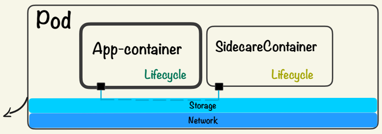
MULTI-CONTAINER PODs: ■ Sidecar Container □ Adapter □ Ambassador

Sidecar container is a container that is deployed alongside a main container in a pod . The main container is typically an application that performs some specific function, while the sidecar container provides support or complementary functionality to the main container

The idea behind the sidecar pattern is to keep the main container focused on a specific task or functionality, while delegating other tasks to the sidecar container. This allows for more modular and flexible deployment architectures, as the sidecar container can be updated or replaced independently of the main container

Although containers inside a pod share a common network and storage, they have independent lifecycles and can be created, updated, and deleted individually



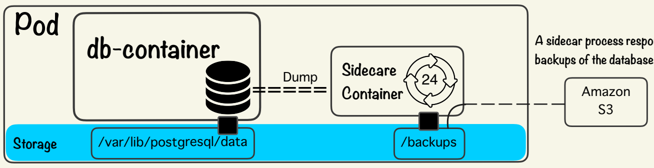
important use cases

- ■ Logging and Monitoring: A side containers can be used to collect and forward logs and metrics from the main application container to a central monitoring system
- ■ Backup and Recovery: A side containers can be used to perform backup and recovery operations on the main application container
- ■ Service mesh: A sidecar container can be used to implement a service mesh such as Istio or Linkerd. A service mesh provides additional functionality for managing and securing communications between services running in Kubernetes

One example of how a sidecar container can be used with a database service in a Kubernetes deployment:

The main container is running a database service and is exposing port 5432 for incoming database connections.

The sidecar container is configured to perform backups of the database



A sidecar process responsible for periodic backups of the database to an S3 bucket

The sidecar container can periodically backup the database to a remote location to ensure data resiliency

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: db-pod
spec:
  containers:
  - name: db-container
    image: my-database-image
    env:
      - name: DATABASE_URL
        value: "postgresql://my-database-hostname:5432/my-database"
    ports:
      - containerPort: 5432
    volumeMounts:
      - name: db-data
        mountPath: /var/lib/postgresql/data

  name: sidecar-container
    image: my-sidecar-image
    env:
      - name: BACKUP_LOCATION
        value: "s3://my-bucket/my-backups"
      - name: DATABASE_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: database-password
    volumeMounts:
      - name: backup-data
        mountPath: /backups
      - name: db-secrets
        mountPath: /secrets
    command: ["/bin/sh", "-c"]
    args:
      - |
        while true; do
          pg_dump -U postgres -h localhost my-database | gzip > /backups/my-
database-$(date +%Y-%m-%d-%H%M%S).sql.gz; s3cmd put /backups/my-database-
*.sql.gz "$BACKUP_LOCATION";
          sleep 86400;
          s3cmd put /backups/my-database-*.sql.gz "$BACKUP_LOCATION";
        done
  volumes:
    - name: db-data
      emptyDir: {}
    - name: backup-data
      emptyDir: {}
    - name: db-secrets
      secret:
        secretName: db-secrets
```

The sidecar container is running a script that periodically backs up the database and stores the backup files in the "/backups" directory. The script is also using the "pg_dump" command to perform the backup and gzip to compress the backup file. The backup location is specified in the environment variable "BACKUP_LOCATION", which is set to an S3 bucket. The script is running in an infinite loop and sleeps for 24 hours between each backup.

The two containers are communicating using shared volumes and environment variables. The main container is using a volume mount called "db-data" to store its data files, while the sidecar container is using a volume mount called "backup-data" to store its backup files

Job & CronJobs

Job is a type of resource that allows you to create and manage a finite or batch process in your cluster. Jobs are commonly used for tasks that need to be run once or a few times, such as data processing, backups, or migrations

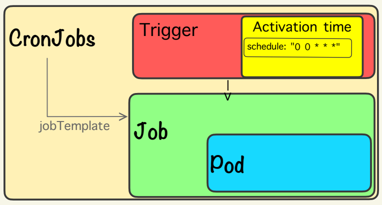
A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-processing-job
spec:
  backoffLimit: 3
  template:
    spec:
      containers:
      - name: data-processor
        image: data-processor:v1.4
        command: ["python", "process_data.py"]
      restartPolicy: Never
```

The backoffLimit specifies the number of times k8s should retry the Job if it fails before giving up

CronJobs in Kubernetes are a way to schedule and automate the execution of Jobs on a recurring basis. A Job is a Kubernetes object that creates one or more Pods to perform a specific task, and a CronJob is a higher-level abstraction that allows Jobs to be scheduled according to a specific time or interval, similar to the Unix cron utility.

Let's say you have a web application that periodically needs to generate reports based on user data. You could create a CronJob that runs a script to generate the report and then terminates when the report is complete.



CronJobs create Jobs which in turn create Pods to run the task

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: report-generation-cronjob
spec:
  schedule: "0 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          ttlSecondsAfterFinished: 100
          containers:
          - name: report-generator
            image: my-django-app:v1
            env:
            - name: DJANGO_SETTINGS_MODULE
              value: myapp.settings
            command: ["python", "manage.py", "generate_report"]
          restartPolicy: Never
```

Job will run at the top of every hour

This will delete the Pod 100 seconds after it finishes

Notice: By default, completed Jobs and Pods are retained after running. To automatically clean up completed Jobs, you can set `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit` on the CronJob