Go Language Testing Guide

This guide covers useful commands and recommended folder structures for testing in Go.

1. Go Test Commands

The `go test` command is the primary tool for running tests in Go.

- **Run all tests in the current package:**

  ```
  go test
  ```

  This command compiles and runs all `_test.go` files in the current directory's package.

- **Run tests verbosely (show test names and results):**

  ```
  go test -v
  ```

  The `-v` flag provides more detailed output, including the name of each test function and its pass/fail status.

- **Run tests matching a specific name/pattern:**

  ```
  go test -run YourTestName
  go test -run "TestSpecificFunction"
  go test -run "Test_.*_Feature" # Run tests matching a regex pattern
  ```

  The `-run` flag takes a regular expression to filter which tests to run.

- **Run tests and also run examples:**

  ```
  go test -run Example
  ```

  Go's testing package supports example functions (prefixed with `Example`). These functions demonstrate how to use a package and are also run as tests to ensure their output remains correct.

- **Run tests and skip caching:**

  ```
  go test -count=1
  ```

By default, `go test` caches successful test results. `-count=1` disables this cache, forcing tests to re-run. Useful when you suspect caching issues or need to ensure fresh execution.

- **Run tests with code coverage analysis:**

```
go test -cover
```

This flag enables code coverage analysis, showing the percentage of your code covered by tests.

- **Run tests with code coverage and generate an HTML report:**

```
go test -coverprofile=coverage.out
go tool cover -html=coverage.out
```

The `-coverprofile` flag writes the coverage data to a file (e.g., `coverage.out`). Then, `go tool cover -html` opens an HTML report in your browser, highlighting covered and uncovered lines.

Aliase

```
alias gocover='go test -coverprofile=coverage.out && go tool cover
-html=coverage.out -o coverage.html && open coverage.html'

# Add this to your ~/.bashrc, ~/.zshrc, or shell configuration file
# Run source ~/.bashrc (or your config file) to load it
# Simply type gocover in your Go project directory
```

Enhanced version (auto-cleanup):

```
alias gocover='go test -coverprofile=coverage.out && go tool cover
-html=coverage.out -o coverage.html && open coverage.html && rm
coverage.out coverage.html'
```

Windows (LINK)

- **Run tests with race detection:**

```
go test -race
```

The `-race` flag enables the built-in data race detector, which helps find concurrency bugs. It can be resource-intensive but is crucial for concurrent applications.

- **Run benchmarks:**

```
go test -bench=.
```

The `-bench` flag runs benchmark functions (prefixed with `Benchmark`). The `.` runs all benchmarks.

- **Run benchmarks matching a specific pattern:**

```
go test -bench="BenchmarkSpecificOperation"
```

- **Run benchmarks for a specific duration:**

```
go test -bench=. -benchtime=5s
```benchtime` specifies the minimum time to run each benchmark.
```

- **Run benchmarks for a specific number of iterations:**

```
go test -bench=. -benchtime=100x
```

- **Show help for `go test`:**

```
go help test
```

2. Go Test Folder Structure

Go's testing philosophy encourages placing tests alongside the code they test.

Standard Package Testing

- **Tests in the same package:**
  By convention, test files (`_test.go`) are placed in the same directory as the source code they are
```

testing. These tests belong to the same package as the source code (e.g., `package mypackage`). They have access to internal (unexported) functions and variables of the package.

```
myproject/
├── mypackage/
│   ├── mypackage.go        // package mypackage
│   └── mypackage_test.go   // package mypackage
└── main.go
```

- **Pros:** Direct access to unexported identifiers, easy to write unit tests.
- **Cons:** Can create tight coupling between test code and implementation details.

External Package Testing (Integration/Black-Box Testing)

- **Tests in a separate `_test` package:**
  For integration or black-box testing, you can place test files in the same directory but declare them as part of a separate package, typically named `packagename_test`. This means the test code can only access the exported (public) functions and types of the package.

```
myproject/
├── mypackage/
│   ├── mypackage.go        // package mypackage
│   └── mypackage_test.go   // package mypackage_test
└── main.go
```

  - **Pros:** Tests only the public API, mimicking how external users would interact with your package. Promotes good API design.
  - **Cons:** Cannot directly test unexported functions.

Dedicated `test` Directory (Less Common, but useful for large projects/e2e)

While not the standard Go way, for larger projects or end-to-end (e2e) tests, some teams might opt for a dedicated `test` directory at the project root or within a module.

```
myproject/
├── pkg/
│   └── mypackage/
│       └── mypackage.go
├── cmd/
│   └── myapp/
│       └── main.go
└── test/                 // Dedicated directory for integration/e2e
tests
    ├── integration/
    │   └── myapp_integration_test.go
```

```
    └── e2e/
        └── myapp_e2e_test.go
```

- **Pros:** Clear separation of concerns, especially for complex integration/e2e tests that might involve setting up external dependencies (databases, APIs).
- **Cons:** Not the idiomatic Go approach for unit tests; requires more explicit import paths. Can make it harder to directly test internal package logic.

Mocks and Test Data

- **Co-locate mocks with tests:**
  If you use mocking frameworks or generate mock interfaces, it's common to place them alongside your `_test.go` files or in a `mocks/` subdirectory within the package.

```
myproject/
├── mypackage/
│   ├── mypackage.go
│   ├── mypackage_test.go
│   └── mocks/
│       └── mock_interface.go // Generated mock
└── main.go
```

- **Test data:**
  For test data (e.g., JSON files, CSVs), a `testdata/` subdirectory within the test package is a common pattern. `go test` ignores directories named `testdata`.

```
myproject/
├── mypackage/
│   ├── mypackage.go
│   ├── mypackage_test.go
│   └── testdata/
│       └── sample.json
└── main.go
```

General Best Practices

- **Test one thing:** Each test function should ideally focus on testing a single, specific aspect of your code.
- **Keep tests fast:** Fast tests encourage frequent execution. For slow tests, consider using `testing.Short()` or separate build tags.
- **Avoid dependencies in unit tests:** Unit tests should be isolated and not depend on external services (databases, network). Use mocks or fakes for dependencies.
- **Use `t.Parallel()`:** For tests that don't share state, use `t.Parallel()` to run them concurrently and speed up test execution.

- **Comprehensive testing:** Aim for good code coverage, but remember that coverage is a metric, not a goal in itself. Focus on testing critical paths and edge cases.