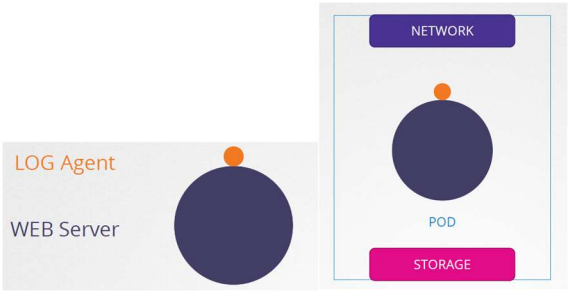
Multi-Container Pod

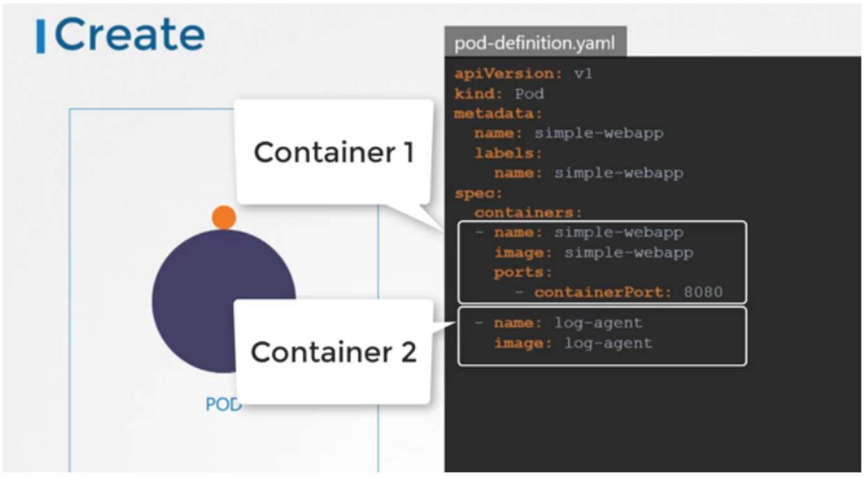
In this section on Multi-Container Pods, the idea of decoupling a large monolithic application into sub-components known as microservices enables us to develop and deploy a set of independent, small, and reusable code. This architecture can then help us scale up, down, as well as modify each service as required, as opposed to modifying the entire application.

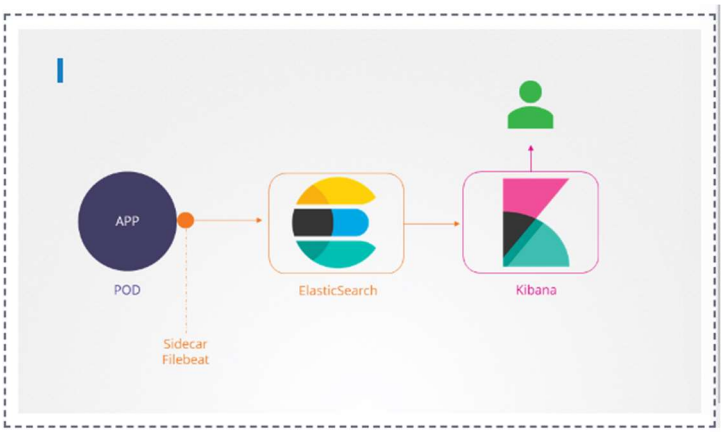
However, at times, you may need two services to work together, such as a web server and a logging service. You need one agent instance per web server instance paired together. You don't want to merge and load the code of the two services, as each of them targets different functionalities and you would still like them to be developed and deployed separately. You only need the two functionalities to work together.

You need one agent per web server instance paired together that can scale up and down together, and that is why you have multi-container pods that share the same lifecycle, which means they are created together and destroyed together. They share the same network space, which means they can refer to each other as localhost, and they have access to the same storage volumes. This way, you do not have to establish volume sharing or services between the pods to enable communication between them.



To create a multi-container pod, add the new container information to the pod definition file. Remember, the container section under the spec section in a pod definition file is an array, and the reason it is an array is to allow multiple containers in a single pod. In this case, we add a new container named "log agent" to our existing pod.






```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app
  namespace: elastic-stack
  labels:
    name: app
spec:
  containers:
  - name: app
    image: kodekloud/event-simulator
    volumeMounts:
    - mountPath: /log
      name: log-volume

  - name: sidecar
    image: kodekloud/filebeat-configured
    volumeMounts:
    - mountPath: /var/log/event-simulator/
      name: log-volume

  volumes:
  - name: log-volume
    hostPath:
      # directory location on host
      path: /var/log/webapp
      # this field is optional
      type: DirectoryOrCreate
```

initContainer

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. If any of them fails, the POD restarts.

But at times you may want to run a process that runs to completion in a container. For example, a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. That's where initContainers comes in.

An **initContainer** is configured in a pod like all other containers, except that it is specified inside a initContainers section, like this:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-application> ; done;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.

You can configure multiple such initContainers as well, like how we did for multi-containers pod. In that case each init container is run **one at a time in sequential order**.

If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```