Lab 1

Lab 1: Test-Driven Development in Go Hello World

Introduction

This lab will help you learn the basics of Go programming. You will learn Go syntax and fundamental concepts using Test-Driven Development (TDD).

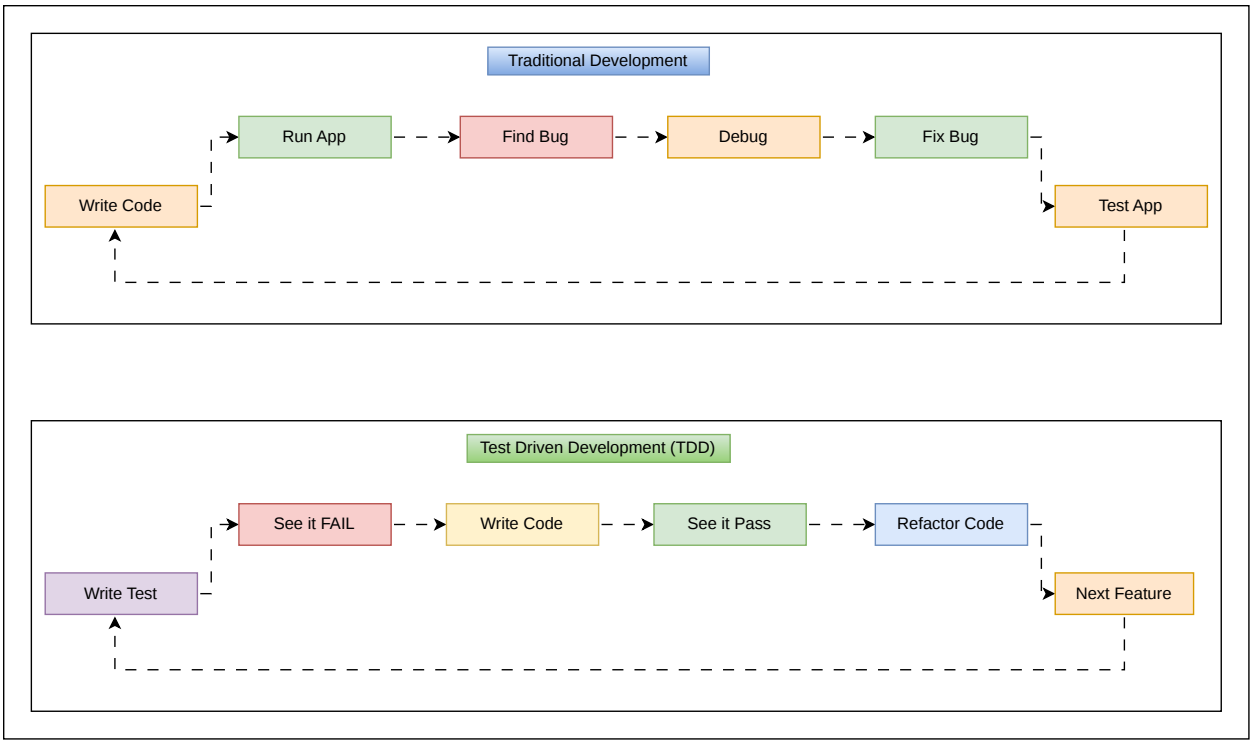
We will introduce what TDD is and show you how to use it to guide your learning process. In this lab, you will start with a simple "Hello World" example using TDD.

What is TDD, Really?

TDD is writing the test before writing the code.

Think of it like this:

1. Write down what you want your code to do (the test)
2. Run it and watch it fail (because the code doesn't exist yet!)
3. Write just enough code to make it pass
4. Clean it up (refactor) while the test keeps you safe



Differences between Traditional Development and Test Driven Development

Traditional Development	Test-Driven Development (TDD)
Code first, then test at the end	Write tests before writing any code
Unclear requirements may lead to rework	Requirements sharpened by writing tests
Debugging happens late	Immediate feedback with each step
Harder to know when you're "done"	"Done" means all tests pass
Refactoring is risky	Refactoring feels safer, tests catch mistakes
Tests can get skipped or forgotten	Tests are part of the workflow

Why TDD Helps You Learn Go

TDD lets you dive into using Go before you fully understand every part of it.When you write a test first, you're essentially making requests to the language, like:

- "I want a Greet function that takes a name"
- "It should return a string"
- "If I call it with 'Karim', it should give back 'Hello, Karim!'"

In doing this, you’re learning:

- How to define functions in Go
- What types look like
- How to manipulate strings
- How Go’s testing framework operates
- How to understand error messages

And you do all of this before writing any of the actual implementation! Now, here's your lab task description for what we'll be doing in the rest of this lab.

