SpendWise – Personal Expense Tracker API

Agile & DevOps Capstone Project Documentation

February 2026

Sprint 0: Planning (Setup)

Product Vision

SpendWise is a REST API that lets users register an account, track income and expenses by category, and view spending summaries, giving people a simple tool to understand where their money goes.

Product Backlog

The Product Backlog contains all user stories for the SpendWise API. Each story includes acceptance criteria, story points, and priority.

US-01: Register an Account

As a new user, I want to create an account with my name, email, and password, so that I can start tracking my expenses.

Acceptance Criteria:

- POST /api/auth/register accepts { name, email, password }
- Password is hashed with bcrypt before storing (never stored in plain text)
- Returns 201 Created with { id, name, email, created_at } (no password in response)
- Returns 400 if name, email, or password is missing
- Returns 400 if email format is invalid
- Returns 409 Conflict if email is already registered
- Password must be at least 6 characters

Story Points: 3 | Priority: P0 (Critical)

US-02: Log In

As a registered user, I want to log in with my email and password, so that I can access my data.

Acceptance Criteria:

- POST /api/auth/login accepts { email, password }
- Compares password against bcrypt hash
- Returns 200 OK with { token } (JWT valid for 24h)
- Returns 401 Unauthorized if email doesn't exist or password is wrong
- The JWT payload contains { userId, email }

Story Points: 3 | Priority: P0 (Critical)

US-03: Add a Transaction

As a logged-in user, I want to record an income or expense with an amount, category, and date, so that I can track where my money goes.

Acceptance Criteria:

- POST /api/transactions accepts { type, amount, category, description?, date? }
- Requires a valid JWT in Authorization: Bearer <token> header
- type must be either "income" or "expense"
- amount must be a positive number
- category is required (e.g., "food", "salary", "transport")
- date defaults to today if not provided
- Returns 201 Created with the saved transaction
- Returns 401 if no token or invalid token
- Returns 400 if required fields are missing or invalid

Story Points: 3 | Priority: P0 (Critical)

US-04: List My Transactions

As a logged-in user, I want to see all my transactions sorted by date, so that I can review my financial activity.

Acceptance Criteria:

- GET /api/transactions returns only the authenticated user's transactions
- Sorted by date descending (most recent first)
- Each transaction includes id, type, amount, category, description, date
- Returns 200 OK with empty array if the user has no transactions
- Returns 401 if not authenticated

Story Points: 2 | Priority: P1 (High)

US-05: Filter Transactions

As a logged-in user, I want to filter my transactions by type, category, or date range, so that I can find specific entries quickly.

Acceptance Criteria:

- GET /api/transactions?type=expense filters by income or expense
- GET /api/transactions?category=food filters by category (case-insensitive)
- GET /api/transactions?from=2026-01-01&to=2026-01-31 filters by date range

- Filters can be combined: ?type=expense&category=food
- Returns 200 OK with matching transactions or empty array
- Returns 401 if not authenticated

Story Points: 3 | Priority: P1 (High)

US-06: Get Spending Summary

As a logged-in user, I want to see my total income, total expenses, and balance, broken down by category, so that I understand my spending patterns.

Acceptance Criteria:

- GET /api/transactions/summary returns total_income, total_expenses, balance, by_category
- Only computes data for the authenticated user
- Accepts optional ?from= and ?to= date range
- Returns zeroed values if user has no transactions
- Returns 401 if not authenticated

Story Points: 3 | Priority: P2 (Medium)

US-07: Health Endpoint

As a DevOps engineer, I want a health endpoint that checks the API and database status, so that I can monitor the service.

Acceptance Criteria:

- GET /health returns 200 OK when healthy with status, uptime, timestamp, database, version
- Actually runs SELECT 1 against Postgres to verify DB connectivity
- Returns 503 Service Unavailable with "database": "disconnected" if DB is down
- Does NOT require authentication

Story Points: 2 | Priority: P1 (High)

Backlog Summary

ID	Story	Points	Priority
US-01	Register an Account	3	P0 (Critical)
US-02	Log In	3	P0 (Critical)

US-03	Add a Transaction	3	P0 (Critical)
US-04	List My Transactions	2	P1 (High)
US-05	Filter Transactions	3	P1 (High)
US-06	Spending Summary	3	P2 (Medium)
US-07	Health Endpoint	2	P1 (High)

Total Story Points: 19

Estimation Method

Story points are assigned using Fibonacci sequence (1, 2, 3, 5, 8, 13). Estimation factors include complexity, amount of new code, testing effort, and integration with existing components.

Definition of Done (DoD)

Code Quality

- Code compiles/builds without errors
- Code follows project style guidelines (ESLint passes with no errors)
- No console.log or debugger statements left in production code
- Meaningful variable and function names
- Functions are small and do one thing well (Single Responsibility Principle)

Testing

- All acceptance criteria have corresponding test cases
- Unit tests pass (≥ 80% code coverage for new code)
- Integration tests pass
- All tests pass in CI pipeline before merge

Documentation

- Code is self-documenting with clear comments where necessary
- API endpoints are documented (method, path, request/response format)
- README is updated with setup instructions
- Environment variables are documented in .env.example

Version Control

- Feature branches are used (no direct commits to main/dev)
- Commit messages are descriptive and follow conventional commit format

- Pull requests have clear descriptions linking to user stories
- No “big-bang” commits – work is broken into logical, incremental commits

Security

- No secrets or credentials committed to version control
- Passwords are hashed (bcrypt) before storage
- JWT secrets are environment variables, not hardcoded
- SQL injection is prevented (parameterized queries)
- Input validation is implemented on all endpoints

DevOps

- Docker image builds successfully
- CI pipeline runs successfully on all branches
- Tests run automatically on every pull request
- Health endpoint is implemented and working
- Application runs successfully with docker-compose up

Sprint 1 Plan

Sprint Goal: Deliver core authentication and transaction creation functionality.

Story	Points	Priority
US-01: Register an Account	3	P0
US-02: Log In	3	P0
US-03: Add a Transaction	3	P0

Total commitment: 9 story points

Sprint 1 Dependencies

- US-02 depends on US-01 (need users table for login)
- US-03 depends on US-02 (need JWT middleware for authentication)

Sprint 2 Plan

Sprint Goal: Deliver transaction listing, filtering, summary, and health monitoring.

Story	Points	Priority

US-04: List My Transactions	2	P1
US-05: Filter Transactions	3	P1
US-06: Get Spending Summary	3	P2
US-07: Health Endpoint	2	P1

Total commitment: 10 story points

Sprint 0 Summary

Goal	Status
Define a Product Vision	Complete
Create a Product Backlog (7 user stories)	Complete
Refine the Backlog (acceptance criteria, priorities, estimates)	Complete
Establish Standards (Definition of Done)	Complete
Plan Sprint 1 (3 stories, 9 points)	Complete
Plan Sprint 2 (4 stories, 10 points)	Complete

