Chapter: Understanding Step Definitions

🧠 What Are Step Definitions?

In Behavior Driven Development (BDD), **Step Definitions** are functions in your programming language (e.g., Go) that match the steps written in your `.feature` files (written in Gherkin syntax) and execute test logic.

🔍 Real World Analogy:

- Think of a **feature file** like a "to-do" checklist written by a product owner:

  > *Scenario: Login with valid credentials*
  > Given I am on the login page
  > When I enter valid username and password
  > Then I should be redirected to the dashboard

- **Step definitions** are the implementation of these steps in code that test whether the app actually behaves that way.

🔄 The Cucumber/godog Flow

```
[.feature file] → [step definitions] → [your Go code/application logic]
```

1. **Feature File**: Describes the behavior using Given/When/Then.
2. **Step Definitions**: Match Gherkin steps to Go functions using regex patterns.
3. **Go Code**: Executes business logic (e.g., login function, DB calls, HTTP request).

💻 Hands-On Example (Go + godog)

Step 1: Feature File (`features/login.feature`)

```
Feature: Login Functionality

  Scenario: Login with valid credentials
    Given I am on the login page
    When I enter username "admin" and password "password123"
    Then I should see the dashboard
```

Step 2: Step Definitions in Go (`login_steps.go`)

```go
package main

import (
  "errors"
  "fmt"
)

type loginContext struct {
  username string
  password string
  loggedIn bool
}

func (l *loginContext) iAmOnTheLoginPage() error {
  fmt.Println("Navigated to login page.")
  return nil
}

func (l *loginContext) iEnterUsernameAndPassword(username, password string)
error {
  l.username = username
  l.password = password
  if username == "admin" && password == "password123" {
    l.loggedIn = true
  }
  return nil
}

func (l *loginContext) iShouldSeeTheDashboard() error {
  if l.loggedIn {
    fmt.Println("Dashboard loaded.")
    return nil
  }
  return errors.New("user not logged in")
}

func InitializeScenario(ctx *godog.ScenarioContext) {
  login := &loginContext{}

  ctx.Step(`^I am on the login page$`, login.iAmOnTheLoginPage)
  ctx.Step(`^I enter username "([^"]*)" and password "([^"]*)"$`,
login.iEnterUsernameAndPassword)
  ctx.Step(`^I should see the dashboard$`, login.iShouldSeeTheDashboard)
}
```

Step 3: Run with godog

```
godog run
```

📌 Important Notes

- Step definition regex must exactly match Gherkin sentence patterns.
- You can reuse steps across scenarios if they match semantically.
- Avoid coupling step definitions with too much logic — keep them readable and test-focused.

❓ Interview Questions

1. **What is the role of step definitions in BDD?**
2. **How does Cucumber/godog connect feature files to executable code?**
3. **What are some common challenges when writing step definitions?**
4. **Can a single step definition be reused across multiple features? How?**
5. **What happens when a Gherkin step has no matching step definition?**
6. **How would you handle test data or shared context in Go's step definitions?**
7. **How do you handle tables or complex inputs from Gherkin to step functions?**

🎥 Curated YouTube Tutorials

> 📌 These videos are beginner-friendly and align with godog & step definition usage.

- **godog Basics Tutorial (Cucumber for Go)**
  🎬 https://www.youtube.com/watch?v=UeOS-oe5rDo
- **Gherkin Syntax & Step Definitions Explained**
  🎬 https://www.youtube.com/watch?v=3YBhRGF3v1o
- **BDD with Cucumber: Feature File to Step Definition**
  🎬 https://www.youtube.com/watch?v=N1DZaY3l3Qc

> (You may skip the Java implementation part and focus on the conceptual explanation and Gherkin-to-step mapping)

☑ Summary

- **Step definitions** connect natural language steps in `.feature` files to executable Go code using regex.
- They play a vital role in bridging the gap between non-technical specifications and technical automation.
- With Go and godog, the process remains structured and powerful, enabling readable and reusable BDD tests.

📌 Tip

Always keep your step definitions readable, maintainable, and DRY (Don’t Repeat Yourself). Group related steps by feature and use helper methods to reduce clutter.