

# Mingus Pytest Implementation: Step-by-Step Guide + Cursor Prompts

## Part 1: Manual Step-by-Step Implementation

### Step 1: Install Testing Dependencies

```
bash
```

```
# In your project directory, create/update requirements.txt
```

```
pip install pytest pytest-flask requests selenium webdriver-manager
```

```
pip install pytest-cov pytest-html # For coverage and HTML reports
```

```
pip freeze > requirements-test.txt
```

### Step 2: Create Testing Directory Structure

```
mingus/
```

```
├── tests/
```

```
│   ├── __init__.py
```

```
│   ├── conftest.py          # Test configuration
```

```
│   ├── test_api_endpoints.py # API testing
```

```
│   ├── test_user_registration.py # Registration flow
```

```
│   ├── test_profile_completion.py # Profile system
```

```
│   ├── test_subscription_management.py # Billing
```

```
│   └── fixtures/
```

```
│       ├── test_data.py      # Test user data
```

```
│       └── mock_services.py  # Mock external services
```

```
│   └── selenium_tests/
```

```
│       ├── test_user_journey.py # UI testing
```

```
│       └── page_objects.py     # Page object pattern
```

```
├── pytest.ini              # Pytest configuration
```

```
└── run_tests.py            # Test runner script
```

### Step 3: Create pytest.ini Configuration

```
ini
```

```
# pytest.ini
[tool:pytest]
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*
addopts =
    -v
    --tb=short
    --strict-markers
    --strict-config
    --cov=backend
    --cov-report=html
    --cov-report=term-missing
    --html=reports/report.html
    --self-contained-html
markers =
    slow: marks tests as slow
    fast: marks tests as fast
    api: marks tests as API tests
    ui: marks tests as UI tests
    integration: marks tests as integration tests
```

## Step 4: Create Test Configuration (conftest.py)

```
python
```

```
# tests/conftest.py
import pytest
import os
import tempfile
from backend import create_app
from backend.database import db, init_db
from backend.models import User, UserProfile

@pytest.fixture(scope='session')
def app():
    """Create and configure a new app instance for each test session."""
    # Create a temporary file to serve as the database
    db_fd, db_path = tempfile.mkstemp()

    # Create the app with test config
    app = create_app({
        'TESTING': True,
        'SQLALCHEMY_DATABASE_URI': f'sqlite:/// {db_path}',
        'SECRET_KEY': 'test-secret-key',
        'WTF_CSRF_ENABLED': False
    })

    # Create the database and the database table
    with app.app_context():
        init_db()
        yield app

    # Clean up
    os.close(db_fd)
    os.unlink(db_path)

@pytest.fixture
def client(app):
    """A test client for the app."""
    return app.test_client()

@pytest.fixture
def runner(app):
    """A test runner for the app's Click commands."""
    return app.test_cli_runner()

@pytest.fixture
def sample_user_data():
```

```

"""Sample user data for testing."""
return {
    'email': 'test@mingus.com',
    'password': 'TestPassword123!',
    'first_name': 'Marcus',
    'last_name': 'Johnson',
    'zip_code': '30309',
    'monthly_income': 65000,
    'industry': 'Technology',
    'job_title': 'Software Developer'
}

@pytest.fixture
def authenticated_user(client, sample_user_data):
    """Create and authenticate a test user."""
    # Register user
    response = client.post('/api/user-profile/register', json=sample_user_data)
    assert response.status_code == 201

    # Login user
    login_response = client.post('/api/auth/login', json={
        'email': sample_user_data['email'],
        'password': sample_user_data['password']
    })
    assert login_response.status_code == 200

    return login_response.get_json()

