Exercise 3: Branch Management and Feature Development

Learning Objectives

By completing this exercise, you will practice:

- Creating and switching between branches
- Understanding branch isolation and independence
- Merging feature branches into main
- Managing multiple features simultaneously
- Branch naming conventions and best practices

Scenario

You're working on a business intelligence dashboard project with multiple features being developed in parallel. You need to manage different feature branches while keeping the main branch stable for releases.

Setup

Initialize the project repository:

```
# Create project directory
mkdir BIDashboardProject
cd BIDashboardProject

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

# Set default branch name
git branch -M main
```

Part 1: Setting Up the Main Branch

Step 1: Create Project Foundation

Create the initial project structure:

`README.md`:

```
# Business Intelligence Dashboard

A comprehensive dashboard for business analytics and reporting.
```

```
## Features
- Customer analytics
- Sales reporting
- Financial metrics
- Performance dashboards

## Status
- Project initialized ✓
- Core infrastructure: In Progress
```

`src/database/schema.sql`:

```
-- Core database schema
CREATE DATABASE BIDashboard;
USE BIDashboard;

-- Core tables
CREATE TABLE audit_log (
    id INT IDENTITY(1,1) PRIMARY KEY,
    table_name NVARCHAR(50),
    operation NVARCHAR(10),
    timestamp DATETIME2 DEFAULT GETDATE(),
    user_name NVARCHAR(50)
);
```

`config/database_config.sql`:

```
-- Database configuration
USE BIDashboard;

-- Set default settings
SET ANSI_NULLS ON;
SET QUOTED_IDENTIFIER ON;
SET ARITHABORT ON;
```

1. Create the directory structure: `mkdir src\database`, `mkdir config`
2. Create all the files above
3. Stage and commit everything: "Initial project setup with core infrastructure"
4. Verify you're on main branch: `git branch`

Part 2: Feature Branch Development

Step 1: Customer Analytics Feature

1. Create and switch to a feature branch: `git switch -c feature/customer-analytics`
2. Verify you're on the new branch: `git branch`

Create `src/database/customer_tables.sql`:

```
-- Customer Analytics Tables
USE BIDashboard;

CREATE TABLE dim_customer (
    customer_key INT IDENTITY(1,1) PRIMARY KEY,
    customer_id INT NOT NULL UNIQUE,
    first_name NVARCHAR(50),
    last_name NVARCHAR(50),
    email NVARCHAR(100),
    registration_date DATE,
    customer_segment NVARCHAR(20),
    created_date DATETIME2 DEFAULT GETDATE()
);

CREATE TABLE fact_customer_activity (
    activity_key INT IDENTITY(1,1) PRIMARY KEY,
    customer_key INT FOREIGN KEY REFERENCES dim_customer(customer_key),
    activity_date DATE,
    page_views INT,
    session_duration_minutes INT,
    actions_taken INT
);
```

Create `src/queries/customer_analytics.sql`:

```
-- Customer Analytics Queries
USE BIDashboard;

-- Customer segmentation analysis
SELECT
    customer_segment,
    COUNT(*) as customer_count,
    AVG(DATEDIFF(day, registration_date, GETDATE())) as
avg_days_since_registration
FROM dim_customer
GROUP BY customer_segment;

-- Top active customers
SELECT TOP 10
    c.first_name + ' ' + c.last_name as customer_name,
    c.email,
    SUM(f.page_views) as total_page_views,
    SUM(f.actions_taken) as total_actions
FROM dim_customer c
JOIN fact_customer_activity f ON c.customer_key = f.customer_key
GROUP BY c.customer_key, c.first_name, c.last_name, c.email
ORDER BY total_page_views DESC;
```

3. Create the queries directory: `mkdir src\queries`
4. Commit your customer analytics work: "Add customer analytics tables and queries"

Step 2: Sales Reporting Feature (Parallel Development)

1. Switch back to main: `git switch main`
2. Create a new feature branch: `git switch -c feature/sales-reporting`

Create `src/database/sales_tables.sql`:

```
-- Sales Reporting Tables
USE BIDashboard;

CREATE TABLE dim_product (
    product_key INT IDENTITY(1,1) PRIMARY KEY,
    product_id INT NOT NULL UNIQUE,
    product_name NVARCHAR(100),
    category NVARCHAR(50),
    subcategory NVARCHAR(50),
    unit_price DECIMAL(10,2),
    created_date DATETIME2 DEFAULT GETDATE()
);

CREATE TABLE fact_sales (
    sale_key INT IDENTITY(1,1) PRIMARY KEY,
    customer_key INT, -- Will link to customer table later
    product_key INT FOREIGN KEY REFERENCES dim_product(product_key),
    sale_date DATE,
    quantity INT,
    unit_price DECIMAL(10,2),
    discount_percentage DECIMAL(5,2),
    total_amount DECIMAL(10,2)
);
```

