# Hand-Written notes on
# Kubernetes Sidecar Containers & BusyBox (A Simple Student Guide)

---

## Point 1: What is a Kubernetes Sidecar Container?

### 1. What is a Sidecar Container?

Sometimes, along with the main container, we need an extra container to help it. This extra container is called a **Sidecar Container**. It runs alongside the main container and provides additional features to improve performance and functionality of main container without losing main containers quality and performance.

Think of it like this:
**Main Container** = Your application (e.g., a web server)
**Sidecar Container** = A helper that manages logs, monitoring, or syncing data

### 2. Why is a Sidecar Container needed?

If your main container is designed to do only one job but needs extra features, instead of modifying it, you can add a Sidecar Container.

**Example:** If a container is running a web app, but you need to store log files, you can use a separate Sidecar Container for log storage.

### 3. Use Cases of Sidecar Containers

- **Log Management** – To store and transfer logs
- **Security** – To enhance the security of the main container
- **Proxy Server** – To optimize networking
- **Data Backup & Sync** – To send data to cloud or storage
- **Monitoring** – To check the health of the container

### 4. Benefits of Sidecar Containers

- You can add new features without modifying the main container
- It improves microservices architecture
- Helps create scalable and repeatable solutions
- Makes logging, monitoring, and security easier

### 5. How to create and use a Sidecar Container?

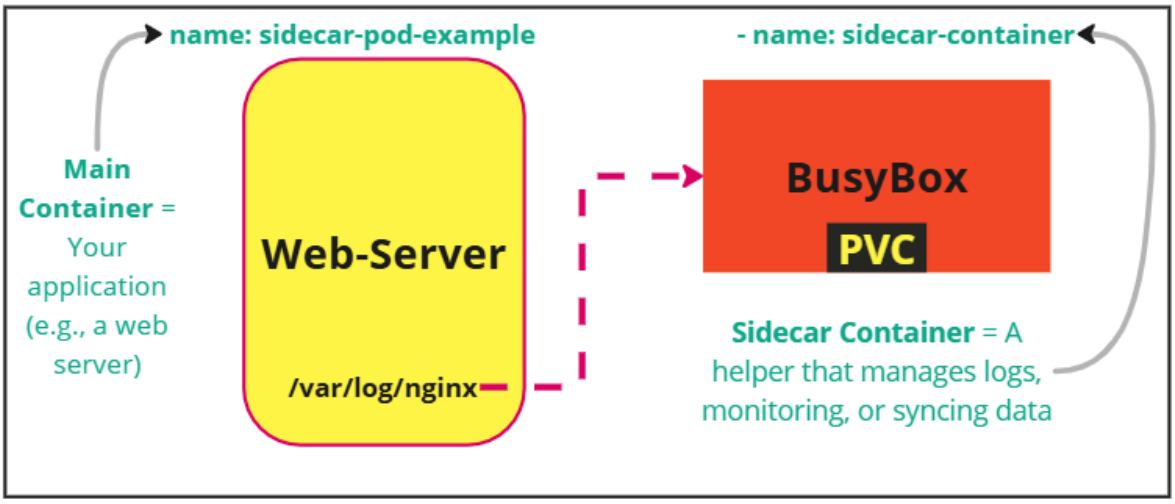
You need a Pod YAML file with two containers:
1**: Main Container** (Nginx Web Server)
2: **Sidecar Container** (BusyBox Log Forwarder)

Sometimes, along with the main container, we need an extra container to help it. This extra container is called a **Sidecar Container**.

**Definition**

It runs alongside the main container and provides additional features to improve performance and functionality of main container without losing main containers quality and performance.

**Diagram**




**Manifests**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sidecar-pod-example
spec:
  containers:
    - name: main-container
      image: nginx
      volumeMounts:
        - name: shared-logs
          mountPath: /var/log/nginx