```

## Step 5: Create Basic API Tests

python

```
# tests/test_api_endpoints.py
import pytest
import json

class TestAPIEndpoints:
    """Test all API endpoints for basic functionality."""

    def test_health_endpoint(self, client):
        """Test application health check."""
        response = client.get('/api/health')
        assert response.status_code == 200
        data = response.get_json()
        assert data['status'] == 'healthy'

    def test_get_user_profile_unauthorized(self, client):
        """Test getting user profile without authentication."""
        response = client.get('/api/user-profile/get')
        assert response.status_code == 401

    def test_get_user_profile_authorized(self, self, client, authenticated_user):
        """Test getting user profile with authentication."""
        headers = {'Authorization': f"Bearer {authenticated_user['token']}"}
        response = client.get('/api/user-profile/get', headers=headers)
        assert response.status_code == 200
        data = response.get_json()
        assert 'user_profile' in data

    def test_update_user_profile(self, self, client, authenticated_user):
        """Test updating user profile."""
        headers = {'Authorization': f"Bearer {authenticated_user['token']}"}
        update_data = {
            'first_name': 'Updated',
            'monthly_income': 75000,
            'dependents': 1
        }
        response = client.post('/api/user-profile/update',
                               json=update_data, headers=headers)
        assert response.status_code == 200
        data = response.get_json()
        assert data['success'] == True

    def test_onboarding_progress(self, self, client, authenticated_user):
        """Test onboarding progress tracking."""
```

```
headers = {'Authorization': f"Bearer {authenticated_user['token']}"}
response = client.get('/api/user-profile/onboarding-progress',
                      headers=headers)
assert response.status_code == 200
data = response.get_json()
assert 'progress_percentage' in data
assert 0 <= data['progress_percentage'] <= 100
```

## Step 6: Create User Registration Tests

python

```
# tests/test_user_registration.py
```

```
import pytest
```

```
class TestUserRegistration:
```

```
    """Test complete user registration flow."""
```

```
    def test_valid_registration(self, client, sample_user_data):
```

```
        """Test successful user registration."""
```

```
        response = client.post('/api/user-profile/register', json=sample_user_data)
```

```
        assert response.status_code == 201
```

```
        data = response.get_json()
```

```
        assert data['success'] == True
```

```
        assert 'user_id' in data
```

```
    def test_duplicate_email_registration(self, client, sample_user_data):
```

```
        """Test registration with duplicate email."""
```

```
        # First registration
```

```
        client.post('/api/user-profile/register', json=sample_user_data)
```

```
        # Attempt duplicate registration
```

```
        response = client.post('/api/user-profile/register', json=sample_user_data)
```

```
        assert response.status_code == 400
```

```
        data = response.get_json()
```

```
        assert 'error' in data
```

```
        assert 'email' in data['error'].lower()
```

```
    def test_invalid_email_format(self, client, sample_user_data):
```

```
        """Test registration with invalid email format."""
```

```
        sample_user_data['email'] = 'invalid-email-format'
```

```
        response = client.post('/api/user-profile/register', json=sample_user_data)
```

```
        assert response.status_code == 400
```

```
        data = response.get_json()
```

```
        assert 'email' in data['error'].lower()
```

```
    def test_weak_password(self, client, sample_user_data):
```

```
        """Test registration with weak password."""
```

```
        sample_user_data['password'] = '123'
```

```
        response = client.post('/api/user-profile/register', json=sample_user_data)
```

```
        assert response.status_code == 400
```

```
        data = response.get_json()
```

```
        assert 'password' in data['error'].lower()
```

```
@pytest.mark.parametrize("missing_field", ['email', 'password', 'first_name'])
```

```
def test_missing_required_fields(self, client, sample_user_data, missing_field):  
    """Test registration with missing required fields."""  
    del sample_user_data[missing_field]  
    response = client.post('/api/user-profile/register', json=sample_user_data)  
    assert response.status_code == 400  
    data = response.get_json()  
    assert missing_field in data['error'].lower()
```