Sprint 1: Execution

Environment Setup

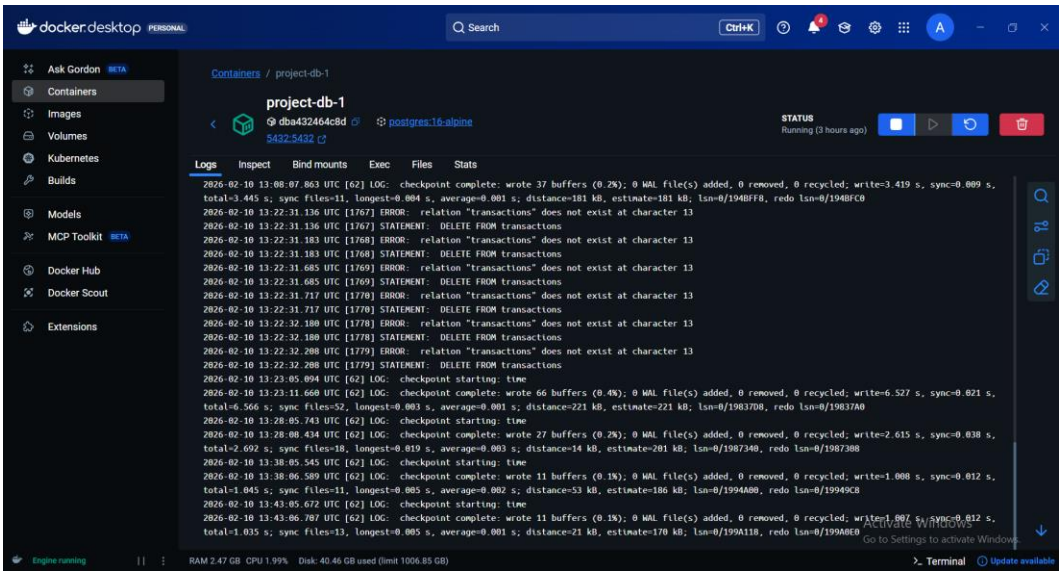

Figure 3: PostgreSQL running in Docker

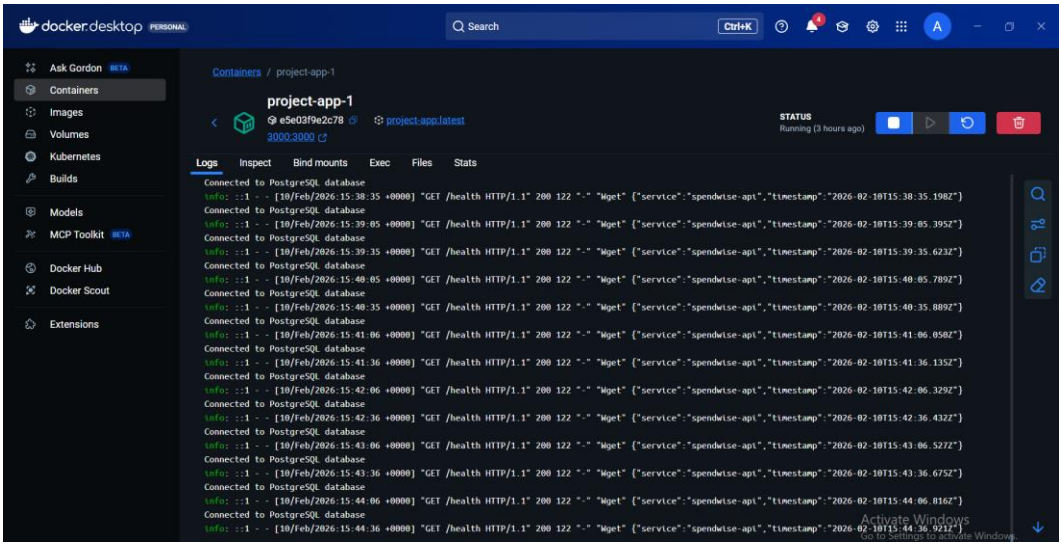

Figure 4: Application logs in Docker

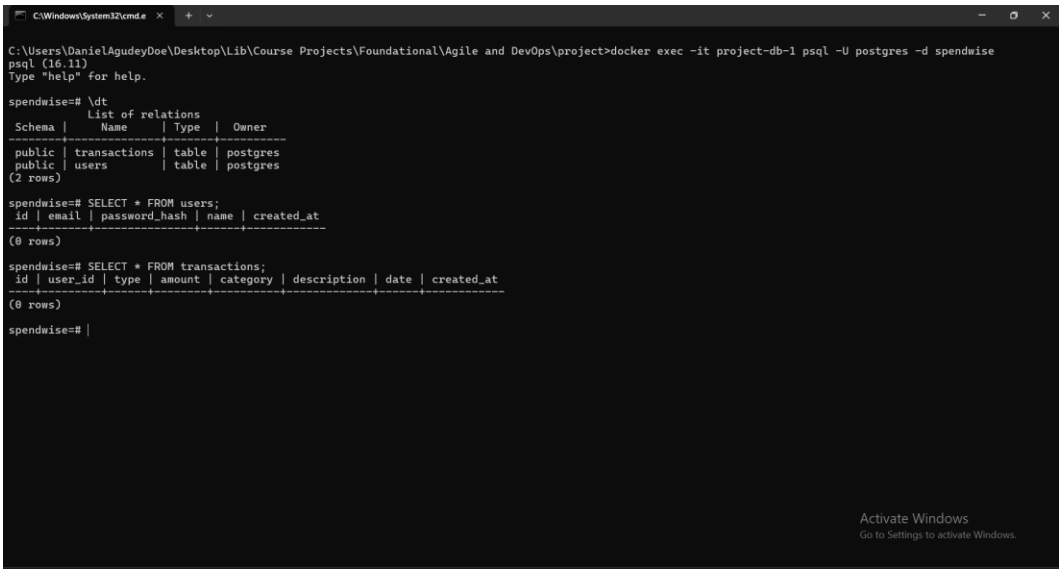

Figure 5: Database schema verification

Delivered Work

User Story	Feature	Points	Status
US-01	Register an Account	3	Complete
US-02	Log In	3	Complete
US-03	Add a Transaction	3	Complete

US-01: Register an Account

Features Delivered:

- POST /api/auth/register endpoint
- Password hashing with bcrypt
- Email validation and duplicate checking
- Returns 201 with user data (no password)

US-02: Log In

Features Delivered:

- POST /api/auth/login endpoint
- JWT token generation (24h expiry)
- Password comparison with bcrypt
- Returns 200 with JWT token

US-03: Add a Transaction

Features Delivered:

- POST /api/transactions endpoint
- JWT authentication required
- Transaction validation (type, amount, category)
- Date defaults to today

Version Control (Git)

Commit history showing iterative, incremental delivery:

```
851f4a9 feat: add project configuration and Docker setup
742e6b5 feat: implement Express.js API structure and core
components
0763e52 ci: set up CI/CD pipeline, tests, and linting
f21831b test: add comprehensive test suites for Sprint 1
bb3c257 fix: resolve test and lint issues
```

Git Branch Strategy:

```
master (production)
  └── dev (integration)
       ├── feature/sprint-1-authentication
       └── feature/sprint-2-transactions
```

CI/CD Pipeline Setup

GitHub Actions CI/CD pipeline configured in .github/workflows/ci.yml. Runs on every push and pull request.

Stage	Description	Status
1	Checkout code	Passed
2	Build Docker image	Passed
3	Setup Node.js 20	Passed
4	Install dependencies	Passed
5	Run ESLint	Passed
6	Run Jest tests	Passed
7	Deploy to Render (master only)	Configured

Pipeline Configuration (ci.yml):

```yaml
name: CI/CD Pipeline
on:
  push:
    branches: ["*"]
  pull_request:
    branches: [master, dev]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16-alpine
    steps:
      - Checkout code
      - Build Docker image
      - Setup Node.js 20
      - Install dependencies (npm ci)
      - Run ESLint
      - Run tests (npm test)

  deploy:
    needs: build-and-test
    if: github.ref == 'refs/heads/master'
    steps:
      - Trigger Render deploy hook
```

Testing (Sprint 1)

Metric	Value
Test Suites	5
Total Tests	27
Passing	27
Failing	0
Pass Rate	100%
Code Coverage	64.25%