    - name: sidecar-container
      image: busybox
      command: ["/bin/sh", "-c", "while true; do cat /var/log/nginx/access.log; sleep 10; done"]
      volumeMounts:
        - name: shared-logs
          mountPath: /var/log/nginx

  volumes:
    - name: shared-logs
      persistentVolumeClaim:
        claimName: pvc2
```

The **main container (nginx)** writes logs to /var/log/nginx.

The **sidecar container (busybox)** reads the same logs from /var/log/nginx and prints them every 10 seconds.

pvc volume "pvc2" is using here to store data persistently

miro

**Note:**

Let's understood the line below in **command** section, that runs inside the busybox container in your Kubernetes Pod:

**command: ["/bin/sh", "-c", "while true; do cat /var/log/nginx/access.log; sleep 5; done"]**

**Breaking it Down:**

1. **/bin/sh** → This starts a shell (sh) inside the container.
2. **-c** → This tells the shell to execute the following command as a script.
3. **while true; do ... done** → This is an infinite loop that keeps running continuously.

4. **cat /var/log/nginx/access.log** → Reads and prints the contents of the Nginx access log.
5. **sleep 5** → Waits for 5 seconds before running the next iteration of the loop.

**What This Does in the Pod:**

- The busybox container keeps reading (cat) the Nginx access logs every 5 seconds.
- This is useful for debugging because it allows you to see real-time logs.
- The loop ensures the container doesn't exit immediately after running the command once.

## How does this work?

- **Main Container (Nginx)** serves the website
- **Sidecar Container (BusyBox)** continuously checks and stores Nginx logs

To apply this in Kubernetes run the manifests file:
kubectl apply -f sidecar.yaml

To check logs:
kubectl logs -f sidecar-example -c sidecar-container

---

## Point 2: What is a BusyBox Image?

**BusyBox** is a very lightweight **Linux-based utility tool** that includes basic Unix commands like ls, cp, mv, echo, cat, grep, wget, etc. without any heavy program or utility tools. It is called **"The Swiss Army Knife of Embedded Linux"** because it provides multiple Unix commands in a single binary. It has 1mb size max. so this is lightweight and fast.

## Use Cases of BusyBox Image

**1. Testing & Debugging**

You can use BusyBox for testing in Docker/Kubernetes:

```
docker run -it busybox sh
```

This opens a lightweight Linux shell where you can run Unix commands.

**2. Getting Shell Access in a Container**

```
kubectl run -it my-busybox --image=busybox -- sh
```

This creates a temporary pod where you can run commands inside the container.

**3. Checking Network Connectivity**

```
kubectl run busybox --image=busybox --restart=Never -- ping google.com
```

This checks if your Kubernetes pod is connected to the internet.

**4. Data Processing & File Operations**

```
kubectl run my-busybox --image=busybox --restart=Never -- cat /var/log/app.log
```

This reads log files inside the container.

**5. Using BusyBox as a Sidecar Container**

BusyBox can be used as a Sidecar to process data from another container:

```
containers:
- name: sidecar-container
  image: busybox
  command: ["/bin/sh", "-c", "while true; do cat /var/log/app.log; sleep 10; done"]
```

This continuously monitors the /var/log/app.log file.

## Benefits of BusyBox Image

- **Lightweight** – Only about **1MB** in size
- **Fast** – Loads quickly and uses fewer resources
- **Ideal for Embedded Systems** – Works well in low-memory and low-CPU environments
- **Complete Unix Toolset** – Includes sh, wget, ping, echo, cat, vi, etc.

---

## Point 3: FAQs (Frequently Asked Questions)

## 1. Can we use the Main Container for Sidecar tasks?

Defiantly we can, it is possible, but it is not a good practice.

**Problems**

- If the **Main Container** (e.g., Nginx, PostgreSQL) handles logging, monitoring, or networking, maintenance becomes difficult.
- The **Single Responsibility Principle (SRP)** suggests that each container should do only one job.
- Updating the Main Container becomes harder if it performs multiple tasks.

## 2. Why is a Sidecar Container better?

- It **does not affect the Main Container**.
- The **application and Sidecar Container can be developed separately**.
- If you only need to change **log forwarding or monitoring**, you can update the Sidecar instead of modifying the Main Container.
- This approach is **better for Microservices and DevOps architectures**.

## 3. Why use BusyBox instead of other Base OS images?

### 1. Lightweight and Fast

- **BusyBox** is **less than 1MB** in size, while Base OS images (Ubuntu, Alpine, Debian) can be **100MB+**.
- A **smaller image means faster loading, less storage, and lower RAM usage**.

### 2. Complete Unix Command Set

- BusyBox includes essential Unix tools (ls, cat, ping, wget, grep, sh).
- We don't need a full Base OS, just basic commands.

### 3. Secure and Minimal

- BusyBox **only includes necessary commands**, reducing security risks.
- Other Base OS images may have unnecessary packages, increasing security vulnerabilities.

### 4. Best for Kubernetes and Containers

- BusyBox is a **scratch-based image** (basic and minimal), making it **perfect for containerized applications**.
- It is **ideal for Sidecar Containers, Debugging Pods, and Testing**.

## 4. Can we use alternatives instead of BusyBox?

Defiantly we can, but BusyBox is **the best and simplest option**. Why because of its size and minimalist nature with complete required tools options.

**Alpine Linux** – Lightweight (~5MB) but heavier than BusyBox
**Debian Slim** – Secure but larger (~20MB+)
**Ubuntu Minimal** – Secure and stable but heavy (~29MB+)
**Scratch** – **Zero-size image** (no built-in tools) but not good for debugging

If you need **Shell Access and Basic Commands in a lightweight container, BusyBox is the best choice**!

---

## My Conclusion

- **Sidecar Containers** help improve the functionality of the Main Container without modifying it.
- They are used for **logging, monitoring, proxy, data sync, and security**.
- Kubernetes **Sidecar Containers** are defined in a **YAML file** with shared **volumes**.
- **BusyBox** is a **lightweight** Linux-based image, ideal for **debugging, networking, file processing, and Sidecar Containers**.
- It is **fast, minimal, and secure**, making it the **best choice for Kubernetes and Docker environments**.
- Using the **Sidecar Pattern** keeps containers modular, scalable, and easy to manage.

---

## Now have some practice

**Practice-1:** Create an Nginx web server pod with a sidecar container that prints logs every 2 seconds. This will demonstrate how a sidecar container can be used for log processing in Kubernetes.

## Answer:

Step-1 Here's the YAML configuration to create a **Pod** with two containers:

1. **Nginx container** - Runs the web server and stores logs in /var/log/nginx.

2. **BusyBox sidecar container** - Reads and prints the logs from /var/log/nginx every 2 seconds.

   # vim sidecar-pod-example.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-sidecar-pod
spec:
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: log-volume
      mountPath: /var/log/nginx
  - name: log-sidecar
    image: busybox
    command: ["/bin/sh", "-c"]
    args: ["while true; do cat /var/log/nginx/access.log; sleep 2; done"]
    volumeMounts:
    - name: log-volume
      mountPath: /var/log/nginx
  volumes:
  - name: log-volume
    emptyDir: {}
```

Manifests yaml file Explanation:

- **Shared Volume (emptyDir)**: Both containers share /var/log/nginx using an emptyDir volume.
- **Nginx Container**: Writes access logs to /var/log/nginx/access.log.
- **BusyBox Sidecar**: Reads the logs from the shared volume and prints them every 2 seconds.

Now question may come, "If **logs are already stored in a volume**, you can view them directly from the volume mount point. So, **why do we need a sidecar container to print logs using:**

**args: ["while true; do cat /var/log/nginx/access.log; sleep 2; done"]**

**So answer would be "You are right If you just need to view logs, use kubectl logs or kubectl exec but If you need to automatically process, filter, stream, or forward logs, a sidecar container is useful."**

**Step-2: See the logs from sidecar**

**# kubectl exec -it nginx-sidecar-pod -c nginx-container -- cat /var/log/nginx/access.log**

```
root@control:~# kubectl exec -it nginx-sidecar-pod -c nginx-container -- cat /var/log/nginx/access.log
127.0.0.1 - - [03/Mar/2025:07:39:33 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.88.1" "-"
```

---

# Real-World Projects (Sidecar Containers in Kubernetes: )

**Assignment: Implementing Sidecar Containers in Kubernetes**

**Objective**

The goal of this assignment is to understand and implement the **sidecar container pattern** in Kubernetes. Each project demonstrates how a sidecar container enhances the primary application by providing additional functionalities like log forwarding, database backup, monitoring, and dynamic configuration updates.

---

## Project 1: Log Forwarding with Fluentd

**Use Case**

In microservices environments, efficient log management is crucial. This project implements a sidecar pattern where an **Nginx web server** writes logs, and a **Fluentd container** collects and forwards them to an external system.

