Kubernetes Objects are like instructions that tell Kubernetes how to manage applications and resources. They act as blueprints that define what should run, how many copies should exist, how networking should work, and how data should be stored.

In Kubernetes, these "objects" are just the *pieces of information* (or "blueprints") that describe how your app should run in a containerized environment. They tell Kubernetes things like:

* How many copies of your app should be running
* How to connect your app to other services
* Where to store your data

**Pod :**

A Pod is the smallest unit in Kubernetes. It is like a Box where application lives or runs.

You can think of it like a **box that holds one or more containers** (usually just one) — along with storage and network settings — all bundled together.

**kubectl create pod nginx –image=nginx-latest**

**Why Pods, Not Just Containers?**

Kubernetes doesn’t manage containers **directly**. It manages **pods**. Here’s why:

* You may need **multiple containers** to work together (like app + log forwarder)

**Which Kubernetes apiVersion Should I Use?**

<https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-apiversion-definition-guide.html>

**Basic Structure of a Kubernetes YAML File**

apiVersion: <Kubernetes API version>

kind: <Type of Kubernetes object>

metadata:

name: <Name of the object>

spec:

<Specification of the object>

**1️.apiVersion**

* Specifies which Kubernetes API version to use for the object.
* Example values:
  + v1 for **Pods, Services, ConfigMaps**
  + apps/v1 for **Deployments, ReplicaSets, StatefulSets**
  + batch/v1 for **CronJobs, Jobs**

**apiVersion: v1**

**2️.kind**

* **Defines the type of Kubernetes object you are creating.**
* Example values:
  + Pod → To create a single pod
  + Deployment → To create a scalable, self-healing application
  + Service → To expose an application
  + ConfigMap → To store non-sensitive configuration data

apiVersion: v1

kind: Pod

**3️.metadata**

* Contains details about the object, such as:
  + name: Unique name of the object.
  + namespace: (Optional) Defines the namespace where the object will be created.
  + labels: Key-value pairs used for identification and selection.
  + annotations: Additional metadata (not used for selection).

apiVersion: v1

kind: Pod

metadata:

name: my-nginx-pod

namespace: default

labels:

app: nginx

environment: dev

annotations:

description: "This is a Nginx pod"

createdBy: "Admin"

**4️.spec**

**What is spec in Kubernetes?**

**spec (**specification**) is the main section in a Kubernetes manifest that defines what we want Kubernetes to do. It describes:**

**✅ How many replicas should run? (for Deployments)  
✅ What containers should run? (for Pods)  
✅ How should a Service expose an application?**

apiVersion: v1

kind: Pod

metadata:

name: my-nginx-pod

namespace: default

labels:

app: nginx

environment: dev

annotations:

description: "This is an Nginx pod"

createdBy: "Admin"

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

**📌 Explanation:**

* **Creates a Pod named nginx-pod.**
* **Runs a single container using the nginx image.**
* **Opens port 80 inside the container.**

**Services:**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: nginx-service**

**namespace: default**

**spec:**

**selector:**

**app: nginx # Matches the Pod label**

**ports:**

**- protocol: TCP**

**port: 80**

**targetPort: 80**

**type: LoadBalancer # Use "NodePort" if LoadBalancer is not available**

**What is selector in Kubernetes?**

**Selector is used to match labels on existing or new Pods. It tells Kubernetes:  
✅ Which Pods belong to this ReplicaSet or Deployment?  
✅ Which Pods should this Service send traffic to?**

apiVersion: v1

kind: Pod

metadata:

name: my-nginx-pod

namespace: default

labels:

app: nginx

environment: dev

annotations:

description: "This is an Nginx pod"

createdBy: "Admin"

spec:

selector: # 🔹 Selector to match the pod labels

matchLabels:

app: nginx

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

env:

- name: APP\_ENV

value: "development"

