docker

Mastering Containerized Local Development and Integration Testing with Testcontainers for Go

ContainerDays London ‘26

Testcontainers Go maintainers

Steven Hartland

VP Engineering + Docker Captain

Rocket Science

Manuel de la Peña

Staff OSS Engineer

Docker

docker

1. Best Practices & Performance
2. Modules
3. AI Skills

docker

1. Best Practices

Writing Fast, Reliable Integration Tests

Why Best Practices Matter

🙈 Slow tests = skipped tests

🤬 Flaky tests erode confidence

🚀 Modern Go + `testcontainers-go` = powerful combo

Starting Point - Basic Container

```go
func TestRawRun(t *testing.T) {
    ctx := context.Background()
    ctr, err := testcontainers.Run(ctx, "mysql:8.0",
        testcontainers.WithEnv(map[string]string{
            "MYSQL_ROOT_PASSWORD": "password",
            "MYSQL_DATABASE":      "testdb",
        }),
        testcontainers.WithExposedPorts("3306/tcp"),
        testcontainers.WithWaitStrategy(
            wait.ForLog("port: 3306  MySQL Community Server"),
        ),
    )
    if err != nil { t.Fatal(err) }
    defer ctr.Terminate(ctx)
    // Use container...
}
```

Improvement #1 - Use Modules

```go
func TestUseModules(t *testing.T) {
    ctx := context.Background()

    // ✅ Use the MySQL module - sensible defaults, less code
    ctr, err := mysql.Run(ctx, "mysql:8.0")
    if err != nil { t.Fatal(err) }
    defer ctr.Terminate(ctx)

    // ✅ Module provides connection string helper
    connStr, err := ctr.ConnectionString(ctx)
    require.NoError(t, err)
}
```

Modules - What's Still Wrong?

- ✅ Lots of boilerplate for MySQL setup
- ❌ Using `defer` for cleanup
- ❌ Inconsistent error handling
- ❌ Using `context.Background()`

Improvement #2 - Proper Cleanup

```go
func TestCleanupContainer(t *testing.T) {
    ctr, err := mysql.Run(t.Context(), "mysql:8.0")

    // ✅ CleanupContainer BEFORE error check
    // Handles partial container starts
    testcontainers.CleanupContainer(t, ctr)
    require.NoError(t, err)

    // ✅ Module provides connection string helper
    connStr, err := ctr.ConnectionString(t.Context())
    require.NoError(t, err)
}
```

Modules - Progress Check

✅ Much less boilerplate!
✅ Proper cleanup with `CleanupContainer`
✅ Using `t.Context()` for test lifecycle
❌ Each test starts its own container - slow!

The Problem - One Container Per Test

```go
func TestCreateUser(t *testing.T) {
    ctr, err := mysql.Run(t.Context(), "mysql:8.0")
    testcontainers.CleanupContainer(t, ctr)
    require.NoError(t, err)
    // Test create user...
}

func TestUpdateUser(t *testing.T) {
    ctr, err := mysql.Run(t.Context(), "mysql:8.0")
    testcontainers.CleanupContainer(t, ctr)
    require.NoError(t, err)
    // Test update user...
}
```

[illegible] 🐢

Table-Driven Tests?

```go
func TestUserOperationsTableDriven(t *testing.T) {
   ctr, err := mysql.Run(t.Context(), "mysql:8.0")
   testcontainers.CleanupContainer(t, ctr)
   require.NoError(t, err)

   tests := []struct {
       name string
       fn   func(t *testing.T, db *sql.DB)
   }{
       {"CreateUser", testCreateUser},
       {"UpdateUser", testUpdateUser},
   }
   for _, tc := range tests {
       t.Run(tc.name, func(t *testing.T) { tc.fn(t, db) })
   }
}
```

Table-Driven - Trade-offs

✅ Single container for sub tests - fast!
✅ Poor IDE integration - can't click to run a single test
❌ Harder to debug - which test case failed?
❌ Loop variable capture issues (pre-Go 1.22)

Improvement #3 - Explicit Subtests

```go
func TestUserOperationsExplicit(t *testing.T) {
    ctr, err := mysql.Run(t.Context(), "mysql:8.0")
    testcontainers.CleanupContainer(t, ctr)
    require.NoError(t, err)

    db := openDB(t, ctr)

    // ✅ Explicit subtests - IDE friendly, easy to debug
    t.Run("CreateUser", func(t *testing.T) { testCreateUser(t, db) })
    t.Run("UpdateUser", func(t *testing.T) { testUpdateUser(t, db) })
    t.Run("DeleteUser", func(t *testing.T) { testDeleteUser(t, db) })
}
```

Explicit Subtests - Benefits

✅ Single container - fast!

