

# Building Docker Containers

Its not uncommon to create a build process where you create a build container first and then use the output of that build container to create your final container

```
FROM golang:latest as builder
LABEL stage=builder
#PHASE 1: Build the GO Binary - aka builder container
WORKDIR /go/src/drexel.edu/bc-service/go
COPY go* .
COPY src ./src
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -v -o bin/bbservice ./src
```

```
#PHASE2: Build the final container, notice the directory naming follows the package prefix in go.mod
FROM alpine
WORKDIR /bin/
COPY --from=builder /go/src/drexel.edu/bc-service/go/bin ./
ENTRYPOINT ["/bin/bbservice"]
EXPOSE 9095
```

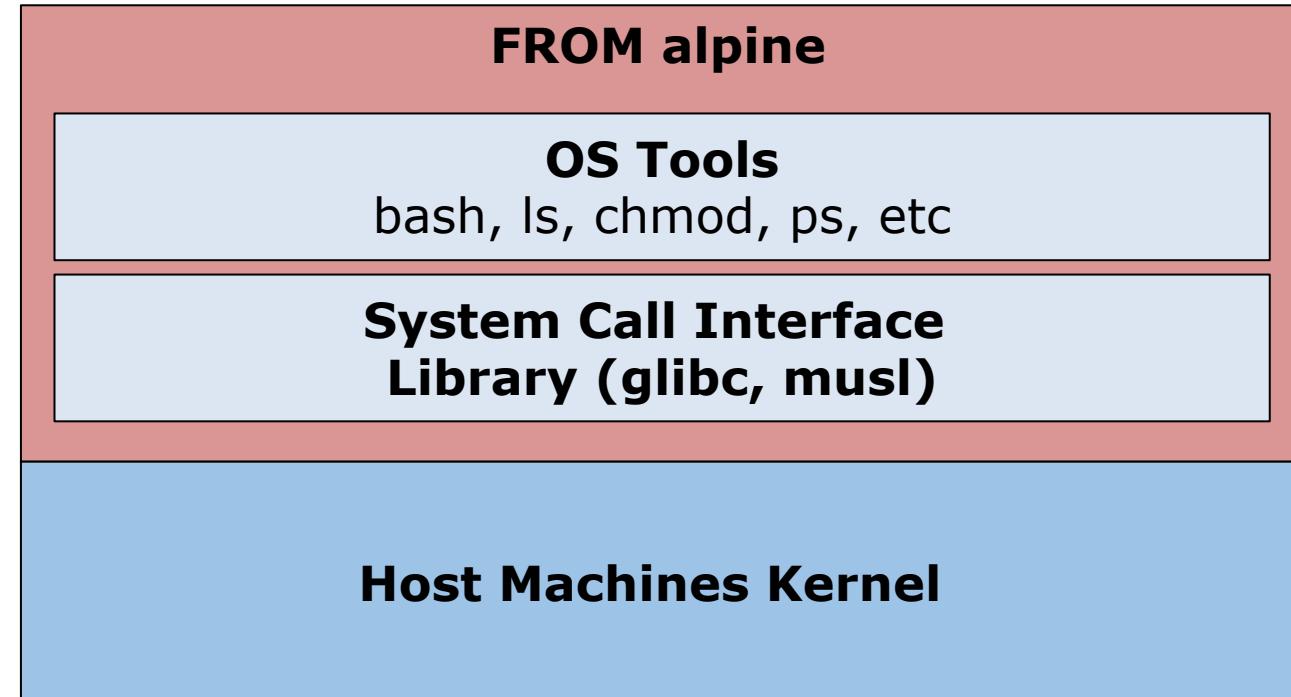
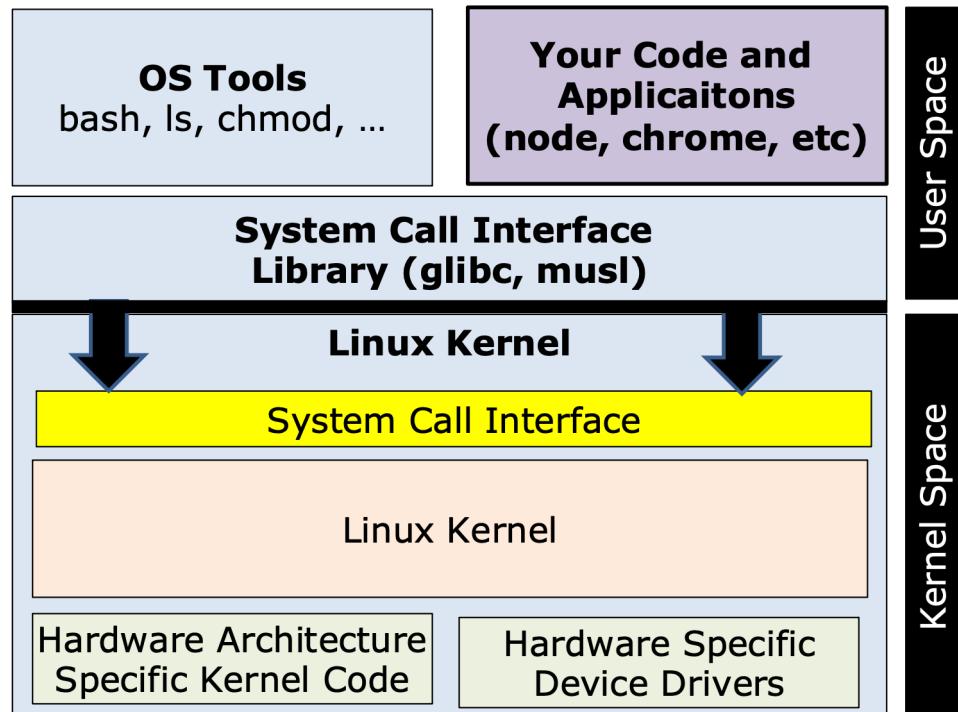
Build Container