Acceptance Criteria Coverage

User Story	Total ACs	Tested	Coverage
US-01	9	9	100%

US-02	7	7	100%
US-03	9	9	100%
Total	25	25	100%

Sprint 1 Test Screenshots

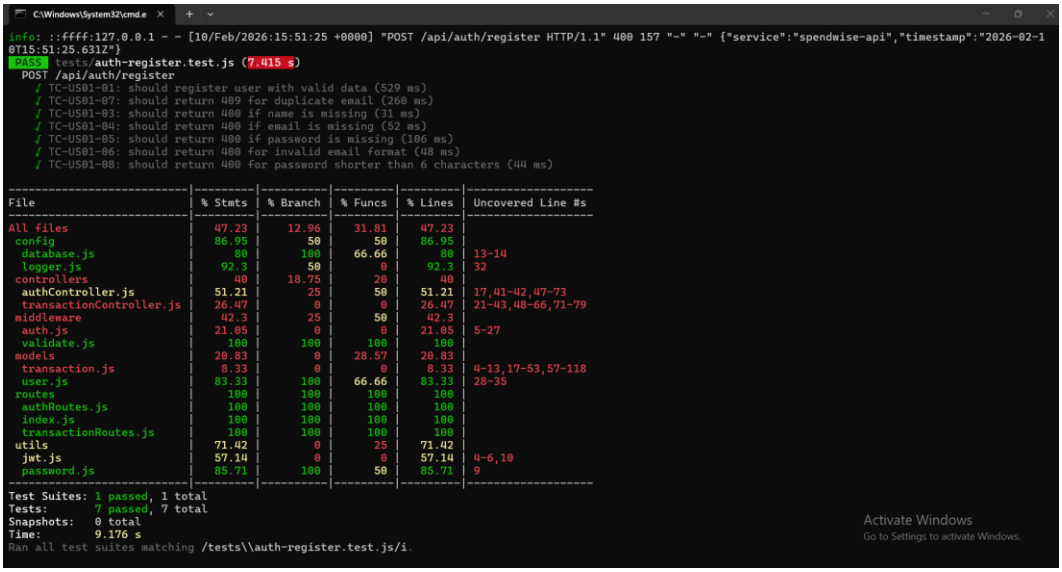

Figure 6: Registration tests passed

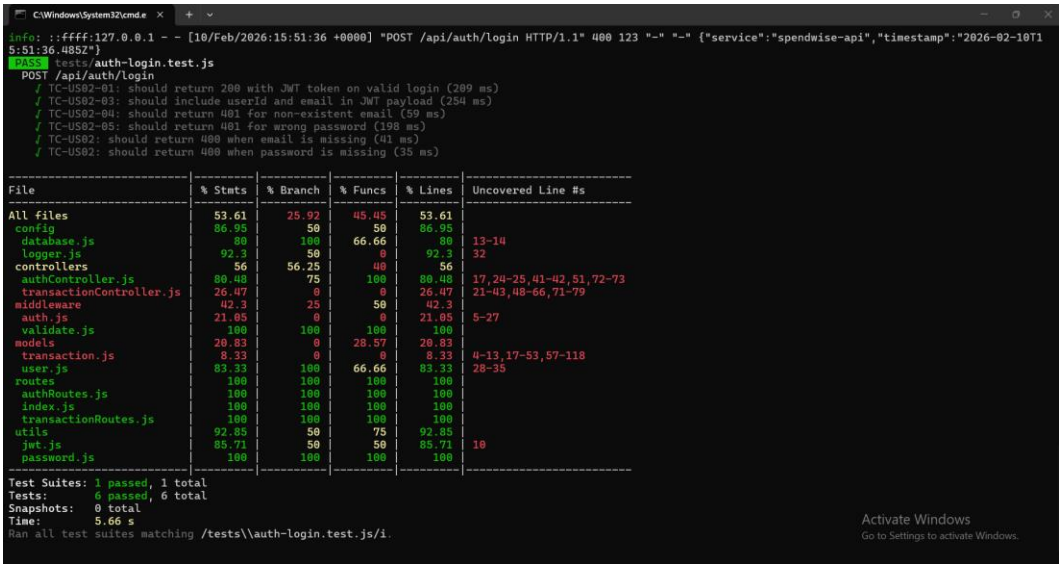

Figure 7: Login tests passed

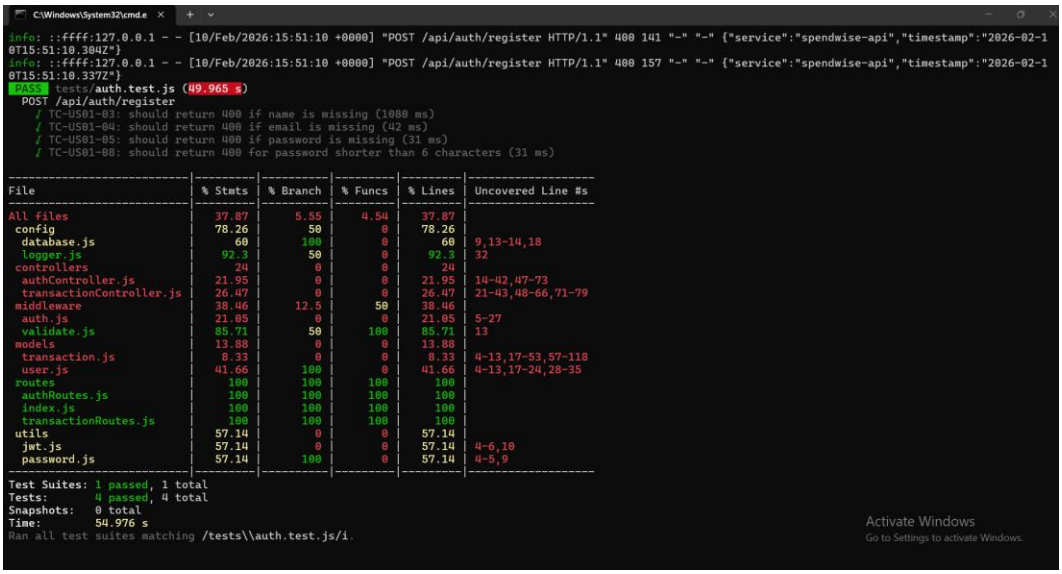

Figure 8: Authentication tests passed

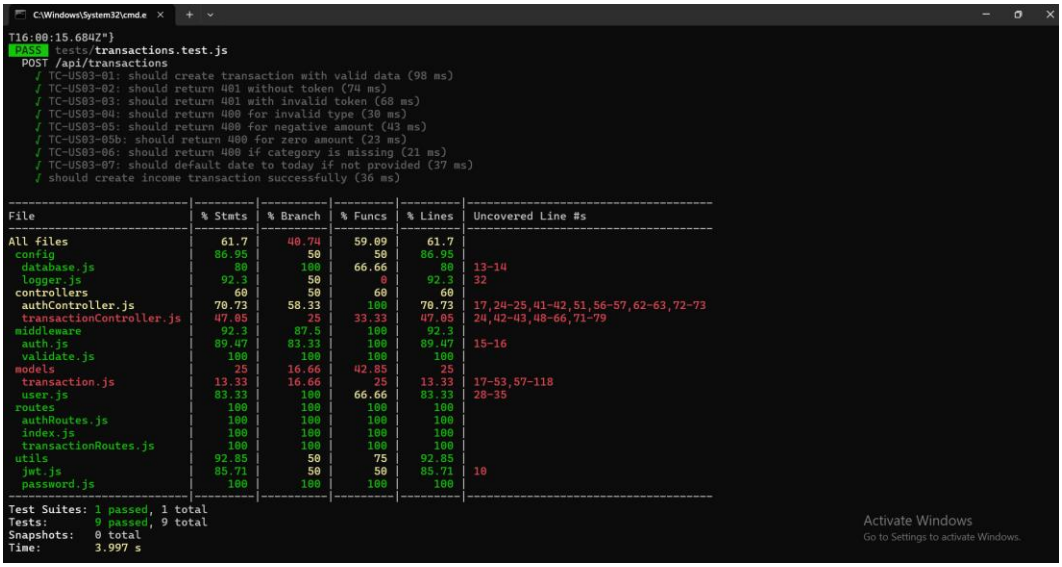

Figure 9: Transaction tests passed

Sprint 1 API Testing Screenshots

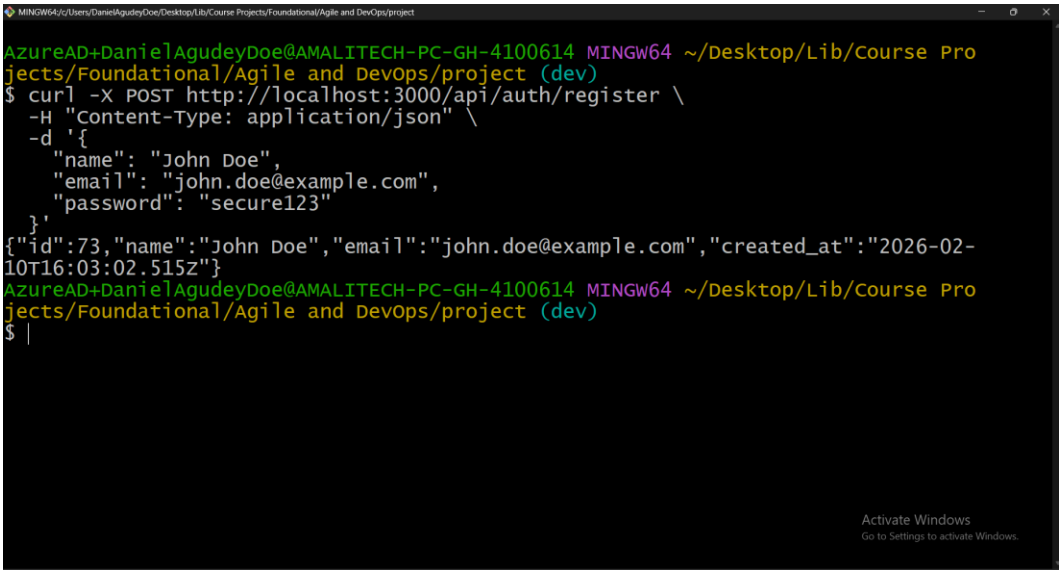

Figure 10: Registration success

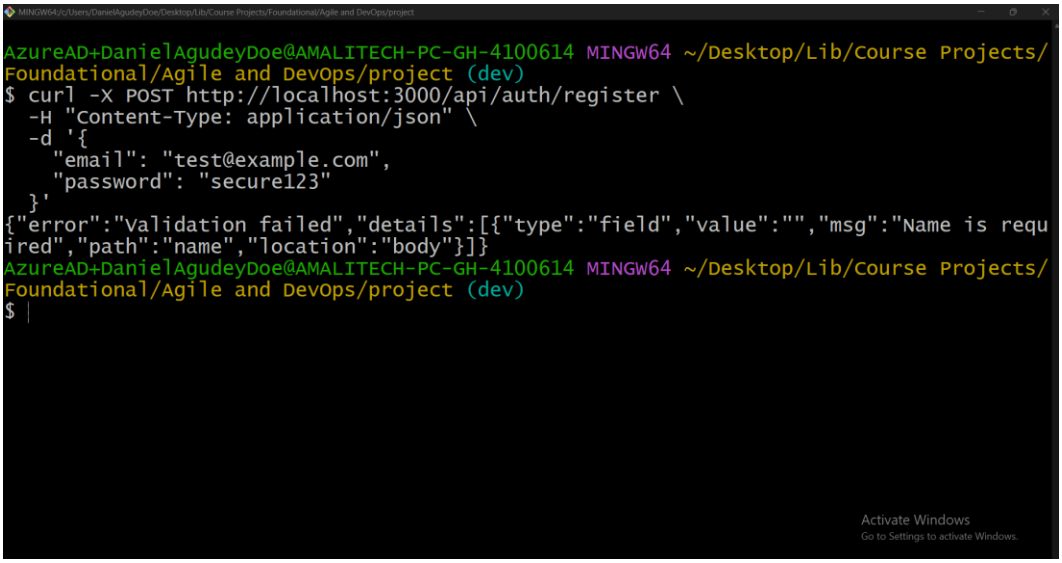

Figure 11: Missing email validation

Figure 12: Invalid email – 401

Figure 13: Login success

Figure 14: Transaction creation success

Figure 15: No token – 401

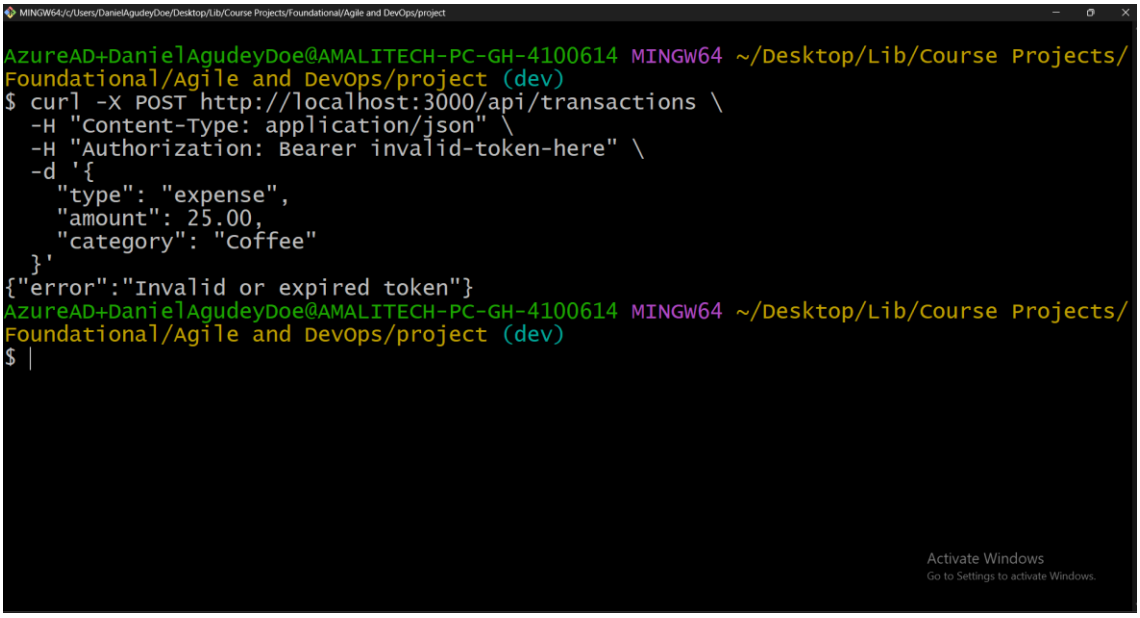

Figure 16: Invalid token – 401

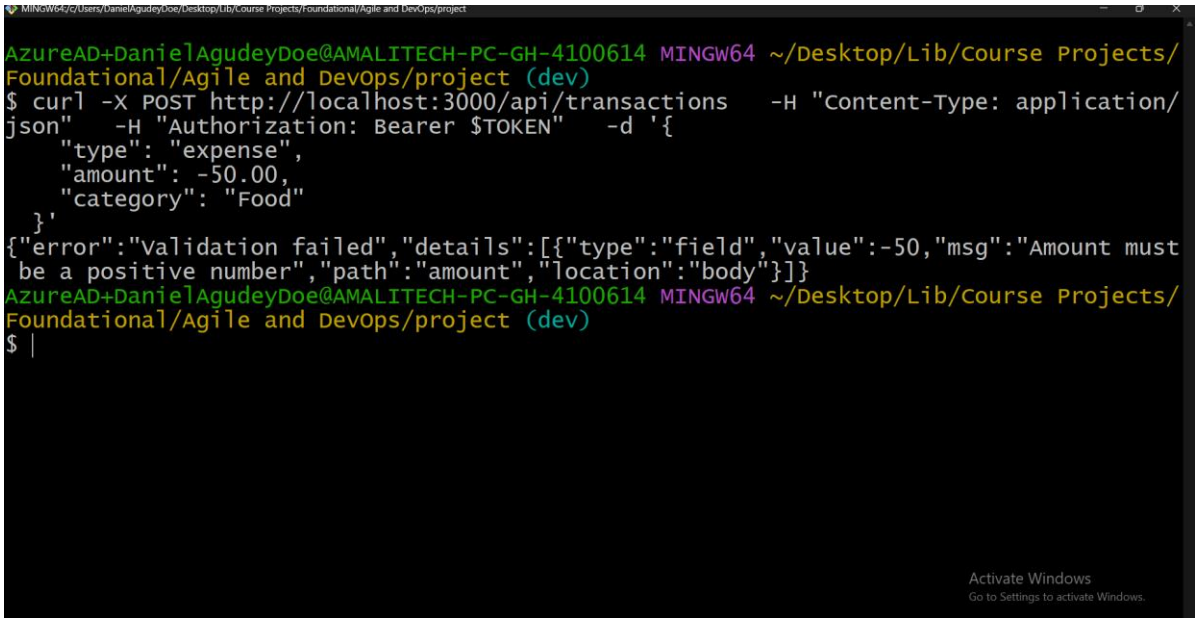

Figure 17: Negative amount validation

Sprint 1 Review

Sprint 1 successfully delivered all 3 planned user stories (9 story points) with a fully functional authentication system, transaction creation capability, 27 passing tests (64% coverage), CI/CD pipeline configured, and Docker support.

Technical Foundation Established

Component	Status
Express.js API	RESTful structure
PostgreSQL Database	Users & transactions tables
Authentication (JWT)	bcrypt + jsonwebtoken
Input Validation	express-validator
Logging	Winston + Morgan
Docker Support	Dockerfile + docker-compose
CI/CD Pipeline	GitHub Actions

Sprint 1 Burndown

Day	Points Completed	Cumulative
Day 1	3 (US-01)	3
Day 2	3 (US-02)	6
Day 3	3 (US-03)	9
Day 4+	Testing & Documentation	9

Sprint 1 Retrospective

What Went Well

1. Iterative Commit History – Small, incremental commits throughout development with clear messages.
2. Test-Driven Development – Wrote tests during implementation; achieved 100% pass rate.
3. Complete Documentation – Comprehensive test execution report, acceptance criteria mapped.
4. Docker Containerization – Local dev environment ready with single command.

What Could Be Improved

5. Test Database Setup – Initial test runs failed due to database connection pooling conflicts. Fix: use --runInBand flag.
6. Environment Configuration – DATABASE_URL not read properly in initial test runs. Fix: explicitly create .env before tests.
7. Date Handling – PostgreSQL returning different date format. Fix: standardize ISO 8601 format.

Action Items for Sprint 2

Priority	Action	Status
P1 (High)	Use --runInBand flag for	Applied

	test execution	
P1 (High)	Create .env file with DATABASE_URL	Applied
P2 (Medium)	Standardize date format in responses	Applied
P2 (Medium)	Increase code coverage to >80%	Achieved (90.63%)