- name: LOG\_LEVEL

value: "info"

resources:

requests:

cpu: "250m" # Requests 0.25 CPU cores

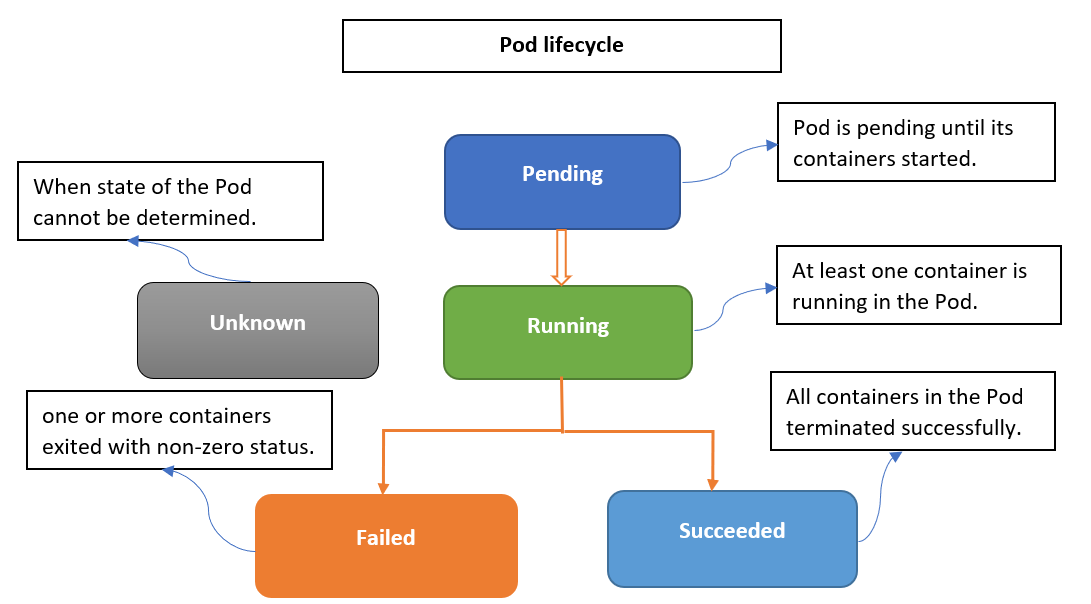
memory: "256Mi" # Requests 256MB RAM

limits:

cpu: "500m" # Limits to 0.5 CPU cores

memory: "512Mi" # Limits to 512MB RAM

**Pod Life Cycle**



**1. Pending**

What’s Happening:

* You submit a pod manifest (e.g., kubectl apply -f pod.yaml)
* The Kube API Server accepts it and writes to etcd (Kubernetes database).
* Scheduler checks for available nodes with enough resources (CPU, RAM, taints/tolerations, affinity, etc.)
* Pending = Pod has been accepted but is not yet running on a node

**2. Running**

What’s Happening:

* The Scheduler assigns the pod to a node.
* The Kubelet on that node sees the assignment and:
  + Pulls container images (via container runtime like containerd)
  + Mounts volumes
  + Creates the pod sandbox (network namespace)
  + Starts containers
* Once at least one container is running, pod phase = Running

**3. Succeeded**

What’s Happening:

* All containers in the pod terminate successfully (exit code 0)
* Kubelet marks the pod as Succeeded
* Example: A batch job that processes a file and exits cleanly.

**A screenshot of a computer

AI-generated content may be incorrect.**

When a liveness probe fails, Kubernetes kills the container and restarts it, but the pod status remains Running — because the pod itself is still alive and managed by the kubelet.

**❌ 2. Probe Fails Multiple Times**

* **The container fails 3 times (as per failureThreshold)**
* **Kubernetes kills (restarts) the container inside the pod**