Run Container

# What OS Kernel does a container use?

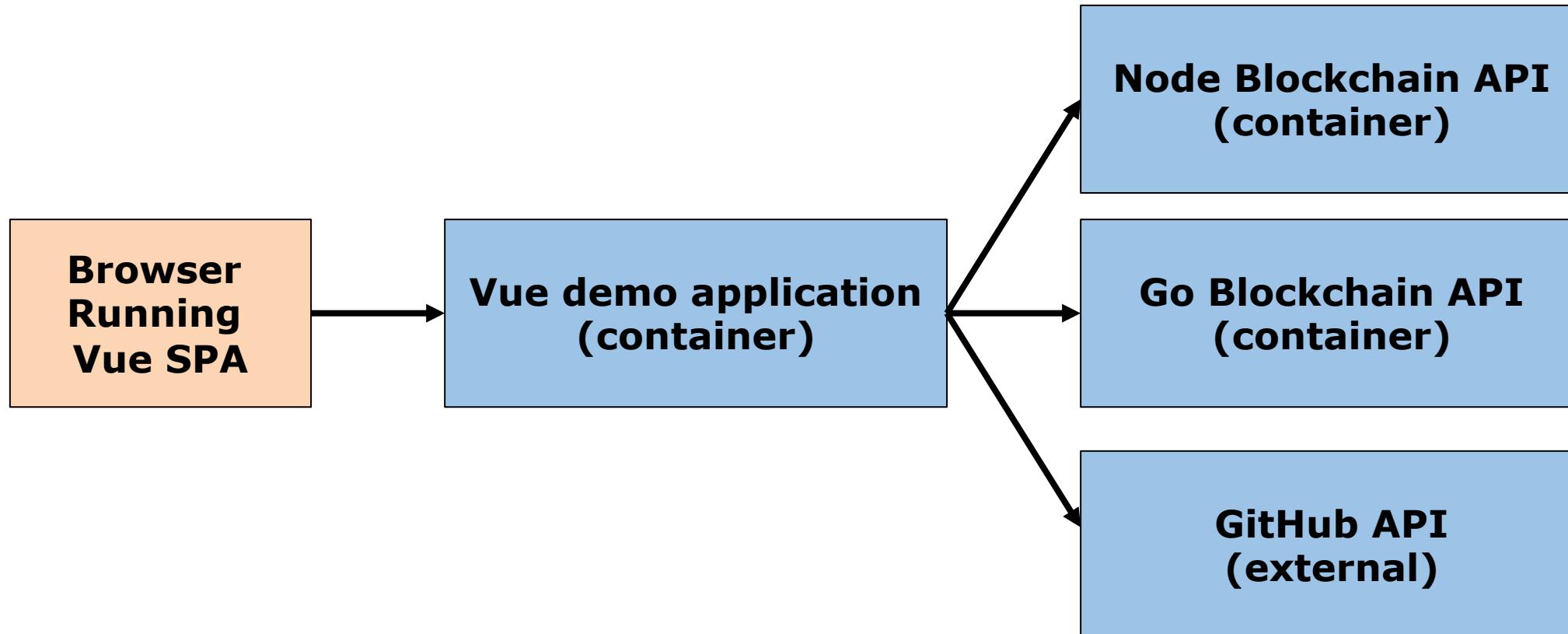
Note the base container is created from an OS image in this example

```
#PHASE2: Build the final container, notice the directory naming follows the package prefix in go.mod
FROM alpine
WORKDIR /bin/
COPY --from=builder /go/src/drexel.edu/bc-service/go/bin .
ENTRYPOINT ["/bin/bcservice"]
EXPOSE 9095
```



