PID 1, SIG Handling, Hooks & Probes: Managing Container Lifecycle Correctly

Anmol Krishan Sachdeva

About Me

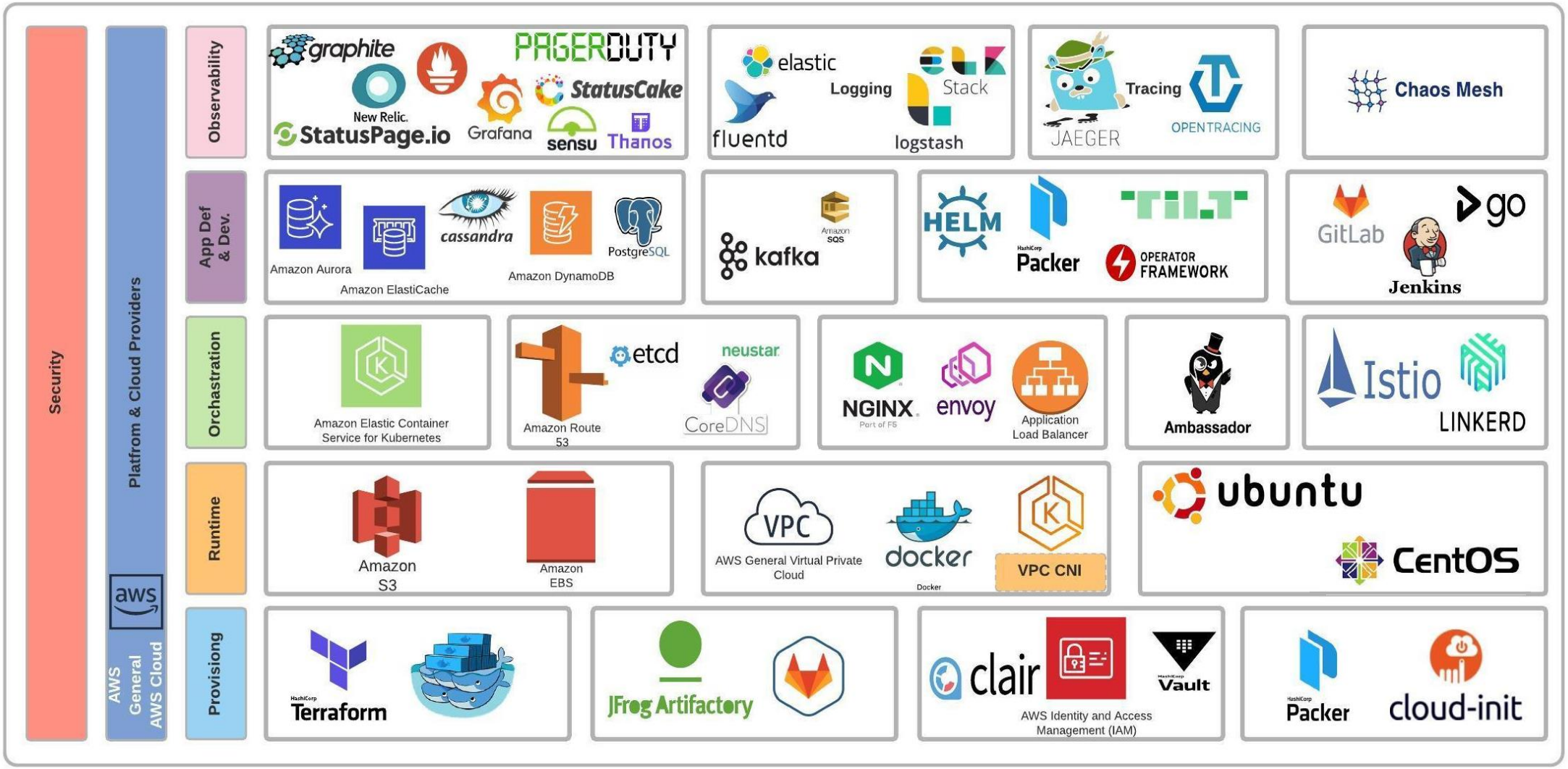
Anmol Krishan Sachdeva
Site Reliability Engineer, OLX Group
MSc Advanced Computing
University of Bristol, United Kingdom
LinkedIn: greatdevaks
Twitter: @greatdevaks

- International Tech Speaker
- Distinguished Guest Lecturer
- Represented India at Reputed International Hackathons
- Deep Learning Researcher
- 8+ International Publications
- ALL STACK DEVELOPER
- Mentor

