# Unit Testing dalam Go

*Unit testing* adalah salah satu jenis pengujian perangkat lunak (*software*) yang berfokus pada pengujian unit-unit terkecil dalam sebuah sistem perangkat lunak. Biasanya, *unit testing* mencakup pengujian *function*, *method*, dan *class*.

**Tujuan *Unit Testing***

- **Deteksi *bug* lebih awal**: dengan melakukan *unit testing*, kita dapat menemukan *bug* lebih awal tanpa perlu menjalankan keseluruhan aplikasi. Bug yang ditemui di unit testing biasanya berhubungan dengan kesalahan logika bisnis, penulisan sintaks, dan kesalahan-kesalahan lain yang dapat membuat kebingungan di kemudian hari.
- **Menulis kode lebih baik**: dengan membuat kode yang dapat dites (*testable*), developer akan secara disiplin menulis kode rapi dan sesuai dengan tanggung jawabnya (menerapkan *single responsibility principle*). Kode yang *testable* adalah kode yang dipecah ke dalam unit-unit kecil dan memiliki fungsi spesifik sehingga mudah untuk diuji secara independen.
- **Menghasilkan dokumentasi**: sebab *unit test* biasanya ditulis dengan format *given-what-then*, hal ini juga sekaligus membuat developer memiliki dokumentasi lengkap mengenai cara fungsi tersebut digunakan, tahapan fungsi bekerja, dan hal yang diharapkan dari penggunaan fungsi tersebut. Hal ini sangat berguna, terutama ketika ada developer baru dalam tim pengembangan.

**Karakteristik *Unit Test* yang Baik**

*Unit test* yang baik pada umumnya memiliki karakteristik sebagai berikut.

- **Cepat**: Biasanya suatu sistem memiliki banyak sekali *unit test* dan jika *unit test* tidak terlalu cepat, butuh waktu yang panjang untuk dieksekusi. Oleh karena itu, *unit test* harus singkat dan *to-the-point*.
- **Sederhana**: Setiap *unit test* seharusnya fokus pada memverifikasi perilaku atau fungsionalitas spesifik (mengikuti aturan "*one assertion per test*"). Struktur tes Anda harus mengikuti pola AAA (*arrange, act, assert*) untuk menjaga kejelasan dan keterbacaan *unit test* yang dibuat. Pilih nama tes yang deskriptif, bermakna, tetapi tetap sederhana agar Anda lebih mudah memahami ribuan tes.
- ***Unit Tests* Harus Dieksekusi secara Terisolasi**: Isolasi kode adalah praktik yang sangat disarankan dalam membuat kode yang *testable*. Data input tes juga harus terkontrol, jadi hindari penggunaan data yang dihasilkan secara dinamis dan mungkin memengaruhi hasil tes.
- **Hasil Tes Harus Sangat Konsisten**: Semakin deterministik *unit test* Anda, semakin baik. Dengan kata lain, hasilnya harus selalu konsisten, tidak peduli terhadap perubahan yang dilakukan pada kode atau dalam urutan pengujian.

- **Refaktor *Unit Test* secara Berkala**: Perlakukan kode *unit test* dengan perhatian yang sama seperti kode sistem utama Anda. Refaktor tes ketika diperlukan untuk meningkatkan *readability*, *maintainability*, dan penggunaan *best practices*.
- **Masukkan ke *Pipeline* CI**: Masukkan *unit test* ke dalam *pipeline continuous integration* (CI) dan otomatiskan eksekusinya. Hal ini memastikan bahwa tes dijalankan secara teratur sehingga memberikan informasi sesegera mungkin mengenai potensi masalah pada *codebase*.

