Exercise 2: Git Areas and Undoing Changes

Learning Objectives

By completing this exercise, you will practice:

- Understanding the working area, staging area, and repository
- Using the stash for temporary storage
- Undoing changes with `git restore`, `git reset`, and `git revert`
- Recovering from various mistakes and unwanted changes

Scenario

You're working on a data warehouse project and need to practice managing different Git areas and recovering from common mistakes. This exercise simulates real-world scenarios where you need to undo changes, manage work-in-progress, and handle various Git states.

Setup

Initialize a new repository:

```
# Create project directory
mkdir DataWarehouseProject
cd DataWarehouseProject

# Initialize repository
git init
git config user.name "Your Name"
git config user.email "your.email@example.com"
```

Part 1: Understanding Git Areas

Step 1: Create Initial Files

Create the following files to establish a baseline:

`schema.sql`:

```
-- Data Warehouse Schema
CREATE SCHEMA dwh;

CREATE TABLE dwh.dim_customer (
    customer_key INT IDENTITY(1,1) PRIMARY KEY,
    customer_id INT NOT NULL,
    first_name NVARCHAR(50),
    last_name NVARCHAR(50),
    email NVARCHAR(100)
);
```

`etl_process.sql`:

```
-- ETL Process Script
-- Load dimension tables

INSERT INTO dwh.dim_customer (customer_id, first_name, last_name, email)
SELECT
    customer_id,
    first_name,
    last_name,
    email
FROM staging.customers
WHERE created_date >= DATEADD(day, -1, GETDATE());
```

1. Stage and commit these files with the message: "Initial schema and ETL scripts"

Step 2: Working Area vs Staging Area Practice

1. Modify `schema.sql` by adding this table at the end:

```


CREATE TABLE dwh.dim_product (
    product_key INT IDENTITY(1,1) PRIMARY KEY,
    product_id INT NOT NULL,
    product_name NVARCHAR(100),
    category NVARCHAR(50),
    price DECIMAL(10,2)
);
```

2. Create a new file `config.sql`:

```
-- Configuration settings
USE DataWarehouse;
SET ANSI_NULLS ON;
SET QUOTED_IDENTIFIER ON;
```

3. Use `git status` to see both files in the working area
4. Stage only `schema.sql`: `git add schema.sql`
5. Use `git status` again - notice one file staged, one untracked
6. Modify `schema.sql` again by adding a comment: `-- Updated: [today's date]`
7. Use `git status` - notice `schema.sql` appears in both staged and unstaged sections

Part 2: Using Git Restore

Step 1: Removing Files from Staging

1. Remove `schema.sql` from staging: `git restore --staged schema.sql`
2. Verify with `git status` that it's now only in working area
3. Stage all files: `git add .`
4. Remove all files from staging: `git restore --staged .`

Step 2: Discarding Working Directory Changes

1. Make sure `schema.sql` has your modifications from earlier
2. Discard changes in working directory: `git restore schema.sql`
3. Verify the file reverted to its original state
4. Add your modifications back and commit them: "Add product dimension table"

Step 3: Practice with Multiple States

1. Create `fact_sales.sql`:

```sql
-- Sales Fact Table
CREATE TABLE dwh.fact_sales (
    sale_key INT IDENTITY(1,1) PRIMARY KEY,
    customer_key INT FOREIGN KEY REFERENCES dwh.dim_customer(customer_key),
    product_key INT FOREIGN KEY REFERENCES dwh.dim_product(product_key),
    sale_date DATE,
    quantity INT,
    unit_price DECIMAL(10,2),
    total_amount DECIMAL(10,2)
);
```

2. Stage the file: `git add fact_sales.sql`
3. Modify the file by adding an index:

```sql


-- Index for better query performance
CREATE INDEX IX_fact_sales_date ON dwh.fact_sales(sale_date);
```

4. Now you have: committed version, staged version, and working directory version
5. Use `git restore fact_sales.sql` - which version is restored?
6. Stage your changes and commit: "Add sales fact table with index"

Part 3: Using the Stash

Step 1: Basic Stash Operations

1. Create `data_quality.sql`:

```
-- Data Quality Checks
SELECT
    'Customers with missing email' as check_name,
    COUNT(*) as issue_count
FROM dwh.dim_customer
WHERE email IS NULL OR email = '';

-- More checks will be added here...
```

2. Add the file to staging: `git add data_quality.sql`
3. Start modifying `etl_process.sql` by adding error handling:

```
-- Error handling
BEGIN TRY
    -- ETL logic here
END TRY
BEGIN CATCH
    PRINT 'Error in ETL process: ' + ERROR_MESSAGE();
    THROW;
END CATCH;
```

4. You realize you need to switch context - stash your work: `git stash`
5. Verify your working directory and staging area are clean: `git status`
6. See your stash: `git stash list`

Step 2: Stash Management

1. Make a quick fix to `schema.sql` by adding a comment: `-- Version 1.1`
2. Commit this change: "Update schema version"
3. Restore your stashed work: `git stash pop`
4. Verify both staged and unstaged changes are back
5. Commit your work: "Add data quality checks and ETL error handling"

Step 3: Multiple Stashes

1. Create `backup_script.sql`:

```
-- Database backup script
BACKUP DATABASE DataWarehouse
TO DISK = 'C:\Backups\DataWarehouse.bak'
WITH FORMAT, INIT;
```

