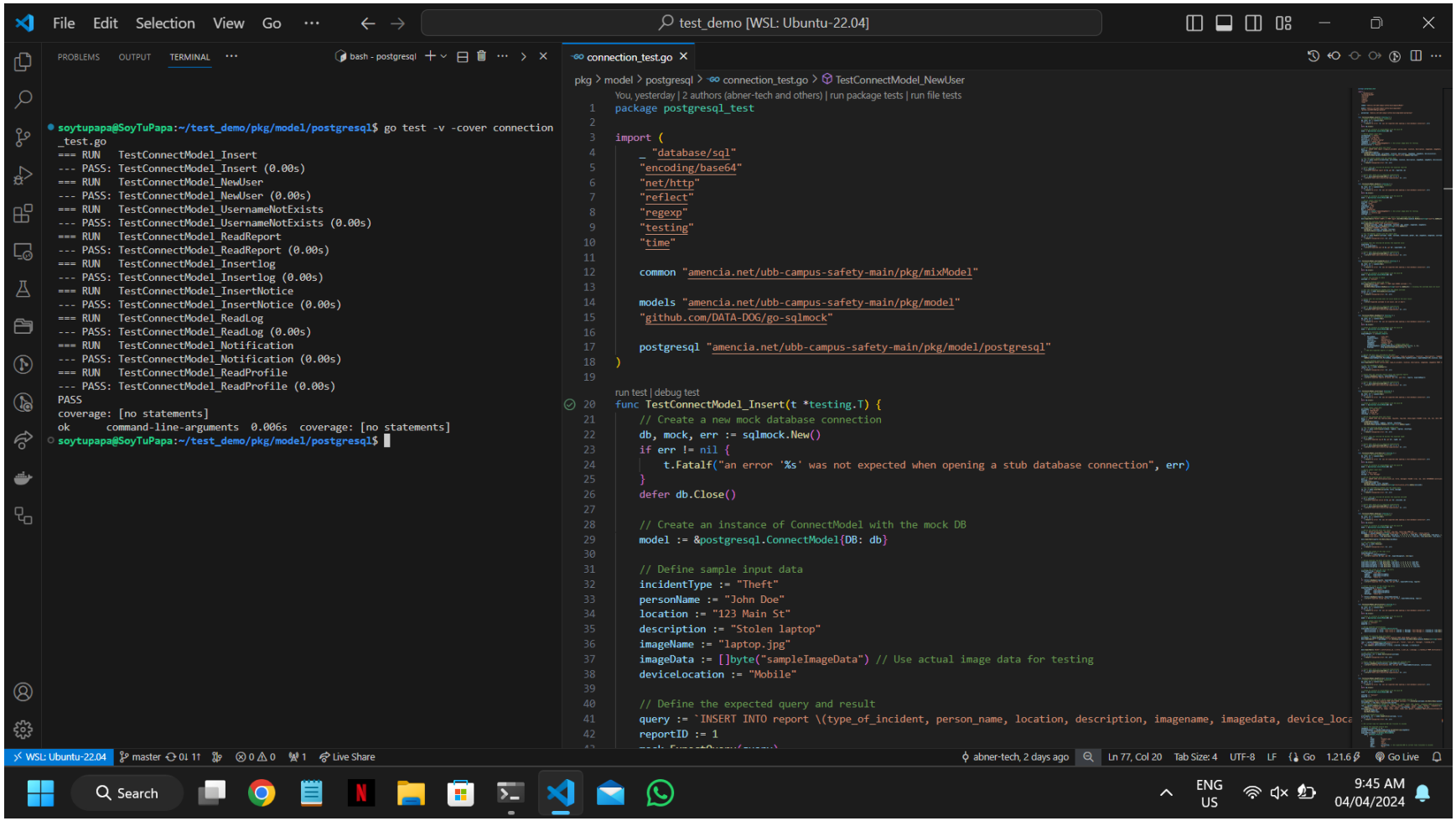
Technical Documentation



```
package postgresql_test

import (
	_ "database/sql"
	"encoding/base64"
	"net/http"
	"reflect"
	"regexp"
	"testing"
	"time"

	common "amencia.net/ubb-campus-safety-main/pkg/mixModel"

	models "amencia.net/ubb-campus-safety-main/pkg/model"
	"github.com/DATA-DOG/go-sqlmock"

	postgresql "amencia.net/ubb-campus-safety-main/pkg/model/postgresql"
)

func TestConnectModel_Insert(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
```

```
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}

	// Define sample input data
	incidentType := "Theft"
	personName := "John Doe"
	location := "123 Main St"
	description := "Stolen laptop"
	imageName := "laptop.jpg"
	imageData := []byte("sampleImageData") // Use actual image data for testing
	deviceLocation := "Mobile"

	// Define the expected query and result
	query := `INSERT INTO report \(type_of_incident, person_name, location, description, imagename, imagedata, device_location\) VALUES \(\$1, \$2, \$3, \$4, \$5, \$6, \$7\) RETURNING report_id;`
	reportID := 1
	mock.ExpectQuery(query).