**Unit test dalam Go**

Dalam Go, unit test dibuat dengan:

- File berakhiran _test.go
- Menggunakan package testing
- Function dimulai dengan Test diikuti nama yang di-test

```go
func TestNamaFungsi(t *testing.T) {
    // 1. ARRANGE (Setup data & dependencies)
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()

    // 2. ACT (Execute fungsi yang di-test)
    product, err := service.CreateProduct(ctx, "Laptop", 1000000, 5)

    // 3. ASSERT (Verify hasil)
    if err != nil {
        t.Errorf("Expected no error, got %v", err)
    }
    if product == nil {
        t.Errorf("Expected product, got nil")
    }
}
```

**Mock Repository**

Mock adalah object tiruan yang menggantikan dependency real (database). Dengan mock, kita bisa:

- Test tanpa database real
- Test scenario sulit (error handling)
- Test lebih cepat
- Test repeatability

**Contoh Unit Test**


**Main.go**

```go
// ==================== FILE: main.go ====================
package main

import (
    "context"
    "errors"
    "fmt"
    "log"
    "time"

    "github.com/gofiber/fiber/v2"
    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/bson/primitive"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
)

// ==================== MODELS ====================

// Product model
type Product struct {
    ID        primitive.ObjectID `bson:"_id,omitempty" json:"id"`
    Name      string             `bson:"name" json:"name"`
    Price     float64            `bson:"price" json:"price"`
    Stock     int                `bson:"stock" json:"stock"`
    CreatedAt time.Time          `bson:"created_at" json:"created_at"`
    UpdatedAt time.Time          `bson:"updated_at" json:"updated_at"`
}

// CreateRequest untuk POST
type CreateRequest struct {
    Name  string  `json:"name"`
    Price float64 `json:"price"`
    Stock int     `json:"stock"`
}
```

```go
// ==================== REPOSITORY INTERFACE ====================

// Repository mendefinisikan contract untuk repository
type Repository interface {
    Create(ctx context.Context, product *Product) (primitive.ObjectID, error)
    GetByID(ctx context.Context, id primitive.ObjectID) (*Product, error)
    GetAll(ctx context.Context) ([]Product, error)
    Update(ctx context.Context, id primitive.ObjectID, product *Product) error
    Delete(ctx context.Context, id primitive.ObjectID) error
}

// ==================== REPOSITORY IMPLEMENTATION ====================

// ProductRepository menangani database operations
type ProductRepository struct {
    collection *mongo.Collection
}

// NewProductRepository create instance
func NewProductRepository(collection *mongo.Collection) *ProductRepository {
    return &ProductRepository{collection: collection}
}

// Create menyimpan product baru
func (r *ProductRepository) Create(ctx context.Context, product *Product)
(primitive.ObjectID, error) {
    if product.Name == "" {
        return primitive.NilObjectID, errors.New("product name cannot be
empty")
    }

    product.ID = primitive.NewObjectID()
    product.CreatedAt = time.Now()
    product.UpdatedAt = time.Now()

    result, err := r.collection.InsertOne(ctx, product)
    if err != nil {
        return primitive.NilObjectID, err
    }

    return result.InsertedID.(primitive.ObjectID), nil
}

// GetByID mengambil product
func (r *ProductRepository) GetByID(ctx context.Context, id
primitive.ObjectID) (*Product, error) {
    var product Product
    err := r.collection.FindOne(ctx, bson.M{"_id": id}).Decode(&product)
```

```
    if err != nil {
        if err == mongo.ErrNoDocuments {
            return nil, errors.New("product not found")
        }
        return nil, err
    }
    return &product, nil
}

// GetAll mengambil semua product
func (r *ProductRepository) GetAll(ctx context.Context) ([]Product, error) {
    cursor, err := r.collection.Find(ctx, bson.M{})
    if err != nil {
        return nil, err
    }
    defer cursor.Close(ctx)

    var products []Product
    if err = cursor.All(ctx, &products); err != nil {
        return nil, err
    }

    return products, nil
}

