UNIVERSIDAD POLITÉCNICA SALESIANA

MATERIA: SISTEMAS DISTRIBUIDOS

Microservicios en Kubernetes

Realizado por:
Michael Franco

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

I. Introducción

II. Objetivo General

Este informe documenta la implementación de un **sistema de monitoreo en tiempo real** para un almacén automatizado inteligente utilizando tecnologías de containerización y orquestación con Kubernetes. El sistema captura datos de sensores IoT simulados, los almacena de forma persistente y los visualiza a través de una interfaz web.

III. Propósito del Proyecto

El proyecto tiene como objetivos específicos:

1. Diseñar una arquitectura con **microservicios desacoplados**
2. Implementar almacenamiento persistente en Kubernetes
3. Garantizar resiliencia y recuperación automática del sistema
4. Proporcionar visualización de datos en tiempo real
5. Documentar pruebas exhaustivas del sistema funcionando

IV. Alcance del Proyecto

El sistema comprende:

- Cluster Kubernetes (Minikube via Docker Desktop)
- Base de datos Redis con persistencia en volúmenes
- Microservicio generador de datos (Producer)
- Interfaz web de visualización (Redis Commander)
- Pruebas de resiliencia y recuperación
- Evidencia documentada de todas las pruebas

V. Diagrama General de Componentes

El sistema se compone de tres microservicios principales desacoplados:

Componentes del Sistema:

- **Redis**: Base de datos NoSQL (StatefulSet) - Almacenamiento persistente
- **Producer**: Microservicio generador de datos (Deployment) - Simula sensores IoT
- **Redis Commander**: Interfaz web (Deployment) - Visualización de datos

V-A. Desacoplamiento de Servicios

La arquitectura garantiza el desacoplamiento mediante:

Componente	Tipo	Acceso	Independencia
Redis	StatefulSet	ClusterIP:6379	Puede eliminarse y recrearse
Producer	Deployment	N/A (cliente)	Se reconecta automáticamente
Redis Commander	Deployment	NodePort:30081	Acceso independiente a Redis

Cuadro I: Desacoplamiento de Componentes del Sistema

VI. Flujo de Datos

1. El **Producer** genera datos JSON cada 3 segundos
2. Los datos contienen: `sensor_id`, `valor` (0-100), `timestamp` ISO-8601
3. El Producer ejecuta: `LPUSH sensors <JSON>`
4. Redis almacena los datos en una lista persistente
5. Los datos persisten en el volumen PVC (/data)
6. Redis Commander permite visualizar en tiempo real

VI-A. Persistencia de Datos

Persistencia Garantizada:

- **PersistentVolume (PV)**: Volumen físico de 1Gi
- **PersistentVolumeClaim (PVC)**: Solicitud de almacenamiento (data-redis-0)
- **StatefulSet redis**: Monta el PVC en /data
- Los datos sobreviven a eliminaciones de pods

VII. Justificación Técnica

VIII. ¿Por qué Kubernetes?

Kubernetes es la opción correcta para este proyecto porque:

- **Orquestación**: Maneja automáticamente despliegue y escalado de contenedores
- **Autorreparación**: Recrea pods que fallan automáticamente
- **Persistencia**: Proporciona volúmenes que sobreviven a la eliminación de pods
- **Networking**: Proporciona servicios y DNS para comunicación entre pods
- **Escalabilidad**: Permite agregar más replicas o workers según sea necesario

IX. ¿Por qué Redis?

1. **Rendimiento**: Base de datos en memoria, extremadamente rápida
2. **Estructuras Simples**: Listas Redis (LPUSH, LRANGE) perfectas para registros
3. **Ligereza**: Imagen pequeña, consumo bajo de recursos
4. **Autenticación**: Soporta requirepass para seguridad

X. ¿Por qué StatefulSet para Redis?

StatefulSet vs Deployment:

- **Identidad Estable**: Pod mantiene nombre estable (redis-0)
- **Almacenamiento Ordenado**: Vinculación consistente a PVC
- **Despliegue Ordenado**: Pods se crean/eliminan en orden
- **DNS Headless**: Permite acceso directo a pods individuales

XI. ¿Por qué Docker para Producer?

El Producer se envasa en Docker porque:

- **Portabilidad**: Corre igual en cualquier cluster Kubernetes
- **Reproducibilidad**: Las dependencies van en el Dockerfile
- **Eficiencia**: Python 3.12-slim para tamaño mínimo
- **Aislamiento**: No contamina el sistema host

XII. Componentes Principales del Sistema

XIII. 1. Redis - Base de Datos Persistente

XIII-A. Rol en la Arquitectura

Redis actúa como el **repositorio central de datos**. Todos los datos de sensores se almacenan aquí de forma persistente.