		WithArgs(incidentType, personName, location, description, imageName, imageData, deviceLocation).
		WillReturnRows(sqlmock.NewRows([]string{"report_id"}).AddRow(reportID))
```

```
	// Call the Insert method with the sample data
	id, err := model.Insert(incidentType, personName, location, description, imageName, imageData, deviceLocation)
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
	}

	// Assert that the returned ID matches the expected reportID
	if id != reportID {
		t.Errorf("expected report ID %d, got %d", reportID, id)
	}

	// Assert that there are no remaining expectations
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}

func TestConnectModel_NewUser(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
```

```
}
defer db.Close()

// Create an instance of ConnectModel with the mock DB
model := &postgresql.ConnectModel{DB: db}

// Define sample input data
username := "testuser"
fname := "John"
lastname := "Doe"
middlename := "M"
gender := "Male"
dob := "1990-01-01"
imagedata := []byte("sampleImageData") // Use actual image data for testing
imagename := "profile.jpg"
usertype := 1

// Mock the usernameExists function to return false (username does not exist)
mock.ExpectQuery("SELECT COUNT(.+) FROM login").WillReturnRows(sqlmock.NewRows([]string{"count"}).AddRow(0))

// Define the expected queries and results
mock.ExpectQuery("INSERT INTO personnelinfotable").
        WithArgs(username, fname, middlename, lastname, dob, gender, imagename, imagedata).
```

```
		WillReturnRows(sqlmock.NewRows([]string{"id"}).AddRow(1))
	mock.ExpectExec("INSERT INTO login").
		WithArgs(1, username, username, usertype).
		WillReturnResult(sqlmock.NewResult(0, 1))

	// Call the NewUser method with the sample data
	id, err := model.NewUser(username, fname, lastname, middlename, gender, dob, imagedata, imagename, usertype)
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
	}

	// Assert that the returned ID matches the expected value
	expectedID := 1
	if id != expectedID {
		t.Errorf("expected user ID %d, got %d", expectedID, id)
	}

	// Assert that there are no remaining expectations
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}
```

```
func TestConnectModel_UsernameNotExists(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}

	// Define the username to check
	username := "testuser"

	// Mock the database query and result
	mock.ExpectQuery("SELECT COUNT(.+) FROM login WHERE username = ?").
		WithArgs(username).
		WillReturnRows(sqlmock.NewRows([]string{"count"}).AddRow(0)) // Assuming the username does not exist

	// Call the usernameExists method with the sample username
	exists, err := model.UsernameExists(username)
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
```

```
	}

	// Assert that the username does not exist based on the mock result
	if exists {
		t.Error("expected username to not exist, but it does")
	}

	// Assert that there are no remaining expectations
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}

func TestConnectModel_ReadReport(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}
```

```
	// Define the expected report data
	expectedReport := []*models.Report{
		{
			PersonName:       "John Doe",
			TypeOfIncident:   "Theft",
			Location:         "Building A",
			Description:      "Stolen laptop",
			ImageName:        "example.jpg",
			ImageData:        []byte{1, 2, 3}, // Sample image data
			EncodedImageData: base64.StdEncoding.EncodeToString([]byte{1, 2, 3}),
			MimeType:         http.DetectContentType([]byte{1, 2, 3}),
		},
		// Add more expected reports if needed
	}

	// Define the mock rows returned by the query
	mockRows := sqlmock.NewRows([]string{"person_name", "type_of_incident", "location", "description", "imagename", "imagedata"}).
