# 🚀 Getting Started with Go Concurrency Projects

This guide walks you through setting up and running all 6 production-ready concurrency projects.

## 📋 Prerequisites

- **Go 1.19+** (check with `go version`)
- Basic terminal/command line knowledge
- Text editor or IDE (VS Code recommended)

## 🏗️ Quick Setup (All Projects)

### Step 1: Navigate to the workspace

```
cd /home/zjunaidz/AI/go-concurrency
```

### Step 2: Initialize Go modules for each project

```
# Initialize all 6 projects at once
for project in rate-limiter job-queue cache web-crawler connection-pool pub-sub; do
  echo "Initializing $project..."
  cd projects/$project/final
  go mod init github.com/yourusername/go-concurrency/$project 2>/dev/null || true
  go mod tidy
  cd ../../..
done
```

### Step 3: Verify setup

```
# This should show "go: downloading" messages and then "ok"
cd projects/rate-limiter/final
go test -v
```

## 🎯 Running Individual Projects

### Project 1: Rate Limiter (Token Bucket with Sharding)

**What it does:** Limits requests per second using token bucket algorithm with 256 shards for high concurrency.

```
cd projects/rate-limiter/final

# Initialize (if not done already)
go mod init github.com/yourusername/go-concurrency/rate-limiter
go mod tidy

# Run all tests
go test -v
```

```
# Run with race detector (catches concurrency bugs)
go test -race -v

# Run benchmarks (see performance)
go test -bench=. -benchmem

# Expected output:
# BenchmarkRateLimiter_Allow-8            5000000    ~250 ns/op (500k ops/sec)
# BenchmarkRateLimiter_Sharded-8         50000000     ~20 ns/op (5M ops/sec - 10x
faster!)
```

**Try the progression:**

```
# Compare naive → improved → final
cd ../naive
go run rate_limiter.go  # Shows race conditions (will panic!)

cd ../improved
go run rate_limiter.go  # Fixed races, but slower

cd ../final
go run rate_limiter.go  # Production-ready with sharding
```

---

### Project 2: Job Queue (Priority Queue with Worker Pool)

**What it does:** Distributed task queue with priorities, retry logic, and bounded worker pool.

```
cd projects/job-queue/final

# Initialize
go mod init github.com/yourusername/go-concurrency/job-queue
go mod tidy

# Run tests
go test -v

# Run stress test (1000 concurrent jobs)
go test -run=TestJobQueue_Concurrent -v

# Run benchmarks
go test -bench=. -benchmem

# Expected output:
# 10k jobs/sec with priority ordering
# Automatic retry with exponential backoff (1s, 2s, 4s, 8s)
```

**Key tests to watch:**

```
# Test priority ordering
go test -run=TestJobQueue_Priority -v
```

```
# Test retry mechanism
go test -run=TestJobQueue_Retry -v

# Test graceful shutdown
go test -run=TestJobQueue_Shutdown -v
```

---

### Project 3: Cache (Thread-Safe LRU with Sharding)

**What it does:** In-memory cache with LRU eviction, 256-way sharding, and TTL support.

```
cd projects/cache/final

# Initialize
go mod init github.com/yourusername/go-concurrency/cache
go mod tidy

# Run tests
go test -v

# Run benchmarks
go test -bench=. -benchmem

# Expected output:
# BenchmarkCache_ShardedGet-8    100000000   ~10 ns/op (100M ops/sec!)
# 256x faster than single-lock version
```

**Explore the sharding magic:**

```
# Run the example showing 1 → 16 → 256 shard progression
go test -run=Example -v

# Output shows:
# 1 shard:   ~50k ops/sec  (high contention)
# 16 shards: ~800k ops/sec (16x improvement)
# 256 shards: ~12M ops/sec (256x improvement!)
```