## Step 7: Create Profile Completion Tests

python



```
# tests/test_profile_completion.py
```

```
import pytest
```

```
class TestProfileCompletion:
```

```
    """Test user profile completion system."""
```

```
def test_initial_profile_completion(self, client, authenticated_user):
```

```
    """Test initial profile completion percentage."""
```

```
    headers = {'Authorization': f"Bearer {authenticated_user['token']}"}
```

```
    response = client.get('/api/user-profile/onboarding-progress', headers=headers)
```

```
    assert response.status_code == 200
```

```
    data = response.get_json()
```

```
    # Should have some completion from registration
```

```
    assert data['progress_percentage'] > 0
```

```
    assert data['progress_percentage'] < 100
```

```
def test_profile_field_updates(self, client, authenticated_user):
```

```
    """Test profile completion increases with field updates."""
```

```
    headers = {'Authorization': f"Bearer {authenticated_user['token']}"}  
  
    # Get initial completion
```

```
    initial_response = client.get('/api/user-profile/onboarding-progress', headers=headers)
```

```
    initial_completion = initial_response.get_json()['progress_percentage']  
  
    # Update additional fields
```

```
    update_data = {
```

```
        'dependents': 2,
```

```
        'marital_status': 'married',
```

```
        'education_level': 'bachelors',
```

```
        'current_savings_balance': 15000,
```

```
        'total_debt_amount': 25000
```

```
    }
```

```
    client.post('/api/user-profile/update', json=update_data, headers=headers)  
  
    # Check completion increased
```

```
    updated_response = client.get('/api/user-profile/onboarding-progress', headers=headers)
```

```
    updated_completion = updated_response.get_json()['progress_percentage']
```

```
    assert updated_completion > initial_completion
```

```
def test_complete_profile_flow(self, client, authenticated_user):
```

```
    """Test completing all profile fields."""
```

```
    headers = {'Authorization': f"Bearer {authenticated_user['token']}"}  
  
    # Get initial completion
```

```

complete_profile_data = {
    'last_name': 'TestUser',
    'zip_code': '30309',
    'dependents': 1,
    'marital_status': 'single',
    'industry': 'Technology',
    'job_title': 'Software Developer',
    'naics_code': '541511',
    'monthly_income': 65000,
    'employment_status': 'full_time',
    'company_size': 'medium',
    'years_experience': 5,
    'education_level': 'bachelors',
    'current_savings_balance': 10000,
    'total_debt_amount': 30000,
    'credit_score_range': 'good',
    'primary_financial_goal': 'emergency_fund',
    'risk_tolerance_level': 'moderate',
    'health_checkin_frequency': 'weekly',
    'notification_preferences': 'email_sms'
}

response = client.post('/api/user-profile/update',
                      json=complete_profile_data, headers=headers)
assert response.status_code == 200

# Check completion percentage
progress_response = client.get('/api/user-profile/onboarding-progress', headers=headers)
progress = progress_response.get_json()['progress_percentage']
assert progress >= 95 # Should be nearly complete

```

## Step 8: Create Test Runner Script

```
python
```

```
# run_tests.py
#!/usr/bin/env python3
"""
Mingus Application Test Runner
Runs comprehensive test suite and generates reports
"""

import os
import sys
import subprocess
from datetime import datetime

def run_tests():
    """Run the complete test suite."""
    print("🚀 Starting Mingus Application Test Suite")
    print("=" * 50)

    # Create reports directory
    os.makedirs('reports', exist_ok=True)

    # Run different test categories
    test_commands = [
        {
            'name': 'API Endpoints',
            'command': ['pytest', 'tests/test_api_endpoints.py', '-v'],
            'marker': 'api'
        },
        {
            'name': 'User Registration',
            'command': ['pytest', 'tests/test_user_registration.py', '-v'],
            'marker': 'fast'
        },
        {
            'name': 'Profile Completion',
            'command': ['pytest', 'tests/test_profile_completion.py', '-v'],
            'marker': 'fast'
        },
        {
            'name': 'All Tests with Coverage',
            'command': ['pytest', '--cov=backend', '--cov-report=html',
                        '--html=reports/test_report.html'],
            'marker': 'all'
        }
    ]
```

```
]
```

```
results = []
```

```
for test_group in test_commands:
```

```
    print(f"\n🕒 Running {test_group['name']} Tests...")
```

```
    try:
```

```
        result = subprocess.run(test_group['command'],
                                capture_output=True, text=True)
```

```
        results.append({
```

```
            'name': test_group['name'],
```

```
            'success': result.returncode == 0,
```

```
            'output': result.stdout,
```

```
            'errors': result.stderr
```

```
        })
```

```
    if result.returncode == 0:
```

```
        print(f"✅ {test_group['name']} - PASSED")
```

```
    else:
```

```
        print(f"❌ {test_group['name']} - FAILED")
```

```
        print(f"Error: {result.stderr[:200]}...")
```

```
except Exception as e:
```

```
    print(f"❌ Error running {test_group['name']}: {e}")
```

```
    results.append({
```

```
        'name': test_group['name'],
```

```
        'success': False,
```

```
        'output': '',
```

```
        'errors': str(e)
```

```
    })
```

```
# Generate summary
```

```
print("\n" + "=" * 50)
```

```
print("📊 TEST SUMMARY")
```

```
print("=" * 50)
```

```
passed = sum(1 for r in results if r['success'])
```

```
total = len(results)
```

```
print(f"Tests Passed: {passed}/{total}")
```

```
print(f"Success Rate: {(passed/total)*100:.1f}%")
```

```
if passed == total:
```

```
    print("🎉 ALL TESTS PASSED! Ready for external developer.")
```

```
else:
```

```
    print("⚠️ Some tests failed. Review and fix before external developer.")
```

```
print(f"\nDetailed reports generated in 'reports/' directory")
```

```
print(f"Timestamp: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
```

```
if __name__ == '__main__':
```

```
    run_tests()
```