// Update mengupdate product
func (r *ProductRepository) Update(ctx context.Context, id primitive.ObjectID,
product *Product) error {
    if product.Name == "" {
        return errors.New("product name cannot be empty")
    }

    product.UpdatedAt = time.Now()

    result, err := r.collection.UpdateOne(ctx, bson.M{"_id": id}, bson.M{
        "$set": bson.M{
            "name":       product.Name,
            "price":      product.Price,
            "stock":      product.Stock,
            "updated_at": product.UpdatedAt,
        },
    })

    if err != nil {
        return err
    }

    if result.MatchedCount == 0 {
```

```
        return errors.New("product not found")
    }

    return nil
}

// Delete menghapus product
func (r *ProductRepository) Delete(ctx context.Context, id primitive.ObjectID)
error {
    result, err := r.collection.DeleteOne(ctx, bson.M{"_id": id})
    if err != nil {
        return err
    }

    if result.DeletedCount == 0 {
        return errors.New("product not found")
    }

    return nil
}

// ==================== SERVICE ====================

// ProductService menangani business logic
type ProductService struct {
    repo Repository // ← Gunakan interface bukan struct konkret
}

// NewProductService create instance
func NewProductService(repo Repository) *ProductService {
    return &ProductService{repo: repo}
}

// CreateProduct membuat product baru
func (s *ProductService) CreateProduct(ctx context.Context, name string, price
float64, stock int) (*Product, error) {
    if name == "" {
        return nil, errors.New("product name cannot be empty")
    }

    if price < 0 {
        return nil, errors.New("price cannot be negative")
    }

    if stock < 0 {
        return nil, errors.New("stock cannot be negative")
    }
```

```
    product := &Product{
        Name:  name,
        Price: price,
        Stock: stock,
    }

    _, err := s.repo.Create(ctx, product)
    if err != nil {
        return nil, err
    }

    return product, nil
}

// GetProductByID mengambil product by ID
func (s *ProductService) GetProductByID(ctx context.Context, id string)
(*Product, error) {
    objID, err := primitive.ObjectIDFromHex(id)
    if err != nil {
        return nil, errors.New("invalid product ID format")
    }

    return s.repo.GetByID(ctx, objID)
}

// GetAllProducts mengambil semua product
func (s *ProductService) GetAllProducts(ctx context.Context) ([]Product,
error) {
    return s.repo.GetAll(ctx)
}

// UpdateProduct mengupdate product
func (s *ProductService) UpdateProduct(ctx context.Context, id string, name
string, price float64, stock int) error {
    if price < 0 {
        return errors.New("price cannot be negative")
    }

    objID, err := primitive.ObjectIDFromHex(id)
    if err != nil {
        return errors.New("invalid product ID format")
    }

    product := &Product{
        Name:  name,
        Price: price,
        Stock: stock,
    }
```

```
    return s.repo.Update(ctx, objID, product)
}

// DeleteProduct menghapus product
func (s *ProductService) DeleteProduct(ctx context.Context, id string) error {
    objID, err := primitive.ObjectIDFromHex(id)
    if err != nil {
        return errors.New("invalid product ID format")
    }

    return s.repo.Delete(ctx, objID)
}

// ApplyDiscount menghitung harga dengan diskon
func (s *ProductService) ApplyDiscount(ctx context.Context, id string,
discountPercent float64) (float64, error) {
    if discountPercent < 0 || discountPercent > 100 {
        return 0, errors.New("discount must be between 0 and 100")
    }

    product, err := s.GetProductByID(ctx, id)
    if err != nil {
        return 0, err
    }

    finalPrice := product.Price * (1 - discountPercent/100)
    return finalPrice, nil
}

// ==================== DATABASE ====================

// ConnectDB menghubungkan ke MongoDB
func ConnectDB() (*mongo.Client, error) {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    client, err := mongo.Connect(ctx,
options.Client().ApplyURI("mongodb://localhost:27017"))
    if err != nil {