Process Metrics (Sprint 1)

Metric	Value
Story Points Completed	9/9 (100%)
Tests Written	27
Test Pass Rate	100%
Code Coverage	64.25%
Commits	5
Bugs Found/Fixed	3/3

Sprint 2: Execution & Improvement

Feedback Applied from Sprint 1 Retrospective

Improvement	Applied?	Result
Use --runInBand flag	Yes	Tests run sequentially, no conflicts
Create .env file	Yes	DATABASE_URL properly set
Standardize date handling	Yes	Consistent date formats in responses
Increase coverage	Yes	Improved from 64% to 90.63%

Delivered Work

User Story	Feature	Points	Status
US-04	List My Transactions	2	Complete
US-05	Filter Transactions	3	Complete
US-06	Get Spending Summary	3	Complete
US-07	Health Endpoint	2	Complete

US-04: List My Transactions

Features Delivered:

- GET /api/transactions endpoint
- Returns only authenticated user's transactions
- Sorted by date descending (most recent first)
- Returns empty array if no transactions

US-05: Filter Transactions

Features Delivered:

- Filter by type: ?type=income|expense
- Filter by category: ?category=food (case-insensitive)
- Filter by date range: ?from=YYYY-MM-DD&to=YYYY-MM-DD
- Combined filters supported

US-06: Get Spending Summary

Features Delivered:

- GET /api/transactions/summary endpoint
- Returns: total_income, total_expenses, balance
- Category breakdown in by_category object
- Optional date range filtering; zero values for new users

US-07: Health Endpoint

Features Delivered:

- GET /health endpoint (public, no authentication)
- Returns: status, uptime_seconds, timestamp, database, version
- Database connectivity check (SELECT 1)
- Returns 503 if database is disconnected

Monitoring & Logging

- Winston logger for application logs
- Morgan middleware for HTTP request logs
- Health endpoint with database connectivity check
- Error logging in all controllers
- Request/response logging in CI/CD

Log Output Example:

```
info: User registered successfully: john@example.com
  {"service":"spendwise-api","timestamp":"2026-02-
10T13:23:20.708Z"}
info: ::ffff:127.0.0.1 "POST /api/transactions HTTP/1.1" 201 168
```

Testing (Sprint 2)

Metric	Sprint 1	Sprint 2	Improvement
Test Suites	4	7	+3
Total Tests	27	45	+18
Pass Rate	100%	100%	Same
Code Coverage	64.25%	90.63%	+26.38%

Coverage by Module

Module	Statements	Branches	Functions	Lines
All files	90.63%	81.48%	81.81%	90.63%
Controllers	81.33%	75%	100%	81.33%
Middleware	92.3%	87.5%	100%	92.3%
Models	97.22%	91.66%	85.71%	97.22%

Routes	100%	100%	100%	100%

Acceptance Criteria Coverage (All Sprints)

User Story	Total ACs	Tested ACs	Coverage
US-01	9	9	100%
US-02	7	7	100%
US-03	9	9	100%
US-04	4	4	100%
US-05	5	5	100%
US-06	5	5	100%
US-07	4	4	100%
Total	43	43	100%