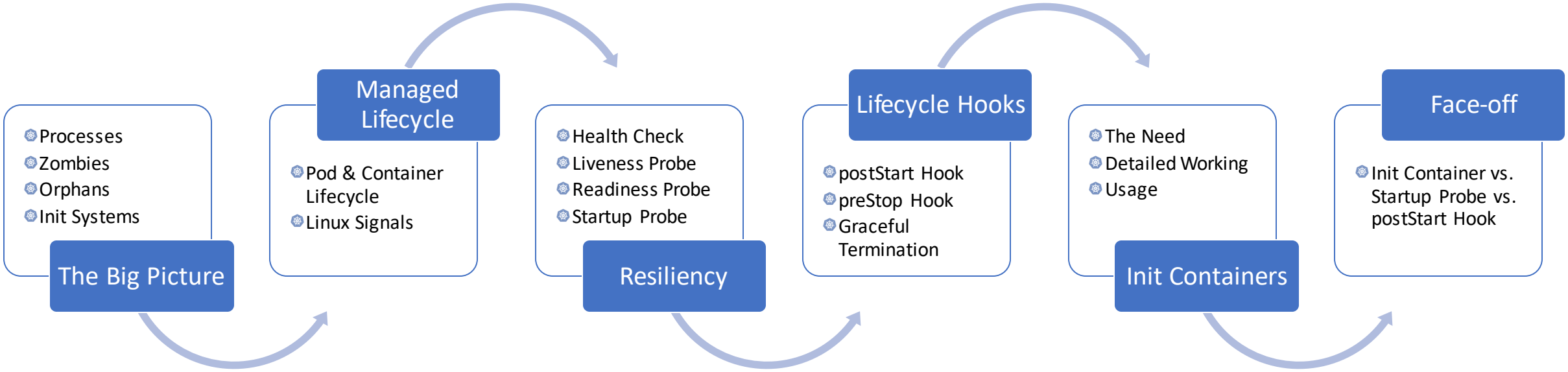
About OLX Group

- Global product and tech group (20+ brands)
- Online buying, selling, and exchange of products and services
- Serving approx. 350 million people per month
- Operating in about 45 countries across 5 continents
- More than 10 million online listings every single month
- Billions of Edge Hits per day
- Hundreds of Thousands of Cache reads per second
- Hundreds of Microservices

Infrastructure Landscape



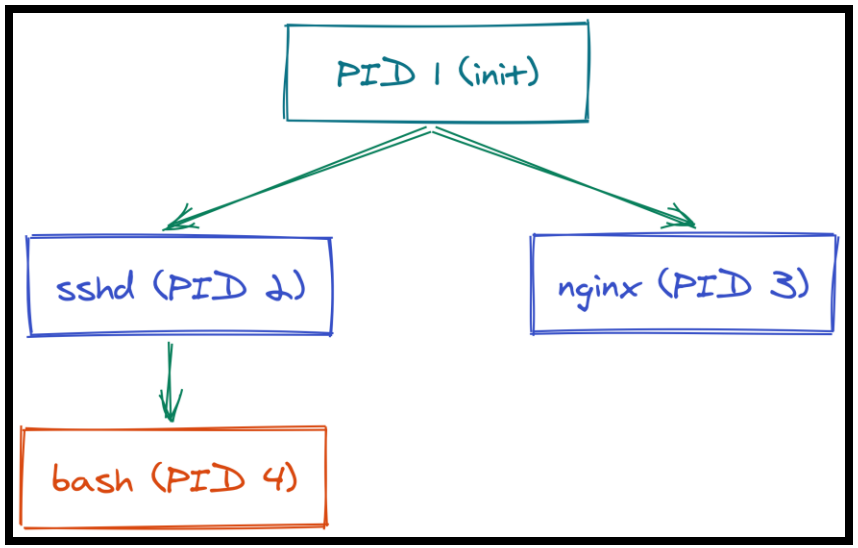
Agenda


The Big Picture
Processes
Zombies
Orphans
Init Systems
Managed Lifecycle
Pod & Container Lifecycle
Linux Signals
Resiliency
Health Check
Liveness Probe
Readiness Probe
Startup Probe
Lifecycle Hooks
postStart Hook
preStop Hook
Graceful Termination
Init Containers
The Need
Detailed Working
Usage
Face-off
Init Container vs. Startup Probe vs. postStart Hook