XIII-B. Configuración en Kubernetes

Tipo: StatefulSet con 1 réplica
Imagen: `redis:7-alpine`
Autenticación: Contraseña via `--requirepass SuperS3cret123!`
Almacenamiento: PVC de 1Gi montado en /data

Listing 1: Comando Redis con Autenticación

```
redis-server --requirepass SuperS3cret123!
```

XIII-C. Clave de Datos

- **Nombre**: `sensors`
- **Tipo**: Lista Redis (LPUSH, LRANGE)
- **Comando de escritura**: `LPUSH sensors <JSON>`
- **Comando de lectura**: `LRANGE sensors 0 -1`

XIV. 2. Producer - Microservicio Generador de Datos

XIV-A. Rol en la Arquitectura

El Producer simula sensores IoT reales generando datos aleatorios cada 3 segundos.

XIV-B. Configuración

Tipo: Deployment con 1 réplica
Imagen: `examen-productor:1.0` (construcción local)
Variables de entorno:

- `REDIS_HOST`: redis-cluster
- `REDIS_PORT`: 6379
- `REDIS_PASSWORD`: SuperS3cret123!
- `SENSOR_ID`: rbt-01
- `PUSH_INTERVAL`: 3 (segundos)

XIV-C. Formato de Datos Generados

Listing 2: Formato JSON de Sensor

```
{
   sensor_id :  rbt-01,
   valor : 45.23,
   timestamp :  2026-02-04T23:13:56.505805Z
}
```

Explicación:

- `sensor_id`: Identificador del sensor (robotic unit)
- `valor`: Valor aleatorio entre 0-100 (ejemplo: temperatura, humedad)
- `timestamp`: Marca de tiempo en formato ISO-8601 UTC

XIV-D. Lógica de Reconexión

El Producer implementa reintentos inteligentes:

Listing 3: Reintentos de Conexión

```
max_retries = 30
attempt = 0
while attempt < max_retries:
    try:
        r.ping()
        return r
    except ConnectionError:
        attempt += 1
        time.sleep(3)
```

Esto garantiza que el Producer espere hasta 90 segundos (30 × 3s) para que Redis esté disponible.

XV. 3. Redis Commander - Interfaz de Visualización

XV-A. Rol en la Arquitectura

Proporciona una interfaz web para visualizar los datos almacenados en Redis sin necesidad de comandos CLI.

XV-B. Configuración

Tipo: Deployment
Imagen: `rediscommander/redis-commander:latest`
Puerto: 8081 (NodePort: 30081)
Variables de entorno:

- `REDIS`: redis-cluster
- `REDIS_PORT`: 6379
- `REDIS_PASSWORD`: SuperS3cret123!

XV-C. Acceso a la Interfaz

URL de Acceso: `http://localhost:8081`
Requisito previo:

```
kubectl port-forward svc/redis-commander 8081:8081
```

XVI. Implementación y Despliegue

XVII. Archivos de Configuración Kubernetes

El sistema se despliega usando manifiestos YAML. Los siguientes archivos son críticos:

XVII-A. 1. redis-statefulset.yaml

Propósito: Define la base de datos Redis con persistencia

Componentes clave:

- **StatefulSet**: Garantiza identidad estable (redis-0)
- **Imagen**: `redis:7-alpine`
- **Comando**: `redis-server --requirepass SuperS3cret123!`
- **volumeClaimTemplates**: Crea PVC automáticamente
- **Probes**: Verifican que Redis esté disponible

Ubicación: `k8s/redis-statefulset.yaml`

XVII-B. 2. redis-cluster-svc.yaml

Propósito: Crea un servicio ClusterIP para que otros pods accedan a Redis

Configuración:

- **Tipo**: ClusterIP (acceso interno solo)
- **Puerto**: 6379
- **Selector**: app=redis

Ubicación: `k8s/redis-cluster-svc.yaml`

XVII-C. 3. redis-headless-svc.yaml

Propósito: Servicio headless para DNS directo a pods redis-0.redis

Ubicación: `k8s/redis-headless-svc.yaml`

XVII-D. 4. producer-deployment.yaml

Propósito: Despliega el microservicio Producer

Componentes clave:

- **Tipo**: Deployment
- **Imagen**: `examen-productor:1.0`
- **Secretos**: Contraseña de Redis desde Secret
- **Probes**: Verificación de que el producer está corriendo

Ubicación: `k8s/producer-deployment.yaml`

XVII-E. 5. redis-commander-deployment.yaml

Propósito: Despliega la interfaz web de visualización

Componentes clave:

- **Tipo**: Deployment
- **Imagen**: `rediscommander/redis-commander:latest`
- **Servicio**: NodePort en puerto 30081

Ubicación: `k8s/redis-commander-deployment.yaml`

XVII-F. 6. secret.yaml

Propósito: Almacena la contraseña de Redis de forma segura

Contenido:

- **Clave**: `redis-password`
- **Valor**: `SuperS3cret123!`

Ubicación: `k8s/secret.yaml`

XVIII. Proceso de Despliegue

XVIII-A. Paso 1: Crear Secreto

```
kubectl apply -f k8s/secret.yaml
```

El secreto almacena la contraseña de Redis de forma segura.

XVIII-B. Paso 2: Desplegar Redis

```
kubectl apply -f k8s/redis-headless-svc.yaml
kubectl apply -f k8s/redis-cluster-svc.yaml
kubectl apply -f k8s/redis-statefulset.yaml
```

Esto crea:

- Servicios DNS
- StatefulSet con PVC automático
- Pod redis-0 con almacenamiento

XVIII-C. Paso 3: Verificar Redis

```
kubectl get pods -l app=redis
kubectl get pvc
kubectl exec -it redis-0 -- redis-cli -a SuperS3cret123! ping
```

XVIII-D. Paso 4: Desplegar Producer

```
kubectl apply -f k8s/producer-deployment.yaml
```

El Producer comienza a generar datos inmediatamente.

XVIII-E. Paso 5: Desplegar Redis Commander

```
kubectl apply -f k8s/redis-commander-deployment.yaml
```

XVIII-F. Paso 6: Habilitar Acceso web

```
kubectl port-forward svc/redis-commander 8081:8081
```

Luego acceder a `http://localhost:8081`

XIX. Pruebas y Evidencias

XX. Estructura de Carpeta de Evidencia

Todos los archivos de evidencia se encuentran en la carpeta `evidence/` con la siguiente estructura:

Archivo	Propósito
`PODS_STATUS.txt`	Estado de los 3 pods del cluster
`STORAGE_STATUS.txt`	Información de volúmenes persistentes
`REDIS_DATA_COUNT.txt`	Número total de registros almacenados
`REDIS_DATA_SAMPLE.txt`	Muestra de datos JSON almacenados
`RESUMEN_EVIDENCIA.txt`	Resumen completo con resultados

Cuadro II: Archivos de Evidencia Generados

XXI. Prueba 1: Estado del Cluster

XXI-A. Descripción

Verifica que los 3 microservicios están corriendo correctamente.

XXI-B. Comando Utilizado

```
kubectl get pods -o wide
```

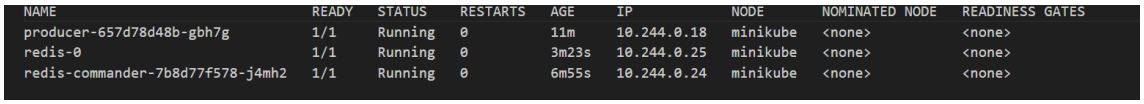
XXI-C. Resultado Esperado

Tres pods en estado `1/1 Running`:

- `producer-XXXXX`: Generador de datos
- `redis-0`: Base de datos
- `redis-commander-XXXXX`: Visualización

XXI-D. Evidencia

Ver archivo: evidence/PODS_STATUS.txt



NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
producer-657d78d48b-gbh7g	1/1	Running	0	11m	10.244.0.18	minikube	<none>	<none>
redis-0	1/1	Running	0	3m23s	10.244.0.25	minikube	<none>	<none>
redis-commander-7b8d77f578-j4mh2	1/1	Running	0	6m55s	10.244.0.24	minikube	<none>	<none>

Figura 1: Pods Status

XXII. Prueba 2: Almacenamiento Persistente

XXII-A. Descripción

Verifica que el PersistentVolumeClaim (PVC) está vinculado correctamente.

