Issue #10: Testing and coverage improvements

Labels: `testing`, `quality`, `enhancement`
Milestone: Feature Complete
Estimated Effort: 8-10 hours

Description

Expand test coverage from current ~40% to >80% by adding comprehensive unit tests, integration tests, and end-to-end tests. Implement code coverage reporting and enforce minimum coverage thresholds in CI/CD.

Current State

- ✅ Basic unit tests for `parseVote()` and `updateLeaderboard()`
- ✅ Jest test framework configured
- ✅ ~40% code coverage (utility functions only)
- ❌ No integration tests
- ❌ No end-to-end tests
- ❌ No database tests
- ❌ No Slack API mocking
- ❌ No coverage reporting in CI
- ❌ No coverage enforcement

Goals

- ✅ Achieve >80% code coverage
- ✅ Add integration tests for database operations
- ✅ Add E2E tests for Slack commands
- ✅ Mock external dependencies properly
- ✅ Set up coverage reporting (Codecov)
- ✅ Enforce coverage thresholds in CI
- ✅ Document testing strategy

Tasks

Phase 1: Test Infrastructure (2-3 hours)

- [ ] Configure Jest for integration tests
- [ ] Set up test database
- [ ] Create test helpers and fixtures
- [ ] Configure Slack API mocking
- [ ] Set up coverage reporting

Phase 2: Unit Tests (2-3 hours)

- [ ] Test environment validation
- [ ] Test logger functionality
- [ ] Test health check logic
- [ ] Test metrics collection
- [ ] Test storage functions

Phase 3: Integration Tests (2-3 hours)

- [ ] Test database operations end-to-end
- [ ] Test Slack bot message handlers
- [ ] Test slash command handlers
- [ ] Test error handling paths
- [ ] Test concurrent operations

Phase 4: E2E Tests (2-3 hours)

- [ ] Test complete vote flow
- [ ] Test leaderboard command flow
- [ ] Test score command flow
- [ ] Test self-vote prevention
- [ ] Test multiple votes in one message

Phase 5: Coverage & CI (1-2 hours)

- [ ] Configure Codecov integration
- [ ] Add coverage badges to README
- [ ] Set minimum coverage thresholds
- [ ] Update CI to fail on coverage drop
- [ ] Document testing guidelines

Implementation

Jest Configuration

```
// jest.config.js
module.exports = {
  // Use ts-jest if TypeScript
  preset: 'ts-jest',
  testEnvironment: 'node',

  // Test paths
  roots: ['<rootDir>/src', '<rootDir>/tests'],
  testMatch: [
    '**/__tests__/**/*.+(ts|tsx|js)',
    '**/?(*.)+(spec|test).+(ts|tsx|js)',
  ],

  // Transform files
  transform: {
    '^.+\\.(ts|tsx)$': 'ts-jest',
  },

  // Coverage configuration
  collectCoverageFrom: [
    'src/**/*.{js,ts}',
    '!src/**/*.test.{js,ts}',
    '!src/**/*.spec.{js,ts}',
    '!src/types.ts',
    '!src/**/*.d.ts',
  ],

  // Coverage thresholds
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
      lines: 80,
      statements: 80,
    },
  },

  // Setup files
  setupFilesAfterEnv: ['<rootDir>/tests/setup.js'],

  // Module paths
  moduleNameMapper: {
    '^@/(.*)$': '<rootDir>/src/$1',
  },

  // Test timeout
  testTimeout: 10000,

  // Reporters
  reporters: [
    'default',
    ['jest-junit', {
      outputDirectory: './coverage',
      outputName: 'junit.xml',
    }],
  ],
};
```