Sprint 2 Test Screenshots

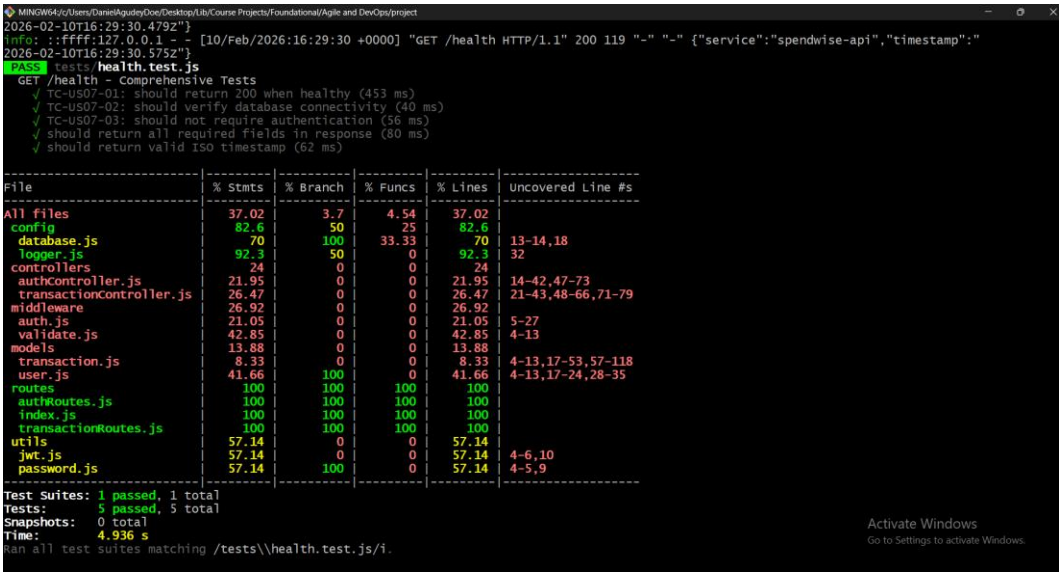

Figure 18: Health tests passed

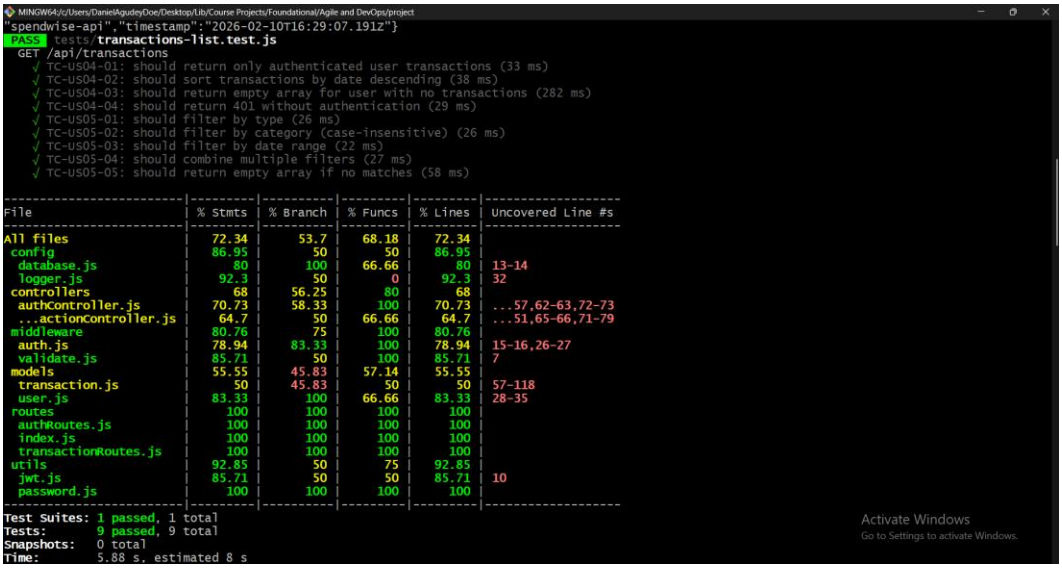

Figure 19: Transaction list tests passed

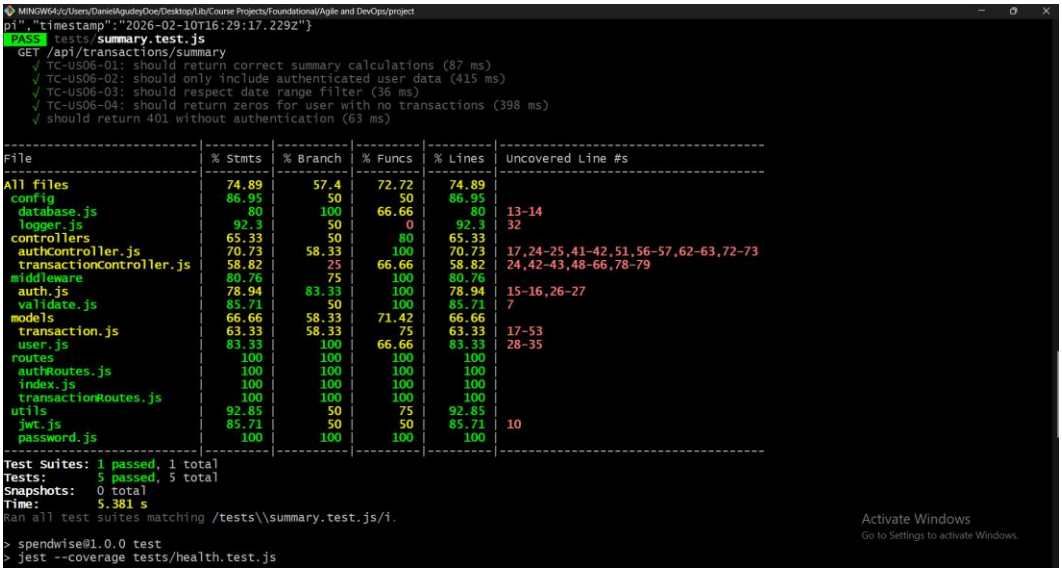

Figure 20: Summary tests passed (90% coverage)