2. Stage it: `git add backup_script.sql`
3. Stash with a message: `git stash push -m "Backup script work in progress"`
4. Make different changes to `config.sql`:

```sql
-- Configuration settings
USE DataWarehouse;
SET ANSI_NULLS ON;
SET QUOTED_IDENTIFIER ON;

-- Performance settings
SET STATISTICS IO ON;
```

5. Stash this too: `git stash push -m "Config file updates"`
6. List stashes: `git stash list`
7. Apply the backup script stash: `git stash apply stash@{1}`
8. Clean up stashes: `git stash clear`

Part 4: Git Reset Practice

Step 1: Soft Reset (Undo Commit, Keep Changes)

1. Make sure all your work is committed
2. Create `monitoring.sql`:

```sql
-- Database monitoring queries
SELECT
    name,
    size_mb = CAST(size * 8.0 / 1024 AS DECIMAL(10,2))
FROM sys.master_files
WHERE database_id = DB_ID('DataWarehouse');
```

3. Commit this file: "Add database monitoring script"
4. Use `git log --oneline` to see your commits
5. Perform a soft reset to undo the last commit: `git reset --soft HEAD~1`
6. Check `git status` - the file should be staged
7. Check the file still exists in your working directory
8. Re-commit: "Add database monitoring script (fixed)"

Step 2: Mixed Reset (Default)

1. Modify `monitoring.sql` by adding another query:

```sql


-- Check database growth
SELECT
    name,
    physical_name,
    size_mb = CAST(size * 8.0 / 1024 AS DECIMAL(10,2))
FROM sys.database_files;
```

2. Commit: "Add database growth monitoring"
3. Reset: `git reset HEAD~1`
4. Check `git status` - changes should be in working directory, not staged

Step 3: Hard Reset (Danger Zone!)

1. Add some temporary debugging code to `etl_process.sql`:

```
-- TEMPORARY DEBUG CODE - REMOVE BEFORE PRODUCTION
PRINT 'Debug: Starting ETL process';
PRINT 'Debug: Processing customer data';
-- END DEBUG CODE
```

2. Stage and commit: "Add debug code (temporary)"
3. Perform hard reset: `git reset --hard HEAD~1`
4. Check that your debug code is completely gone
5. Verify with `git log --oneline` that the debug commit disappeared

Part 5: Git Revert Practice

Step 1: Safe "Undo" with Revert

1. Create `stored_procedures.sql`:

```
-- Stored Procedures
CREATE PROCEDURE GetCustomerSales
    @customer_id INT
AS
BEGIN
    SELECT
        c.first_name,
        c.last_name,
        SUM(f.total_amount) as total_sales
    FROM dwh.dim_customer c
    JOIN dwh.fact_sales f ON c.customer_key = f.customer_key
    WHERE c.customer_id = @customer_id
    GROUP BY c.first_name, c.last_name;
END;
```

2. Commit: "Add customer sales stored procedure"
3. Realize the procedure has a bug - revert it: `git revert HEAD`
4. Check `git log --oneline` - you should see both the original commit and the revert commit
5. Verify the file is gone from your working directory

Step 2: Reverting Multiple Commits

1. Create three small commits:

- Add `views.sql` with a simple view: "Add customer view"
- Add `functions.sql` with a date function: "Add date utility function"
- Modify `config.sql` to add a setting: "Update configuration"

2. Use `git log --oneline` to see all commits

3. Revert the middle commit (the functions one): `git revert HEAD~1`

4. Observe how Git handles this selective revert

Part 6: Recovery Scenarios

Scenario 1: Accidentally Staged Wrong Files

1. Create both `temp_analysis.sql` (a temporary file) and `permanent_analysis.sql` (important file)
2. Accidentally stage both: `git add .`
3. Remove only the temp file from staging: `git restore --staged temp_analysis.sql`
4. Commit only the permanent file

Scenario 2: Mixed Up Changes

1. Make changes to both `schema.sql` and `etl_process.sql`
2. Stage everything: `git add .`
3. Realize you want to commit each file separately
4. Unstage everything: `git restore --staged .`
5. Stage and commit each file individually

Scenario 3: Committed Too Early

1. Create `incomplete_feature.sql` with just:

```
-- New feature - work in progress
CREATE TABLE dwh.dim_time (
    date_key INT PRIMARY KEY
    -- More columns to be added
);
```

2. Commit: "Add time dimension (incomplete)"
3. Add the missing columns:

```
-- New feature - work in progress
CREATE TABLE dwh.dim_time (
    date_key INT PRIMARY KEY,
    full_date DATE,
    year INT,
    month INT,
    day INT,
    day_name NVARCHAR(10)
);
```

4. Use `git add` and `git commit --amend` to fix the previous commit
5. Verify with `git log` that you still have only one commit for this feature

Verification and Review

Check Your Understanding

1. Use `git log --oneline` to review your commit history
2. Use `git status` to ensure working directory is clean
3. Count how many times you used each command:
   - `git restore`
   - `git restore --staged`
   - `git reset`
   - `git revert`
   - `git stash`

Challenge Questions

1. What's the difference between `git reset --soft`, `git reset --mixed`, and `git reset --hard`?
2. When would you use `git revert` instead of `git reset`?
3. How is the stash different from creating a commit?
4. What happens to uncommitted changes when you use `git stash`?

Expected Outcomes

After completing this exercise, you should:

- Understand the difference between working area, staging area, and repository
- Know how to undo changes at different stages
- Be comfortable using the stash for temporary storage
- Understand when to use restore vs reset vs revert
- Have practiced recovering from common Git mistakes

Warning: Be very careful with `git reset --hard` and similar destructive commands in real projects. Always ensure you have backups or that your work is safely committed elsewhere!