# Architecting for containers

Generally containers run one process, and are expected to do one thing. Thus we need a way for containers to work together to build useful architectures



# Container Repositories

**After containers are created, they are generally pushed to a container repository. These could be public or private**

The screenshot shows the Docker Hub interface. At the top, there's a navigation bar with links like 'Explore', 'Repositories', 'Organizations', and 'Help'. Below the navigation is a search bar with placeholder text 'Search for great content (e.g., mysql)'. A dropdown menu shows the current namespace 'architectingsoftware'. There are two repository cards displayed:

- architectingsoftware / bcservcego**  
Last pushed: 2 years ago  
Not Scanned | 0 stars | 11 downloads | Public
- architectingsoftware / node-pubs-test**  
Last pushed: 2 years ago  
Not Scanned | 0 stars | 13 downloads | Public

A tip at the bottom left suggests switching namespaces via the dropdown.

**Manage Container Versions**

**Push Containers on Build**

**Pull Containers When Needed**

# Container Repositories

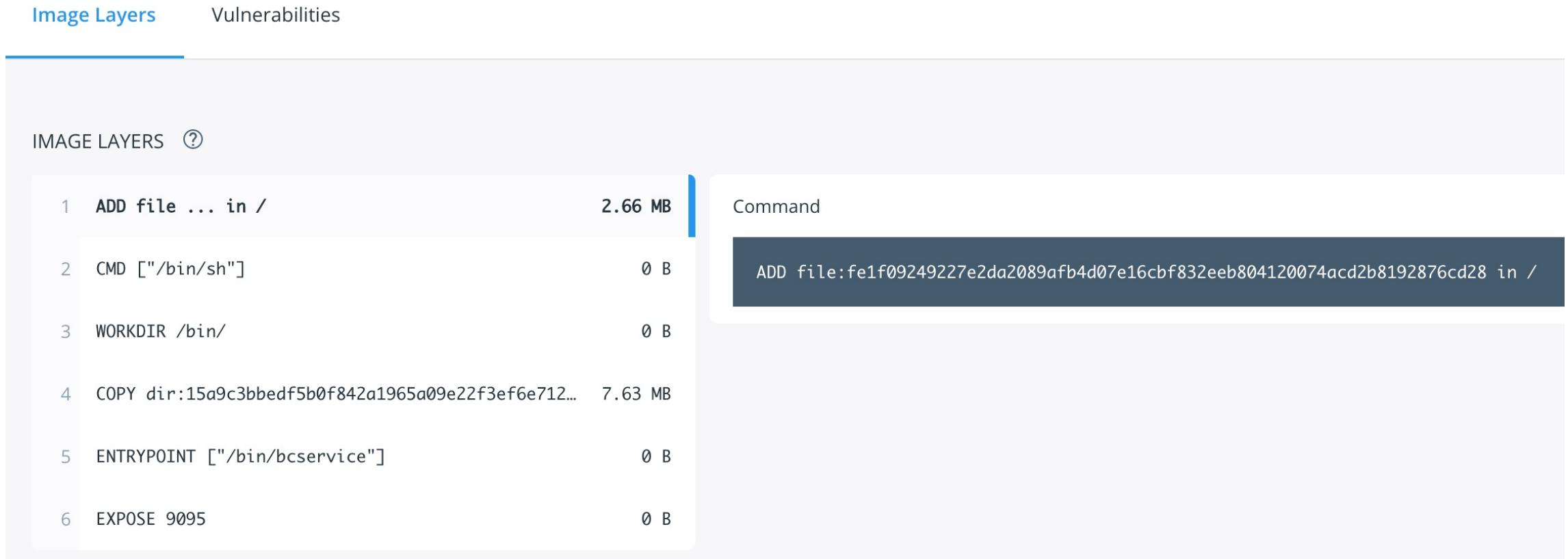
**Container repositories also keep track of the container layers and can do other things like scan for security vulnerabilities**

