Go Apps on Kubernetes

Production Best Practices

v1.0.0

Pocketbook Series

by Luca Sepe

LS 71

About This Pocketbook

Why:

Provide a practical production baseline for building and operating Go services on Kubernetes.

This pocketbook is for:

- Go backend engineers shipping containerized APIs/services.
- Platform/SRE teams defining shared production guardrails.
- Tech leads standardizing reliability and operability across services.

What you get:

- Essential defaults for probes, shutdown, timeouts, retries, and runtime limits.
- Reusable snippets for observability, deployment safety, and incident triage.
- Checklists to validate readiness before production rollout.

Scope

- Kubernetes-ready application behavior and service runtime practices.
- Day-1 and Day-2 operational reliability for HTTP-centric microservices.

Out of scope:

- Deep Kubernetes platform administration (cluster provisioning/CNI/storage internals).
- Service-specific business logic and domain-level architecture choices.

How to use it

1. Apply the baseline templates first.
2. Adapt values to your SLOs, traffic profile, and dependency behavior.
3. Keep changes explicit, measured, and testable under load/failure.

HTTP Server Hardening

Why:

> Explicit server limits reduce resource exhaustion and slow-client attack surface.

- Never run `http.ListenAndServe` without timeouts.
- Configure `ReadHeaderTimeout`, `ReadTimeout`, `WriteTimeout`, `IdleTimeout`.
- Set `MaxHeaderBytes` to prevent header abuse.
- Keep handlers context-aware and cancellable.

```
srv := &http.Server{
    Addr:              cfg.HTTPAddr,
    Handler:           mux,
    ReadHeaderTimeout: 5 * time.Second,
    ReadTimeout:       15 * time.Second,
    WriteTimeout:      30 * time.Second,
    IdleTimeout:       60 * time.Second,
    MaxHeaderBytes:    1 << 20, // 1 MiB
}
```

Prometheus Metrics Endpoint (Go)

Why:

> Stable low-cardinality metrics keep monitoring accurate, affordable, and queryable at scale.

- Keep Prometheus scraping simple: one `/metrics` endpoint.
- Prefer stable metric names and low-cardinality labels.
- Never put user IDs, emails, tokens, or raw URLs in labels.

```
import (
   "net/http"
   "github.com/prometheus/client_golang/prometheus/promhttp"
)

mux := http.NewServeMux()
mux.Handle("/metrics", promhttp.Handler())
```

Label Hygiene

- Good labels: `method`, `route`, `status_class`, `dependency`.
- Risky labels: `user_id`, `session_id`, full path/query string.

Error Handling and Retries

Why:

Bounded retries with jitter improve resilience without amplifying upstream outages.

When:

Retry only transient and idempotent operations with explicit caps.

- Return explicit errors with context
  - `fmt.Errorf("...: %w", err)`
- Retry only transient failures and use bounded exponential backoff + jitter.
- Respect idempotency before retrying writes.
- Use circuit breakers/bulkheads for fragile dependencies.

Generic Retry Utility (Go)

```
type RetryPolicy struct {
   MaxAttempts int
   BaseDelay   time.Duration
   MaxDelay    time.Duration
}

func DoWithRetry(
   ctx context.Context,
   p RetryPolicy,
   operation func(context.Context) error,
   shouldRetry func(error) bool,
) error {
   if p.MaxAttempts < 1 {
      p.MaxAttempts = 1
   }

   if p.BaseDelay <= 0 {
      p.BaseDelay = 100 * time.Millisecond
   }

   if p.MaxDelay < p.BaseDelay {
      p.MaxDelay = p.BaseDelay
   }

   var lastErr error
   for attempt := 1; attempt <= p.MaxAttempts; attempt++ {
      if err := operation(ctx); err == nil {
         return nil
      } else {
         lastErr = err
      }

      if attempt == p.MaxAttempts || !shouldRetry(lastErr) {
         break
      }

      backoff := p.BaseDelay << (attempt - 1)
      if backoff > p.MaxDelay {
         backoff = p.MaxDelay
      }

      jitter := time.Duration(rand.Int63n(int64(backoff / 2)))
      wait := backoff/2 + jitter
```

```
            timer := time.NewTimer(wait)
            select {
            case <-ctx.Done():
               timer.Stop()
               return fmt.Errorf("retry canceled: %w", ctx.Err())
            case <-timer.C:
            }
         }

         return fmt.Errorf("failed after %d attempts: %w", p.
           MaxAttempts, lastErr)
      }
```

Example usage:

```
err := DoWithRetry(ctx, RetryPolicy{
   MaxAttempts: 4,
   BaseDelay:   100 * time.Millisecond,
   MaxDelay:    2 * time.Second,
}, callUpstream, isTransient)
```

`isTransient` example:

```
func isTransient(err error) bool {
   if err == nil {
      return false
   }

   if errors.Is(err, context.DeadlineExceeded) {
      return true
   }

   var netErr net.Error
   if errors.As(err, &netErr) && netErr.Timeout() {
      return true
   }

   var httpErr interface{ StatusCode() int }
   if errors.As(err, &httpErr) {
      code := httpErr.StatusCode()
      return code == 429 || code == 502 || code == 503 || code
     == 504
   }

   return false
}
```

Retryability Quick Map

Condition	Retry?	Note
`429 Too Many Requests`	Yes	Respect `Retry-After` when available
`502/503/504`	Yes	Transient upstream/gateway failures
Timeout / temporary network error	Yes	Use bounded attempts and jitter
`400 Bad Request`	No	Usually caller/input bug
`401/403`	No	Authn/authz issue, needs fix or refresh
`404 Not Found`	Usually no	Retry only if eventual consistency is expected

CI Commands (Go)

Why:

Fast CI quality gates catch correctness, race, and security issues before deployment.

```
go test ./...
go test -race ./...
go vet ./...
staticcheck ./...
govulncheck ./...
```

Final Practical Advice

- Keep each endpoint and dependency budgeted with timeout + retry policy.
- Prefer predictable degradation (readiness false) over random failures.
- Design shutdown and startup behavior as part of the API contract.