Sprint 2 API Testing Screenshots

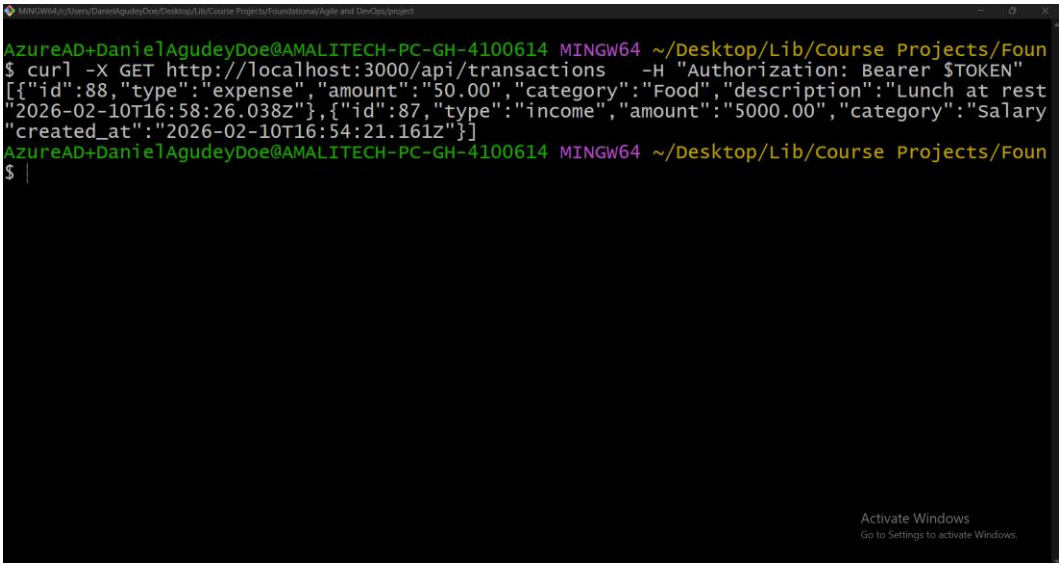

Figure 21: List transactions

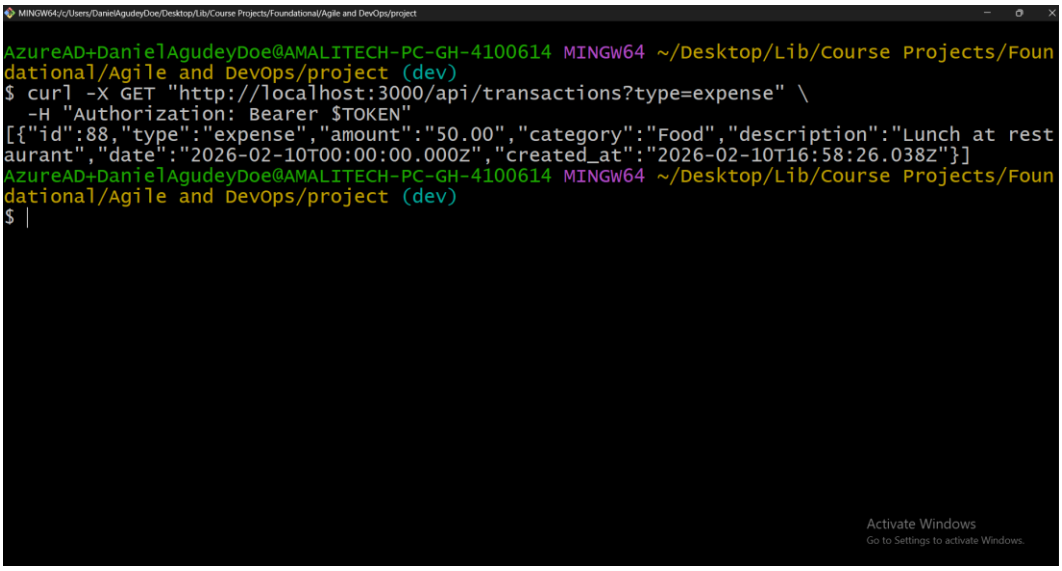

Figure 22: Filter by type

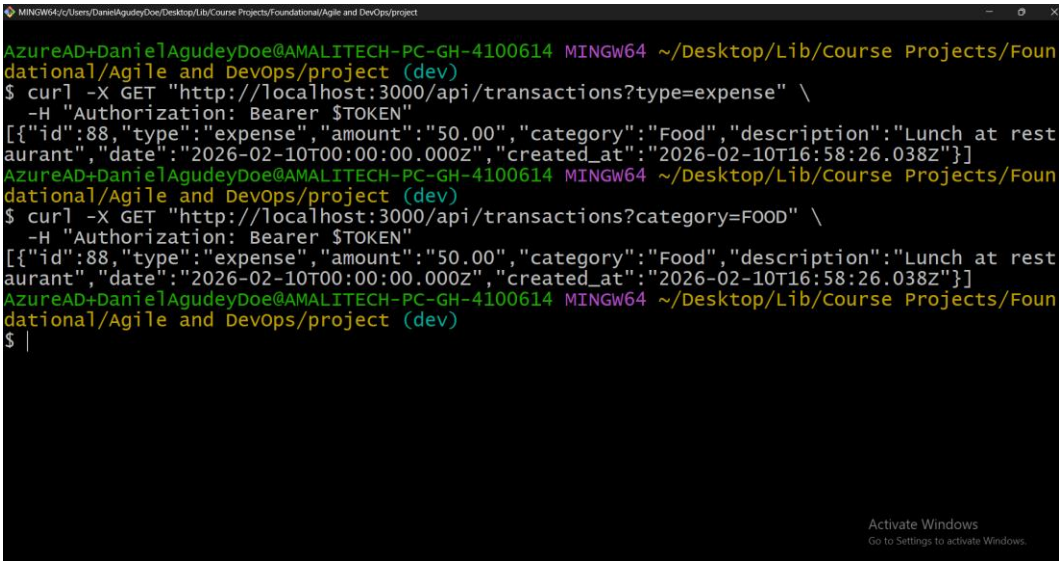

Figure 23: Filter by category

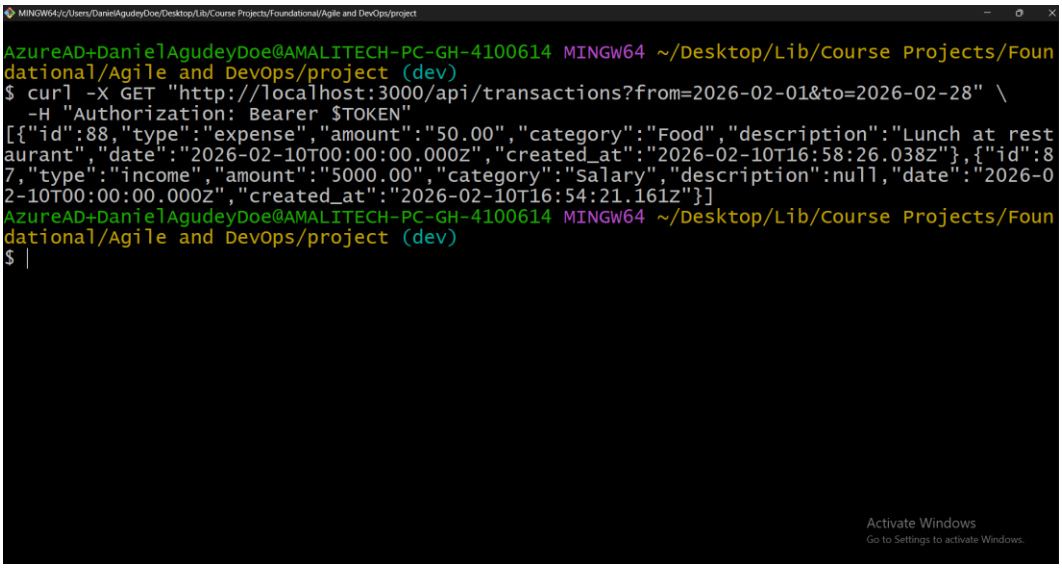

Figure 24: Filter by date range

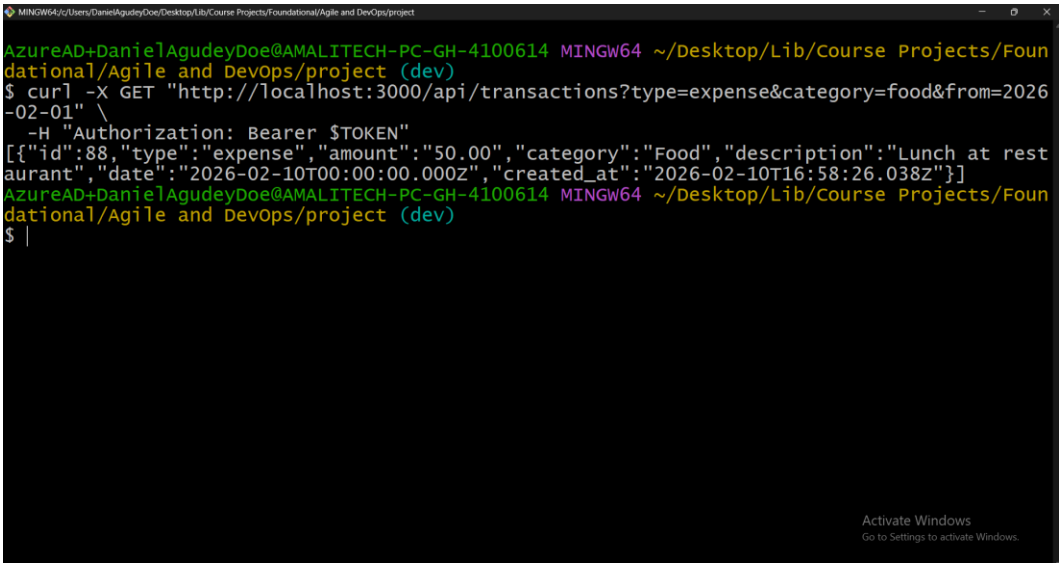

Figure 25: Combined filters

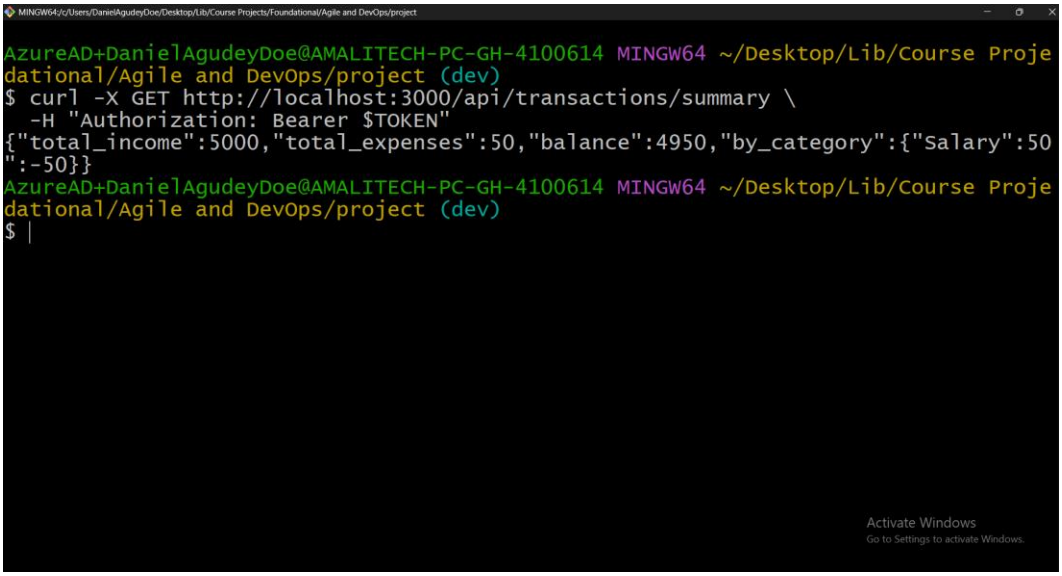

Figure 26: Summary endpoint

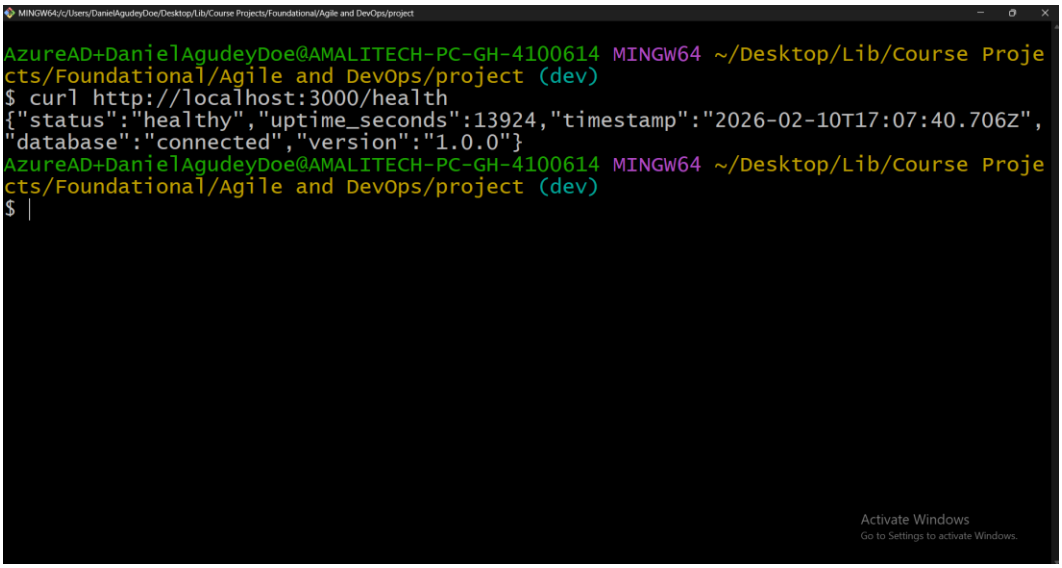

Figure 27: Health endpoint

Sprint 2 Review

Sprint 2 successfully delivered all 4 planned user stories (10 story points) with significant improvements: 90.63% code coverage (up from 64.25%), 45 tests passing (up from 27), full monitoring (health endpoint + logging), and all acceptance criteria tested.