Image Layers      Vulnerabilities

---

IMAGE LAYERS [?](#)

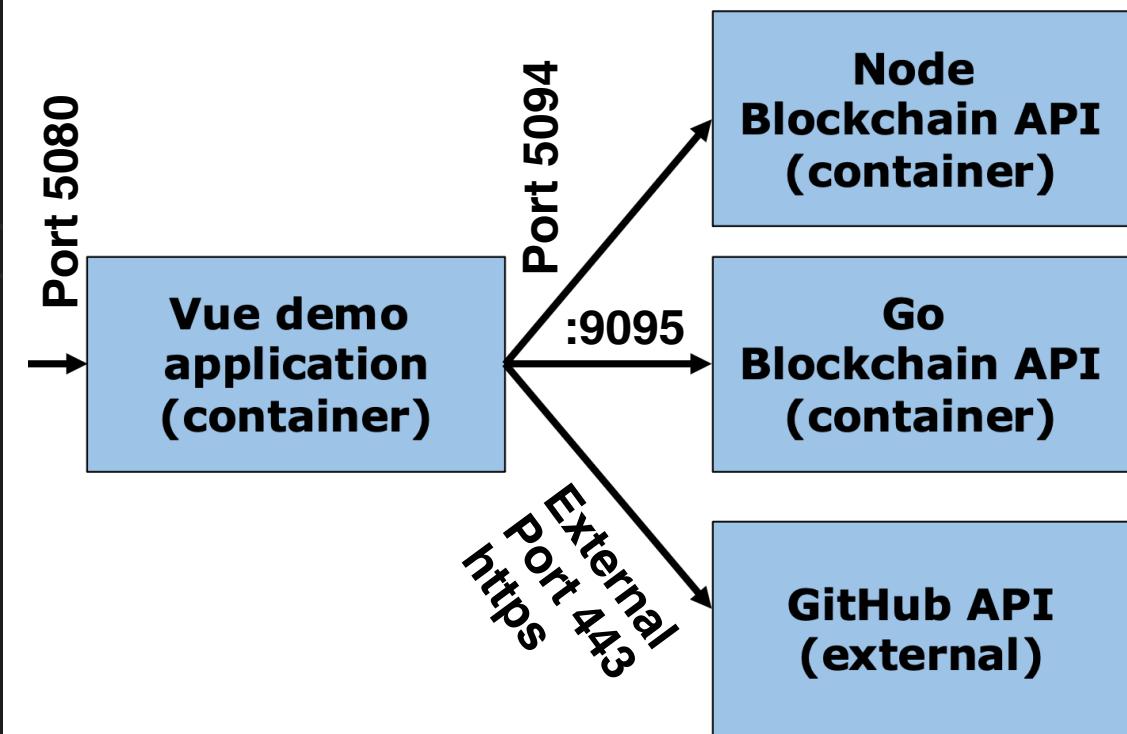
1 ADD file ... in /	2.66 MB	Command
2 CMD ["/bin/sh"]	0 B	ADD file:fe1f09249227e2da2089afb4d07e16cbf832eeb804120074acd2b8192876cd28 in /
3 WORKDIR /bin/	0 B	
4 COPY dir:15a9c3bbdef5b0f842a1965a09e22f3ef6e712... 7.63 MB		
5 ENTRYPOINT ["/bin/bcservice"]	0 B	
6 EXPOSE 9095	0 B	