Build a Greeting System by Practicing TDD

In this lab, you're going to build a greeting system step by step. Each step adds a new feature, and you'll write the test first every time.

Steps:

1. **Create a function that returns** `"Hello, World!"`
2. **Make it greet people by name**
3. **Handle when no name is provided**
4. **Add formal vs. casual greetings**
5. **Greet differently based on time of day**

Step 0: Project Setup

First, let's initialize our Go project. This tells Go "hey, this is a module!"

```
mkdir greeting
cd greeting
go mod init github.com/yourusername/greeting
```

You'll see a `go.mod` file appear. That's Go's way of tracking your project.

Step 1: Create a function that returns "Hello, World!"

Step 1.1: Create a file called `greeting_test.go`

In Go, test files always end with `_test.go`. Let's write our first test:

```
package greeting

import"testing"

funcTestGreet(t*testing.T){
    got:=Greet()
    want:="Hello, World!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

What's happening here?

- `package greeting` – Declares the package name.
- `import "testing"` – Imports Go's built-in testing tools.
- `func TestGreet(t *testing.T)` – All test functions in Go start with `Test`.
- `t.Errorf()` – Reports a test failure.

Step 1.2: Run the test (it will ***Fail***!)

```
gotest
```

You'll see an error like:

```
./greeting_test.go:6:9: undefined: Greet
```

This is good! We want to see it fail first. It's failing because `Greet()` doesn't exist yet.

Step 1.3: Create `greeting.go` and write just enough code

```
package greeting

funcGreet()string{
return"Hello, World!"
}
```

Step 1.4: Run the test again

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.001s
```

Learning Outcomes:

- How to write a test in Go
- How to create a function that returns a string
- The TDD cycle: Write a test that specifies your requirement (it will fail) → Write just enough code to make the test pass → Refactor the code to improve it

---

Step 2: Make our function greet specific people by Name

Step 2.1: Add a new test (add this to `greeting_test.go`)

```
funcTestGreetWithName(t*testing.T){
```

```
    got:=Greet("Karim")
    want:="Hello, Karim!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 2.2: Run the test (watch it fail)

```
gotest
```

```
./greeting_test.go:15:15: too many arguments in call to Greet
```

It's complaining because `Greet()` doesn't accept any parameters yet!

Step 2.3: Update your function in `greeting.go`

```
package greeting

funcGreet(namestring)string{
return"Hello, "+ name+"!"
}
```

Step 2.4: Run tests again

```
gotest
```

Now first test is broken:

```
```

```
./greeting_test.go:6:15: not enough arguments in call to Greet
```

When we change function signatures, we need to update all our tests.

Step 2.5: Fix the first test

```
funcTestGreet(t*testing.T){
    got:=Greet("World")// Added "World" as the name
    want:="Hello, World!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 2.6: Run tests one more time

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.002s
```

Learning Outcomes:

- How to add parameters to functions
- String concatenation with `+`
- How changing code affects existing tests (this is why tests are valuable!)

Step 3: When someone doesn't provide a name, default to "Poridhian"

In real programs, things go wrong. Users forget to fill in forms, data comes in incomplete. Good code handles these edge cases gracefully.

Step 3.1: Write a test for the edge case

```
funcTestGreetWithEmptyName(t*testing.T){
    got:=Greet("")
    want:="Hello, Poridhian!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 3.2: Run the test (see it fail)

```
gotest
```

```
--- FAIL: TestGreetWithEmptyName (0.00s)
    greeting_test.go:25: got "Hello, !" but wanted "Hello, Poridhian!"
```

Our test shows exactly what's wrong.

Step 3.3: Add the logic to handle empty names

```
package greeting

funcGreet(namestring)string{
if name==""{
        name="Poridhian"
}
```

```
return"Hello, "+ name+"!"
}
```

Step 3.4: Run all tests

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.001s
```

Learning Outcomes

- How to use `if` statements in Go
- Checking for empty strings

Step 4: Add formal vs. casual greetings

Think about how you greet your best friend ("Hey, Karim!") versus your uncle ("Good day, Rahim uncle"). Let's add that to our code.

Step 4.1: Write a test for casual greetings

```
funcTestGreetCasual(t*testing.T){
    got:=Greet("Karim","casual")
    want:="Hey, Karim!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 4.2: Write a test for formal greetings

```
funcTestGreetFormal(t*testing.T){
    got:=Greet("Rahim uncle","formal")
    want:="Good day, Rahim uncle!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 4.3: Run tests (they'll fail because we need a second parameter)