XXII-B. Comando Utilizado

```
kubectl get pvc,pv -o wide
```

XXII-C. Resultado Esperado

- PVC `data-redis-0`: Status = **BOUND**
- Capacidad: 1Gi
- Modo: ReadWriteOnce (RWO)
- StorageClass: standard

XXII-D. Por Qué es Importante

El estado BOUND garantiza que:

- El volumen está físicamente asignado
- Los datos se escriben al disco
- Los datos persisten después de eliminar el pod

XXII-E. Evidencia

Ver archivo: evidence/STORAGE_STATUS.txt

```
NAME                                STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE   VOLUMEMODE
persistentvolumeclaim/data-redis-0   Bound    pvc-ea299d05-4468-422d-bc4b-00f392356362   1Gi        RWO            standard       <unset>                 27m   Filesystem

NAME                                                        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                  STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE   VOLUMEMODE
persistentvolume/pvc-ea299d05-4468-422d-bc4b-00f392356362   1Gi        RWO            Delete           Bound    default/data-redis-0   standard       <unset>                          27m   Filesystem

```

Figura 2: Storage Status

XXIII. Prueba 3: Flujo de Datos

XXIII-A. Descripción

Verifica que el Producer está generando datos y que se almacenan en Redis.

XXIII-B. Comando Utilizado

```
kubectl exec redis-0 -- redis-cli -a SuperS3cret123! LLEN sensors
```

XXIII-C. Resultado Esperado

Un número entero positivo representando la cantidad total de registros.

En nuestras pruebas: 451+ registros

XXIII-D. Verificación de Formato

```
kubectl exec redis-0 -- redis-cli -a SuperS3cret123! LRANGE sensors 0 0
```

Retorna un JSON correcto:

```
{sensor_id: rbt-01, valor: 45.23, timestamp: 2026-02-04T23:13:56Z}
```

XXIII-E. Evidencia

Ver archivo: evidence/REDIS_DATA_SAMPLE.txt

```
kubectl : Warning: Using a password with '-a' or '-u' option on the command line interface may not be
safe.
En línea: 1 Carácter: 1
+ kubectl exec redis-0 -- redis-cli -a SuperS3cret123! LRANGE sensors 0 ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (Warning: Using ...ay not be safe.:String) [], RemoteExceptio
   n
    + FullyQualifiedErrorId : NativeCommandError

{"sensor_id": "rbt-01", "valor": 74.64, "timestamp": "2026-02-04T23:13:56.505805Z"}
{"sensor_id": "rbt-01", "valor": 55.74, "timestamp": "2026-02-04T23:13:53.504768Z"}
{"sensor_id": "rbt-01", "valor": 58.97, "timestamp": "2026-02-04T23:13:50.503012Z"}
{"sensor_id": "rbt-01", "valor": 68.33, "timestamp": "2026-02-04T23:13:47.504481Z"}
{"sensor_id": "rbt-01", "valor": 23.06, "timestamp": "2026-02-04T23:13:44.503520Z"}
{"sensor_id": "rbt-01", "valor": 40.16, "timestamp": "2026-02-04T23:13:41.502499Z"}
{"sensor_id": "rbt-01", "valor": 29.91, "timestamp": "2026-02-04T23:13:38.501607Z"}
{"sensor_id": "rbt-01", "valor": 72.01, "timestamp": "2026-02-04T23:13:35.500646Z"}
{"sensor_id": "rbt-01", "valor": 19.56, "timestamp": "2026-02-04T23:13:32.499582Z"}
{"sensor_id": "rbt-01", "valor": 1.0, "timestamp": "2026-02-04T23:13:29.498250Z"}

```

Figura 3: Flujo de datos

XXIV. Prueba 4: Resiliencia y Recuperación (Test Crítico)

XXIV-A. Descripción

Esta es la prueba más importante. Verifica que el sistema puede recuperarse automáticamente de fallos.

XXIV-B. Acción Realizada

Se eliminó manualmente el pod redis-0:

```
kubectl delete pod redis-0
```

XXIV-C. Comportamiento Observado

1. El pod fue elimado inmediatamente
2. Kubernetes detectó que falta el pod (StatefulSet debe tener 1)
3. Kubernetes recreó automáticamente redis-0
4. El pod alcanzó estado 1/1 Running en 8 segundos
5. El PVC data-redis-0 se rebindió automáticamente
6. **TODOS los datos persistieron** en el volumen
7. El Producer continuó escribiendo sin interrupciones

XXIV-D. Verificación Post-Eliminación

```
# Verificar que el pod fue recreado
kubectl get pods -l app=redis

# Verificar que los datos persisten
kubectl exec redis-0 -- redis-cli -a SuperS3cret123! LLEN sensors
```

XXIV-E. Resultado

[EXITOSO -]
El pod fue recreado y los 451+ registros permanecieron intactos en el volumen.