✅ Click to run any subtest in IDE

✅ Clear test names in output

✅ Easy to add/remove tests

❌ Subtests share same database - data conflicts!

Improvement #4 - Isolated Databases

```go
t.Run("CreateUser", func(t *testing.T) {
    t.Parallel()
    db := openUniqueDB(t, ctr) // Creates unique DB for this subtest
    testCreateUser(t, db)
})

t.Run("UpdateUser", func(t *testing.T) {
    t.Parallel()
    db := openUniqueDB(t, ctr) // Creates unique DB for this subtest
    testUpdateUser(t, db)
})
```

Isolated Databases - Helper

```go
func openUniqueDB(t *testing.T, ctr *mysql.MySQLContainer) *sql.DB {
    t.Helper()
    ctx := t.Context()
    // Generate unique database name using test name
    dbName := strings.ReplaceAll(t.Name(), "/", "_")

    db, err := sql.Open("mysql", connStr)
    require.NoError(t, err)
    t.Cleanup(func() { db.Close() })

    // Create isolated database for this test
    _, err = db.ExecContext(ctx, fmt.Sprintf("CREATE DATABASE IF NOT EXISTS `%s`", dbName))
    require.NoError(t, err)

    return db
}
```

Isolated Databases - Checkpoint

✅ Single container - fast!
✅ Subtests can run in parallel
✅ No data conflicts between tests
✅ Each test starts with clean state
❌ Only subtests benefit from shared container

Improvement #5 - Reusable Containers

```go
// runPostgres is a helper to start a Postgres container with reuse enabled.
func runPostgres(ctx context.Context) (*postgres.PostgresContainer, error) {
    return postgres.Run(ctx, "postgres:16", testcontainers.WithReuseByName("shared-postgres"))
}

// runPostgresReuse is a test helper to start a Postgres container with reuse enabled.
func runPostgresReuse(t *testing.T) *postgres.PostgresContainer {
    db, err := runPostgres(t.Context())
    if err != nil {
        // If the create failed ensure it's fully cleaned up.
        testcontainers.CleanupContainer(t, db)
        t.Fatal(err)
    }

    return db
}
```

Reusable Containers - Tests

```go
func TestReuse_A(t *testing.T) {
   db := runPostgresReuse(t)
   t.Log("container id", db.GetContainerID())
   require.True(t, db.IsRunning())
}

func TestReuse_B(t *testing.T) {
   db := runPostgresReuse(t)
   t.Log("container id", db.GetContainerID())
   require.True(t, db.IsRunning())
}
```