The Big Picture

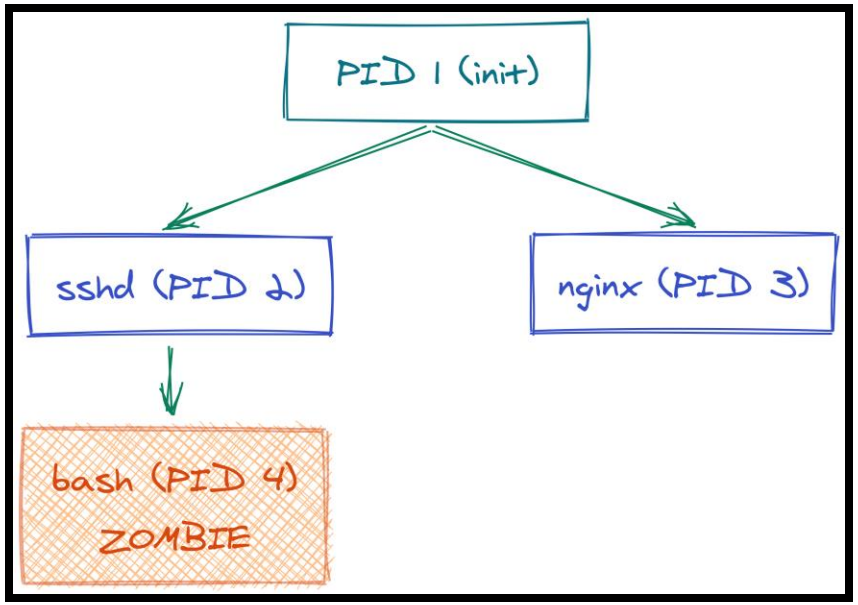
Processes, Zombies, Orphans, and Init Systems

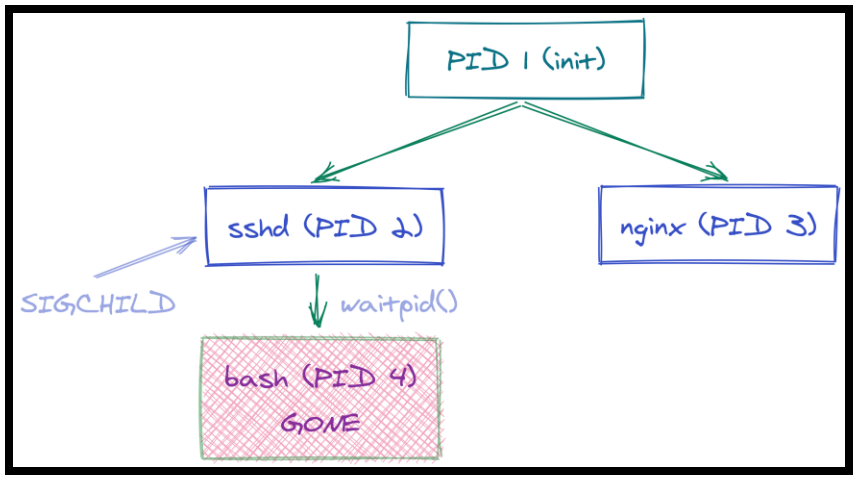
Unix Processes



- Instance of a running application
- Ordered in form of a tree
- Each process can spawn several child processes
- Each process has a parent, except for the top-most process ***(init / PID 1)***
- PID 1 is started by the kernel
- PID 1 acts as a parent process and starts the rest of the system and processes

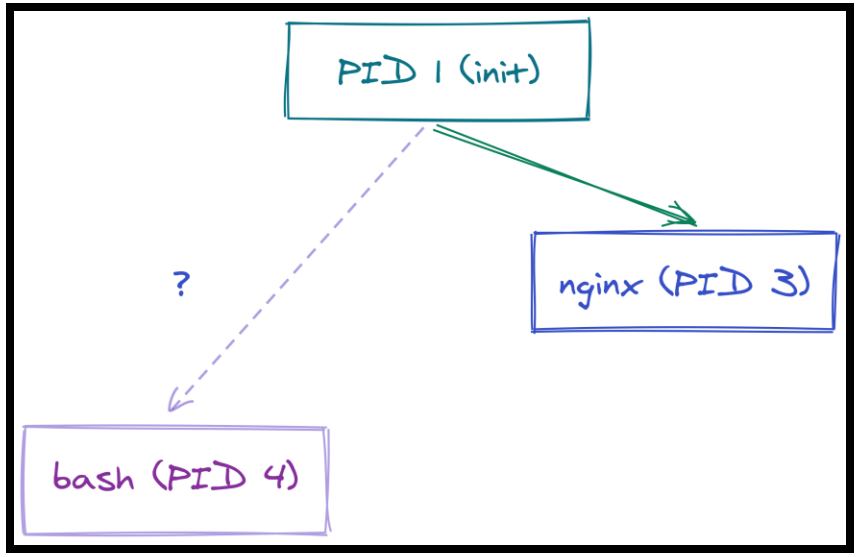
Zombies





- ***Defunct / Zombie*** processes?
- ***waitpid()*** system call
- ***Reaping?***
- ***SIGCHILD*** signal
- *ZOMBIES ARE THE PROCESSES THAT HAVE TERMINATED BUT HAVE NOT YET BEEN WAITED FOR BY THEIR PARENT PROCESSES*

Orphans



- What if the parent process terminates somehow?
- What happens to its children? ***Orphaned?***
- Time for PID 1 to take over? Adoption?
- Who does the reaping now? PID 1?

