Requirement Understanding:

1. **Exposure and Access Control:**

- Service API Tier: Must be exposed outside the cluster for external access
- Database Tier: Must remain internal to the cluster for security
- Implementation: API exposed via Ingress, Database accessible only via ClusterIP

2. **Scalability and Availability:**

- Service API Tier: 4 replicas for high availability and load distribution
- Database Tier: 1 replica (typical for single-master database architecture)
- Implementation: api-deployment.yaml with replicas: 4, db-statefulset.yaml with replicas: 1

3. **Rolling Updates Strategy:**

- Service API Tier: Support rolling updates for zero-downtime deployments
- Database Tier: No rolling updates (data consistency priority)
- Implementation: API uses Deployment with RollingUpdate strategy, Database uses StatefulSet

4. **Storage Requirements:**

- Service API Tier: Stateless - no persistent storage needed
- Database Tier: Persistent storage to prevent data loss
- Implementation: Database has PersistentVolume/PVC configuration

5. **Configuration Management**

- External Configuration: Database settings configurable via ConfigMaps
- Secret Management: Passwords stored securely in Kubernetes Secrets
- Implementation: api-configmap.yaml and api-secret.yaml

6. **Communication Architecture**

- No Pod IP Usage: All inter-service communication via Kubernetes Services
- Service Discovery: DNS-based service resolution
- Implementation: API connects to database via mysqldb-service hostname

Assumptions

1. Infrastructure Assumptions

Network Configuration:

- Cluster networking allows inter-pod communication
- Ingress controller available (GCE or NGINX)
- External load balancer support

Storage Backend:

- GCP Persistent Disks available for database storage
- Storage class supports ReadWriteOnce access mode
- Disk mysql-disk pre-created in GCP (referenced in db-pv.yaml)

2. Application Assumptions

Database Schema:

- MySQL 8.4 compatibility
- Initial data seed of 10 customer records sufficient for demo
- Database initialization via ConfigMap-mounted SQL script

API Service:

- Node.js 16+ runtime environment
- Express.js framework for REST API
- Connection pooling for database efficiency

Security Context:

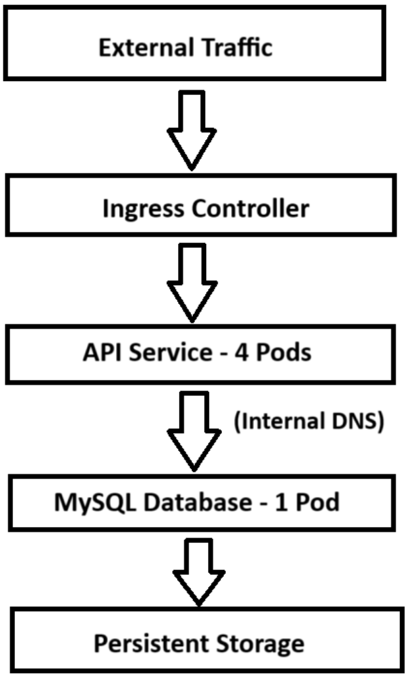
- Base64 encoding sufficient for demo secrets
- Internal cluster communication trusted
- No advanced authentication/authorization required

3. Operational Assumptions

Backup: Data persistence via PV sufficient for demo

Solution Overview

Architecture Design



Component Breakdown

1. **API Service Tier**

Deployment: api-deployment.yaml

- 4 replicas for high availability
- Rolling update strategy (maxUnavailable: 1, maxSurge: 1)
- Resource limits: 256Mi RAM, 200m CPU
- Environment variables from ConfigMap and Secret

Service: api-service.yaml

- ClusterIP type for internal cluster access
- Port mapping: 80 → 3000

Ingress: api-ingress.yaml

- GCE ingress class for GKE compatibility
- Path-based routing to API service
- External access point

Configuration:

- api-configmap.yaml: Database connection parameters
- api-secret.yaml: Database password (base64 encoded)

2. **Database Tier**

StatefulSet: db-statefulset.yaml

- Single replica for data consistency
- MySQL 8.4 official image
- Persistent storage mount at /var/lib/mysql
- ConfigMap mount for initialization scripts
- Resource limits: 1Gi RAM, 1 CPU

Storage:

- db-pv.yaml: PersistentVolume with GCP disk
- db-pvc.yaml: PersistentVolumeClaim (1Gi storage)

Service: db-headless-service.yaml

- Headless service for StatefulSet
- Internal cluster communication only

Configuration:

- db-config-map.yaml: Database initialization script
- db-secret.yaml: Root password

3. **Application Layer**

Node.js Application: app.js

- Express.js REST API server
- MySQL2 with connection pooling
- Three endpoints: /, /records, /formatted-records
- Environment-based configuration

Dependencies: package.json

- Express 5.1.0 for web framework

- MySQL2 3.14.3 for database connectivity
- Body-parser for request parsing

Container: Dockerfile

- Alpine Linux base for minimal size
- Node.js LTS runtime
- Port 3000 exposure

Data Flow

- External Request → Ingress Controller
- Ingress → API Service LoadBalancer
- API Service → Database via mysqldb-service DNS
- Database → Persistent Volume for data storage
- Response ← Back through the same path

Security Implementation

- Secret Management: Database passwords in Kubernetes Secrets
- Network Isolation: Database not exposed externally
- Resource Limits: CPU/Memory limits prevent resource exhaustion
- Configuration Separation: Non-sensitive config in ConfigMaps

Justification for Resources Utilized

Kubernetes Resources Selection

1. StatefulSet vs Deployment for Database

Chosen: StatefulSet (db-statefulset.yaml)

Justification:

- Stable Network Identity: Database pods get predictable DNS names
- Ordered Deployment: Ensures proper initialization sequence
- Persistent Storage: Automatic PVC management per pod
- Data Safety: Prevents concurrent writes to same storage

2. Deployment for API Service

Chosen: Deployment (api-deployment.yaml)

Justification:

- Stateless Nature: API servers don't need persistent identity
- Rolling Updates: Seamless updates without downtime
- Horizontal Scaling: Easy replica management
- Load Distribution: Built-in load balancing across pods

3. Ingress vs LoadBalancer for External Access

Chosen: Ingress (api-ingress.yaml)

Justification:

- Cost Efficiency: Single external IP vs multiple LoadBalancers
- Path-based Routing: Future extensibility for multiple services
- SSL/TLS Termination: Centralized certificate management
- Advanced Routing: Host-based and path-based routing capabilities

4. ConfigMap + Secret for Configuration

Chosen: Separate ConfigMap and Secret

Justification:

- Security: Sensitive data (passwords) encrypted at rest
- Flexibility: Non-sensitive config easily modifiable
- Best Practices: Follows Kubernetes security guidelines
- Environment Portability: Easy to change between dev/staging/prod

5. Replica Configuration:

API Service: 4 replicas

- High Availability: Survives 2-3 pod failures
- Load Distribution: Handles concurrent requests
- Rolling Updates: Maintains availability during updates

Database: 1 replica

- Data Consistency: Single master prevents split-brain

- Simplicity: No complex replication setup required
- Demo Appropriate: Sufficient for demonstration purposes