Test Setup

```javascript
// tests/setup.js
const { Pool } = require('pg');

// Test database configuration
const testDbConfig = {
  connectionString: process.env.TEST_DATABASE_URL || 'postgresql://localhost:5432/
ppbot_test',
  ssl: false,
};

let testPool;

// Setup before all tests
beforeAll(async () => {
  // Create test database pool
  testPool = new Pool(testDbConfig);

  // Create test schema
  await testPool.query(`
    DROP TABLE IF EXISTS leaderboard CASCADE;
    DROP TABLE IF EXISTS vote_history CASCADE;

    CREATE TABLE leaderboard (
      user_id VARCHAR(20) PRIMARY KEY,
      score INTEGER DEFAULT 0 NOT NULL,
      created_at TIMESTAMP DEFAULT NOW(),
      updated_at TIMESTAMP DEFAULT NOW()
    );

    CREATE INDEX idx_leaderboard_score ON leaderboard(score DESC);

    CREATE TABLE vote_history (
      id SERIAL PRIMARY KEY,
      voter_id VARCHAR(20) NOT NULL,
      voted_user_id VARCHAR(20) NOT NULL,
      vote_type VARCHAR(2) NOT NULL CHECK (vote_type IN ('++', '--')),
      channel_id VARCHAR(20),
      message_ts VARCHAR(20),
      created_at TIMESTAMP DEFAULT NOW()
    );

    CREATE INDEX idx_vote_history_user ON vote_history(voted_user_id);
    CREATE INDEX idx_vote_history_created ON vote_history(created_at DESC);
  `);
});

// Cleanup after all tests
afterAll(async () => {
  await testPool.end();
});

// Reset database before each test
beforeEach(async () => {
  await testPool.query('TRUNCATE leaderboard, vote_history RESTART IDENTITY CASCADE');
});

// Export test pool for use in tests
global.testPool = testPool;
```

Test Helpers

```
// tests/helpers.js

/**
 * Create a mock Slack message
 */
function createMockMessage(overrides = {}) {
  return {
    user: 'U12345',
    text: 'test message',
    channel: 'C12345',
    ts: '1234567890.123456',
    ...overrides,
  };
}

/**
 * Create a mock Slack command
 */
function createMockCommand(overrides = {}) {
  return {
    command: '/leaderboard',
    text: '',
    user_id: 'U12345',
    channel_id: 'C12345',
    trigger_id: 'trigger123',
    ...overrides,
  };
}

/**
 * Create mock Slack client
 */
function createMockSlackClient() {
  return {
    chat: {
      postMessage: jest.fn().mockResolvedValue({ ok: true, ts: '1234567890.123456' }),
      update: jest.fn().mockResolvedValue({ ok: true }),
      delete: jest.fn().mockResolvedValue({ ok: true }),
    },
    users: {
      info: jest.fn().mockResolvedValue({
        ok: true,
        user: {
          id: 'U12345',
          name: 'testuser',
          real_name: 'Test User',
        },
      }),
    },
  };
}

/**
 * Wait for a condition to be true
 */
async function waitFor(conditionFn, timeout = 5000) {
  const startTime = Date.now();
  while (Date.now() - startTime < timeout) {
    if (await conditionFn()) {
      return true;
    }
    await new Promise(resolve => setTimeout(resolve, 100));
  }
  throw new Error('Timeout waiting for condition');
}

module.exports = {
  createMockMessage,
  createMockCommand,
  createMockSlackClient,
  waitFor,
};
```

Unit Tests Examples

```
// src/vote.test.js
const { parseVote } = require('./vote');

describe('parseVote', () => {
  test('parses single upvote', () => {
    const text = '<@U12345> ++';
    const result = parseVote(text);
    expect(result).toEqual([{ userId: 'U12345', action: '++' }]);
  });

  test('parses single downvote', () => {
    const text = '<@U12345> --';
    const result = parseVote(text);
    expect(result).toEqual([{ userId: 'U12345', action: '--' }]);
  });

  test('parses multiple votes', () => {
    const text = '<@U12345> ++ <@U67890> --';
    const result = parseVote(text);
    expect(result).toEqual([
      { userId: 'U12345', action: '++' },
      { userId: 'U67890', action: '--' },
    ]);
  });

  test('parses vote with emojis and text after', () => {
    const text = '<@U12345> ++ 🎉 great job!';
    const result = parseVote(text);
    expect(result).toEqual([{ userId: 'U12345', action: '++' }]);
  });

  test('returns empty array for no votes', () => {
    const text = 'just a regular message';
    const result = parseVote(text);
    expect(result).toEqual([]);
  });

  test('ignores invalid user IDs', () => {
    const text = '<@invalid> ++';
    const result = parseVote(text);
    expect(result).toEqual([]);
  });
});
```

