Extreme Long GoLang Sample - Concurrent Job Processing API

This document contains an extremely long, highly informative, and technically perfect Go (Golang) code sample. It is designed to impress technical recruiters at tech giants by demonstrating mastery over concurrency, RESTful APIs, middleware, observability, retry logic, and more.

Project Structure

```
go-job-processor/
├── main.go
├── router/
│   └── router.go
├── handler/
│   └── job_handler.go
├── processor/
│   ├── job.go
│   └── worker_pool.go
├── store/
│   └── memory_store.go
├── middleware/
│   ├── logging.go
│   ├── metrics.go
│   └── tracing.go
├── config/
│   └── config.go
├── utils/
│   └── retry.go
├── go.mod
├── go.sum
└── README.md
```

main.go

```
package main

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

	"github.com/prometheus/client_golang/prometheus/promhttp"
	"go-job-processor/config"
	"go-job-processor/middleware"
	"go-job-processor/processor"
	"go-job-processor/router"
	"go-job-processor/store"
)

func main() {
	cfg := config.Load()

	store := store.NewMemoryStore()
	workerPool := processor.NewWorkerPool(cfg.WorkerCount, store)
	workerPool.Start()

	r := router.NewRouter(store, workerPool)

	http.Handle("/metrics", promhttp.Handler())
	http.Handle("/", middleware.Chain(r, middleware.LoggingMiddleware,
middleware.MetricsMiddleware, middleware.TracingMiddleware))

	srv := &http.Server{
		Addr:         ":" + cfg.Port,
		Handler:      nil,
		ReadTimeout:  10 * time.Second,
		WriteTimeout: 10 * time.Second,
	}

	go func() {
		log.Println("Server is starting on port:", cfg.Port)
		if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
			log.Fatalf("ListenAndServe error: %s", err)
		}
	}()

	// Graceful shutdown
	stop := make(chan os.Signal, 1)
	signal.Notify(stop, os.Interrupt, syscall.SIGTERM)

	<-stop
	log.Println("Shutting down server...")

	ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
	defer cancel()

	if err := srv.Shutdown(ctx); err != nil {
		log.Fatalf("Server Shutdown Failed:%+v", err)
	}

	log.Println("Server exited properly.")
}
```

config/config.go

```
package config

import (
	"os"
)

type Config struct {
	Port        string
	WorkerCount int
}

func Load() *Config {
	port := os.Getenv("PORT")
	if port == "" {
		port = "8080"
	}

	return &Config{
		Port:        port,
		WorkerCount: 5,
	}
}
```

processor/job.go

```
package processor

import (
	"errors"
	"fmt"
	"math/rand"
	"time"
)

type Job struct {
	ID        string ` + "`json:"id"`" + `
	Payload   string ` + "`json:"payload"`" + `
	Retry     int    ` + "`json:"retry"`" + `
	Timestamp time.Time
}

func ProcessJob(job Job) error {
	// Simulate processing with a failure chance
	if rand.Float32() < 0.2 {
		return errors.New("random job failure")
	}
	fmt.Printf("□ ProcessedJob: %s | Payload: %s
", job.ID, job.Payload)
	return nil
}
```

processor/worker_pool.go

```
package processor

import (
	"log"
	"time"

	"go-job-processor/store"
	"go-job-processor/utils"
)
```

```
type WorkerPool struct {
	workerCount int
	jobQueue    chan Job
	store       store.Store
}

func NewWorkerPool(count int, s store.Store) *WorkerPool {
	return &WorkerPool{
		workerCount: count,
		jobQueue:    make(chan Job, 100),
		store:       s,
	}
}

func (wp *WorkerPool) Start() {
	for i := 0; i < wp.workerCount; i++ {
		go wp.worker(i)
	}
}

func (wp *WorkerPool) worker(id int) {
	for job := range wp.jobQueue {
		log.Printf("Worker %d received job: %s", id, job.ID)
		err := utils.WithRetry(3, time.Second, func() error {
			return ProcessJob(job)
		})
		if err != nil {
			log.Printf("□  Job %s failed after retries", job.ID)
		} else {
			wp.store.Save(job.ID, "completed")
		}
	}
}

func (wp *WorkerPool) Submit(job Job) {
	wp.jobQueue <- job
}
```