A screenshot of a web-based Docker image viewer. At the top, there are two tabs: 'Image Layers' (which is active and highlighted in blue) and 'Vulnerabilities'. Below the tabs, the heading 'IMAGE LAYERS' is followed by a question mark icon. A horizontal line separates this from the main content area. The content area displays a table of six image layers. The first layer, which has a blue vertical bar next to it, is expanded to show its command: 'ADD file:fe1f09249227e2da2089afb4d07e16cbf832eeb804120074acd2b8192876cd28 in /'. This expanded layer is enclosed in a dark grey box. The other five layers are shown in their collapsed state.

# Using Docker-Compose

Generally containers run one process, and are expected to do one thing. Thus we need a way for containers to work together to build useful architectures

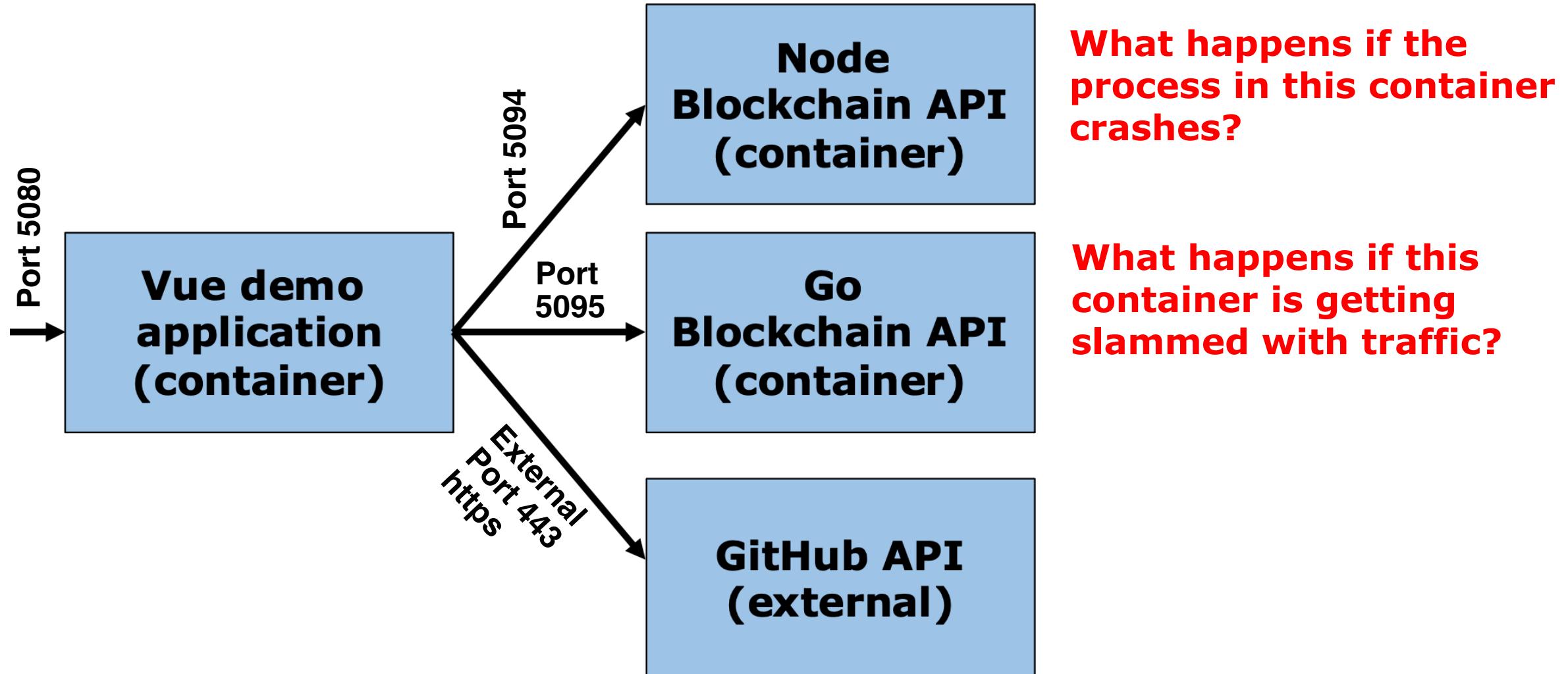
```
version: '3.8'
services:
  gowebservice:
    image: 'architecting-software/bc-service-go'
    ports:
      - '5095:9095'
  nodewebservice:
    image: 'architecting-software/bc-service-node-fastify'
    ports:
      - '5094:9094'
  webserver:
    image: 'architecting-software/se577-demo-app'
    ports:
      - '5080:9080'
```