		AddRow(expectedReport[0].PersonName, expectedReport[0].TypeOfIncident, expectedReport[0].Location, expectedReport[0].Description, expectedReport[0].ImageName, expectedReport[0].ImageData)

	// Mock the database query and result
	mock.ExpectQuery("SELECT person_name, type_of_incident, location, description, imagename, imagedata FROM report").WillReturnRows(mockRows)
```

```
	// Call the ReadReport method
	reports, err := model.ReadReport()
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
	}

	// Assert that the returned reports match the expected reports
	if !reflect.DeepEqual(reports, expectedReport) {
		t.Errorf("expected reports to match, got %+v, want %+v", reports, expectedReport)
	}

	// Assert that there are no remaining expectations
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}

func TestConnectModel_Insertlog(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
```

```
}
defer db.Close()

// Create an instance of ConnectModel with the mock DB
model := &postgresql.ConnectModel{DB: db}

// Define sample input data
personName := "John Doe"
logDate := "2024-04-01"
logTime := "12:00 PM"
checkType := "Check In"

// Define the expected query and result
query := `INSERT INTO log \(person_name, log_date, log_time, check_type\) VALUES \(\$1, \$2, \$3, \$4\) RETURNING id;`
logID := 1
mock.ExpectQuery(query).
        WithArgs(personName, logDate, logTime, checkType).
        WillReturnRows(sqlmock.NewRows([]string{"id"}).AddRow(logID))

// Call the Insertlog method with the sample data
id, err := model.Insertlog(personName, logDate, logTime, checkType)
if err != nil {
        t.Fatalf("unexpected error: %s", err)
```

```
	}

	// Assert that the returned ID matches the expected logID
	if id != logID {
		t.Errorf("expected log ID %d, got %d", logID, id)
	}

	// Assert that there are no remaining expectations
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}

func TestConnectModel_InsertNotice(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}
```

```
// Define sample input data
userID := 1
title := "Test Title"
message := "Test Message"

// Define the expected query and result
query := `INSERT INTO notification\(user_id, title, message\) VALUES \(\$1, \$2, \$3\) RETURNING notification_id;`
noticeID := 1
mock.ExpectQuery(query).
	WithArgs(userID, title, message).
	WillReturnRows(sqlmock.NewRows([]string{"notification_id"}).AddRow(noticeID))

// Call the InsertNotice method with the sample data
id, err := model.InsertNotice(userID, title, message)
if err != nil {
	t.Fatalf("unexpected error: %s", err)
}

// Assert that the returned ID matches the expected noticeID
if id != noticeID {
	t.Errorf("expected notice ID %d, got %d", noticeID, id)
}
```

```
	// Assert that there are no remaining expectations
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}

func TestConnectModel_ReadLog(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}

	// Define the expected query and result
	query := `SELECT person_name, log_date, log_time, check_type FROM log`
	mockRows := sqlmock.NewRows([]string{"person_name", "log_date", "log_time", "check_type"}).
		AddRow("John Doe", time.Date(2024, time.April, 1, 0, 0, 0, 0, time.UTC), time.Date(2024, time.April, 1, 8, 0, 0, 0, time.UTC),
"Check In").
```

```
		AddRow("Jane Smith", time.Date(2024, time.April, 1, 0, 0, 0, 0, time.UTC), time.Date(2024, time.April, 1, 12, 0, 0, 0, time.UTC), "Check Out")

	mock.ExpectQuery(query).WillReturnRows(mockRows)

	// Call the ReadLog method
	logs, err := model.ReadLog()
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
	}

	// Assert the length of the logs slice
	expectedLogCount := 2
	if len(logs) != expectedLogCount {
		t.Errorf("expected %d logs, got %d", expectedLogCount, len(logs))
	}

	// Define expected log dates and times for 2024
	expectedFirstLogDate := time.Date(2024, time.April, 1, 0, 0, 0, 0, time.UTC)