Integration Tests Examples

```javascript
// tests/integration/database.test.js
const { getUserScore, updateUserScore, getTopUsers } = require('../../src/storage');

describe('Database Integration', () => {
  test('getUserScore returns 0 for new user', async () => {
    const score = await getUserScore('U12345');
    expect(score).toBe(0);
  });

  test('updateUserScore creates new user with score', async () => {
    const newScore = await updateUserScore('U12345', 1);
    expect(newScore).toBe(1);

    const score = await getUserScore('U12345');
    expect(score).toBe(1);
  });

  test('updateUserScore increments existing user score', async () => {
    await updateUserScore('U12345', 5);
    const newScore = await updateUserScore('U12345', 3);
    expect(newScore).toBe(8);
  });

  test('updateUserScore decrements score', async () => {
    await updateUserScore('U12345', 5);
    const newScore = await updateUserScore('U12345', -2);
    expect(newScore).toBe(3);
  });

  test('updateUserScore can go negative', async () => {
    await updateUserScore('U12345', -5);
    const score = await getUserScore('U12345');
    expect(score).toBe(-5);
  });

  test('getTopUsers returns sorted list', async () => {
    await updateUserScore('U11111', 10);
    await updateUserScore('U22222', 5);
    await updateUserScore('U33333', 15);
    await updateUserScore('U44444', -5);

    const topUsers = await getTopUsers(3);
    expect(topUsers).toHaveLength(3);
    expect(topUsers[0].user_id).toBe('U33333');
    expect(topUsers[0].score).toBe(15);
    expect(topUsers[1].user_id).toBe('U11111');
    expect(topUsers[2].user_id).toBe('U22222');
  });

  test('getTopUsers excludes users with zero score', async () => {
    await updateUserScore('U11111', 10);
    await updateUserScore('U22222', 0);

    const topUsers = await getTopUsers(10);
    expect(topUsers).toHaveLength(1);
    expect(topUsers[0].user_id).toBe('U11111');
  });

  test('concurrent updates maintain consistency', async () => {
    const userId = 'U12345';

    // Simulate concurrent vote processing
    await Promise.all([
      updateUserScore(userId, 1),
      updateUserScore(userId, 1),
      updateUserScore(userId, 1),
      updateUserScore(userId, 1),
      updateUserScore(userId, 1),
    ]);

    const finalScore = await getUserScore(userId);
    expect(finalScore).toBe(5);
  });
});
```

E2E Tests Examples

```javascript
// tests/e2e/vote-flow.test.js
const { App } = require('@slack/bolt');
const { createMockMessage, createMockSlackClient } = require('../helpers');

describe('Vote Flow E2E', () => {
  let app, mockClient;

  beforeEach(() => {
    mockClient = createMockSlackClient();
    app = new App({
      token: 'xoxb-test-token',
      signingSecret: 'test-secret',
      receiver: { app: jest.fn() },
    });
  });

  test('processes upvote and updates leaderboard', async () => {
    const message = createMockMessage({
      user: 'U11111',
      text: '<@U22222> ++ great work!',
    });

    // Simulate message handler
    await app.processEvent({
      event: { type: 'message', ...message },
      say: jest.fn(),
    });

    // Verify score was updated
    const score = await getUserScore('U22222');
    expect(score).toBe(1);
  });

  test('blocks self-votes', async () => {
    const sayMock = jest.fn();
    const message = createMockMessage({
      user: 'U11111',
      text: '<@U11111> ++ vote for myself',
    });

    await app.processEvent({
      event: { type: 'message', ...message },
      say: sayMock,
    });

    // Verify score was not updated
    const score = await getUserScore('U11111');
    expect(score).toBe(0);

    // Verify error message was sent
    expect(sayMock).toHaveBeenCalledWith(
      expect.stringContaining('cannot vote for themselves')
    );
  });

  test('processes multiple votes in one message', async () => {
    const message = createMockMessage({
      user: 'U11111',
      text: '<@U22222> ++ <@U33333> ++ <@U44444> --',
    });

    await app.processEvent({
      event: { type: 'message', ...message },
      say: jest.fn(),
    });

    // Verify all votes were processed
    expect(await getUserScore('U22222')).toBe(1);
    expect(await getUserScore('U33333')).toBe(1);
    expect(await getUserScore('U44444')).toBe(-1);
  });

  test('leaderboard command returns top users', async () => {
    // Set up some scores
    await updateUserScore('U11111', 10);
    await updateUserScore('U22222', 5);
    await updateUserScore('U33333', 15);

    const ackMock = jest.fn();
    const sayMock = jest.fn();

    await app.processEvent({
      event: {
        type: 'slash_command',
        command: '/leaderboard',
        user_id: 'U11111',
      },
      ack: ackMock,
      say: sayMock,
    });

    expect(ackMock).toHaveBeenCalled();
    expect(sayMock).toHaveBeenCalledWith(
      expect.objectContaining({
        text: expect.stringContaining('Leaderboard'),
      })
    );
  });
});
```