** RESTARTS increases**

** Status still shows Running because:**

* **The pod is healthy**
* **Only the container was restarted**

kubectl apply -f pod.yaml

kubectl get pods -w

kubectl describe pod <pod-name>

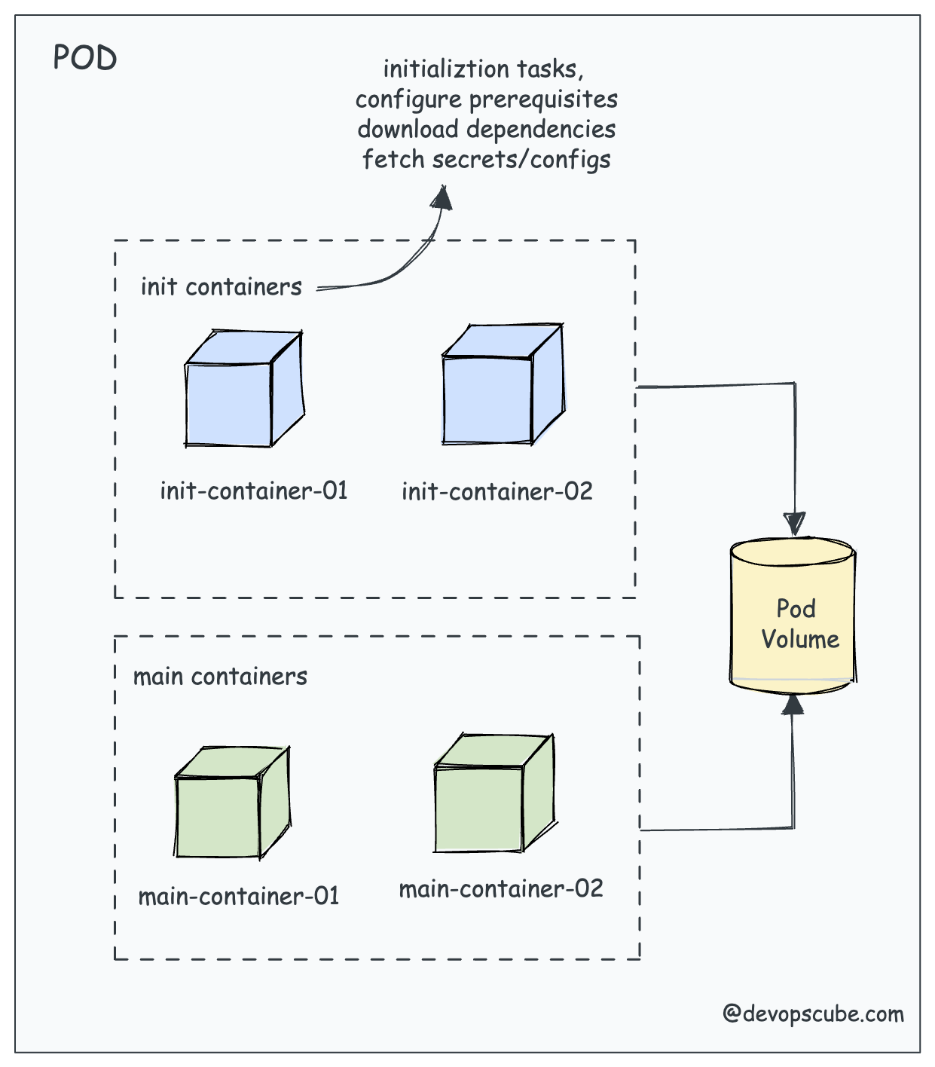
kubectl logs <pod-name>

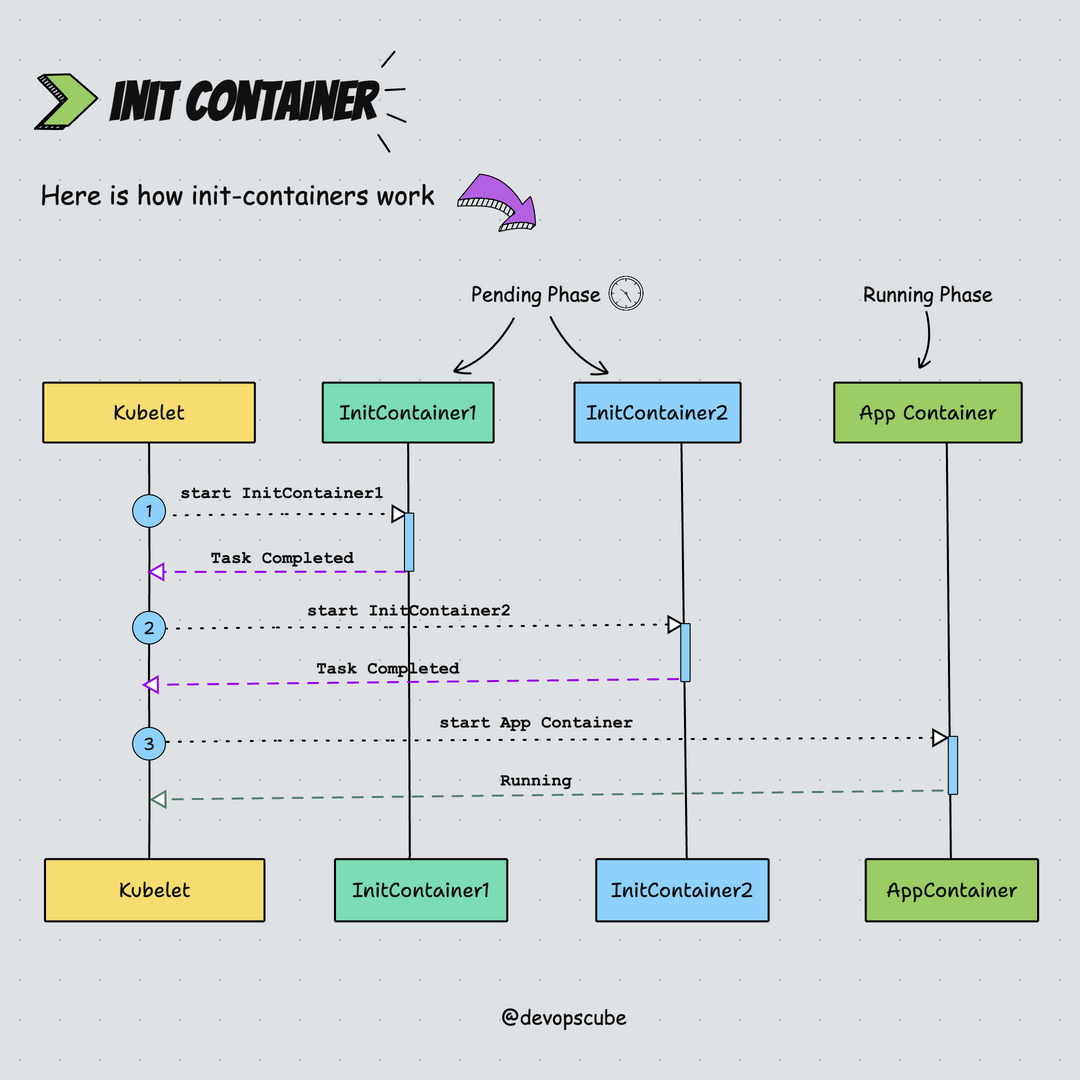
**What is an Init Container?**

**Init containers** are special containers that **run before** the main application container in a Pod.

[Kubernetes pods](https://devopscube.com/kubernetes-pod/) can have more than one container. These containers in a pod work together to achieve a common goal.

Init Containers are containers that start and run to completion before starting the main containers in the pod. It acts as a preparatory step, allowing us to perform initialization tasks, configure prerequisites, or configure dependencies required by the application in the main containers.





🎯 Real-Time Use Case

Let’s say you're deploying a web app that requires:

* A database to be ready.
* Configuration files to be present in a shared volume.

**✅ Init Container Use Cases**

Init containers run before the main app container starts and are useful for setup tasks.

**1. Waiting for External Services**

* **Scenario**: Your application requires a database like PostgreSQL to be ready.
* **Use Case**: Init container uses curl or nc to probe the database endpoint until it's reachable.

**2. Cloning Git Repositories**

* **Scenario**: Application needs config/scripts/code from a Git repo.
* **Use Case**: Init container clones the repo to a shared volume before the app container starts.

**3. Permission Fixes**

* **Scenario**: Mounted volume has wrong permissions.
* **Use Case**: Init container runs chmod or chown on the volume before app starts.

**4. Schema Migration**

* **Scenario**: Web app needs DB schema to be up-to-date.
* **Use Case**: Init container runs DB migration tools like flyway or alembic.

**5. Security Checks or Token Fetch**

* **Scenario**: Your service needs a security token.
* **Use Case**: Init container fetches a token from Azure Key Vault or a HashiCorp Vault and stores it in a shared volume.

**🔄 Sidecar Container Use Cases**

Sidecars run **with** the main container and add functionality like logging, monitoring, or proxying.