        return nil, err
    }

    if err := client.Ping(ctx, nil); err != nil {
        return nil, err
    }

    fmt.Println("✓ Connected to MongoDB")
```

```
    return client, nil
}

// ==================== MAIN ====================

func main() {
    // Connect MongoDB
    mongoClient, err := ConnectDB()
    if err != nil {
        log.Fatalf("Failed to connect MongoDB: %v", err)
    }
    defer mongoClient.Disconnect(context.Background())

    // Setup layers
    collection := mongoClient.Database("ecommerce").Collection("products")
    repo := NewProductRepository(collection)
    service := NewProductService(repo)

    // Setup Fiber
    app := fiber.New()

    // ==================== ROUTES ====================

    // POST /products - Create product
    app.Post("/products", func(c *fiber.Ctx) error {
        var req CreateRequest
        if err := c.BodyParser(&req); err != nil {
            return c.Status(fiber.StatusBadRequest).JSON(fiber.Map{
                "error": "Invalid request body",
            })
        }

        product, err := service.CreateProduct(c.Context(), req.Name,
req.Price, req.Stock)
        if err != nil {
            return c.Status(fiber.StatusBadRequest).JSON(fiber.Map{
                "error": err.Error(),
            })
        }

        return c.Status(fiber.StatusCreated).JSON(product)
    })

    // GET /products - Get all products
    app.Get("/products", func(c *fiber.Ctx) error {
        products, err := service.GetAllProducts(c.Context())
        if err != nil {
            return c.Status(fiber.StatusInternalServerError).JSON(fiber.Map{
```

```
                "error": err.Error(),
            })
        }

        return c.JSON(products)
    })

    // GET /products/:id - Get product by ID
    app.Get("/products/:id", func(c *fiber.Ctx) error {
        id := c.Params("id")
        product, err := service.GetProductByID(c.Context(), id)
        if err != nil {
            return c.Status(fiber.StatusNotFound).JSON(fiber.Map{
                "error": err.Error(),
            })
        }

        return c.JSON(product)
    })

    // PUT /products/:id - Update product
    app.Put("/products/:id", func(c *fiber.Ctx) error {
        id := c.Params("id")
        var req CreateRequest

        if err := c.BodyParser(&req); err != nil {
            return c.Status(fiber.StatusBadRequest).JSON(fiber.Map{
                "error": "Invalid request body",
            })
        }

        err := service.UpdateProduct(c.Context(), id, req.Name, req.Price,
req.Stock)
        if err != nil {
            return c.Status(fiber.StatusBadRequest).JSON(fiber.Map{
                "error": err.Error(),
            })
        }

        return c.JSON(fiber.Map{"message": "Product updated successfully"})
    })

    // DELETE /products/:id - Delete product
    app.Delete("/products/:id", func(c *fiber.Ctx) error {
        id := c.Params("id")
        err := service.DeleteProduct(c.Context(), id)
        if err != nil {
            return c.Status(fiber.StatusNotFound).JSON(fiber.Map{
```

```
                "error": err.Error(),
            })
        }

        return c.JSON(fiber.Map{"message": "Product deleted successfully"})
    })

    // GET /products/:id/discount?percent=20 - Apply discount
    app.Get("/products/:id/discount", func(c *fiber.Ctx) error {
        id := c.Params("id")
        percentStr := c.Query("percent")

        var percent float64
        _, err := fmt.Sscanf(percentStr, "%f", &percent)
        if err != nil {
            return c.Status(fiber.StatusBadRequest).JSON(fiber.Map{
                "error": "Invalid discount percentage",
            })
        }

        finalPrice, err := service.ApplyDiscount(c.Context(), id, percent)
        if err != nil {
            return c.Status(fiber.StatusBadRequest).JSON(fiber.Map{
                "error": err.Error(),
            })
        }

        return c.JSON(fiber.Map{
            "final_price": finalPrice,
        })
    })

    // Start server
    app.Listen(":3000")
}
```