	expectedFirstLogTime := time.Date(2024, time.April, 1, 8, 0, 0, 0, time.UTC)
	expectedSecondLogDate := time.Date(2024, time.April, 1, 0, 0, 0, 0, time.UTC)
	expectedSecondLogTime := time.Date(2024, time.April, 1, 12, 0, 0, 0, time.UTC)
```

```
	// Assert the values of the first log entry
	expectedFirstLog := &models.Log{
		PersonName: "John Doe",
		LogDate:    expectedFirstLogDate,
		LogTime:    expectedFirstLogTime,
		CheckType:  "Check In",
	}
	if !reflect.DeepEqual(logs[0], expectedFirstLog) {
		t.Errorf("expected first log %+v, but got %+v", expectedFirstLog, logs[0])
	}

	// Assert the values of the second log entry
	expectedSecondLog := &models.Log{
		PersonName: "Jane Smith",
		LogDate:    expectedSecondLogDate,
		LogTime:    expectedSecondLogTime,
		CheckType:  "Check Out",
	}
	if !reflect.DeepEqual(logs[1], expectedSecondLog) {
		t.Errorf("expected second log %+v, but got %+v", expectedSecondLog, logs[1])
	}
}
```

```
func TestConnectModel_Notification(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}

	// Define sample input data
	username := "testuser"
	memberID := 1

	// Define expected notifications
	expectedNotifications := []*models.Notification{
		{Notificationid: 1, Title: "Test Title 1", UserID: 1, Message: "Test Message 1", Created_at: time.Now()},
		{Notificationid: 2, Title: "Test Title 2", UserID: 1, Message: "Test Message 2", Created_at: time.Now()},
	}

	// Define the expected query and result
	queryRegex := regexp.QuoteMeta("SELECT memberID FROM LOGIN WHERE username = $1")
```

```
	mock.ExpectQuery("^" + queryRegex +
"$").WithArgs(username).WillReturnRows(sqlmock.NewRows([]string{"memberID"}).AddRow(memberID))


	rows := sqlmock.NewRows([]string{"notification_id", "title", "user_id", "message", "created_at"})
	for _, n := range expectedNotifications {
		rows.AddRow(n.Notificationid, n.Title, n.UserID, n.Message, n.Created_at)
	}


	mock.ExpectQuery(`SELECT n.notification_id, n.title, n.user_id, n.message, n.created_at FROM notification n LEFT JOIN
notification_seen ns ON n.notification_id = ns.notification_id AND ns.user_id = \$1 WHERE ns.notification_id IS NULL ORDER BY n.notification_id
DESC;`).WithArgs(memberID).WillReturnRows(rows)


	// Call the Notification method
	notifications, err := model.Notification(username)
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
	}


	// Assert that the returned notifications match the expected ones
	if !reflect.DeepEqual(notifications, expectedNotifications) {
		t.Errorf("expected notifications %+v, but got %+v", expectedNotifications, notifications)
	}


	// Assert that there are no remaining expectations
```

```
	if err := mock.ExpectationsWereMet(); err != nil {
		t.Errorf("there were unfulfilled expectations: %s", err)
	}
}

func TestConnectModel_ReadProfile(t *testing.T) {
	// Create a new mock database connection
	db, mock, err := sqlmock.New()
	if err != nil {
		t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
	}
	defer db.Close()

	// Create an instance of ConnectModel with the mock DB
	model := &postgresql.ConnectModel{DB: db}

	username := "testuser"
	memberID := 1

	// Mock database query for SELECT memberID FROM LOGIN WHERE username = $1
	mock.ExpectQuery("SELECT memberID FROM LOGIN WHERE username =
?").WithArgs(username).WillReturnRows(sqlmock.NewRows([]string{"memberID"}).AddRow(memberID))
	currentTime := time.Now().Truncate(time.Second)
```

```
	// Mock database query for SELECT id, image, fname, mname, lname, dob, gender, imagedata FROM personnelinfotable WHERE id = ? LIMIT 1;
	rows := sqlmock.NewRows([]string{"id", "image", "fname", "mname", "lname", "dob", "gender", "imagedata"}).
		AddRow(1, "example.jpg", "John", "Doe", "Smith", currentTime, "Male", []byte{1, 2, 3})
	mock.ExpectQuery("^SELECT id, image, fname, mname, lname, dob, gender, imagedata FROM personnelinfotable WHERE id = \\$1 LIMIT 1$").
		WithArgs(memberID).