```
gotest
```

```
./greeting_test.go:32:15: too many arguments in call to Greet
        have (string, string)
        want (string)
./greeting_test.go:40:15: too many arguments in call to Greet
        have (string, string)
        want (string)
FAIL    github.com/poridhian/greeting [build failed]
```

Step 4.4: Update the function

```
package greeting

funcGreet(namestring, stylestring)string{
if name==""{
        name="Poridhian"
}
```

```
if style=="casual"{
return"Hey, "+ name+"!"
}

if style=="formal"{
return"Good day, "+ name+"!"
}

return"Hello, "+ name+"!"
}
```

Step 4.5: Update all previous tests to include a style parameter

Go back to your earlier tests and add `"normal"` as the second parameter. For example:

```
funcTestGreet(t*testing.T){
   got:=Greet("World","normal")
   want:="Hello, World!"

if got!= want{
      t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 4.6: Let's refactor to make it cleaner

Looking at all those `if` statements, there's a better way. Let's use a `switch`:

```
package greeting

funcGreet(namestring, stylestring)string{
if name==""{
      name="Poridhian"
}
```

```
    prefix:=getPrefix(style)
return prefix+ name+"!"
}

funcgetPrefix(stylestring)string{
switch style{
case"casual":
return"Hey, "
case"formal":
return"Good day, "
default:
return"Hello, "
}
}
```

Step 7: Run tests to make sure refactoring didn't break anything

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.002s
```

This is the positive side of TDD. We refactored our code completely, and we know immediately that everything still works.

Learning Outcomes:

- Adding multiple parameters to functions
- Using `if` statements for different conditions
- Using `switch` statements (cleaner than many `if`s)
- Breaking code into smaller functions
- Refactoring safely with tests as a safety net

Conclusion

In this lab, you accomplished a lot:

- You experienced the core cycle of Test-Driven Development (TDD)—writing tests before code, watching them fail, making them pass, and refactoring with confidence.
- You learned foundational Go concepts: package structure, functions, string manipulation, parameters, conditionals (`if`, `switch`), and basic types.
- You practiced Go's testing tools: test functions, `testing.T`, running tests with `go test`, and understanding the results.

Lab 1: Test-Driven Development in Go Hello World

Introduction

This lab will help you learn the basics of Go programming. You will learn Go syntax and fundamental concepts using Test-Driven Development (TDD).

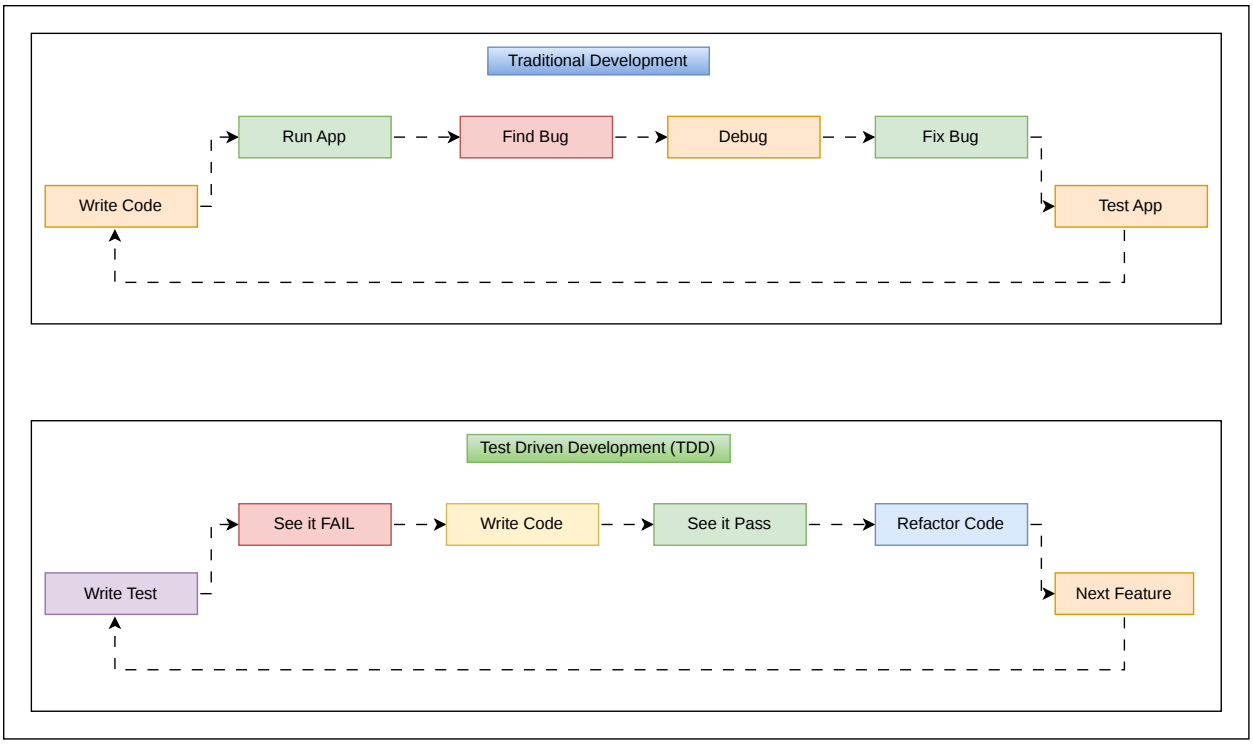
We will introduce what TDD is and show you how to use it to guide your learning process. In this lab, you will start with a simple "Hello World" example using TDD.

What is TDD, Really?

TDD is writing the test before writing the code.

Think of it like this:

1. Write down what you want your code to do (the test)
2. Run it and watch it fail (because the code doesn't exist yet!)
3. Write just enough code to make it pass
4. Clean it up (refactor) while the test keeps you safe



Differences between Traditional Development and Test Driven Development

Traditional Development	Test-Driven Development (TDD)
Code first, then test at the end	Write tests before writing any code
Unclear requirements may lead to rework	Requirements sharpened by writing tests
Debugging happens late	Immediate feedback with each step
Harder to know when you're "done"	"Done" means all tests pass
Refactoring is risky	Refactoring feels safer, tests catch mistakes
Tests can get skipped or forgotten	Tests are part of the workflow

Why TDD Helps You Learn Go

TDD lets you dive into using Go before you fully understand every part of it.When you write a test first, you’re essentially making requests to the language, like:

- “I want a Greet function that takes a name”
- “It should return a string”
- “If I call it with ‘Karim’, it should give back ‘Hello, Karim!’”

In doing this, you’re learning:

- How to define functions in Go
- What types look like
- How to manipulate strings
- How Go’s testing framework operates
- How to understand error messages

And you do all of this before writing any of the actual implementation! Now, here's your lab task description for what we'll be doing in the rest of this lab.

Build a Greeting System by Practicing TDD

In this lab, you're going to build a greeting system step by step. Each step adds a new feature, and you'll write the test first every time.

Steps:

1. **Create a function that returns** `"Hello, World!"`
2. **Make it greet people by name**
3. **Handle when no name is provided**
4. **Add formal vs. casual greetings**
5. **Greet differently based on time of day**

Step 0: Project Setup

First, let's initialize our Go project. This tells Go "hey, this is a module!"