Are Zombies Harmful?

- Entry in the process table?
- Kernel resources?
- Creation of new processes?
- Resource starvation?

Zombies and Containers

- Generally, one main application process runs per container
- Does this main process act like an init process?
- What about reaping?
- Zombies all around?
- What about Docker containers managed by some third-party?
- Need for a proper init system?
- How about using bash?

Init System To The Rescue

- Upstart? Systemd? The heavyweight systems…
- Tini or dumb-init?

Tini Init System

- https://github.com/krallin/tini
- Simple and lightweight
- Appropriate for containers
- Reaps zombies
- Performs signal forwarding
- Adding or removing Tini doesn't have any negative impact

Setting Up Tini


```
FROM debian:stretch
RUN apt-get update && apt-get -y install procps python3
COPY defunct.py /app/
# CMD ["python3", "/app/defunct.py"]

# Add Tini
ENV TINI_VERSION v0.19.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /tini
RUN chmod +x /tini
ENTRYPOINT ["/tini", "--"]

# Run your program under Tini
CMD ["python3", "/app/defunct.py"]
```

```
#!/usr/bin/env python3
import os
import subprocess

child_pid = os.fork()
if child_pid == 0:  # child process
    pid2 = os.fork()
    if pid2 != 0:
        print("The zombie pid will be: {}".format(pid2))
else:  # parent
    print("Parent PID: {}".format(os.getpid()))
    os.waitpid(child_pid, 0)
    subprocess.check_call(('ps', 'xawuf'))
```

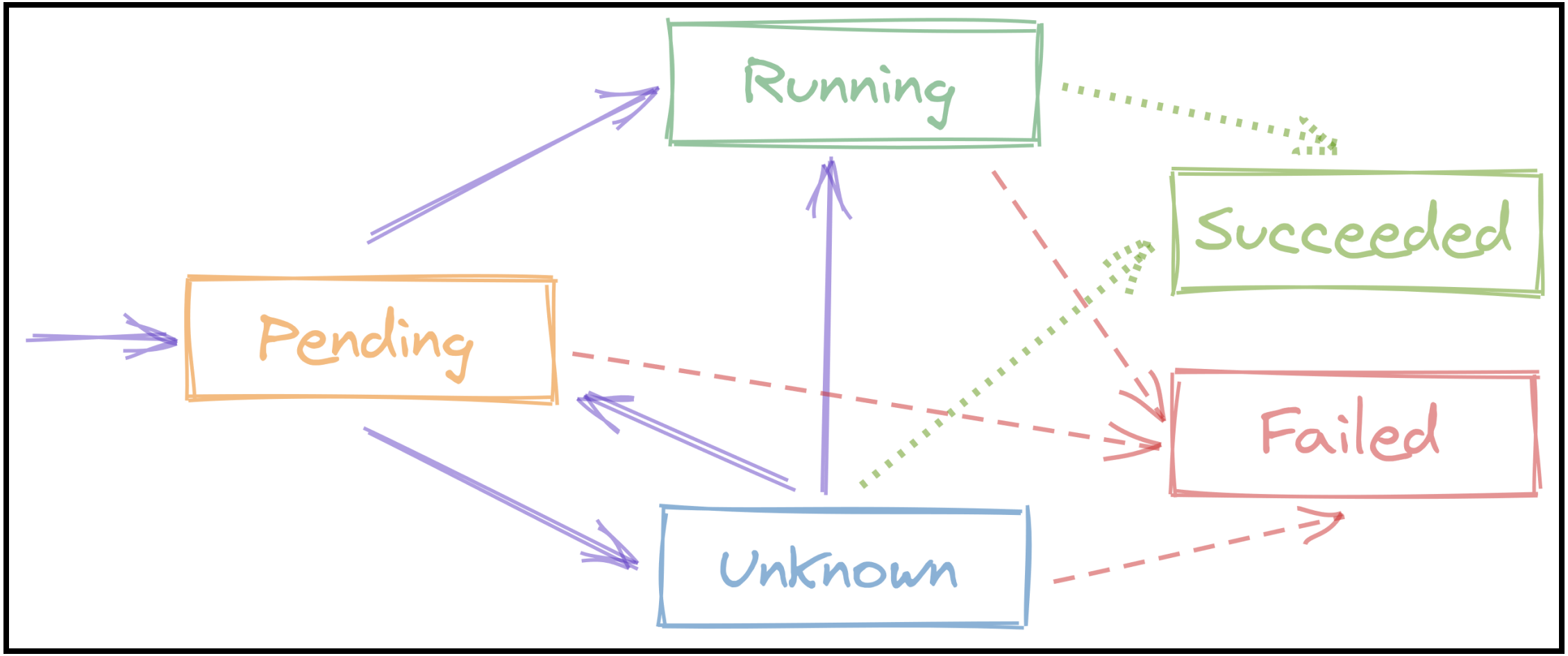
More Details

- Tini needs to run as PID 1 in order to reap zombies
- Can act as a process sub-reaper if not started as PID 1
  - Passing `-s` argument to Tini (tini –s -- …)
- Exits with child's exit code; remapping possible