		WillReturnRows(rows)

	// Call the ReadProfile method
	profileData, err := model.ReadProfile(username, false)
	if err != nil {
		t.Fatalf("unexpected error: %s", err)
	}

	// Get current time for expected DOB and truncate to seconds

	// Define the expected profile data
	imageData := []byte{1, 2, 3}
	encodedImageData := base64.StdEncoding.EncodeToString(imageData)
	mimeType := http.DetectContentType(imageData)
	expectedProfileData := &common.ProfileDATA{
		DATA: []*models.Profile{
			{
```

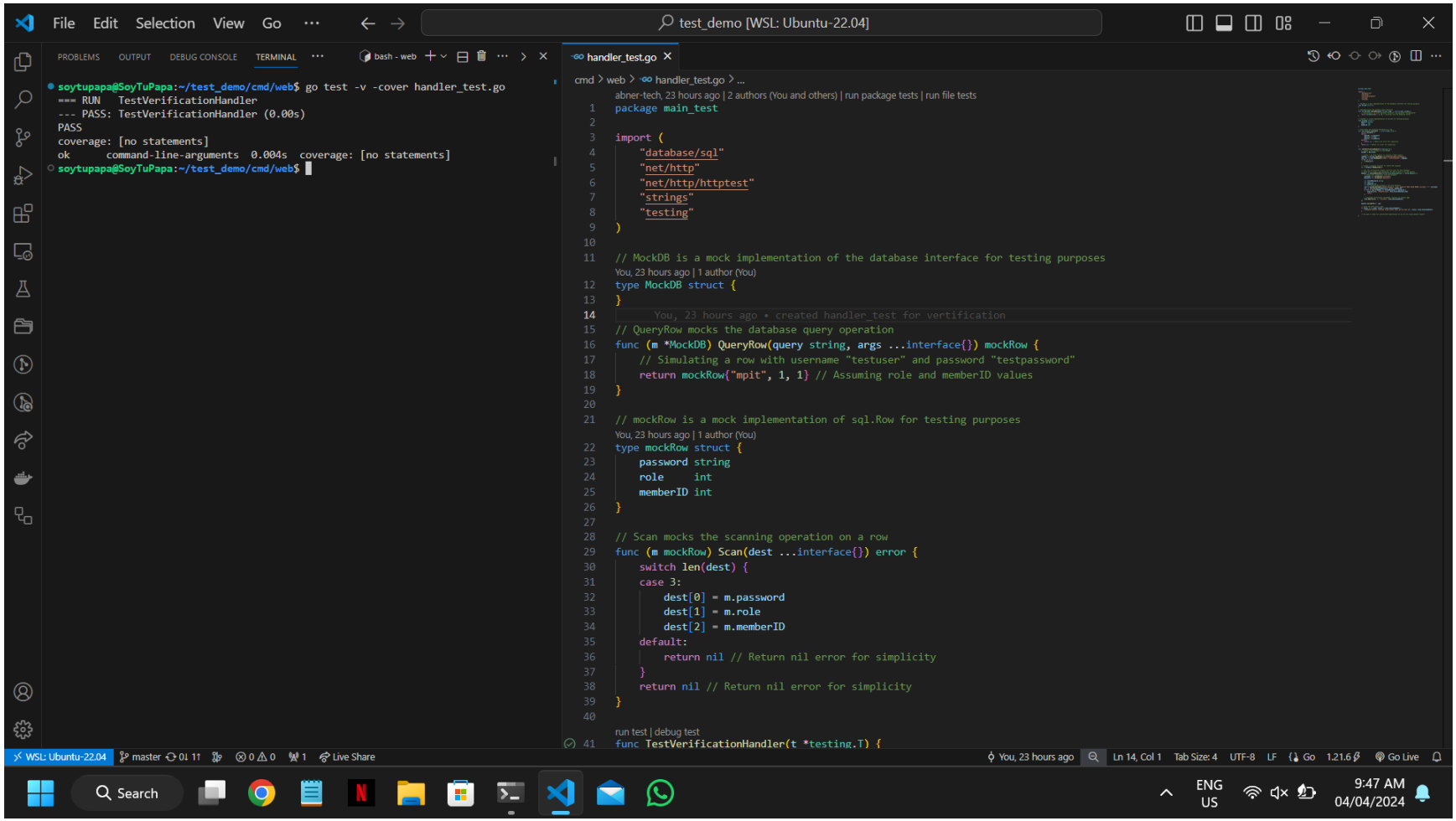
```
                        ID:          1,
                        Image:       "example.jpg",
                        Fname:       "John",
                        Mname:       "Doe",
                        LName:       "Smith",
                        DOB:         currentTime, // Set expected DOB to current time truncated to seconds
                        Gender:      "Male",
                        ImageData:   imageData,
                        EncodedImage: encodedImageData,
                        MimeType:    mimeType,
                    },
                },
                Notification: nil,
        }

        // Compare actual profile data with expected profile data
        if len(profileData.DATA) != len(expectedProfileData.DATA) {
                t.Errorf("expected %d profiles, but got %d profiles", len(expectedProfileData.DATA), len(profileData.DATA))
        }

        for i, expectedProfile := range expectedProfileData.DATA {
                if !reflect.DeepEqual(profileData.DATA[i], expectedProfile) {
                        t.Errorf("expected profile data %+v, but got %+v", expectedProfile, profileData.DATA[i])
```

```
                }
        }

        // Assert that there are no remaining expectations
        if err := mock.ExpectationsWereMet(); err != nil {
                t.Errorf("there were unfulfilled expectations: %s", err)
        }
}
```


File Edit Selection View Go
test_demo [WSL: Ubuntu-22.04]
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
bash - web
soytupapa@SoyTuPapa:~/test_demo/cmd/web$ go test -v -cover handler_test.go
=== RUN TestVerificationHandler
--- PASS: TestVerificationHandler (0.00s)
PASS
coverage: [no statements]
ok command-line-arguments 0.004s coverage: [no statements]
soytupapa@SoyTuPapa:~/test_demo/cmd/web$
handler_test.go
cmd > web > handler_test.go > ...