```
mkdir greeting
cd greeting
go mod init github.com/yourusername/greeting
```

You'll see a `go.mod` file appear. That's Go's way of tracking your project.

Step 1: Create a function that returns "Hello, World!"

Step 1.1: Create a file called `greeting_test.go`

In Go, test files always end with `_test.go`. Let's write our first test:

```
package greeting

import"testing"

funcTestGreet(t*testing.T){
    got:=Greet()
    want:="Hello, World!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

What's happening here?

- `package greeting` – Declares the package name.
- `import "testing"` – Imports Go's built-in testing tools.
- `func TestGreet(t *testing.T)` – All test functions in Go start with `Test`.
- `t.Errorf()` – Reports a test failure.

Step 1.2: Run the test (it will ***Fail***!)

```
gotest
```

You'll see an error like:

```
./greeting_test.go:6:9: undefined: Greet
```

This is good! We want to see it fail first. It's failing because `Greet()` doesn't exist yet.

Step 1.3: Create `greeting.go` and write just enough code

```
package greeting

funcGreet()string{
return"Hello, World!"
}
```

Step 1.4: Run the test again

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.001s
```

Learning Outcomes:

- How to write a test in Go
- How to create a function that returns a string
- The TDD cycle: Write a test that specifies your requirement (it will fail) → Write just enough code to make the test pass → Refactor the code to improve it

---

Step 2: Make our function greet specific people by Name

Step 2.1: Add a new test (add this to `greeting_test.go`)

```
funcTestGreetWithName(t*testing.T){
```

```
    got:=Greet("Karim")
    want:="Hello, Karim!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 2.2: Run the test (watch it fail)

```
gotest
```

```
./greeting_test.go:15:15: too many arguments in call to Greet
```

It's complaining because `Greet()` doesn't accept any parameters yet!