Docker compose allows us to bring up collections of containers via docker-compose up, and docker-compose down

# Architectural Issues with Docker-Compose

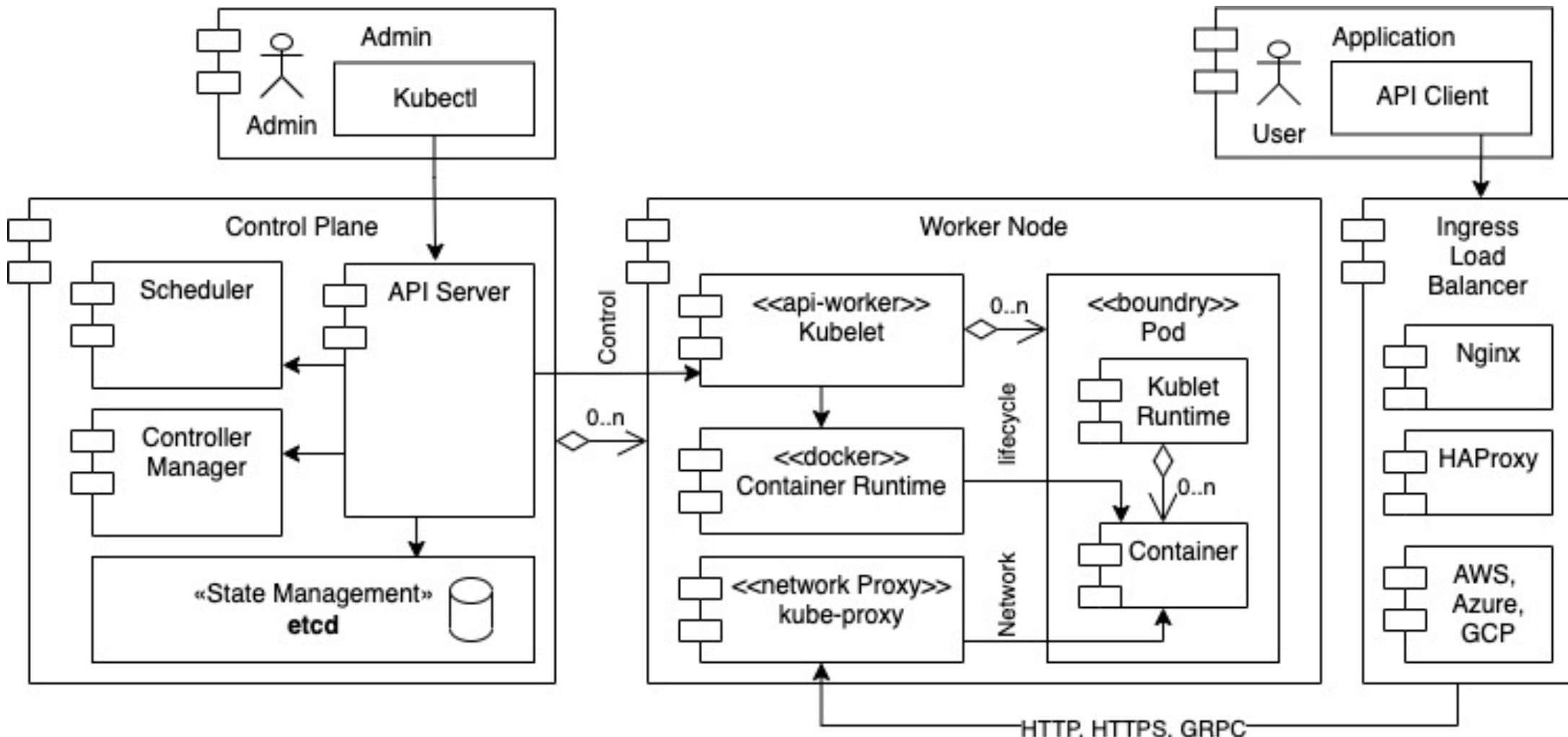
Docker compose does not supervise the running environment