Managed Lifecycle

Pod & Container Lifecycle, and Linux Signals

Pod Lifecycle Phases



Container Lifecycle States

Terminating Gracefully

- Very important
- Need for cleanups?
- Forced termination? Downtime?
- ***SIGTERM?*** A gentle poke…
- ***SIGKILL?*** The hard kill…

Termination Lifecycle

Set the grace period | Enter TERMINATING state | Stop getting traffic

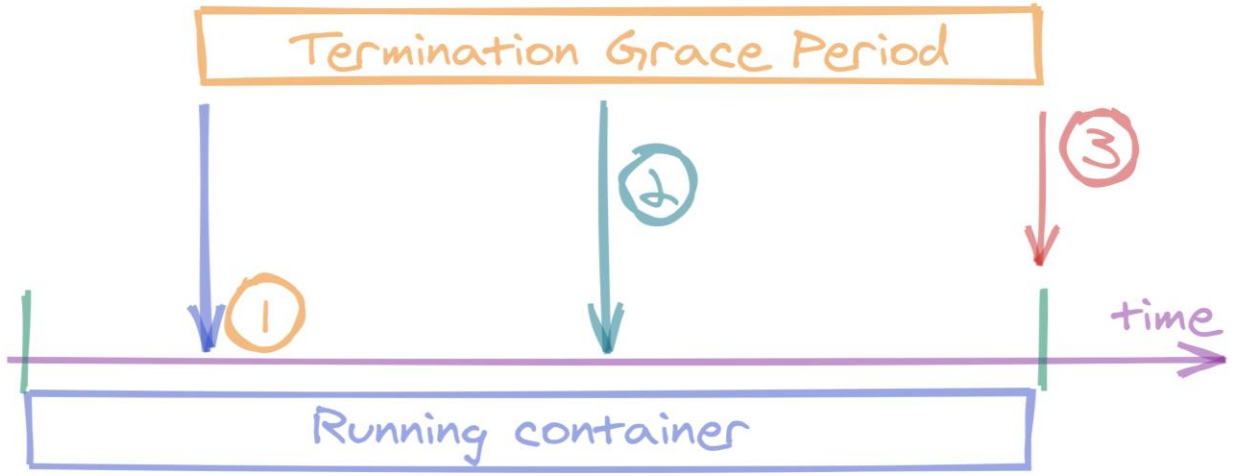
Execute preStop hook, if present | grace of extra 2 seconds possible

Send SIGTERM to PID 1 of each container

Grace period ends | Issue SIGKILL

API server deletes the Pod's API Object

Termination Lifecycle



1. preStop hook executed | Shutdown process begins
2. SIGTERM issued
3. SIGKILL issued | Forcibly shutdown

Resiliency

Health Check,
Liveness Probe,
Readiness Probe,
and Startup Probe

Health Probe Pattern

- About how an application can communicate its health state to Kubernetes
- Kubernetes should know the state of the Pod so that it can decide whether to send requests to the Pod or not
- Containers must provide APIs for different kinds of health checks

Self-Healing Containers

- Kubelet brings up the containers and keep them running until the Pod dies
- Kubelet restarts a container in case if it crashes; done via the Process Health Checks (generic)
- How can a container's main process crash?

Problems

- What if an application stops working without its main process crashing? Deadlock, Infinite Loop, Memory Leak, Thrashing, and numerous other reasons could be there
- Can applications handle these situations well using some complex logic?
- Should other services be able to access or send requests to a crashed application?
- How to tackle such problems?