**Main_test.go**

```
package main

import (
    "context"
    "errors"
    "testing"
    "time"

    "go.mongodb.org/mongo-driver/bson/primitive"
)

// ==================== MOCK REPOSITORY ====================

// MockProductRepository adalah repository tiruan untuk testing
// Implement interface Repository dengan method yang sama
type MockProductRepository struct {
    products map[primitive.ObjectID]*Product
}

// NewMockProductRepository membuat mock repository baru
func NewMockProductRepository() *MockProductRepository {
    return &MockProductRepository{
        products: make(map[primitive.ObjectID]*Product),
    }
}

// Create implement Repository.Create
func (m *MockProductRepository) Create(ctx context.Context, product *Product)
(primitive.ObjectID, error) {
    if product.Name == "" {
        return primitive.NilObjectID, errors.New("product name cannot be
empty")
    }
    product.ID = primitive.NewObjectID()
    product.CreatedAt = time.Now()
    product.UpdatedAt = time.Now()
    m.products[product.ID] = product
    return product.ID, nil
}

// GetByID implement Repository.GetByID
func (m *MockProductRepository) GetByID(ctx context.Context, id
primitive.ObjectID) (*Product, error) {
    if product, exists := m.products[id]; exists {
        return product, nil
    }
    return nil, errors.New("product not found")
```

```
}

// GetAll implement Repository.GetAll
func (m *MockProductRepository) GetAll(ctx context.Context) ([]Product, error)
{
    var products []Product
    for _, p := range m.products {
        products = append(products, *p)
    }
    return products, nil
}

// Update implement Repository.Update
func (m *MockProductRepository) Update(ctx context.Context, id
primitive.ObjectID, product *Product) error {
    if _, exists := m.products[id]; !exists {
        return errors.New("product not found")
    }
    product.ID = id
    product.UpdatedAt = time.Now()
    m.products[id] = product
    return nil
}

// Delete implement Repository.Delete
func (m *MockProductRepository) Delete(ctx context.Context, id
primitive.ObjectID) error {
    if _, exists := m.products[id]; !exists {
        return errors.New("product not found")
    }
    delete(m.products, id)
    return nil
}

// ==================== TEST CASES ====================

// TestCreateProduct menguji pembuatan product
func TestCreateProduct(t *testing.T) {
    // SETUP: Gunakan pointer ke MockProductRepository
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()

    tests := []struct {
        name    string
        pName   string
        price   float64
        stock   int
```

```
        wantErr bool
        errMsg  string
    }{
        {
            name:    "Valid Product",
            pName:   "Laptop",
            price:   10000000,
            stock:   5,
            wantErr: false,
            errMsg:  "",
        },
        {
            name:    "Negative Price",
            pName:   "Mouse",
            price:   -5000,
            stock:   10,
            wantErr: true,
            errMsg:  "price cannot be negative",
        },
        {
            name:    "Negative Stock",
            pName:   "Keyboard",
            price:   500000,
            stock:   -3,
            wantErr: true,
            errMsg:  "stock cannot be negative",
        },
        {
            name:    "Empty Name",
            pName:   "",
            price:   1000,
            stock:   1,
            wantErr: true,
            errMsg:  "product name cannot be empty",
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            product, err := service.CreateProduct(ctx, tt.pName, tt.price,
tt.stock)

            if (err != nil) != tt.wantErr {
                t.Errorf("CreateProduct() error = %v, wantErr %v", err,
tt.wantErr)
                return
            }
```

```
            if tt.wantErr && err.Error() != tt.errMsg {
                t.Errorf("error = %v, want %v", err.Error(), tt.errMsg)
                return
            }

            if !tt.wantErr {
                if product == nil {
                    t.Errorf("got nil, want product")
                    return
                }
                if product.Name != tt.pName {
                    t.Errorf("Name = %v, want %v", product.Name, tt.pName)
                }
                if product.Price != tt.price {
                    t.Errorf("Price = %v, want %v", product.Price, tt.price)
                }
                if product.Stock != tt.stock {
                    t.Errorf("Stock = %v, want %v", product.Stock, tt.stock)
                }
            }
        })
    }
}

// TestGetProductByID menguji pengambilan product berdasarkan ID
func TestGetProductByID(t *testing.T) {
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()