Create `src/queries/sales_reports.sql`:

```
-- Sales Reporting Queries
USE BIDashboard;

-- Monthly sales summary
SELECT
    YEAR(sale_date) as sale_year,
    MONTH(sale_date) as sale_month,
    COUNT(*) as total_transactions,
    SUM(total_amount) as total_revenue,
    AVG(total_amount) as avg_transaction_value
FROM fact_sales
GROUP BY YEAR(sale_date), MONTH(sale_date)
ORDER BY sale_year DESC, sale_month DESC;
```

```
-- Top selling products
SELECT TOP 10
    p.product_name,
    p.category,
    SUM(s.quantity) as total_quantity_sold,
    SUM(s.total_amount) as total_revenue
FROM dim_product p
JOIN fact_sales s ON p.product_key = s.product_key
GROUP BY p.product_key, p.product_name, p.category
ORDER BY total_revenue DESC;
```

3. Commit the sales reporting work: "Add sales reporting tables and queries"

Step 3: Financial Metrics Feature

1. Switch back to main: `git switch main`
2. Create another feature branch: `git switch -c feature/financial-metrics`

Create `src/database/financial_tables.sql`:

```
-- Financial Metrics Tables
USE BIDashboard;

CREATE TABLE dim_account (
    account_key INT IDENTITY(1,1) PRIMARY KEY,
    account_code NVARCHAR(20) NOT NULL UNIQUE,
    account_name NVARCHAR(100),
    account_type NVARCHAR(20), -- Revenue, Expense, Asset, Liability
    parent_account_code NVARCHAR(20),
    is_active BIT DEFAULT 1
);

CREATE TABLE fact_financial_transactions (
    transaction_key INT IDENTITY(1,1) PRIMARY KEY,
    account_key INT FOREIGN KEY REFERENCES dim_account(account_key),
    transaction_date DATE,
    description NVARCHAR(200),
    debit_amount DECIMAL(15,2),
    credit_amount DECIMAL(15,2),
    reference_number NVARCHAR(50)
);
```

Create `src/queries/financial_reports.sql`:

```
-- Financial Reporting Queries
USE BIDashboard;

-- Profit and Loss Statement
SELECT
```

```
    a.account_type,
    a.account_name,
    SUM(
        CASE
            WHEN a.account_type IN ('Revenue') THEN f.credit_amount -
f.debit_amount
            WHEN a.account_type IN ('Expense') THEN f.debit_amount -
f.credit_amount
            ELSE 0
        END
    ) as net_amount
FROM dim_account a
JOIN fact_financial_transactions f ON a.account_key = f.account_key
WHERE a.account_type IN ('Revenue', 'Expense')
    AND f.transaction_date >= DATEADD(month, -1, GETDATE())
GROUP BY a.account_type, a.account_name
ORDER BY a.account_type, net_amount DESC;

-- Cash flow analysis
SELECT
    DATEPART(week, transaction_date) as week_number,
    SUM(credit_amount) as total_inflow,
    SUM(debit_amount) as total_outflow,
    SUM(credit_amount) - SUM(debit_amount) as net_flow
FROM fact_financial_transactions
WHERE transaction_date >= DATEADD(month, -1, GETDATE())
GROUP BY DATEPART(week, transaction_date)
ORDER BY week_number;
```

3. Commit the financial metrics work: "Add financial metrics tables and reports"

Part 3: Branch Inspection and Comparison

Step 1: Examine Branch Differences

1. List all branches: `git branch`
2. Switch to main: `git switch main`
3. List files in src/database: `dir src\database`
4. Switch to customer analytics: `git switch feature/customer-analytics`
5. List files again: `dir src\database`
6. Notice how the file contents change between branches

Step 2: View Branch History

1. See the commit history of current branch: `git log --oneline`
2. Switch to sales reporting: `git switch feature/sales-reporting`
3. Compare the commit history: `git log --oneline`
4. Switch to main and compare: `git switch main` then `git log --oneline`

Step 3: Compare Branch Contents

From main branch, compare what's different:

1. `git diff main feature/customer-analytics --name-only`
2. `git diff main feature/sales-reporting --name-only`
3. `git diff main feature/financial-metrics --name-only`

Part 4: Merging Features

Step 1: Merge Customer Analytics

1. Ensure you're on main: `git switch main`
2. Merge the customer analytics feature: `git merge feature/customer-analytics`
3. Check the commit history: `git log --oneline`
4. Verify the new files are present: `dir src\database`

Step 2: Update README After First Merge

Update `README.md` to reflect completed feature:

```
# Business Intelligence Dashboard

A comprehensive dashboard for business analytics and reporting.

## Features
- Customer analytics ✓
- Sales reporting: In Progress
- Financial metrics: In Progress
- Performance dashboards: Planned

## Status
- Project initialized ✓
- Core infrastructure ✓
- Customer analytics feature ✓
```

Commit this update: "Update README after customer analytics completion"