Probes

- A diagnostic performed by the Kubelet on a container
- Provides resiliency
- Helps in better load balancing and request routing
- Ensures timely response to each request

Technicalities of Probes

- Periodically performed by the Kubelet
- Possible via calling Handlers implemented by the container
- **Type of Handlers / Probing Mechanisms:**
  - **Exec:** Executes a command in a container
  - **TCP Socket:** Performs a TCP check against the specified port of the Pod
  - **HTTP GET:** Makes an HTTP GET request on container's IP address, a specified port and path; success is considered for 2xx or 3xx HTTP response codes
- **Resultant states:**
  - Success
  - Failure
  - Unknown

Container Probes

- Liveness Probe
- Readiness Probe
- Startup Probe

Liveness Probe

- Tells whether a container is alive or dead
- In case of failure, the Kubelet kills the container
- Whether the container will restart or not depends on the `***restartPolicy***` of the container (which can be ***Always***, ***OnFailer***, or ***Never***)
- **TIPS:**
  - Always define a liveness probe for pods running in production
  - Have the application expose a health-check API endpoint (like */health*)
  - The health-check API endpoint should not require authentication, else the probe will always fail
  - Keep it light on the computational resources (probe's CPU time is part of the container's CPU time quota)

Readiness Probe

- A container may need to perform some warm-up procedure
- Signals whether a container is ready to accept requests / connections
- Until all the containers of a Pod are ready, the Pod isn't treated to be ready
- Unlike the Liveness Probe, on failure, a container isn't killed
- **Note:** After receiving a **SIGTERM** signal, even though if the readiness check passes, Kubernetes tries to prevent the container from receiving new requests

Code Walkthrough


```yaml
spec:
  containers:
  - name: hello-python
    image: python-checks:latest
    imagePullPolicy: Never
    livenessProbe:
      httpGet:
        path: /health/live
        port: 5000
      initialDelaySeconds: 2
      failureThreshold: 2
      periodSeconds: 2
    readinessProbe:
      httpGet:
        path: /health/ready
        port: 5000
      initialDelaySeconds: 2
      failureThreshold: 2
      periodSeconds: 2
```

```python
@app.route("/health/ready")
def readiness():
    global POD_READY
    if POD_READY == 0:
        return jsonify({'message': 'The Pod is ready.'}), 200
    else:
        return jsonify({'message': 'The Pod is not ready.'}), 502

@app.route("/health/stop/ready")
def stopReady():
    global POD_READY
    POD_READY = 1
    return jsonify({'message': 'The POD_READY status has been set to false.'}), 200

@app.route("/health/start/ready")
def startReady():
    global POD_READY
    POD_READY = 0
    return jsonify({'message': 'The POD_READY status has been set to true.'}), 200
```

Startup Probe

- Indicates whether the application within the container has started
- All other probes are disabled until Startup Probe succeeds
- Useful for slow-starting containers
- The Startup Probe is meant to be executed only at the startup, unlike others
- A decent `***failureThreshold***` should be provided

Lifecycle Hooks

postStart Hook,
preStop Hook, and
Graceful Termination

Need for Lifecycle Hooks

- Using only process signals for managing container / application lifecycle is somewhat limited
- Helps maintain the container / application lifecycle in a better manner

postStart Hook

- Executed just after a container is created, asynchronously with the main container process
- Warm-up logic can be implemented
- Can be used to delay the startup state of the container while giving time to the main container process to initialize
- Precondition checks can be done – any failure would result in the main process getting killed
- Can be used to signal to an external listener about the application getting started

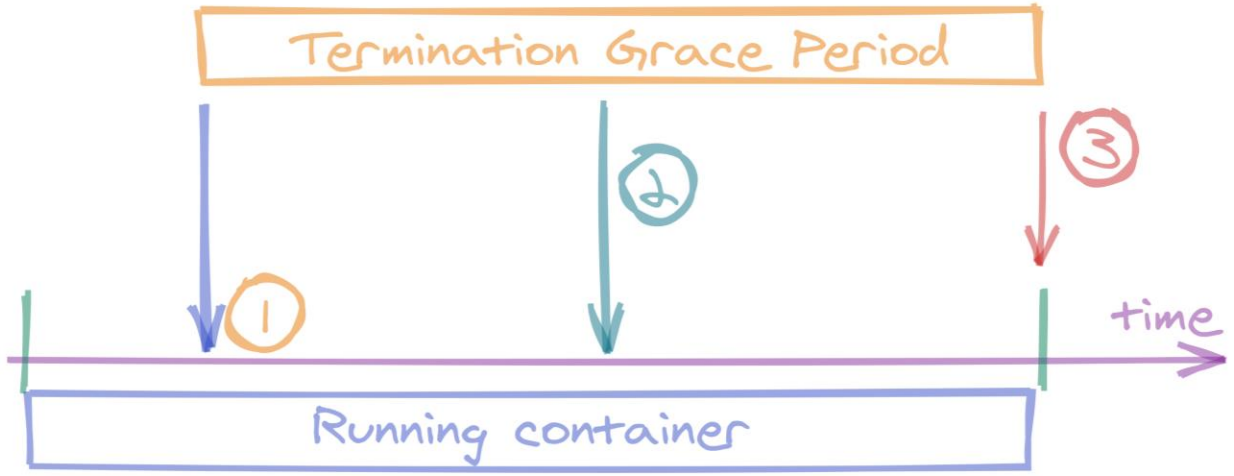
postStart Hook Behaviour

- No guarantees of running
- postStart action is a blocking call
- Container status remains `***Waiting***` until the postStart handler completes, which in turn keeps the Pod status in the `***Pending***`status
- postStart hook runs in parallel with the main container process – ***it may happen that the hook gets executed before the container has started***
- No retries happens

preStop Hook

- Call sent to the container before it is terminated
- Initiates graceful termination
- Use when reacting to **SIGTERM** signal is not possible from within the application
- Useful when using third-party managed container images