---

### Project 4: Web Crawler (Ethical with Robots.txt)

**What it does:** Concurrent web crawler respecting robots.txt, with per-domain rate limiting and circuit breakers.

```
cd projects/web-crawler/final

# Initialize (needs external dependency for HTML parsing)
go mod init github.com/yourusername/go-concurrency/web-crawler
go get golang.org/x/net/html
go mod tidy
```

```
# Run tests
go test -v

# Run specific tests
go test -run=TestCrawler_RobotsTxt -v    # Verifies ethical crawling
go test -run=TestCrawler_RateLimiting -v # Checks politeness
go test -run=TestCrawler_CircuitBreaker -v # Auto-disables dead domains

# Run benchmarks
go test -bench=. -benchmem

# Expected output:
# 200-500 pages/min with proper rate limiting
# Respects robots.txt Disallow and Crawl-Delay directives
```

**Try crawling a test site:**

```
# The tests include a mock HTTP server
go test -run=TestCrawler_Basic -v

# Shows crawling behavior:
# ✅ Fetches pages respecting depth limits
# ✅ Extracts links and queues them
# ✅ Rate limits per domain (1 req/sec default)
# ✅ Stops on context cancellation
```

---

### Project 5: Connection Pool (with Lifecycle Management)

**What it does:** Database/network connection pool with health checks, idle timeout, and max lifetime.

```
cd projects/connection-pool/final

# Initialize
go mod init github.com/yourusername/go-concurrency/connection-pool
go mod tidy

# Run tests
go test -v

# Run key lifecycle tests
go test -run=TestPool_IdleTimeout -v    # 5min idle → closes
go test -run=TestPool_MaxLifetime -v    # 30min max → recycles
go test -run=TestPool_HealthCheck -v    # Dead conns filtered out

# Run benchmarks
go test -bench=. -benchmem

# Expected output:
# BenchmarkPool_Acquire-8    50000000   ~20 ns/op (50M ops/sec!)
# <1µs latency for acquire/release cycle
```

**Watch the pool in action:**

```
# Run the full test suite with verbose output
go test -v

# You'll see:
# ✅ Min connections pre-created (warm pool)
# ✅ Max connections enforced (bounded resources)
# ✅ Health checks prevent returning dead connections
# ✅ Circuit breaker auto-recovers after failures
# ✅ Background maintenance cleans up old connections
```

---

### Project 6: Pub-Sub (At-Least-Once Delivery)

**What it does:** Message broker with fan-out, pattern matching, acknowledgments, and Dead Letter Queue.

```
cd projects/pub-sub/final

# Initialize
go mod init github.com/yourusername/go-concurrency/pub-sub
go mod tidy

# Run tests
go test -v

# Run specific features
go test -run=TestBroker_PatternMatching -v  # Wildcard topics "orders.*"
go test -run=TestBroker_Retry -v            # 3 attempts with backoff
go test -run=TestBroker_DLQ -v              # Dead letter queue for failures

# Run benchmarks
go test -bench=. -benchmem

# Expected output:
# BenchmarkBroker_Publish-8    10000000   ~100 ns/op (100M msgs/sec!)
# At-least-once delivery with ack mechanism
```

**Explore pub-sub features:**

```
# Test pattern matching
go test -run=TestBroker_PatternMatching -v
# Shows: "orders.*" matches "orders.created", "orders.updated"

# Test reliability
go test -run=TestBroker_Retry -v
# Shows: Failed messages retry 3x with exponential backoff

# Test concurrent load
```

```
go test -run=TestBroker_Concurrent -v
# Shows: 10 publishers × 100 msgs × 10 subscribers = 10k messages
```

---

## 🧪 Running All Tests (Full Validation)

```
# From workspace root
cd /home/zjunaidz/AI/go-concurrency/projects

# Run all tests with race detector
for project in rate-limiter job-queue cache web-crawler connection-pool pub-sub; do
  echo "======================================="
  echo "Testing $project..."
  echo "======================================="
  cd $project/final
  go test -race -v
  echo ""
  cd ../..
done
```