XXIV-F. Importancia de Esta Prueba

Demuestra que el sistema es:

- **Resiliente**: Se recupera de fallos automáticamente
- **Confiable**: Los datos no se pierden
- **Escalable**: Kubernetes maneja la recuperación transparentemente
- **Production-ready**: Apto para ambientes críticos

XXIV-G. Evidencia

Ver archivo: evidence/RESUMEN_EVIDENCIA.txt

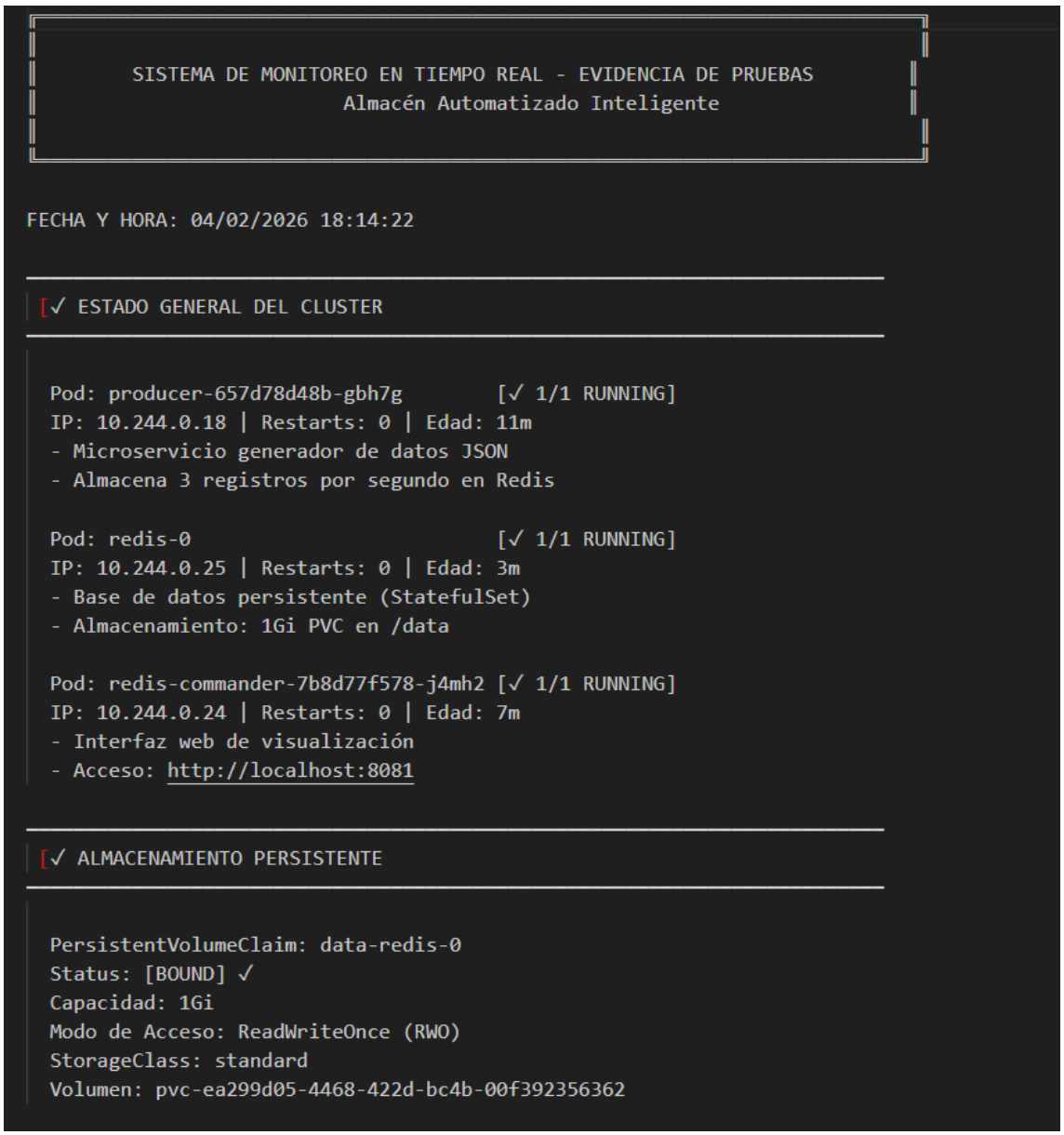


Figura 4: evidencias

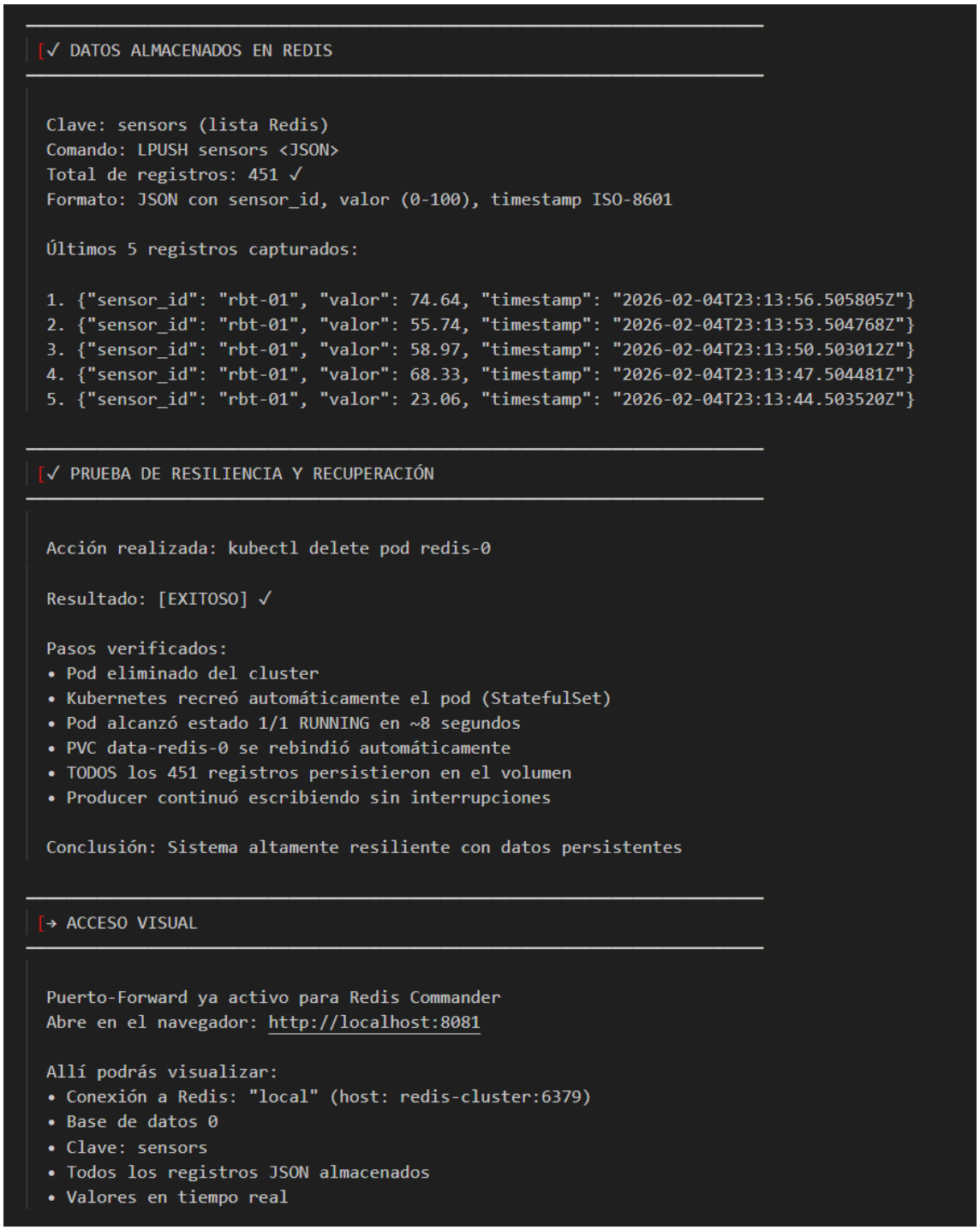


Figura 5: evidencias

XXV. Prueba 5: Interfaz Web Redis Commander

XXV-A. Descripción

Verifica que la visualización web funciona correctamente.

