Week 4 Go: Testing

What you'll learn

What you'll need

- Session 1
  - Unit Test
    - Definition
    - Go Test
    - Demo
    - Go Test Commands
  - Test Doubles: Stub and Mock
  - Dependency Injection
    - Definition
    - Example
  - Integration Test
    - Definition
    - Persistence integration tests
    - Integration testing interactions between services
  - Resources
  - Exercise
- Session 2: Assignment
- Session 3: Review Assignment

What you'll learn

- Create unit tests
- Create integration tests

What you'll need

- Basic understanding of Go syntax

Session 1

Unit Test

Definition

"In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use." - Source: Wikipedia.

Go Test

In Go, we use the `go test` subcommand for unit testing. The `go test` subcommand is a test driver for Go packages that follow certain conventions:

- Files that end with `_test.go` are not part of the package built with normal `go build`, but they are part of the package when built with `go test`.
- A test function is a function beginning with `Test`. For example, `TestGetUserInfo`, `TestPlaceOrder`, …
- A benchmark function is a function beginning with `Benchmark`. Run the command `go test -bench regexp` to report the mean execution time of the operation.

```
func BenchmarkPalindrome(b *testing.B)  {
	// If there is any complex initialization, perform it here and call b.ResetTimer to reset timer.
	// Therefore, it will not add to the measured time of each iteration.
	for i := 0; i < b.N; i++ {
		IsPalindrome("aaaa")
	}
}
```

- An example function, whose name starts with Example, provides machine-checked documentation. It neither has parameters nor results.

```
func ExampleIsPalindrome()  {
	fmt.Printf("%t", IsPalindrome("aaaa"))
	// go test will check output is printed into standard output
	// Output:
	// true
}
```

Demo

`TestPalindrome` shows table-driving testing style which is very common in Go. The t parameter provides methods for reporting errors and logging additional info. The code can be found here.

`palindrome.go`

```
package palindrome

import (
"unicode"
)

func IsPalindrome(s string) bool {
	var runes []rune
	for _ , r := range s {
		runes = append(runes, unicode.ToLower(r))
	}
	for i := range runes {
		if runes[i] != runes[len(runes) - 1 -i] {
			return false
		}
	}
	return true
}
```

`palindrome_test.go`

```
func TestPalindrome(t *testing.T)  {
	testCases := []struct{
		input string
		output bool
	}{
		 {"aa", true},
		{"ábá", true},
		{"Aba", true},
		{"ab", false},
		{"Abc", false},
	}
	for _, tc := range testCases {
		if IsPalindrome(tc.input) != tc.output {
			t.Errorf("IsPalindrome(%s) != %t", tc.input,
tc.output)
		}
```

```
      }
}
```

Go Test Commands

Here is a list of available Go Test commands:

```
// Build and run tests inside current directory
go test

// Build and run tests for all sub-folders
go test ./...

// Print name and execution time of each test in the package
go test -v

// Build and run tests whose names match a regex
go test -run regexp

// Run tests with test coverage
go test -cover -coverprofile=output.out .
go tool cover -html output.out

// For more.
go help test
```

Test Doubles: Stub and Mock

System Under Test (SUT) usually has dependencies. These dependencies can complicate and slow down unit tests. A good practice is to replace dependencies with test doubles. There are two types of test doubles:

- Stub which has a "fixed" set of "canned" responses that are specific to your test(s).
- Mock which has a set of expectations about calls that are made. If these expectations are not met, the test fails. - Source
- Some mock packages in Go: golang/mock, stretchr/testify/mock,

Dependency Injection

Definition

"Dependency injection (DI) means giving an object its instance variables."
- Source

One benefit of DI is that we can easily mock external dependencies in unit tests.