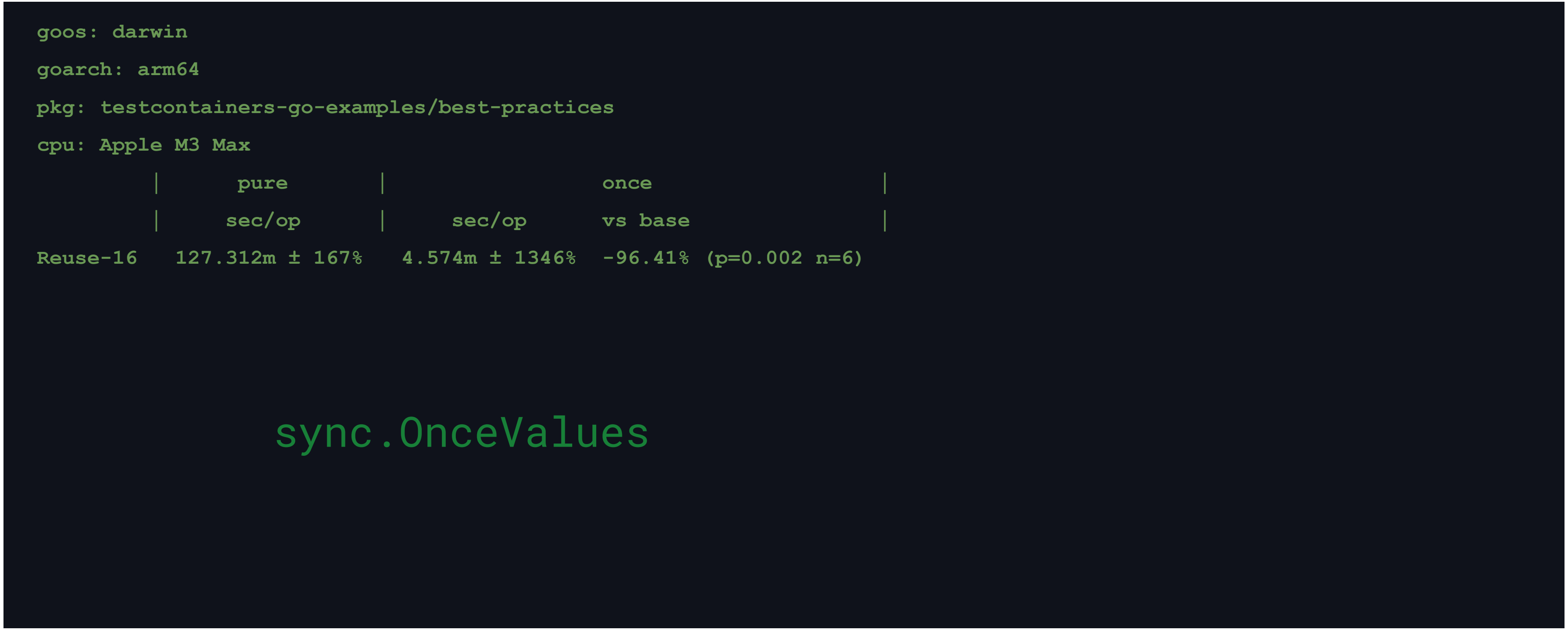
Even Better - sync.OnceValues

```
var runPostgresOnce = sync.OnceValues(func() (*postgres.PostgresContainer, error) {
    return runPostgres(context.Background())
})

func TestOnce_A(t *testing.T) {
    db, err := runPostgresOnce()
    require.NoError(t, err)
    t.Log("container id", db.GetContainerID())
    require.True(t, db.IsRunning())
}

func TestOnce_B(t *testing.T) {
    db, err := runPostgresOnce()
    require.NoError(t, err)
    t.Log("container id", db.GetContainerID())
    require.True(t, db.IsRunning())
}
```

Performance: Pure Reuse vs sync.OnceValues

```
goos: darwin
goarch: arm64
pkg: testcontainers-go-examples/best-practices
cpu: Apple M3 Max
         │      pure       │                 once                  │
         │     sec/op      │    sec/op     vs base                 │
Reuse-16   127.312m ± 167%   4.574m ± 1346%  -96.41% (p=0.002 n=6)
```