Sprint 2 Burndown

Day	Points Completed	Cumulative
Day 1	5 (US-04, US-05 tests)	5
Day 2	5 (US-06, US-07 tests)	10
Day 3	Documentation & Review	10

Definition of Done Status

Criterion	Sprint 1	Sprint 2
Code follows style guidelines	Pass	Pass
Tests pass	100%	100%
Code merged to dev	Done	Done
CI pipeline passes	Pass	Pass
Documentation updated	Done	Done
Code coverage > 80%	64% (below)	90.6% (above)

Final Retrospective

Sprint-by-Sprint Comparison

Metric	Sprint 1	Sprint 2	Improvement
Story Points	9	10	+1
Tests	27	45	+18
Coverage	64.25%	90.63%	+26.38%
Pass Rate	100%	100%	Same
User Stories	3	4	+1

Lessons Learned

Technical Lessons:

8. Docker First Approach – Starting with Docker simplifies environment setup. CI/CD integrates naturally.
9. Test-Driven Development – Writing tests during implementation catches bugs early. 100% pass rate maintained.
10. CI/CD is Non-Negotiable – Automated testing catches issues immediately. Quality gates keep code clean.
11. Git Flow Discipline – Feature branches enable parallel work. Merge to dev keeps master clean.

Agile Lessons:

12. Sprint Planning Matters – Well-defined acceptance criteria = successful delivery.
13. Retrospectives Drive Improvement – Identified specific, actionable improvements and applied all 3.
14. Definition of Done Enforces Quality – Clear DoD prevented cutting corners.

What I Would Do Differently

15. Start CI/CD earlier – Set up pipeline skeleton in Sprint 0 before writing code.
16. More detailed acceptance criteria – Add edge cases and boundary conditions to ACs.
17. Separate test database – Use test containers for true isolation from the start.
18. Plan pagination earlier – Even if not required for MVP, saves refactoring later.

Skills Developed

Skill	Before	After
Node.js/Express	Basic	Proficient
PostgreSQL	Basic	Intermediate
Docker	Basic	Intermediate

Git/GitHub	Basic	Proficient
CI/CD (GitHub Actions)	None	Proficient
Jest Testing	Basic	Proficient
Agile Practices	Theoretical	Practical

