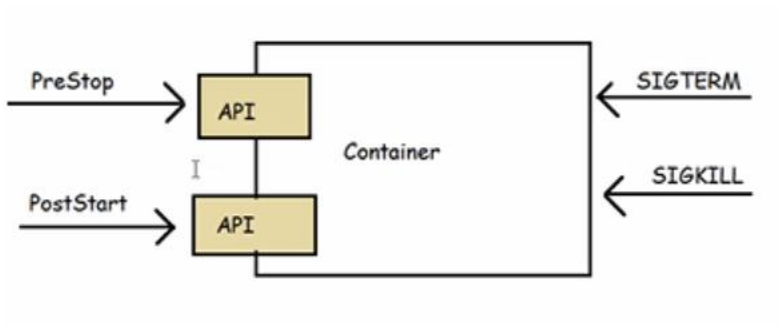
Managed Life Cycle

- Overview: Containerized applications managed by cloud-native platforms have no control over their lifecycle & to be good cloud native citizens they have to listen to the events emitted by managing platforms and adapt their lifecycles according.
- Problem:
  - In addition to monitoring the state of container, the platform sometimes might issue commands and expect the application to react to those
  - Driven by external factors and policies, platform may decide to start or stop the applications it is managing at any moment & it is up to the containerized application to determine which events are important to react & how to react.
- Solution:
  - For this usecase some events are emitted by the platform, that the container can

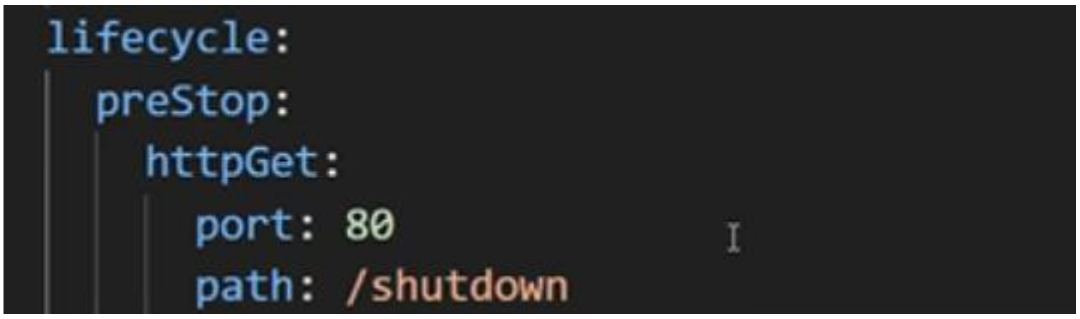
    listen to and react to if desired
  - SIGTERM signal: whenever k8s decides to shut down a container (failed liveness probe or Pod it belongs to is shutting), the container receives a SIGTERM signal. Once the SIGTERM signal has been received, the application should shut down as quickly as possible.
  - SIGKILL signal: If a container process has not shut down after SIGTERM signal, it is shutdown forcefully by following the SIGKILL signal. k8s doesn't send the SIGKILL signal immediately after SIGTERM it will wait for a grace period (default is 30 second). The grace period can be defined per pod spec. terminationGracePeriodSeconds



Container hooks provide information to the container about events in its management lifecycle. For example, immediately after a container is started, it receives a *Post Start* hook. These hooks are broadcast *into* the container with information about the life-cycle of the container. They are different from the events provided by Docker and other systems which are *output* from the container. Output events provide a log of what has already happened. Input hooks provide real-time notification about things that are happening, but no historical log.

PostStart This hook is sent immediately after a container is created. It notifies the container that it has been created. No parameters are passed to the handler. It is NOT guaranteed that the hook will execute before the container entrypoint.(Exp: Starting Agent/ Splunk forwarder)

PreStop This hook is called immediately before a container is terminated. No parameters are passed to the handler. This event handler is blocking, and must complete before the call to delete the container is sent to the Docker daemon. The SIGTERM notification sent by Docker is also still sent. A more complete description of termination behavior can be found in Termination of Pods.(taking backup)



```yaml
lifecycle:
  preStop:
    httpGet:
      port: 80
      path: /shutdown
```