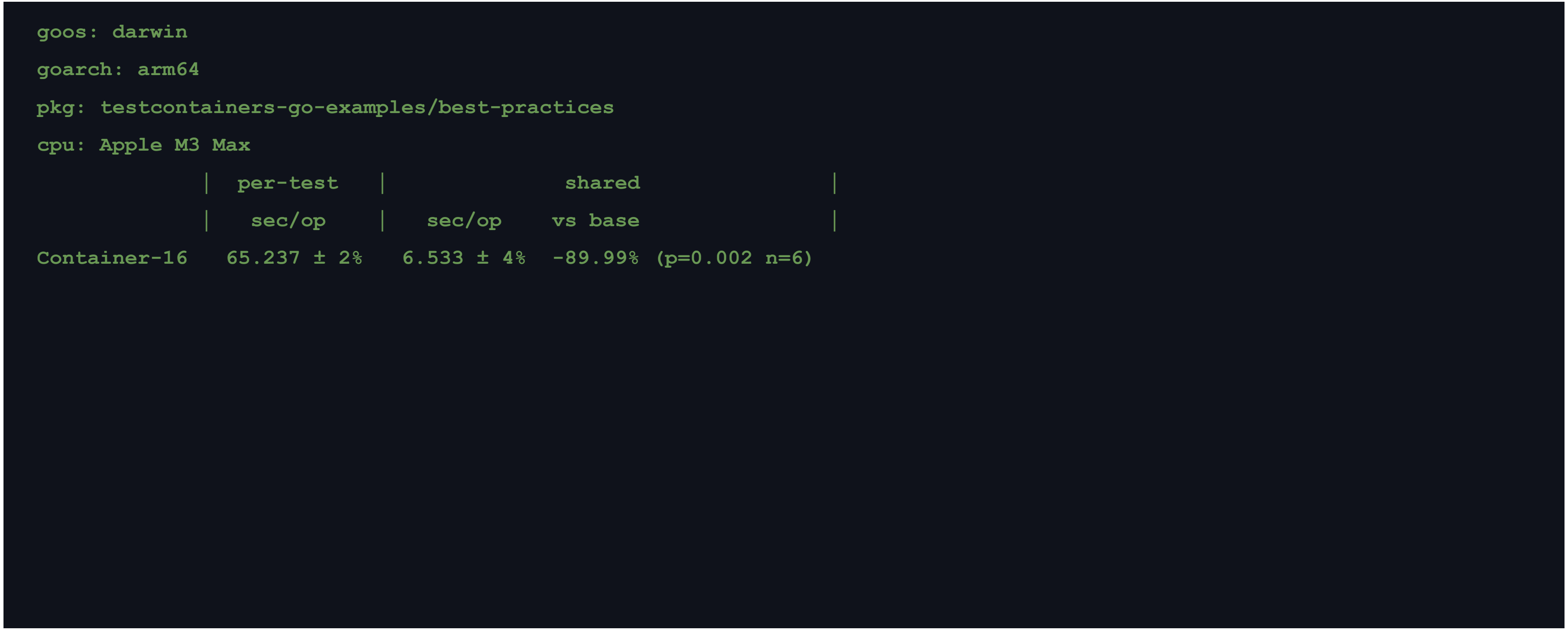
Performance: Pure Reuse vs sync.OnceValues



```
goos: darwin
goarch: arm64
pkg: testcontainers-go-examples/best-practices
cpu: Apple M3 Max
         │       pure       │                    once                    │
         │      sec/op      │     sec/op       vs base                   │
Reuse-16   127.312m ± 167%   4.574m ± 1346%  -96.41% (p=0.002 n=6)
```

```
sync.OnceValues
```

Performance: Pure Reuse vs sync.OnceValues



```
goos: darwin
goarch: arm64
pkg: testcontainers-go-examples/best-practices
cpu: Apple M3 Max
             │  per-test   │               shared               │
             │   sec/op    │   sec/op    vs base                │
Container-16   65.237 ± 2%   6.533 ± 4%  -89.99% (p=0.002 n=6)
```

Bonus: Effective Wait Strategies

```
// ✅ Best: HTTP health check
testcontainers.WithWaitStrategy(
   wait.ForHTTP("/").WithPort("80/tcp")
        .WithStartupTimeout(30*time.Second),
)

// ✅ Good: SQL ping check
testcontainers.WithWaitStrategy(
   wait.ForSQL("5432/tcp", "postgres", connFunc)
        .WithStartupTimeout(60*time.Second),
)

// ❌ Avoid: Log-based checks (logs change between versions)
wait.ForLog("Ready to accept connections")
```

Key Takeaways

✅ Use modules for less boilerplate
✅ `CleanupContainer` before error check
✅ `t.Context()` for test lifecycle
✅ Explicit subtests over table-driven tests
✅ Shared containers for speed
✅ Functional wait strategies (avoid `ForLog`)

docker

2. DRY, use modules

Optional subtitle

What are modules?