CI/CD Pipeline Evidence

CI/CD Screenshots

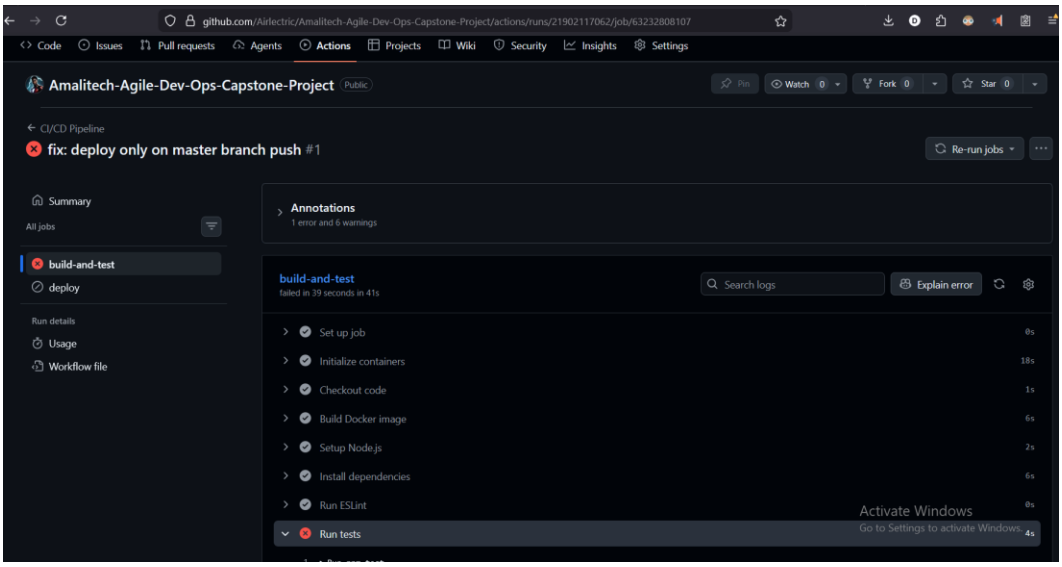

Figure 28: First CI/CD test on master failed

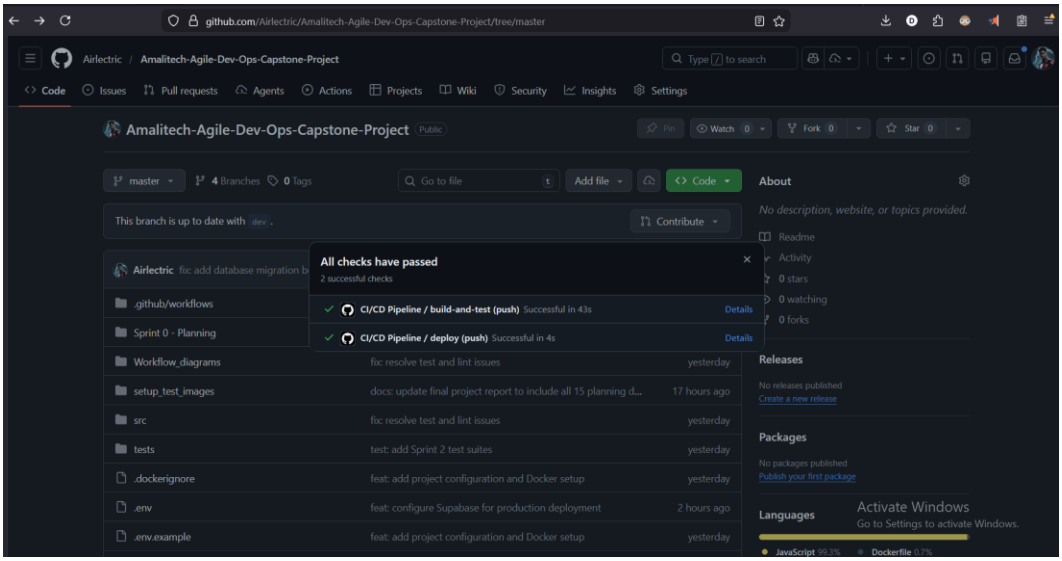

Figure 29: After fixing DB error – all CI/CD tests passed

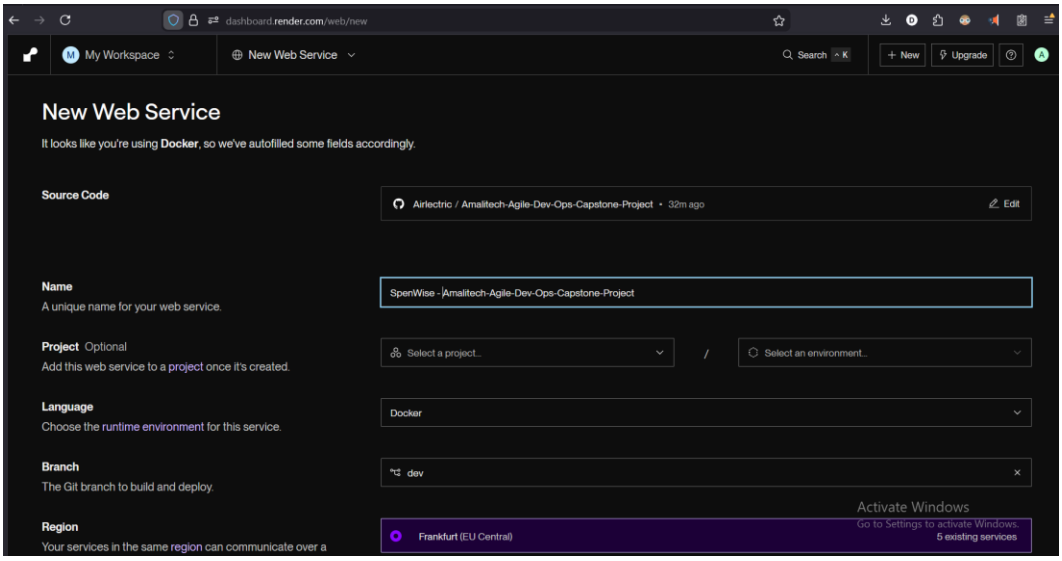

Figure 30: Deployment configuration in Docker

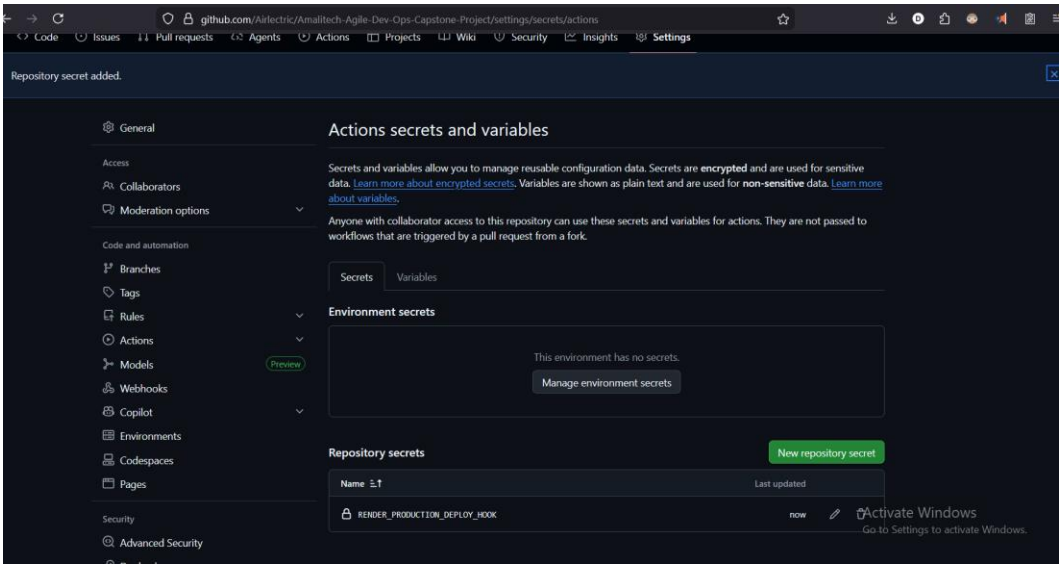

Figure 31: Render deploy hook added to GitHub Actions secrets

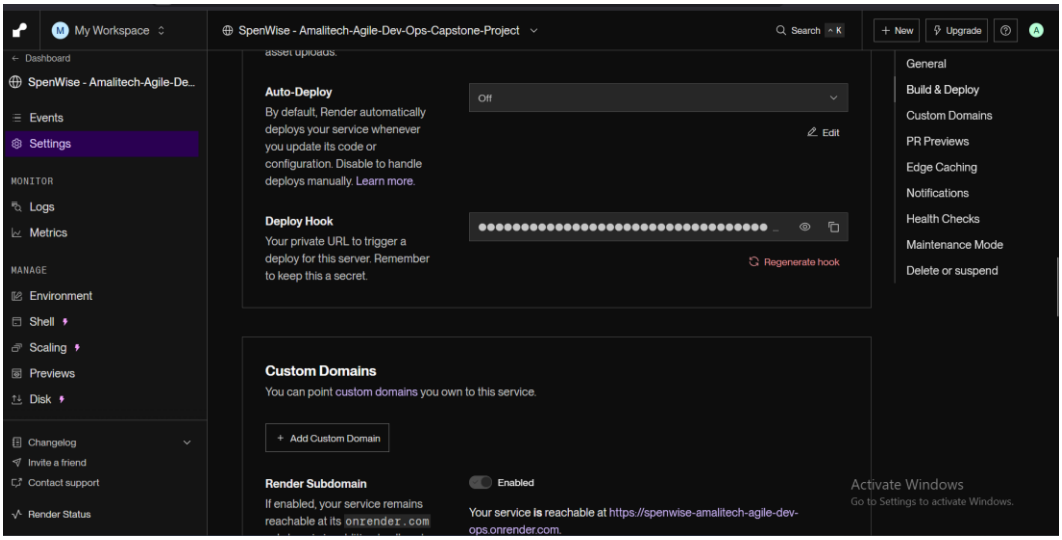

Figure 32: Retrieving Render deploy hooks

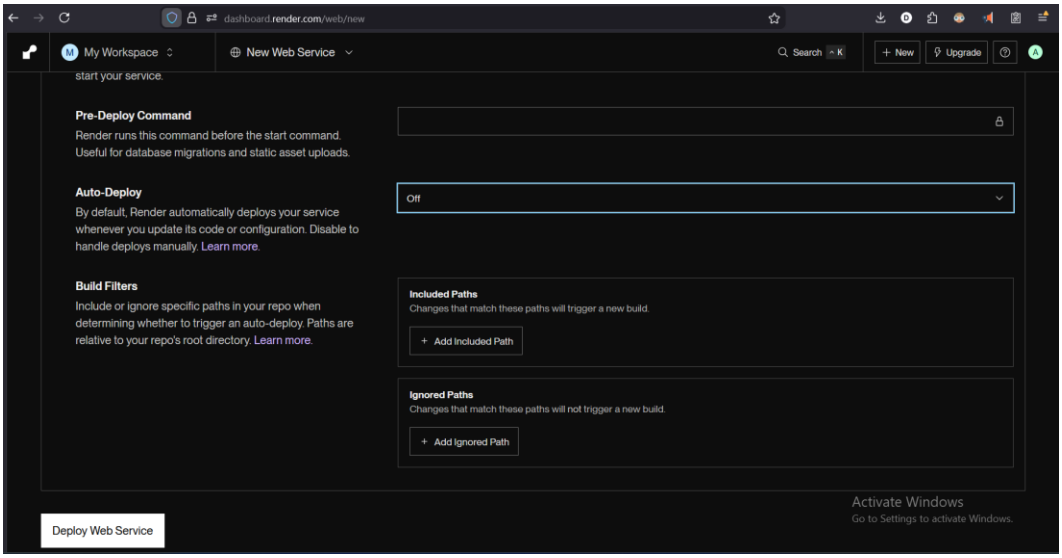

Figure 33: Disabled Render auto-deploy (CI/CD handles deployment)

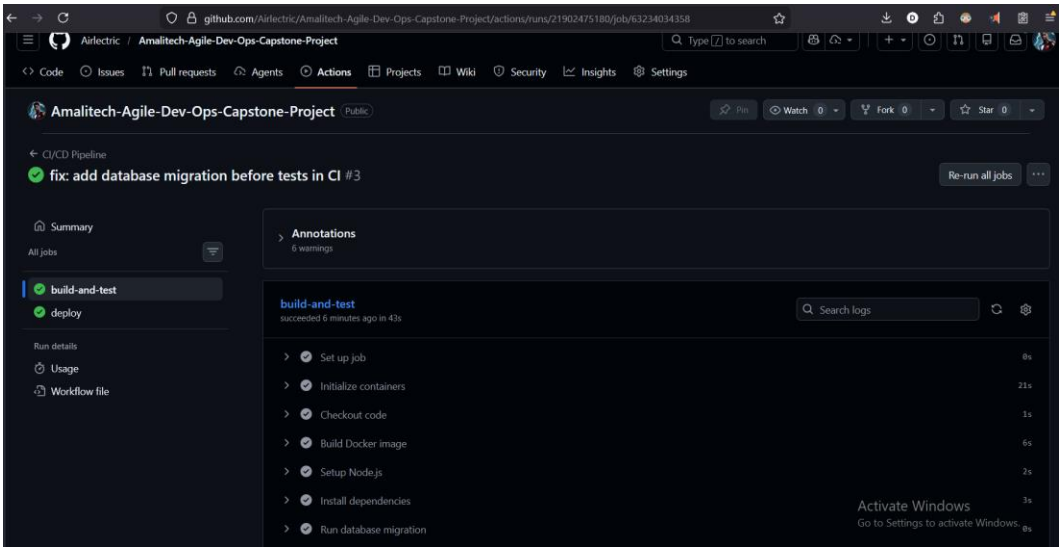

Figure 34: Successful deployment on GitHub Actions

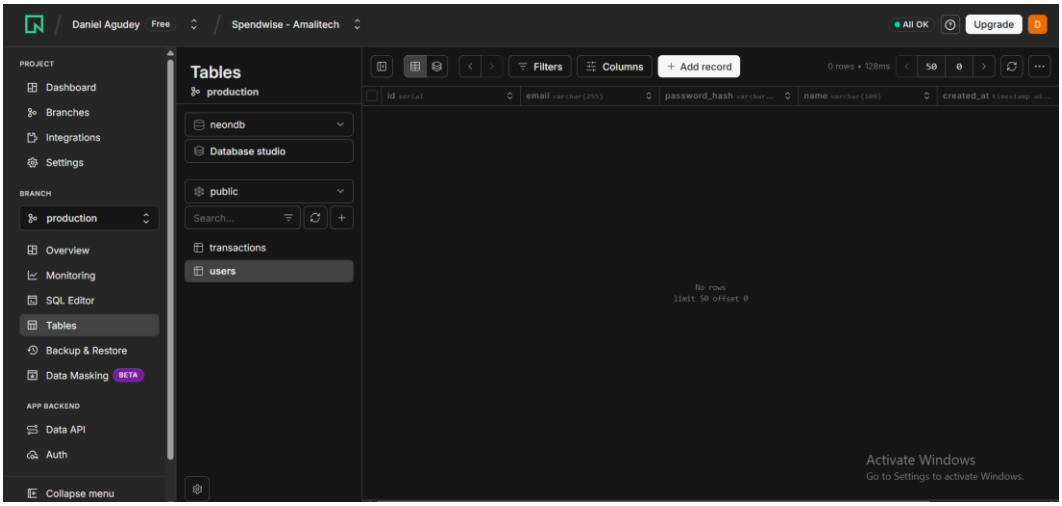

Figure 35: Tables migrated successfully to remote DB

Secrets Configuration

Secret	Purpose	Status
DATABASE_URL	Production database connection	Configured
JWT_SECRET	JWT signing key	Configured
RENDER_DEPLOY_HOOK_URL	Auto-deploy trigger	Configured

Final Project Completion Summary

Metric	Value
Total User Stories	7
Total Story Points	19
Completed Stories	7 (100%)
Completed Points	19 (100%)
Total Test Cases	45
Passing Tests	45 (100%)
Code Coverage	90.63%
CI/CD Pipeline	Configured & Passing
Docker Support	Configured

API Endpoints

Method	Endpoint	Description	Status
POST	/api/auth/register	Register new user	Complete
POST	/api/auth/login	Login, get JWT	Complete
POST	/api/transactions	Create transaction	Complete
GET	/api/transactions	List user's transactions	Complete
GET	/api/transactions?filters	Filter transactions	Complete
GET	/api/transactions/summary	Get spending summary	Complete
GET	/health	Health check	Complete

The SpendWise Personal Expense Tracker API has been successfully implemented following Agile and DevOps best practices. All 7 user stories completed (19 story points), 45 tests passing with 90.63% coverage, fully functional CI/CD pipeline, and comprehensive documentation provided.