Coverage Configuration

```
// package.json
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage",
    "test:ci": "jest --ci --coverage --maxWorkers=2"
  },
  "jest": {
    "coverageReporters": ["text", "lcov", "html", "json-summary"]
  }
}
```

Codecov Integration

```yaml
# .github/workflows/ci.yml (add to CI workflow)
- name: Upload coverage to Codecov
  uses: codecov/codecov-action@v4
  with:
    token: ${{ secrets.CODECOV_TOKEN }}
    files: ./coverage/lcov.info
    flags: unittests
    name: codecov-umbrella
    fail_ci_if_error: true
```

Coverage Badges

Add to README.md:

```markdown
[![codecov](https://files.readme.io/d19875a-Screenshot_2024-08-06_at_11.20.59.png)
```

Testing Strategy Documentation

Create `TESTING.md`:

```markdown
# Testing Strategy

## Overview
pp-bot uses a comprehensive testing strategy covering unit tests, integration tests,
and end-to-end tests.

## Test Structure
```

tests/
├── unit/ # Unit tests for individual functions
├── integration/ # Integration tests for database, APIs
├── e2e/ # End-to-end tests for complete flows
├── helpers.js # Test helper functions
└── setup.js # Test environment setup

~~~
## Running Tests

```bash
# Run all tests
npm test

# Run specific test file
npm test -- vote.test.js

# Run with coverage
npm run test:coverage

# Run in watch mode
npm run test:watch
~~~

Writing Tests

Unit Tests

Test individual functions in isolation:

```
describe('parseVote', () => {
  test('parses single upvote', () => {
    const result = parseVote('<@U12345> ++');
    expect(result).toEqual([{ userId: 'U12345', action: '++' }]);
  });
});
```

Integration Tests

Test interactions between components:

```
describe('Database Integration', () => {
  test('updateUserScore persists to database', async () => {
    await updateUserScore('U12345', 1);
    const score = await getUserScore('U12345');
    expect(score).toBe(1);
  });
});
```

E2E Tests

Test complete user flows:

```
describe('Vote Flow E2E', () => {
  test('processes vote and updates leaderboard', async () => {
    // Simulate user action
    // Verify end result
  });
});
```

Coverage Requirements

- Minimum 80% coverage for all metrics
- Critical paths must have 100% coverage
- New features must include tests

Continuous Integration

- Tests run on every PR
- Coverage reports uploaded to Codecov
- CI fails if coverage drops below threshold
  ```

Acceptance Criteria

- [ ] Code coverage >80% (branches, functions, lines, statements)
  ```

- [ ] All critical paths have tests
- [ ] Unit tests for all utility functions
- [ ] Integration tests for database operations
- [ ] Integration tests for Slack handlers
- [ ] E2E tests for complete flows
- [ ] Test database setup works correctly
- [ ] Slack API mocking works
- [ ] Codecov integration active
- [ ] Coverage badge in README
- [ ] CI enforces coverage thresholds
- [ ] Testing strategy documented
- [ ] All tests pass consistently

## Performance Considerations

- Tests should complete in <30 seconds
- Use parallel test execution
- Mock external APIs (Slack, Sentry)
- Use test database (not production)
- Clean up test data after each test

## Resources

- Jest Documentation (https://jestjs.io/docs/getting-started)
- Testing Best Practices (https://testingjavascript.com/)
- Codecov Documentation (https://docs.codecov.com/)
- @slack/bolt Testing Guide (https://slack.dev/bolt-js/tutorial/testing)

## Dependencies

**Recommended after:**
- Issue #2: PostgreSQL integration (to test database operations)
- Issue #3: TypeScript migration (to test with types)