    // Pre-condition: Buat product dulu
    product, err := service.CreateProduct(ctx, "Monitor", 2000000, 8)
    if err != nil {
        t.Fatalf("Failed to create product: %v", err)
    }
    validID := product.ID.Hex()

    tests := []struct {
        name    string
        id      string
        wantErr bool
    }{
        {"Valid ID", validID, false},
        {"Invalid Format", "invalid-id", true},
        {"Non-existent ID", primitive.NewObjectID().Hex(), true},
    }

    for _, tt := range tests {
```

```
        t.Run(tt.name, func(t *testing.T) {
            result, err := service.GetProductByID(ctx, tt.id)

            if (err != nil) != tt.wantErr {
                t.Errorf("error = %v, wantErr %v", err, tt.wantErr)
                return
            }

            if !tt.wantErr && result == nil {
                t.Errorf("got nil, want product")
            }
        })
    }
}

// TestApplyDiscount menguji perhitungan diskon
func TestApplyDiscount(t *testing.T) {
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()

    // Pre-condition: Buat product
    product, _ := service.CreateProduct(ctx, "Phone", 5000000, 10)
    productID := product.ID.Hex()

    tests := []struct {
        name            string
        id              string
        discountPercent float64
        want            float64
        wantErr         bool
    }{
        {"20% Discount", productID, 20, 4000000, false},
        {"No Discount", productID, 0, 5000000, false},
        {"50% Discount", productID, 50, 2500000, false},
        {"Invalid Over 100%", productID, 150, 0, true},
        {"Invalid Negative", productID, -10, 0, true},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            result, err := service.ApplyDiscount(ctx, tt.id,
tt.discountPercent)

            if (err != nil) != tt.wantErr {
                t.Errorf("error = %v, wantErr %v", err, tt.wantErr)
                return
            }
```

```
            if !tt.wantErr && result != tt.want {
                t.Errorf("got %v, want %v", result, tt.want)
            }
        })
    }
}

// TestUpdateProduct menguji update product
func TestUpdateProduct(t *testing.T) {
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()

    // Pre-condition: Buat product dulu
    product, _ := service.CreateProduct(ctx, "Tablet", 3000000, 10)
    productID := product.ID.Hex()

    tests := []struct {
        name    string
        id      string
        pName   string
        price   float64
        stock   int
        wantErr bool
    }{
        {"Valid Update", productID, "Tablet Pro", 3500000, 5, false},
        {"Invalid Price", productID, "Tablet", -1000, 5, true},
        {"Invalid ID", "invalid", "Tablet", 3000000, 5, true},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            err := service.UpdateProduct(ctx, tt.id, tt.pName, tt.price,
tt.stock)

            if (err != nil) != tt.wantErr {
                t.Errorf("error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}

// TestDeleteProduct menguji penghapusan product
func TestDeleteProduct(t *testing.T) {
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()
```

```
    // Pre-condition: Buat product dulu
    product, _ := service.CreateProduct(ctx, "Headset", 500000, 15)
    validID := product.ID.Hex()

    tests := []struct {
        name    string
        id      string
        wantErr bool
    }{
        {"Valid Delete", validID, false},
        {"Already Deleted", validID, true},
        {"Invalid ID", "invalid-id", true},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            err := service.DeleteProduct(ctx, tt.id)

            if (err != nil) != tt.wantErr {
                t.Errorf("error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}

// TestGetAllProducts menguji mengambil semua products
func TestGetAllProducts(t *testing.T) {
    mockRepo := NewMockProductRepository()
    service := NewProductService(mockRepo)
    ctx := context.Background()

    // Pre-condition: Buat beberapa products
    service.CreateProduct(ctx, "Product 1", 100000, 5)
    service.CreateProduct(ctx, "Product 2", 200000, 10)
    service.CreateProduct(ctx, "Product 3", 300000, 15)

    tests := []struct {
        name        string
        expectedLen int
        wantErr     bool
    }{
        {"Get All Products", 3, false},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            products, err := service.GetAllProducts(ctx)
```

```
            if (err != nil) != tt.wantErr {
                t.Errorf("error = %v, wantErr %v", err, tt.wantErr)
                return
            }

            if !tt.wantErr && len(products) != tt.expectedLen {
                t.Errorf("got %d products, want %d", len(products),
tt.expectedLen)
            }
        })
    }
}
```