**1. Log Forwarding**

* **Scenario**: Centralized log collection needed.
* **Use Case**: Sidecar runs Fluent Bit or Fluentd to ship logs to Azure Monitor, Splunk, or ELK.

**5. Telemetry & Metrics Export**

* **Scenario**: Need custom metrics in Prometheus.
* **Use Case**: Sidecar exposes metrics via a lightweight app (e.g., node-exporter or custom collector).

Lifecycle:

* Init Containers: Run to completion before the main application starts and do not continue running after the main container starts.
* Sidecar Containers: Run alongside the main containers throughout the Pod’s lifecycle, often providing auxiliary functionality like logging, monitoring, or proxying.

Probes:

* Init Containers: Do not support lifecycle probes (livenessProbe, readinessProbe, startupProbe) as they are intended to run only once to completion.
* Sidecar Containers: Support lifecycle probes since they need to stay healthy and ready throughout the Pod’s lifecycle.

Interaction:

* Init Containers: Can prepare the environment or perform tasks like fetching configurations, but they do not interact directly with the running main application containers.
* Sidecar Containers: Continuously interact with the main containers, often providing ongoing services such as log aggregation, monitoring, or configuration management.

Execution Order:

* Init Containers: Must complete sequentially before any main application container starts.
* Sidecar Containers: Start alongside the main containers and run concurrently.

**ReplicaSet Controller → Kitchen Manager**

* 📌 Ensures a fixed number of running pods
* 👨‍🍳 Example: You have 3 chefs working in your restaurant. If one chef quits, the kitchen manager hires another immediately to maintain the count.
* 📌 Purpose: Ensures that a specified number of identical pods are running at all times.
* **🔹 Real-Time Use Case:  
  Imagine you are running a web application with 3 replicas for high availability. If one pod crashes due to a node failure, the ReplicaSet Controller automatically starts a new pod to replace it, ensuring uninterrupted service.**

**Deployment Controller**

* **📌 Purpose: 📌 Manages smooth updates and rollbacks for applications, ensuring zero downtime during deployments.**
* **👨‍🍳 Example: Your restaurant wants to change its menu from "old dishes" to "new dishes" gradually, making sure customers don't face issues. The head chef introduces the new menu step by step.**
* **🔹 Real-Time Use Case:  
  You want to update your application from version 1.0 to version 2.0 without downtime. The Deployment Controller performs a rolling update, gradually replacing old pods with new ones without disrupting users. If an issue occurs, it automatically rolls back to the previous version**

**4️.Service (Exposes Pods for Communication)**

Definition: A Service allows Pods to communicate internally or externally.

* ClusterIP (default) → Internal access.
* NodePort → Exposes service on a fixed port (30000-32767).
* LoadBalancer → Exposes service externally with cloud provider’s load balancer.

**Purpose:**

**📌 Manages smooth updates and rollbacks for applications, ensuring zero downtime during deployments.**

**👨‍🍳 Example: Your restaurant wants to change its menu from "old dishes" to "new dishes" gradually, making sure customers don't face issues. The head chef introduces the new menu step by step.**

**🔹 Real-Time Use Case:  
You want to update your application from version 1.0 to version 2.0 without downtime. The Deployment Controller performs a rolling update, gradually replacing old pods with new ones without disrupting users. If an issue occurs, it automatically rolls back to the previous version.**

apiVersion: apps/v1 # API version for Deployment

kind: Deployment # Specifies this is a Deployment

metadata:

name: nginx-deployment # Name of the Deployment

labels:

app: nginx # Labels help identify the deployment

spec: **# This 'spec' defines the Deployment level**

replicas: 3 # Runs 3 replicas of the application

selector:

matchLabels:

app: nginx # Matches pods with this label

template: # Template for creating pods

metadata:

labels:

app: nginx # Label assigned to created pods

spec:

containers:

- name: nginx-container

image: nginx:latest # Image to be used for the container

ports:

- containerPort: 80 # Port exposed inside the pod

**Spec at the Deployment Level:**

* This spec section defines the desired state of the Deployment itself. It tells Kubernetes how to manage the Deployment and its replicas.
* Purpose: It specifies how many replicas of the Pod you want to run, what selector to use to match the Pods, and the template for creating the Pods. In other words, it tells Kubernetes how to create and manage the Pods that will be running your application.