## 📊 Running All Benchmarks (Performance Validation)

```
# From projects directory
for project in rate-limiter job-queue cache web-crawler connection-pool pub-sub; do
  echo "======================================="
  echo "Benchmarking $project..."
  echo "======================================="
  cd $project/final
  go test -bench=. -benchmem -benchtime=3s
  echo ""
  cd ../..
done
```

## 🎓 Learning Path Recommendations

### Path 1: By Complexity (Easiest → Hardest)

1. **rate-limiter** - Learn sharding basics (1 day)
2. **cache** - Practice LRU + sharding (1 day)
3. **job-queue** - Master worker pools (2 days)
4. **connection-pool** - Understand lifecycle (2 days)
5. **web-crawler** - Complex coordination (3 days)
6. **pub-sub** - Advanced patterns (3 days)

### Path 2: By Pattern (Group Similar Concepts)

- **Sharding Projects:** rate-limiter, cache (learn lock reduction)
- **Worker Pool Projects:** job-queue, web-crawler (learn bounded concurrency)
- **Lifecycle Projects:** connection-pool, pub-sub (learn resource management)

### Path 3: Interview Focus (Most Asked First)

1. **rate-limiter** - Extremely common (Stripe, Shopify, Twitter)
2. **cache** - Universal favorite (Google, Facebook, Amazon)
3. **job-queue** - Backend systems (Uber, DoorDash, Airbnb)

## 🐛 Comparing Naive → Improved → Final

Each project has 3 implementations showing the learning progression:

### Example: Rate Limiter Evolution

```
cd projects/rate-limiter

# 1. naive/ - BROKEN (demonstrates common bugs)
cd naive
go run rate_limiter.go
# Expected: Race detector catches concurrent map writes
# Learning: Always protect shared state!

# 2. improved/ - FIXED (works but not optimized)
cd ../improved
go test -bench=. -benchmem
# Expected: ~50k ops/sec (safe but slow)
# Learning: Single mutex becomes bottleneck

# 3. final/ - PRODUCTION (optimized and feature-complete)
cd ../final
go test -bench=. -benchmem
# Expected: ~500k ops/sec (10x faster with sharding!)
# Learning: Reduce contention with smart data structures
```

### Try This Learning Exercise

For each project, run this comparison:

```
cd projects/[PROJECT_NAME]

# Compare the implementations side-by-side
echo "=== NAIVE (shows bugs) ===" && cat naive/*.go | head -50
echo "=== FINAL (production) ===" && cat final/*.go | head -50

# Run naive tests (see failures)
cd naive && go test -race 2>&1 | grep -i "race\|panic" || echo "No tests in naive"

# Run final tests (see success)
cd ../final && go test -race -v
```

## 🔍 Understanding the Test Output

### What to Look For:

✅ **Good Signs:**

```
PASS
ok      github.com/yourusername/go-concurrency/rate-limiter    2.134s
```

# 🏆 Race-Free (Most Important): ``` PASS

WARNING: DATA RACE ← Should NEVER see this in final/

```
**📈 Performance (Benchmarks):**
```

BenchmarkRateLimiter_Allow-8 5000000 250 ns/op ^^^^^^^^ ^^^^^^^^ operations per operation Higher ops/sec = better throughput Lower ns/op = lower latency

```
## 🚨 Troubleshooting

### Issue: "go: cannot find main module"
**Solution:**
```bash
# Make sure you're in a project's final/ directory
cd projects/rate-limiter/final

# Initialize the module
go mod init github.com/yourusername/go-concurrency/rate-limiter
go mod tidy
```

### Issue: "package golang.org/x/net/html is not in GOROOT"

**Solution:**

```
# Only web-crawler needs this external dependency
cd projects/web-crawler/final
go get golang.org/x/net/html
go mod tidy
```

### Issue: Tests fail with "context deadline exceeded"

**Solution:** This is expected in some tests (testing timeout behavior). Look for:

```
TestCrawler_Cancellation ... ok (explicitly testing cancellation)
TestJobQueue_Timeout ... ok (explicitly testing timeout)
```

### Issue: "too many open files"

**Solution:**
```

```
# Increase file descriptor limit (connection-pool tests may need this)
ulimit -n 4096

# Or run fewer concurrent operations
go test -parallel=4  # Default is GOMAXPROCS
```

## 📚 Next Steps

### 1. Read the Documentation

```
# Start with foundations
cd /home/zjunaidz/AI/go-concurrency
cat 00-foundations/01-why-concurrency.md
cat 00-foundations/02-processes-threads-goroutines.md

# Then study each section
ls -la 0*-*/
```

### 2. Study the Project READMEs

```
# Each project has detailed explanations
cat projects/rate-limiter/README.md
cat projects/web-crawler/README.md
# etc.
```

### 3. Modify and Experiment

```
# Copy a project and make it your own
cp -r projects/rate-limiter/final my-rate-limiter
cd my-rate-limiter

# Try these experiments:
# - Change shard count from 256 to 512 (what happens to performance?)
# - Add metrics tracking (requests served, rejected, etc.)
# - Implement sliding window instead of token bucket
# - Add Redis backend for distributed rate limiting
```

### 4. Interview Preparation

```
# Read the interview prep section
cat 08-interview-prep/01-common-questions.md
cat 08-interview-prep/02-system-design.md

# Practice explaining each project:
# "I built a rate limiter using token bucket with 256-way sharding.
#  This reduced lock contention 256x, achieving 500k ops/sec..."
```

## 🎯 Quick Wins (30-Minute Challenges)

Try these to quickly validate your understanding:

### Challenge 1: Rate Limiter

"Modify final implementation to support burst capacity of 2x the rate"

```
cd projects/rate-limiter/final
# Hint: Change maxTokens = rate * 2
# Test: Send burst of 20 requests when rate=10, should allow ~15 through
```

### Challenge 2: Cache

"Add cache hit rate metric to the LRU cache"

```
cd projects/cache/final
# Hint: Add atomic counters for hits/misses
# Test: Verify hit rate increases with repeated gets
```

### Challenge 3: Job Queue

"Implement job cancellation by ID"

```
cd projects/job-queue/final
# Hint: Store jobs in sync.Map, add Cancel(id) method
# Test: Cancel mid-processing, verify job stops
```

## 📊 Performance Targets (Your Goals)

When you run the benchmarks, aim for these numbers:

| Project | Operation | Target Throughput | Notes |
|---|---|---|---|
| rate-limiter | Allow() | 500k+ ops/sec | With 256 shards |
| job-queue | Submit() | 10k+ jobs/sec | With 10 workers |
| cache | Get() | 100M+ ops/sec | With 256 shards |
| web-crawler | Crawl() | 200-500 pages/min | Respects rate limits |
| connection-pool | Acquire() | 50k+ ops/sec | <1µs latency |
| pub-sub | Publish() | 100k+ msgs/sec | At-least-once |

## 🎉 Success Criteria

You'll know you've mastered this when you can:

✅ Run all tests with `-race` flag (no data races)
✅ Explain why sharding improves performance (lock contention)
✅ Implement worker pool from memory (bounded concurrency)
✅ Design graceful shutdown (context + WaitGroup)
✅ Debug deadlocks using stack traces ( `SIGQUIT` signal)
✅ Write production-ready concurrent code confidently

## 💬 Need Help?

**Check these resources in order:**

1. **Project README** - `projects/[PROJECT]/README.md`
2. **Documentation** - `0X-[SECTION]/` folders
3. **Test files** - Shows usage examples: `*_test.go`
4. **PROGRESS.md** - Overall curriculum guide

---

🚀 **Ready to start? Begin with:**

```
cd projects/rate-limiter/final
go mod init github.com/yourusername/go-concurrency/rate-limiter
go mod tidy
go test -v
```

**Then compare with naive to see why it's broken:**

```
cd ../naive
go test -race  # Watch it panic! 🔥
```

Happy learning! 🎓