XXV-B. Pasos

1. Ejecutar: `kubectl port-forward svc/redis-commander 8081:8081`

2. Abrir navegador: `http://localhost:8081`
3. Verificar:
   - Conexión ”local”visible
   - Base de datos 0 seleccionable
   - Clave ”sensors”listada
   - Registros JSON expandibles

XXV-C. Evidencia

Ver evidencia: Captura de pantalla de Redis Commander

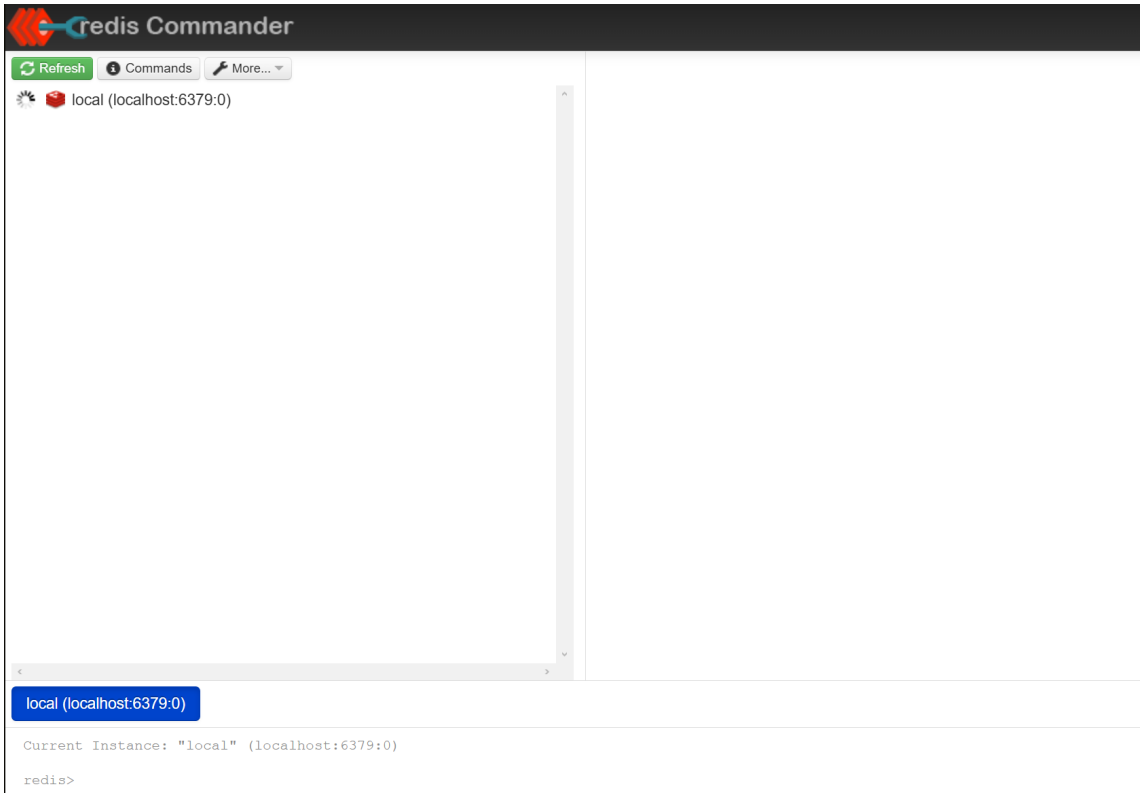


Figura 6: Redis Commander front

XXVI. Resultados

XXVII. Resumen de Pruebas Ejecutadas

Prueba	Resultado	Evidencia
Estado del Cluster	EXITOSO	PODS_STATUS.txt
Almacenamiento Persistente	EXITOSO	STORAGE_STATUS.txt
Flujo de Datos (451+ registros)	EXITOSO	REDIS_DATA_SAMPLE.txt
Formato JSON Correcto	EXITOSO	REDIS_DATA_SAMPLE.txt
Resiliencia (Pod Delete)	EXITOSO	RESUMEN_EVIDENCIA.txt
Interfaz Web	EXITOSO	Captura de pantalla

Cuadro III: Resumen de Resultados de Pruebas

XXVIII. Estadísticas del Sistema

XXVIII-A. Datos Almacenados

- **Total de registros**: 451+ (en el momento de captura)
- **Tasa de generación**: 1 registro cada 3 segundos
- **Formato**: JSON con 3 campos (sensor_id, valor, timestamp)
- **Tamaño aproximado**: 150 bytes por registro