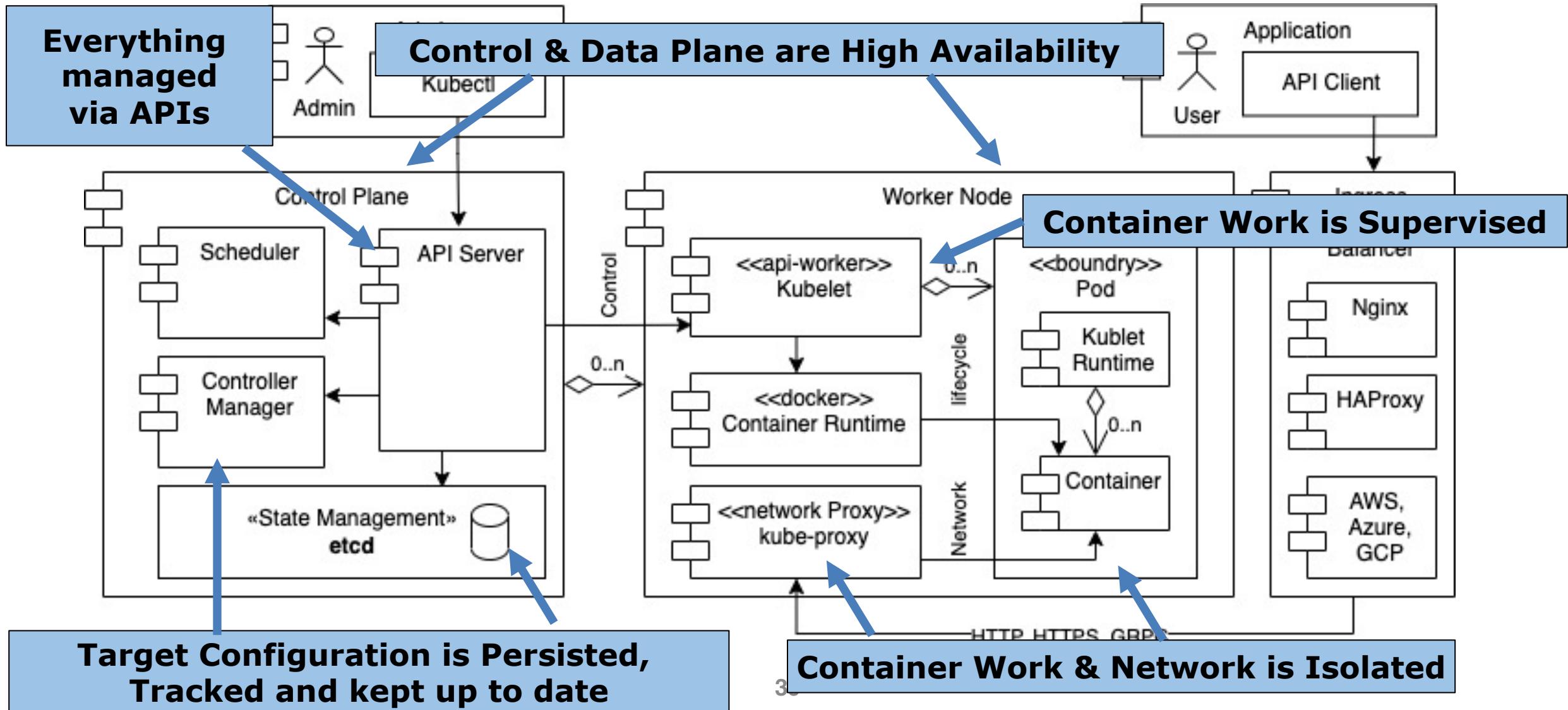
# Running Containers at Scale In Production