abner-tech, 23 hours ago | 2 authors (You and others) | run package tests | run file tests
package main_test
import (
"database/sql"
"net/http"
"net/http/httptest"
"strings"
"testing"
)
// MockDB is a mock implementation of the database interface for testing purposes
You, 23 hours ago | 1 author (You)
type MockDB struct {
}
You, 23 hours ago • created handler_test for vertification
// QueryRow mocks the database query operation
func (m *MockDB) QueryRow(query string, args ...interface{}) mockRow {
// Simulating a row with username "testuser" and password "testpassword"
return mockRow{"mpit", 1, 1} // Assuming role and memberID values
}
// mockRow is a mock implementation of sql.Row for testing purposes
You, 23 hours ago | 1 author (You)
type mockRow struct {
password string
role int
memberID int
}
// Scan mocks the scanning operation on a row
func (m mockRow) Scan(dest ...interface{}) error {
switch len(dest) {
case 3:
dest[0] = m.password
dest[1] = m.role
dest[2] = m.memberID
default:
return nil // Return nil error for simplicity
}
return nil // Return nil error for simplicity
}
run test | debug test
func TestVerificationHandler(t *testing.T) {
WSL: Ubuntu-22.04
master
Live Share
You, 23 hours ago
Ln 14, Col 1
Tab Size: 4
UTF-8
LF
Go
1.21.6
Go Live
Search
ENG
US
9:47 AM
04/04/2024

```
package main_test

import (
	"database/sql"
	"net/http"
	"net/http/httptest"
	"strings"
	"testing"
)

// MockDB is a mock implementation of the database interface for testing purposes
type MockDB struct {
}

// QueryRow mocks the database query operation
func (m *MockDB) QueryRow(query string, args ...interface{}) mockRow {
	// Simulating a row with username "testuser" and password "testpassword"
	return mockRow{"mpit", 1, 1} // Assuming role and memberID values
}

// mockRow is a mock implementation of sql.Row for testing purposes
type mockRow struct {
	password string
```

```
	role     int
	memberID int
}

// Scan mocks the scanning operation on a row
func (m mockRow) Scan(dest ...interface{}) error {
	switch len(dest) {
	case 3:
		dest[0] = m.password
		dest[1] = m.role
		dest[2] = m.memberID
	default:
		return nil // Return nil error for simplicity
	}
	return nil // Return nil error for simplicity
}

func TestVerificationHandler(t *testing.T) {
	// Create a new instance of the MockDB
	mockDB := &MockDB{}

	// Create a new HTTP request to simulate a POST request
	reqBody := strings.NewReader("username=mpit&password=mpit")
```

```
req, err := http.NewRequest("POST", "/verification", reqBody)
if err != nil {
	t.Fatal(err)
}

// Create a response recorder to record the response
rr := httptest.NewRecorder()

// Call the verification handler function with the mock database
// Since we're using a mock database, we need to pass it to the handler
handler := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
	// Fetch user credentials from the database
	username := r.FormValue("username")
	password := r.FormValue("password")

	var storedPassword string
	var role int
	var memberID int
	// Call the QueryRow method of the mock database
	row := mockDB.QueryRow("SELECT password, role, memberID FROM LOGIN WHERE username = ?", username)
	err := row.Scan(&storedPassword, &role, &memberID)
	if err == sql.ErrNoRows || storedPassword != password {
		http.Error(w, "Unauthorized", http.StatusUnauthorized)
```

```
                return
        }

        // Assuming verification succeeded, redirect to success page
        http.Redirect(w, r, "/success", http.StatusSeeOther)
    })

    handler.ServeHTTP(rr, req)

    // Check the HTTP status code
    if status := rr.Code; status != http.StatusSeeOther {
        t.Errorf("handler returned wrong status code: got %v want %v", status, http.StatusSeeOther)
    }

    // No need to check for unfulfilled expectations as we are not using sqlmock anymore
}
```