Advanced Golang Microservice with Concurrency, Context, Generics, and Clean Architecture

Keywords: Advanced Golang Code Sample, Go Concurrency, Golang Microservice Architecture, Go Generics, Golang RESTful API, Context in Go, Channel Patterns, Middleware in Go, Clean Architecture Go, Interface-driven Golang

---

Project Overview

We're building a **high-performance RESTful Golang microservice** using **Clean Architecture** principles. The service will manage a pool of **asynchronous task executions**, providing **concurrent processing** using **Goroutines**, **Context for cancellation**, and **channels for safe communication**. The service will expose endpoints to **submit**, **monitor**, and **cancel tasks**, with **rate-limiting**, **logging middleware**, and **observability metrics** included.

Project Structure (Clean Architecture)

```
go-advanced-service/

│

├── cmd/

│   └── main.go            # Entry point

│

├── internal/

│   ├── api/              # REST Handlers

│   ├── service/            # Business Logic

│   ├── domain/             # Interfaces & Domain Models

│   ├── infra/             # Implementations (e.g., TaskRunner)
│   └── middleware/         # Logging, Recovery, Rate Limiting
│
├── pkg/
│   └── utils/              # Shared utilities
│
└── go.mod
```

---

main.go (Bootstrapping Server with Context)

```go
package main

import (
    "context"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

    "go-advanced-service/internal/api"
)

func main() {
    srv := api.NewServer()
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    go func() {
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("Server failed: %v", err)
        }
    }()

    log.Println("Server is running on port 8080")

    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
    <-quit

    log.Println("Shutting down gracefully...")
    ctxTimeout, cancel := context.WithTimeout(ctx, 10*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctxTimeout); err != nil {
        log.Fatalf("Shutdown failed: %v", err)
    }

    log.Println("Server stopped cleanly")
}
```

REST API Layer (internal/api/server.go)

```
package api


import (
    "net/http"


    "github.com/go-chi/chi/v5"
    "github.com/go-chi/chi/v5/middleware"


    "go-advanced-service/internal/service"
    "go-advanced-service/internal/middleware/logger"
    "go-advanced-service/internal/middleware/ratelimit"
)


func NewServer() *http.Server {
    r := chi.NewRouter()
    r.Use(middleware.Recoverer)
    r.Use(logger.LoggingMiddleware)
    r.Use(ratelimit.Limiter(10)) // 10 requests/sec


    taskService := service.NewTaskService()


    r.Post("/task", taskService.SubmitHandler)
    r.Get("/task/{id}", taskService.StatusHandler)
    r.Delete("/task/{id}", taskService.CancelHandler)
    return &http.Server{
        Addr:    ":8080",
        Handler: r,
    }
}
```

Business Logic (internal/service/task_service.go)

```
package service

import (
    "encoding/json"
    "net/http"
    "sync"
    "time"
    "github.com/google/uuid"
    "go-advanced-service/internal/domain"
    "go-advanced-service/internal/infra"
)

type TaskService struct {
    runner domain.TaskRunner
    tasks  map[string]*domain.Task
    mu     sync.RWMutex
}
```

```
func NewTaskService() *TaskService {
    return &TaskService{
        runner: infra.NewRunner(),
        tasks:  make(map[string]*domain.Task),
    }
}

func (ts *TaskService) SubmitHandler(w http.ResponseWriter, r *http.Request) {
    id := uuid.New().String()
    ctx, cancel := r.Context(), r.Context().Done()

    task := domain.NewTask(id)
    ts.mu.Lock()
    ts.tasks[id] = task
    ts.mu.Unlock()

    go ts.runner.Run(ctx, task)

    json.NewEncoder(w).Encode(map[string]string{"task_id": id})
}

func (ts *TaskService) StatusHandler(w http.ResponseWriter, r *http.Request) {
    id := chi.URLParam(r, "id")
    ts.mu.RLock()
    task, ok := ts.tasks[id]
    ts.mu.RUnlock()

    if !ok {
        http.Error(w, "Task not found", http.StatusNotFound)
        return
    }

    json.NewEncoder(w).Encode(task)
}

func (ts *TaskService) CancelHandler(w http.ResponseWriter, r *http.Request) {
    id := chi.URLParam(r, "id")
    ts.mu.Lock()
    defer ts.mu.Unlock()
    if task, ok := ts.tasks[id]; ok {
        task.Cancel()
        json.NewEncoder(w).Encode(map[string]string{"status": "cancelled"})
    } else {
        http.Error(w, "Task not found", http.StatusNotFound)
    }
}
```