Step 3: Test Merge Conflict Resolution

1. Switch to sales-reporting branch: `git switch feature/sales-reporting`
2. Modify the README.md in this branch:

```
# Business Intelligence Dashboard

A comprehensive dashboard for business analytics and reporting.

## Features
- Customer analytics
- Sales reporting ✓
- Financial metrics: In Progress
```

```
- Performance dashboards: Planned

## Status
- Project initialized ✓
- Core infrastructure ✓
- Sales reporting feature ✓
```

3. Commit: "Update README for sales reporting completion"
4. Switch to main: `git switch main`
5. Try to merge: `git merge feature/sales-reporting`
6. You should get a merge conflict in README.md
7. Open README.md in a text editor and resolve the conflict by combining both updates:

```
# Business Intelligence Dashboard

A comprehensive dashboard for business analytics and reporting.

## Features
- Customer analytics ✓
- Sales reporting ✓
- Financial metrics: In Progress
- Performance dashboards: Planned

## Status
- Project initialized ✓
- Core infrastructure ✓
- Customer analytics feature ✓
- Sales reporting feature ✓
```

8. Stage the resolved file: `git add README.md`
9. Complete the merge: `git commit -m "Merge sales-reporting feature with conflict resolution"`

Part 5: Branch Cleanup and Management

Step 1: Clean Up Merged Branches

1. List all branches: `git branch`
2. Delete the merged customer analytics branch: `git branch -d feature/customer-analytics`
3. Delete the merged sales reporting branch: `git branch -d feature/sales-reporting`
4. Verify branches are deleted: `git branch`

Step 2: Continue with Remaining Features

1. Switch to financial metrics: `git switch feature/financial-metrics`
2. Check if it's up to date with main's latest changes
3. If needed, merge main into your feature branch: `git merge main`
4. Switch back to main: `git switch main`

5. Merge financial metrics: `git merge feature/financial-metrics`
6. Delete the merged branch: `git branch -d feature/financial-metrics`

Step 3: Final Project State

Update the final README.md:

```
# Business Intelligence Dashboard

A comprehensive dashboard for business analytics and reporting.

## Features
- Customer analytics ✓
- Sales reporting ✓
- Financial metrics ✓
- Performance dashboards: Next Phase

## Status
- Project initialized ✓
- Core infrastructure ✓
- All core features completed ✓

## Next Steps
- Add performance dashboards
- Create data visualization components
- Set up automated reporting
```

Commit: "Final README update - all core features complete"

Part 6: Advanced Branch Scenarios

Challenge 1: Hotfix Branch

1. Create a hotfix branch: `git switch -c hotfix/fix-schema-bug`
2. "Fix" a bug in `config/database_config.sql` by adding:

```
-- Database configuration
USE BIDashboard;

-- Set default settings
SET ANSI_NULLS ON;
SET QUOTED_IDENTIFIER ON;
SET ARITHABORT ON;

-- Fix: Add missing isolation level setting
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

3. Commit: "Fix missing transaction isolation level setting"
4. Merge into main: `git switch main` then `git merge hotfix/fix-schema-bug`

5. Delete hotfix branch: `git branch -d hotfix/fix-schema-bug`

Challenge 2: Experimental Branch

1. Create an experimental branch: `git switch -c experiment/new-reporting-engine`
2. Create `src/experimental/new_engine.sql`:

```
-- Experimental new reporting engine
-- This is just a proof of concept

CREATE VIEW v_unified_metrics AS
SELECT
    'Customer' as metric_type,
    COUNT(*) as metric_value,
    'Active customers' as metric_description
FROM dim_customer
WHERE customer_segment IS NOT NULL

UNION ALL

SELECT
    'Sales' as metric_type,
    SUM(total_amount) as metric_value,
    'Total revenue' as metric_description
FROM fact_sales;
```

3. Commit: "Add experimental unified metrics view"
4. Switch back to main: `git switch main`
5. Decide not to merge this experiment (keep branch for later): `git branch`

Verification and Review

Review Your Work

1. Check final branch list: `git branch`
2. View complete project history: `git log --oneline --graph`
3. List all files in your project: `git ls-files`
4. Verify main branch has all merged features

Understanding Check

Answer these questions:

1. How many branches did you create total?
2. How many branches did you merge into main?
3. What happens to files when you switch branches?
4. Why is it important to switch to main before creating new feature branches?
5. What's the difference between `git branch` and `git checkout -b`?

Expected Outcomes

After completing this exercise, you should:

- Understand how branches provide isolated development environments
- Know how to create, switch, and merge branches
- Have experience with merge conflicts and resolution
- Understand branch cleanup and maintenance
- Be comfortable with parallel feature development workflows

Best Practices Learned

- Always create feature branches from an up-to-date main branch
- Use descriptive branch names with prefixes (feature/, hotfix/, experiment/)
- Keep feature branches focused on single functionality
- Delete merged branches to keep repository clean
- Resolve merge conflicts carefully by understanding both changes
- Test branches before merging into main

Note: In real projects, you would typically use pull requests for code review before merging feature branches into main.