**Spec at the Pod Template Level (inside template):**

* This spec section is nested inside the Pod template. It describes the configuration of each individual Pod that will be created by the Deployment.
* Purpose: This section defines the configuration of the containers within the Pods, such as:
  + Which image to use (nginx:latest).
  + The container ports to expose.
  + Resource requests and limits.
  + Environment variables.
  + Volumes and volume mounts.
* This spec essentially describes the Pod's container settings, which will be replicated across the Pods managed by the Deployment.

kubectl apply -f nginx-deployment.yaml

kubectl get deployments

kubectl get pods

kubectl get deployments

**🟢 4️.Service (Exposes Pods for Communication)**

**Definition: A Service allows Pods to communicate internally or externally.**

* **ClusterIP (default) → Internal access.**
* **NodePort → Exposes service on a fixed port (30000-32767).**
* **LoadBalancer → Exposes service externally with cloud provider’s load balancer.**

**📌 Example YAML for Service (NodePort)**

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

nodePort: 30080

type: NodePort

**3. StatefulSet Controller**

📌 **Purpose:** Manages stateful applications like databases by maintaining **persistent identity and storage** for each pod.

🔹 **Real-Time Use Case:**  
A **MySQL database** running in Kubernetes must retain its data even if a pod restarts. The **StatefulSet Controller** ensures that each database instance gets a unique identifier and persistent storage so data is not lost.

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: mysql-db

spec:

serviceName: "mysql"

replicas: 2

selector:

matchLabels:

app: mysql

template:

metadata:

labels:

app: mysql

spec:

containers:

- name: mysql

image: mysql:latest

volumeMounts:

- name: mysql-storage

mountPath: /var/lib/mysql

volumeClaimTemplates:

- metadata:

name: mysql-storage

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi

**4. DaemonSet Controller**

📌 **Purpose:** Ensures that a specific pod runs on **every node** in the cluster.

👨‍🍳 **Example:** In a restaurant, **cleaning staff** is needed at every table, regardless of how many tables are occupied.

🔹 **Real-Time Use Case:**  
If you are running a **log collection agent** (e.g., Fluentd or Prometheus Node Exporter), you need it on **every** node. The **DaemonSet Controller** ensures that the logging agent runs on each node automatically.

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: logging-agent

spec:

selector:

matchLabels:

name: logging-agent

template:

metadata:

labels:

name: logging-agent

spec:

containers:

- name: fluentd

image: fluentd:latest

**5. Job Controller**

📌 **Purpose:** Runs short-lived batch jobs and ensures completion.

👨‍🍳 **Example:** A restaurant hosts a **birthday event** where a chef makes a **custom cake** only for that day. Once done, the task is complete.

🔹 **Real-Time Use Case:**  
A company needs to process **a large number of files overnight** (e.g., data migration, backup, or report generation). The **Job Controller** runs a batch job, processes the files, and stops automatically after completion.

apiVersion: batch/v1

kind: Job

metadata:

name: data-processing-job

spec:

template:

spec:

containers:

- name: processor

image: data-processor:latest

restartPolicy: Never

**6. CronJob Controller**

📌 **Purpose:** Runs scheduled jobs at specific times (similar to a Linux cron job).

👨‍🍳 **Example:** Every morning, the chef prepares a **special dish** for the day at exactly 8 AM.

🔹 **Real-Time Use Case:**

**Kubernetes Equivalent:** If you need **automated scheduled tasks** in Kubernetes, **CronJob** helps run them at specific times.

* **Example Task:**
  + Running a **database backup every midnight**
  + Sending an **email report every Friday at 10 AM**
  + **Clearing old logs every 7 days**

📌 **Real-Time Use Case:**

* Running a **data cleanup job** in a banking system every weekend to delete old transactions.

apiVersion: batch/v1

kind: CronJob

metadata:

name: daily-backup

spec:

schedule: "0 0 \* \* \*" # Runs at midnight every day

jobTemplate:

spec:

template:

spec:

containers:

- name: backup

image: backup-script:latest

restartPolicy: OnFailure