Revisiting Termination



1. preStop hook executed | Shutdown process begins
2. SIGTERM issued
3. SIGKILL issued | Forcibly shutdown

Code Walkthrough


```yaml
spec:
  containers:
  - name: lifecycle
    image: nginx
    lifecycle:
      postStart:
        exec:
          command:
            - /bin/sh
            - -c
            - echo '<h1>postStart Hook</h1>' >> /usr/share/nginx/html/index.html && sleep 20
      preStop:
        exec:
          command:
            - /bin/sh
            - -c
            - |
              echo '<h1>preStop Hook</h1>' >> /usr/share/nginx/html/index.html && sleep 20
              nginx -s quit
              while killall -0 nginx; do sleep 1; done
```

The Specialized Init Containers

Need, Working, and Usage

Init Containers

- Run before the application containers
- Contains the utilities or setup which is not present in the application's image
- A Pod can have one or more init containers
- Must run to successful completion
- Don't support lifecycle hooks or probes

Init Containers Continued

- If an init container fails during execution and the Pod's `***restartPolicy***` is not set to `***Never***`, the Kubelet would repeatedly restart the init container until it succeeds
- If the Pod's `***restartPolicy***` is set to `***Never***` and an init container fails during execution, the Pod is treated as failed
- Use separate image(s)
- If multiple init containers are defined, they run sequentially in the specified order; one must successfully complete before the next one starts executing
- Can share the same volume with the application containers
- Altering an init container leads to restarting of the Pod

Code Walkthrough

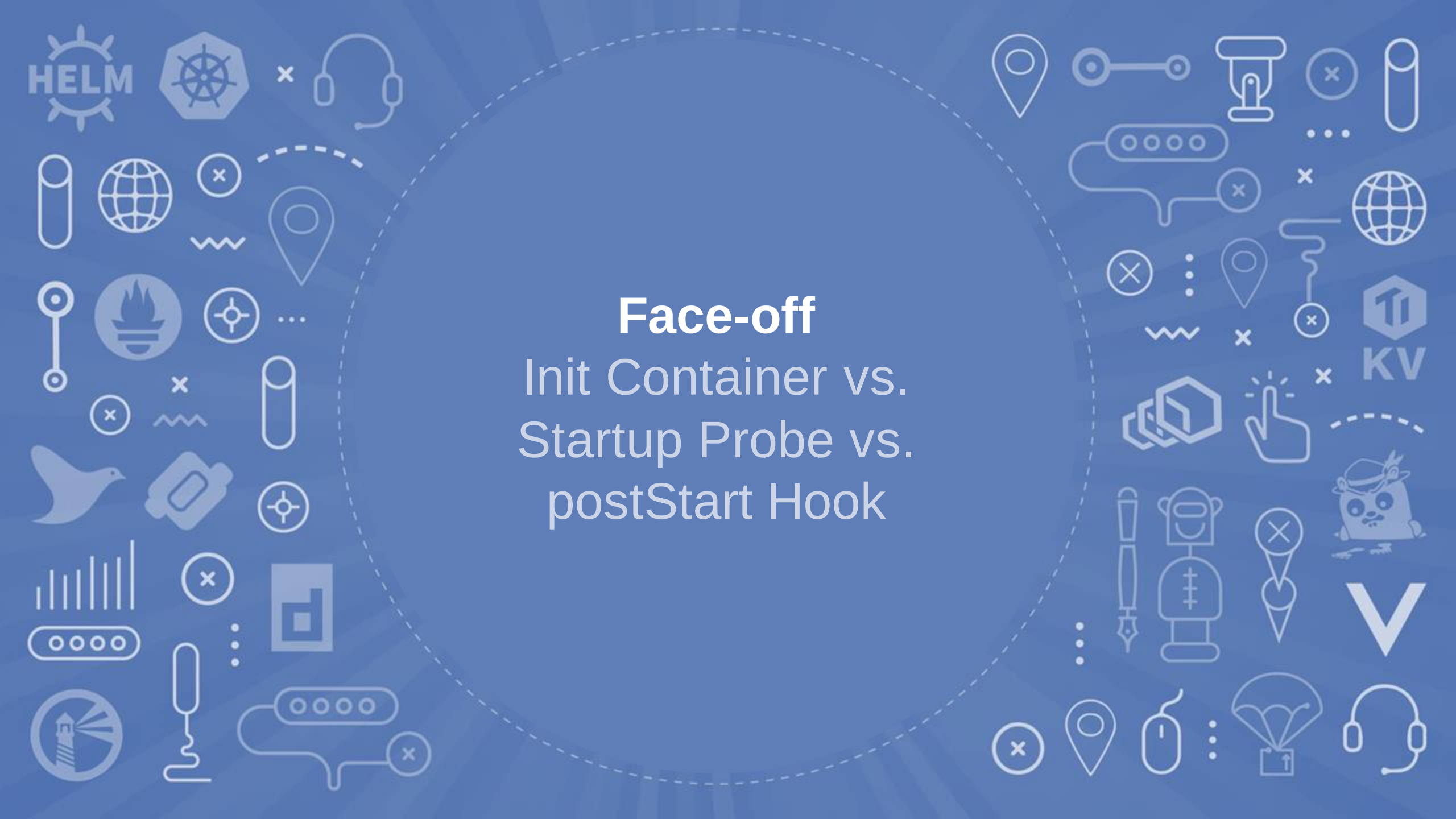

```
 5 spec:
 6   initContainers:
 7   - name: initializer
 8     image: busybox
 9     command:
10     - bin/sh
11     - -c
12     - echo '<h1>Hello World from Init Container</h1>' >> /initmessage/index.html
13     volumeMounts:
14     - mountPath: /initmessage
15       name: nginxinit
16   containers:
17   - name: helloworld
18     image: nginx
19     volumeMounts:
20     - mountPath: /usr/share/nginx/html
21       name: nginxinit
22   volumes:
23   - name: nginxinit
24     emptyDir: {}
```