Go modules representing a given technology for you to simply consume them in one-liner

Internally hide the usage of core options to build a given technology, adding custom configuration options and custom behavior exposed through API methods.

➔ Specific API to deal with state and/or behavior of the given technology

- ◆ Connection strings, HTTP endpoints, Credentials, add config files, ...

➔ Community modules

- ◆ Not tested on our CI
- ◆ Not released alongside testcontainers-go core library

➔ https://testcontainers.com/modules

Compose

It just works!

Your project already has a dev environment using Docker Compose to start dependencies

- ➔ Instead of calling it from an external process, you can control it with Testcontainers, directly into your tests.
- ➔ Transition path in case you want to use other Testcontainers modules
- ➔ Plug wait strategies to the services in the Compose file.

Toxiproxy

Chaos Engineering at your hands!

Test low network conditions for verifying failure modes and resiliency

- ➔ Create a Docker network
- ➔ Attach the service/s under test to it
- ➔ Attach toxiproxy to it, proxying a given port
- ➔ Build connection strings using toxiproxy instead of the real service/s
- ➔ Add toxics: latency, down, bandwidth (kb/s), timeouts, reset_peer, limit_data...
- ➔ Use the proxied client
- ➔ Fun & Profit!

k3s

Two personas:

1) Application Developers

➔ Get LoadBalancer’s IP, and run e2e tests against the deployed application, using Playwright, Cypress, k6, etc

2) Cluster Administrators

➔ Get cluster’s kubeconfig and create a k8s Go client

➔ Check against the resources

docker

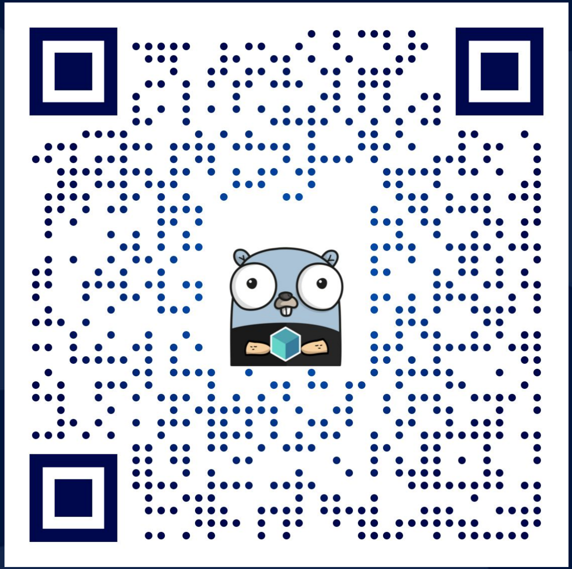
3. Working with coding agents

AGENTS.md, SKILL.md

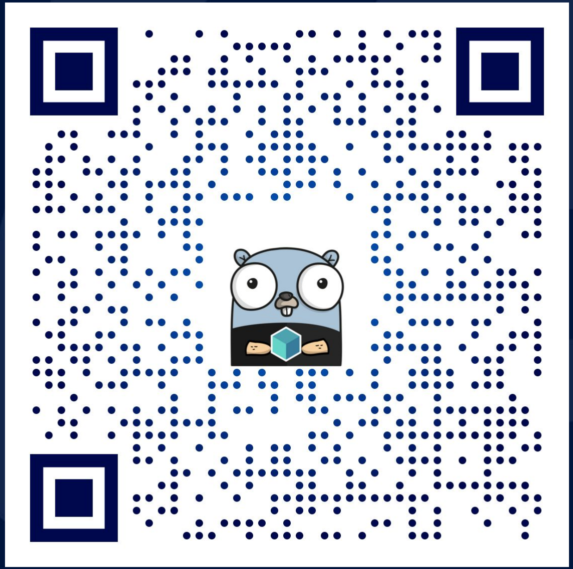
Agent SKILLS

- ➔ https://github.com/testcontainers/claude-skills
  - ◆ Valid for CLAUDE and Copilot
  - ◆ Adds knowledge about Testcontainers Go when using Coding Agents
  - ◆ Remember the Best Practices!

docker



Testcontainers Go Modules



Testcontainers Go Examples

docker

Thank you!

ContainerDays London ‘26