Step 2.3: Update your function in `greeting.go`

```
package greeting

funcGreet(namestring)string{
return"Hello, "+ name+"!"
}
```

Step 2.4: Run tests again

```
gotest
```

Now first test is broken:

```
```

```
./greeting_test.go:6:15: not enough arguments in call to Greet
```

When we change function signatures, we need to update all our tests.

Step 2.5: Fix the first test

```
funcTestGreet(t*testing.T){
    got:=Greet("World")// Added "World" as the name
    want:="Hello, World!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 2.6: Run tests one more time

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.002s
```

Learning Outcomes:

- How to add parameters to functions
- String concatenation with `+`
- How changing code affects existing tests (this is why tests are valuable!)

Step 3: When someone doesn't provide a name, default to "Poridhian"

In real programs, things go wrong. Users forget to fill in forms, data comes in incomplete. Good code handles these edge cases gracefully.

Step 3.1: Write a test for the edge case

```
funcTestGreetWithEmptyName(t*testing.T){
    got:=Greet("")
    want:="Hello, Poridhian!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 3.2: Run the test (see it fail)

```
gotest
```

```
--- FAIL: TestGreetWithEmptyName (0.00s)
    greeting_test.go:25: got "Hello, !" but wanted "Hello, Poridhian!"
```

Our test shows exactly what's wrong.

Step 3.3: Add the logic to handle empty names

```
package greeting

funcGreet(namestring)string{
if name==""{
        name="Poridhian"
}
```

```
return"Hello, "+ name+"!"
}
```

Step 3.4: Run all tests

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.001s
```

Learning Outcomes

- How to use `if` statements in Go
- Checking for empty strings

Step 4: Add formal vs. casual greetings

Think about how you greet your best friend ("Hey, Karim!") versus your uncle ("Good day, Rahim uncle"). Let's add that to our code.

Step 4.1: Write a test for casual greetings

```
funcTestGreetCasual(t*testing.T){
    got:=Greet("Karim","casual")
    want:="Hey, Karim!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 4.2: Write a test for formal greetings

```
funcTestGreetFormal(t*testing.T){
    got:=Greet("Rahim uncle","formal")
    want:="Good day, Rahim uncle!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 4.3: Run tests (they'll fail because we need a second parameter)

```
gotest
```

```
./greeting_test.go:32:15: too many arguments in call to Greet
        have (string, string)
        want (string)
./greeting_test.go:40:15: too many arguments in call to Greet
        have (string, string)
        want (string)
FAIL    github.com/poridhian/greeting [build failed]
```

Step 4.4: Update the function

```
package greeting

funcGreet(namestring, stylestring)string{
if name==""{
        name="Poridhian"
}
```

```
if style=="casual"{
return"Hey, "+ name+"!"
}

if style=="formal"{
return"Good day, "+ name+"!"
}

return"Hello, "+ name+"!"
}
```

Step 4.5: Update all previous tests to include a style parameter

Go back to your earlier tests and add `"normal"` as the second parameter. For example:

```
funcTestGreet(t*testing.T){
    got:=Greet("World","normal")
    want:="Hello, World!"

if got!= want{
        t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 4.6: Let's refactor to make it cleaner

Looking at all those `if` statements, there's a better way. Let's use a `switch`:

```
package greeting

funcGreet(namestring, stylestring)string{
if name==""{
        name="Poridhian"
}
```

```
    prefix:=getPrefix(style)
return prefix+ name+"!"
}

funcgetPrefix(stylestring)string{
switch style{
case"casual":
return"Hey, "
case"formal":
return"Good day, "
default:
return"Hello, "
}
}
```

Step 7: Run tests to make sure refactoring didn't break anything

```
gotest
```

```
PASS
ok      github.com/poridhian/greeting    0.002s
```

This is the positive side of TDD. We refactored our code completely, and we know immediately that everything still works.

Learning Outcomes:

- Adding multiple parameters to functions
- Using `if` statements for different conditions
- Using `switch` statements (cleaner than many `if`s)
- Breaking code into smaller functions
- Refactoring safely with tests as a safety net

Conclusion

In this lab, you accomplished a lot:

- You experienced the core cycle of Test-Driven Development (TDD)—writing tests before code, watching them fail, making them pass, and refactoring with confidence.
- You learned foundational Go concepts: package structure, functions, string manipulation, parameters, conditionals (`if`, `switch`), and basic types.
- You practiced Go's testing tools: test functions, `testing.T`, running tests with `go test`, and understanding the results.