**Menjalankan unit test**

```
C:\project\gotest> go test -v ./...
=== RUN   TestCreateProduct
=== RUN   TestCreateProduct/Valid_Product
=== RUN   TestCreateProduct/Negative_Price
=== RUN   TestCreateProduct/Negative_Stock
=== RUN   TestCreateProduct/Empty_Name
--- PASS: TestCreateProduct (0.00s)
    --- PASS: TestCreateProduct/Valid_Product (0.00s)
    --- PASS: TestCreateProduct/Negative_Price (0.00s)
    --- PASS: TestCreateProduct/Negative_Stock (0.00s)
    --- PASS: TestCreateProduct/Empty_Name (0.00s)
=== RUN   TestGetProductByID
=== RUN   TestGetProductByID/Valid_ID
=== RUN   TestGetProductByID/Invalid_Format
=== RUN   TestGetProductByID/Non-existent_ID
--- PASS: TestGetProductByID (0.00s)
    --- PASS: TestGetProductByID/Valid_ID (0.00s)
    --- PASS: TestGetProductByID/Invalid_Format (0.00s)
    --- PASS: TestGetProductByID/Non-existent_ID (0.00s)
=== RUN   TestApplyDiscount
=== RUN   TestApplyDiscount/20%_Discount
=== RUN   TestApplyDiscount/No_Discount
=== RUN   TestApplyDiscount/50%_Discount
=== RUN   TestApplyDiscount/Invalid_Over_100%
=== RUN   TestApplyDiscount/Invalid_Negative
--- PASS: TestApplyDiscount (0.00s)
    --- PASS: TestApplyDiscount/20%_Discount (0.00s)
    --- PASS: TestApplyDiscount/No_Discount (0.00s)
    --- PASS: TestApplyDiscount/50%_Discount (0.00s)
    --- PASS: TestApplyDiscount/Invalid_Over_100% (0.00s)
    --- PASS: TestApplyDiscount/Invalid_Negative (0.00s)
```

```
=== RUN   TestUpdateProduct
=== RUN   TestUpdateProduct/Valid_Update
=== RUN   TestUpdateProduct/Invalid_Price
=== RUN   TestUpdateProduct/Invalid_ID
--- PASS: TestUpdateProduct (0.00s)
    --- PASS: TestUpdateProduct/Valid_Update (0.00s)
    --- PASS: TestUpdateProduct/Invalid_Price (0.00s)
    --- PASS: TestUpdateProduct/Invalid_ID (0.00s)
=== RUN   TestDeleteProduct
=== RUN   TestDeleteProduct/Valid_Delete
=== RUN   TestDeleteProduct/Already_Deleted
=== RUN   TestDeleteProduct/Invalid_ID
--- PASS: TestDeleteProduct (0.00s)
    --- PASS: TestDeleteProduct/Valid_Delete (0.00s)
    --- PASS: TestDeleteProduct/Already_Deleted (0.00s)
    --- PASS: TestDeleteProduct/Invalid_ID (0.00s)
=== RUN   TestGetAllProducts
=== RUN   TestGetAllProducts/Get_All_Products
--- PASS: TestGetAllProducts (0.00s)
    --- PASS: TestGetAllProducts/Get_All_Products (0.00s)
PASS
ok      gote
```