- Containers are ephemeral – this is a big deal – every time its run it has the same starting state
- Containers can be distributed via container repositories and inherit other benefits like ensuring the proper containers are used, they are secure, etc
- But containers alone are not enough:
  - They need to be “supervised” – are they healthy, have they crashed, do they need to be restarted?
  - They need to be distributed to prevent against issues when their runtime crashes
  - They need to be aware of load so they can autoscale up, and autoscale down
  - They need to be isolated, so only container workloads that interact with each other can interact with each other

# Running Containers at Scale In Production – Kubernetes or k8s

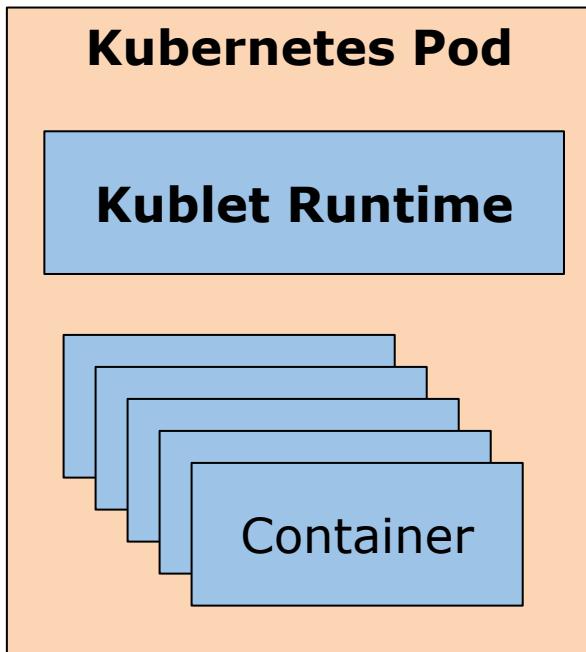


# Running Containers at Scale In Production – Kubernetes or k8s

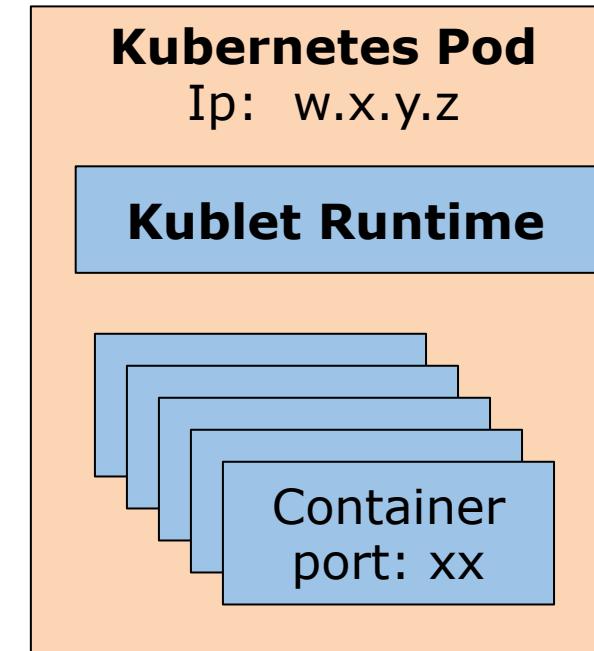


# Running Containers at Scale In Production – Kubernetes or k8s - Pods

**Most of the time pods run one container, the kubelet manages the pod runtime**



**Namespaces are used to govern what is shared between containers in the “pod” such as IPC, and things across the Kubernetes cluster such as network**



**Each k8s pod is given a routable IP address  
Each container in the pod can be reached via a port on that IP address**

# Running Containers at Scale In Production – Kubernetes or k8s – Core K8s Objects

Managing Kubernetes can be complex, but you can get by with just a few of the k8s objects