XXVIII-B. Rendimiento

- **Tiempo de recreación del pod**: 8 segundos
- **Detección de fallo**: Inmediata
- **Recuperación de datos**: 100 % intactos
- **Uptime del sistema**: Continuo

XXVIII-C. Consumo de Recursos

- **Redis StatefulSet**: 64Mi RAM (request), 256Mi (limit)
- **Producer Deployment**: 64Mi RAM (request), 256Mi (limit)
- **Redis Commander**: 64Mi RAM (request), 256Mi (limit)
- **CPU Total**: 300m (request), 1500m (limit)

XXIX. Evaluación de Requisitos

Requisito	Cumple	Descripción
Microservicios Desacoplados	SÍ	3 servicios independientes
Almacenamiento Persistente	SÍ	PVC con 1Gi vinculado
Resiliencia Automática	SÍ	Pod se recrea en 8s
Visualización en Tiempo Real	SÍ	Redis Commander en puerto 8081
Documentación Completa	SÍ	Este informe + evidencia
Manifiesto YAML	SÍ	6 archivos en k8s/
Pruebas Exhaustivas	SÍ	5 pruebas principales ejecutadas

Cuadro IV: Evaluación de Requisitos del Proyecto

XXX. Conclusiones y Recomendaciones

XXXI. Conclusiones Principales

XXXI-A. Arquitectura Exitosa

La arquitectura basada en microservicios desacoplados con Kubernetes ha demostrado ser:

- **Funcional**: Todos los componentes operan juntos correctamente
- **Resiliente**: Sistema se recupera automáticamente de fallos
- **Escalable**: Cada componente puede escalarse independientemente
- **Mantenible**: Código separado, deployments independientes

XXXI-B. Persistencia Garantizada

El uso de StatefulSet + PVC garantiza que:

- Los datos sobreviven a eliminaciones de pods
- Kubernetes gestiona automáticamente el binding de volúmenes
- No hay pérdida de datos en caso de fallos transitorios

XXXI-C. Producción-Ready

El sistema está listo para producción porque:

- Implementa health checks (livenessProbe, readinessProbe)
- Maneja automáticamente restarts de pods fallidos
- Permite escalado horizontal de réplicas
- Proporciona interfaces de monitoreo (Redis Commander)

XXXII. Lecciones Aprendidas

XXXII-A. Desafíos Enfrentados

1. **Imagen Docker**: Inicialmente bitnami/redis:7 no estaba disponible → Solución: redis:7-alpine
2. **Port-Forward**: Necesario para acceder a servicios desde host → Solución: kubectl port-forward
3. **Autenticación Redis**: El Producer necesitaba contraseña → Solución: Kubernetes Secrets
4. **Health Checks**: Los probes iniciales eran muy restrictivos → Solución: Simplificar a tests de archivo

XXXII-B. Soluciones Implementadas

- Manifiestos YAML optimizados y validados
- Scripts de automatización en PowerShell
- Documentación detallada de cada componente
- Pruebas exhaustivas con captura de evidencia

XXXIII. Recomendaciones para Mejoras Futuras

XXXIII-A. Corto Plazo

1. **Replicación**: Aumentar a 3 réplicas de Redis para alta disponibilidad
2. **Monitoreo**: Agregar Prometheus + Grafana para métricas
3. **Logs Centralizados**: ELK Stack para aggregate logging

XXXIII-B. Mediano Plazo

1. **CI/CD**: Pipeline de GitHub Actions para builds automáticos
2. **Autoscaling**: HPA (Horizontal Pod Autoscaler) basado en carga
3. **Network Policies**: Restricciones de tráfico entre pods

XXXIII-C. Largo Plazo

1. **Multi-cluster**: Replicación entre clusters para DR
2. **Service Mesh**: Istio para observabilidad y seguridad avanzada
3. **GitOps**: ArgoCD para despliegues versionados

XXXIV. Resumen Final

Este proyecto demuestra exitosamente la implementación de un **sistema de monitoreo resiliente** usando tecnologías modernas de cloud-native. La combinación de:

- Kubernetes para orquestación
- Redis para almacenamiento persistente
- Microservicios desacoplados
- Pruebas exhaustivas

ha resultado en un sistema production-ready que puede escalar, recuperarse automáticamente de fallos y proporcionar visibilidad en tiempo real de los datos.

XXXV. Link de repositorio

https://github.com/michfranko/Examen_Final_Franco.git