**Kubernetes Pod Manifest:**

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-fluentd-pod  # Pod name
spec:
  volumes:
    - name: log-volume
      emptyDir: {}
  containers:
    - name: nginx  # Primary container: Nginx (Web Server)
      image: nginx
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
    - name: fluentd  # Sidecar container: Fluentd (Log Processor)
      image: fluent/fluentd
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
```

**How It Works?**

- Nginx writes logs to /var/log/nginx.
- Fluentd reads these logs and forwards them to an external system.

**Service Manifest (ClusterIP):**

```
apiVersion: v1
kind: Service
metadata:
  name: fluentd-service
spec:
  selector:
    app: nginx-fluentd
  ports:
    - protocol: TCP
      port: 24224
      targetPort: 24224
  type: ClusterIP
```

**Access Fluentd Logs:**

```
kubectl port-forward svc/fluentd-service 24224:24224
```

---

## Project 2: Automated Database Backup

**Use Case**

Automated backups are essential for databases running inside Kubernetes. This project introduces a **sidecar container** that periodically backs up **MySQL database data** to a shared volume.

**Kubernetes Pod Manifest:**

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-backup-pod
```

```
spec:
  volumes:
    - name: db-backup
      nfs:
        server: <NFS_SERVER_IP>
        path: /exported/path
  containers:
    - name: mysql
      image: mysql:5.7
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "root"
      volumeMounts:
        - name: db-backup
          mountPath: /var/backups
    - name: backup-sidecar
      image: busybox
      volumeMounts:
        - name: db-backup
          mountPath: /var/backups
      command: ["/bin/sh", "-c", "while true; do cp -r /var/lib/mysql /var/backups; sleep 3600; done"]
```

**How It Works?**

- MySQL stores data in its primary volume.
- The sidecar runs an automated backup process every hour.

**Service Manifest (ClusterIP):**

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-backup-service
spec:
  selector:
    app: mysql-backup
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
  type: ClusterIP
```

**Access MySQL Database:**

```
kubectl port-forward svc/mysql-backup-service 3306:3306
```

---

## Project 3: Application Monitoring with Prometheus

**Use Case**

Observability is key in modern applications. This project integrates **Prometheus as a sidecar container** to monitor application metrics in real-time.

**Kubernetes Pod Manifest:**

```
apiVersion: v1
kind: Pod
metadata:
  name: app-monitoring-pod
spec:
  containers:
    - name: my-app
      image: myapp:latest
      ports:
        - containerPort: 8080
    - name: prometheus-exporter
      image: prom/prometheus
```

```
    ports:
      - containerPort: 9090
```

**How It Works?**

- The application runs on port 8080.
- The Prometheus sidecar scrapes and exposes application metrics.

**Service Manifest (NodePort):**

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus-exporter-service
spec:
  selector:
    app: app-monitoring
  ports:
    - protocol: TCP
      port: 9090
      targetPort: 9090
      nodePort: 30090
  type: NodePort
```

**Access Prometheus Dashboard:**

```
kubectl get svc prometheus-exporter-service
```

Then, open:

```
http://<NODE_IP>:30090/metrics
```

---

## Project 4: Dynamic Configuration Updates

**Use Case**

Many applications require **dynamic configuration updates** without restarting. This sidecar container **fetches updates periodically** and updates the config file.

**Kubernetes Pod Manifest:**

```
apiVersion: v1
kind: Pod
metadata:
  name: config-sync-pod
spec:
  volumes:
    - name: config-volume
      emptyDir: {}
  containers:
    - name: my-app
      image: my-app:latest
      volumeMounts:
        - name: config-volume
          mountPath: /etc/app/config
    - name: config-updater
      image: busybox
      volumeMounts:
        - name: config-volume
          mountPath: /etc/app/config
      command: ["/bin/sh", "-c", "while true; do echo 'updated' > /etc/app/config/config.yaml; sleep 60;
done"]
```

**How It Works?**

- The application reads configuration from /etc/app/config/config.yaml.
- The sidecar updates this file every 60 seconds.

**Service Manifest (ClusterIP):**

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: config-sync-service
spec:
  selector:
    app: config-sync
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: ClusterIP
```

**Access Configuration Update Logs:**

```
kubectl logs -f config-sync-pod -c config-updater
```

---

**Deployment & Testing**

**Apply All Manifests:**

```
kubectl apply -f fluentd-service.yaml
kubectl apply -f mysql-backup-service.yaml
kubectl apply -f prometheus-exporter-service.yaml
kubectl apply -f config-sync-service.yaml
```

**Check Running Services:**

```
kubectl get svc
```

**Delete Any Service If Needed:**

```
kubectl delete -f fluentd-service.yaml
kubectl delete -f mysql-backup-service.yaml
kubectl delete -f prometheus-exporter-service.yaml
kubectl delete -f config-sync-service.yaml
```

Now, verify that all your **sidecar containers** are correctly configured and functional!

---

Follow this channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**