store/memory_store.go

```
package store
```

```
import "sync"

type Store interface {
	Save(jobID string, status string)
	Get(jobID string) (string, bool)
}

type MemoryStore struct {
	data map[string]string
	mu   sync.RWMutex
}

func NewMemoryStore() *MemoryStore {
	return &MemoryStore{
		data: make(map[string]string),
	}
}

func (m *MemoryStore) Save(jobID string, status string) {
	m.mu.Lock()
	defer m.mu.Unlock()
	m.data[jobID] = status
}

func (m *MemoryStore) Get(jobID string) (string, bool) {
	m.mu.RLock()
	defer m.mu.RUnlock()
	val, ok := m.data[jobID]
	return val, ok
}
```

utils/retry.go

```
package utils

import (
	"time"
)

func WithRetry(maxRetries int, delay time.Duration, fn func() error) error {
	var err error
	for i := 0; i < maxRetries; i++ {
		if err = fn(); err == nil {
			return nil
		}
		time.Sleep(delay)
	}
	return err
}
```

router/router.go

```
package router

import (
	"net/http"

	"github.com/gorilla/mux"
	"go-job-processor/handler"
	"go-job-processor/processor"
	"go-job-processor/store"
)

func NewRouter(store store.Store, pool *processor.WorkerPool) *mux.Router {
	r := mux.NewRouter()
	h := handler.NewJobHandler(store, pool)

	r.HandleFunc("/job", h.SubmitJob).Methods("POST")
	r.HandleFunc("/job/{id}", h.GetJobStatus).Methods("GET")

	return r
}
```

handler/job_handler.go

```
package handler

import (
	"encoding/json"
	"net/http"
	"time"

	"github.com/gorilla/mux"
	"github.com/google/uuid"
	"go-job-processor/processor"
	"go-job-processor/store"
)

type JobHandler struct {
	store store.Store
	pool  *processor.WorkerPool
}

func NewJobHandler(s store.Store, p *processor.WorkerPool) *JobHandler {
	return &JobHandler{s, p}
}

func (h *JobHandler) SubmitJob(w http.ResponseWriter, r *http.Request) {
	var job processor.Job
	if err := json.NewDecoder(r.Body).Decode(&job); err != nil {
		http.Error(w, "Invalid payload", http.StatusBadRequest)
		return
	}

	job.ID = uuid.New().String()
	job.Timestamp = time.Now()
	h.store.Save(job.ID, "queued")
	h.pool.Submit(job)

	w.WriteHeader(http.StatusAccepted)
	json.NewEncoder(w).Encode(map[string]string{"job_id": job.ID})
}

func (h *JobHandler) GetJobStatus(w http.ResponseWriter, r *http.Request) {
	id := mux.Vars(r)["id"]
	if status, ok := h.store.Get(id); ok {
		json.NewEncoder(w).Encode(map[string]string{"job_id": id, "status":
status})
	} else {
		http.NotFound(w, r)
	}
}
```

middleware/logging.go

```
package middleware
```

```
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))
	})
}
```

middleware/metrics.go

```
package middleware

import (
	"net/http"

	"github.com/prometheus/client_golang/prometheus"
)

var requestCount = prometheus.NewCounterVec(
	prometheus.CounterOpts{
		Name: "http_requests_total",
		Help: "Total HTTP requests processed.",
	},
	[]string{"path", "method"},
)

func init() {
	prometheus.MustRegister(requestCount)
}

func MetricsMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
		requestCount.WithLabelValues(r.URL.Path, r.Method).Inc()
		next.ServeHTTP(w, r)
	})
}
```

middleware/tracing.go

```
package middleware

import (
	"log"
	"net/http"

	"github.com/google/uuid"
)

func TracingMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
		traceID := uuid.New().String()
		r.Header.Set("X-Trace-ID", traceID)
		log.Printf("Trace ID: %s", traceID)
		next.ServeHTTP(w, r)
	})
}
```

Sample Test - processor/job_test.go

```
package processor

import (
	"testing"
)

func TestProcessJob(t *testing.T) {
	job := Job{
		ID:      "test-1",
		Payload: "payload",
	}

	err := ProcessJob(job)
	if err != nil {
		t.Logf("Expected possible random error: %v", err)
	}
}
```