Usage of Init Containers

- Delaying the application container startup
- Perform precondition checks
- Run utilities or code that is not part of the application container or is not secure to be run through the application container
- Seed data in the database before the application starts
- Wait for some service to become available before the application starts
- Configure things at the runtime
- Perform database schema preparation
- Perform database migrations
- Create user accounts
- And much more…

Scheduling and Resources

- Init containers and application containers co-exist inside a Pod
- Pod's effective request/limit for a resource depends on what is specified for the init containers as well as the application containers
- **Pod's effective request/limit for a resource is the higher of:**
  - The sum of request/limit for a resource of all the application containers
  - The effective request/limit for a resource of the init containers – ***it is the highest of any particular resource request/limit defined on all the init containers***



Face-off

Init Container vs. Startup Probe vs. postStart Hook

Face-off

Parameters	postStart Hook	Init Container	Startup Probe

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application
Scope	Per container	Whole Pod	Per Container

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application
Scope	Per container	Whole Pod	Per Container
Container Image	Same as application	Separate image	Same as application

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application
Scope	Per container	Whole Pod	Per Container
Container Image	Same as application	Separate image	Same as application
Run Guarantee	No	Must run successfully	Must run successfully

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application
Scope	Per container	Whole Pod	Per Container
Container Image	Same as application	Separate image	Same as application
Run Guarantee	No	Must run successfully	Must run successfully
Failure Threshold / Restarts	Kills container if fails	Restart until successful, depends on the Pod's restartPolicy	Can be specified

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application
Scope	Per container	Whole Pod	Per Container
Container Image	Same as application	Separate image	Same as application
Run Guarantee	No	Must run successfully	Must run successfully
Failure Threshold / Restarts	Kills container if fails	Restart until successful, depends on the Pod's restartPolicy	Can be specified
Usage	Precondition checks, signal to external listeners, and introducing delays	Initialization and precondition checks	Appropriate for slow-starting containers and for checking if the application has started functioning

Face-off

Parameters	postStart Hook	Init Container	Startup Probe
Container	Same as application	Separate container	Same as application
Scope	Per container	Whole Pod	Per Container
Container Image	Same as application	Separate image	Same as application
Run Guarantee	No	Must run successfully	Must run successfully
Failure Threshold / Restarts	Kills container if fails	Restart until successful, depends on the Pod's restartPolicy	Can be specified
Usage	Precondition checks, signal to external listeners, and introducing delays	Initialization and precondition checks	Appropriate for slow-starting containers and for checking if the application has started functioning
Count	Exactly one but supports multiple commands using Exec mechanism	Multiple	Exactly one but supports multiple commands using Exec mechanism

Time for Q/A

@Slack #2-kubecon-101


KEEP CLOUD NATIVE
EVERYWHERE
KubeCon
CloudNativeCon
North America 2020
Virtual