Example

`todo/server/repository/todo.go`

```
type ToDo interface {
 Get(id string) (*pb.Todo, error)
}

type ToDoImpl struct {
}


func (r ToDoImpl) Get(id string) (*pb.Todo, error) {
// Actual implementation.
}
```

`todo/server/service/todo/todo.go`

```
type Service struct {
     ToDoRepo repository.ToDo
}

func (s Service) DoSomeThing() error {
    // Use s.ToDoRepo
}
```

`client.go`

```
todoRepository := repository.ToDoImpl{}
todoService := todo.Service{ToDoRepo: todoRepository}
```

`todo/server/service/todo/todo_test.go`

```
func TestDoSomeThing(t *testing.T)  {
     mockToDoRep := &mocks.ToDo{}
     mockToDoRep.On("Get", req.Id).Return(toDo, nil)

     service := Service{ToDoRepo: mockToDoRep}

     err :=  service.DoSomeThing()

     assert.Nil(t, err)
     mockToDoRep.AssertExpectations(t)
}
```

Integration Test

In this section, we'll define integration tests based on Chris Richardson's book entitled *Microservices patterns*.

We will also only focus on two types of integration tests:

- Persistence integration tests
- Integration testing interactions between services

Definition

"Integration tests verify that a service can interact with infrastructure services such as databases and other application services." - Source: Chris Richardson

Persistence integration tests

These kinds of tests check if the database access logic works as expected. For instance, we write integration tests for the repository package and verify whether data is saved into a database. The database could be set up using Docker.

```
package repository


type ToDoRepositorySuite struct {
	suite.Suite
	db *pg.DB
	todoRep ToDo
}

func (s *ToDoRepositorySuite) SetupSuite()  {
	// Connect to PostgresQL
	s.db = pg.Connect(&pg.Options{
		User:     "postgres",
		Password: "example",
		Database: "todo",
		Addr:     "localhost" + ":" + "5433",
		RetryStatementTimeout: true,
		MaxRetries:            4,
		MinRetryBackoff:       250 * time.Millisecond,
	})

	// Create Table
	s.db.CreateTable(&pb.Todo{}, nil)
	s.todoRep = ToDoImpl{DB: s.db}
}

func (s *ToDoRepositorySuite) TearDownSuite() {
	s.db.Close()
}

func (s *ToDoRepositorySuite)TestInsert()  {
	item := &pb.Todo{ Id: "new_item", Title: "meeting"}
	err := s.todoRep.Insert(item)

	s.Nil(err)

	newTodo, err := s.todoRep.Get(item.Id)
	s.Nil(err)
	s.Equal(item, newTodo)
}

func TestExampleTestSuite(t *testing.T) {
	suite.Run(t, new(ToDoRepositorySuite))
}
```

Integration testing interactions between services

A good strategy for this is to use Consumer-driven contract test. There are two popular contract testing frameworks::

- Spring Cloud Contract (for JVM-based applications)
- Pact family of frameworks (supports Go)
- At Grab, we built Mocker, a Go SDK for integration and functional tests on local boxes and CI.

Resources

Here are some useful unit testing package in Go:
https://github.com/stretchr/testify
https://github.com/vektra/mockery

Exercise

- Write unit test for todo service.
- Write integration tests to verify if the data is saved to the database, and whether the to-do service follows a predefined contract. Use `docker-compose` to:
    - Spin up a postgresql database
    - Run integration tests to verify if data persists in the database.

- Run integration tests to verify if the to-do service follows a contract using pack-go.

Session 2: Assignment

Complete the unit tests and integration tests for all endpoints of the *to-do* service.

- A sample unit test is provided for the `GetToDo` method.
- A sample integration test is provided for the `Insert` method of ToDo repository.
- Use pact-go to write integration tests for endpoints of the to-do service.

Session 3: Review Assignment

Things to look for:

- All logic of the `service` package is unit-tested.
- Mock all dependencies in unit tests.
- Write integration tests for all methods inside `repository` package to verify that data is saved into database.
- Write integration tests for all endpoints of the to-do service using pact-go