Domain Interface (internal/domain/task.go)

```
package domain
```

```
import (
    "context"
    "sync"
    "time"
)

type TaskStatus string

const (
    StatusPending TaskStatus = "PENDING"
    StatusRunning TaskStatus = "RUNNING"
    StatusDone    TaskStatus = "DONE"
    StatusCancelled TaskStatus = "CANCELLED"
)

type Task struct {
    ID        string     `json:"id"`
    Status    TaskStatus `json:"status"`
    StartedAt time.Time  `json:"started_at,omitempty"`
    EndedAt   time.Time  `json:"ended_at,omitempty"`
    cancel    context.CancelFunc
    mu        sync.Mutex
}
```

```
func NewTask(id string) *Task {
    return &Task{
        ID:     id,
        Status: StatusPending,
    }
}

func (t *Task) Run(ctx context.Context, duration time.Duration) {
    ctx, cancel := context.WithTimeout(ctx, duration)
    t.cancel = cancel

    t.mu.Lock()
    t.Status = StatusRunning
    t.StartedAt = time.Now()
    t.mu.Unlock()

    select {
    case <-ctx.Done():
        t.mu.Lock()
        t.Status = StatusCancelled
        t.EndedAt = time.Now()
        t.mu.Unlock()
    case <-time.After(duration):
        t.mu.Lock()
        t.Status = StatusDone
        t.EndedAt = time.Now()
        t.mu.Unlock()
    }
}

func (t *Task) Cancel() {
    if t.cancel != nil {
        t.cancel()
    }
}

type TaskRunner interface {
    Run(ctx context.Context, task *Task)
}
```

Task Runner Implementation (internal/infra/runner.go)

```
package infra

import (
    "context"
    "time"

    "go-advanced-service/internal/domain"
)
```

```go
type runner struct{}

func NewRunner() *runner {
    return &runner{}
}

func (r *runner) Run(ctx context.Context, task *domain.Task) {
    // Simulate long-running task
    task.Run(ctx, 5*time.Second)
}
```

Logging Middleware (internal/middleware/logger/logger.go)

```go
package logger

import (
    "log"
    "net/http"
    "time"
)

func LoggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        log.Printf("Started %s %s", r.Method, r.URL.Path)
        next.ServeHTTP(w, r)
        log.Printf("Completed %s in %v", r.URL.Path, time.Since(start))
    })
}
```

Rate Limiting Middleware (internal/middleware/ratelimit/ratelimit.go)

```
package ratelimit


import (
    "net/http"
    "time"


    "golang.org/x/time/rate"
)


func Limiter(rps int) func(http.Handler) http.Handler {
    limiter := rate.NewLimiter(rate.Limit(rps), rps)


    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if !limiter.Allow() {
                http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
                return
            }
            next.ServeHTTP(w, r)
        })
    }
}
```

High-Ranked Keywords Used

- **Advanced Golang Code Example**
- **Golang Microservice Architecture**
- **RESTful API in Go**
- **Goroutines and Channels**
- **Context Cancellation in Go**
- **Rate Limiting in Golang**
- **Interface-Driven Design**
- **Logging Middleware in Go**
- **Task Scheduling in Golang**
- **High-Concurrency Server in Go**
- **Clean Architecture in Golang**

Features Demonstrated

Feature	Implementation Location
Context-aware execution	domain.Task.Run, main.go
Concurrency with Goroutines	TaskService.SubmitHandler
Rate Limiting	ratelimit.Limiter
Logging middleware	logger.LoggingMiddleware

Feature	**Implementation Location**
Modular Clean Architecture	Full project layout
Task cancellation	CancelHandler, task.Cancel()
RESTful API handlers	SubmitHandler, StatusHandler, CancelHandler
Interface abstraction	domain.TaskRunner
Real-time task tracking	Task model with status, start/end timestamps
Robust error handling	Standard Go idioms in API