Here, when you are calling application with port 80, and accessing /shutdown, it would shut down your application gracefully.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hooks-demo
spec:
  containers:
    - name: hooks-container
      image: nginx
      ports:
        - containerPort: 80
      livenessProbe:
        httpGet:
          port: 80
          path: /
        initialDelaySeconds: 30
      lifecycle:
        preStop:
          httpGet:
            port: 80
            path: /shutdown
```

Kubernetes best practices: terminating with grace

1 - Pod is set to the "Terminating" State and removed from the endpoints list of all Services
At this point, the pod stops getting new traffic. Containers running in the pod will not be affected.

2 - preStop Hook is executed
The preStop Hook is a special command or http request that is sent to the containers in the pod.

If your application doesn't gracefully shut down when receiving a SIGTERM you can use this hook to trigger a graceful shutdown. Most programs gracefully shut down when receiving a SIGTERM, but if you are using third-party code or are managing a system you don't have control over, the preStop hook is a great way to trigger a graceful shutdown without modifying the application.

3 - SIGTERM signal is sent to the pod
At this point, Kubernetes will send a SIGTERM signal to the containers in the pod. This signal lets the containers know that they are going to be shut down soon.

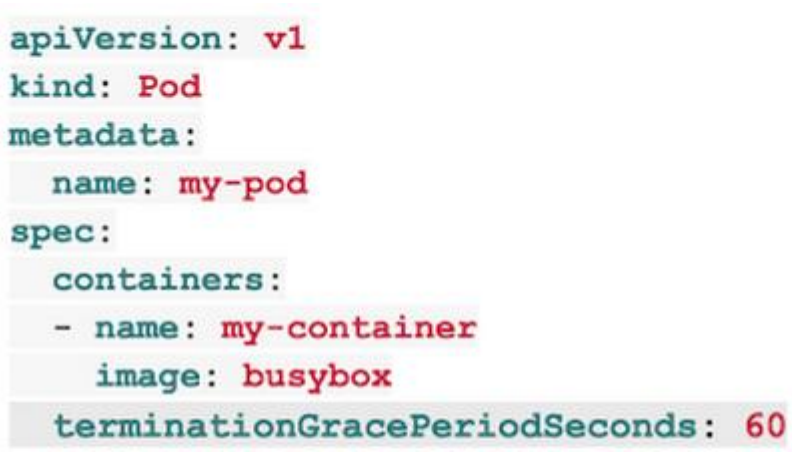
Your code should listen for this event and start shutting down cleanly at this point. This may include stopping any long-lived connections (like a database connection or WebSocket stream), saving the current state, or anything like that.

Even if you are using the preStop hook, it is important that you test what happens to your application if you send it a SIGTERM signal, so you are not surprised in production!

4 - Kubernetes waits for a grace period
At this point, Kubernetes waits for a specified time called the termination grace period. By default, this is 30 seconds. It's important to note that this happens in parallel to the preStop hook and the SIGTERM signal. Kubernetes does not wait for the preStop hook to finish.

If your app finishes shutting down and exits before the terminationGracePeriod is done, Kubernetes moves to the next step immediately.

If your pod usually takes longer than 30 seconds to shut down, make sure you increase the grace period. You can do that by setting the terminationGracePeriodSeconds option in the Pod YAML. For example, to change it to 60 seconds:



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: busybox
  terminationGracePeriodSeconds: 60
```

5 - SIGKILL signal is sent to pod, and the pod is removed

If the containers are still running after the grace period, they are sent the SIGKILL signal and forcibly removed. At this point, all Kubernetes objects are cleaned up as well.

Conclusion
Kubernetes can terminate pods for a variety of reasons and making sure your application handles these terminations gracefully is core to creating a stable system and providing a great user experience.

Automated Placement

- Overview: Automated Placement is the core function of the k8s scheduler for signing new Pods to nodes. This pattern is about exploring the ways to influence placement decisions from outside.
- Problem: A decent microservice will consist of tens to hundreds of isolated processes (Containers in a Pod). With large and ever-growing number of microservices, assigning and placing them individually to nodes is not a manageable activity
- Solution:
    - In k8s, assigning Pods to nodes is done by kube-scheduler.
    - kube-scheduler whenever it finds a newly created Pod defintion for API Server assigns the pod to node.
    - Allocatable Node Resources: “` Allocatable = Node Capacity – kube-Reserved (kubelet, container) – System-Reserved (sshd, udev)