---

## Part 2: Cursor Prompts for Automated Generation

### Cursor Prompt 1: Complete Testing Setup

I'm building a Flask financial wellness application called Mingus targeting African American professionals (age 25-35, income \$40K-\$100K).

Current application status:

- Flask backend with PostgreSQL database
- 6 API endpoints for user management
- User profile system with 25+ fields
- 3 subscription tiers (\$10, \$20, \$50)
- Authentication system
- User registration and onboarding flow

Create a complete pytest testing suite with:

1. conftest.py with proper fixtures for:

- Flask app with test configuration
- Test database setup/teardown
- Sample user data for target demographic
- Authentication helpers

2. Test files for:

- API endpoint testing (all 6 endpoints)
- User registration flow
- Profile completion system
- Subscription management
- Data validation
- Error handling

3. pytest.ini configuration with:

- Coverage reporting
- HTML report generation
- Test markers for different categories
- Proper test discovery

4. Test runner script that:

- Runs tests in logical order
- Generates comprehensive reports
- Provides clear pass/fail summary

5. Sample test data that's culturally appropriate for African American professionals including:

- Realistic names and locations
- Relevant industries and job titles
- Appropriate income ranges
- Metro areas (Atlanta, Houston, DC, Dallas, etc.)

Make tests comprehensive but maintainable. Include edge cases and error scenarios.

## **Cursor Prompt 2: Selenium UI Testing**

Create a comprehensive Selenium testing suite for complete user interface testing of my Flask financial wellness application.

#### Application Details:

- Financial planning app for African American professionals
- User registration, profile completion, dashboard
- Mobile-responsive design (critical for target demographic)
- 3 subscription tiers with different feature access
- Health and financial data integration

Create Selenium tests for:

#### 1. Complete user journey testing:

- Landing page load and responsiveness
- Registration form (all validation scenarios)
- Login process and session management
- Profile completion flow (25+ fields)
- Dashboard functionality and data display
- Feature access based on subscription tier
- Mobile responsiveness testing

#### 2. Cross-browser testing setup:

- Chrome and Firefox drivers
- Mobile viewport simulation
- Responsive design validation

#### 3. Page Object Pattern implementation:

- Separate page objects for each major page
- Reusable element locators
- Action methods for user interactions

#### 4. Visual and functional testing:

- Screenshot capture for verification
- Form validation testing
- Error message display validation
- Loading time measurement
- Cultural appropriateness review

#### 5. Test data management:

- Multiple user personas for African American professionals
- Various demographic scenarios
- Different subscription levels
- Edge cases and error conditions



Include proper setup for WebDriver management, comprehensive reporting, and parallel test execution capabilities.

### **Cursor Prompt 3: Performance and Load Testing**

Create a performance testing suite for my Flask financial wellness application to ensure it can handle my target of 1,000 users in year one.

#### Application Architecture:

- Flask backend with PostgreSQL
- Redis caching for performance
- User profile system with complex calculations
- Real-time financial forecasting
- Mobile-first design for target demographic

Create performance tests for:

#### 1. Load Testing:

- Simulate 100-500 concurrent users
- Test critical user paths (registration, login, dashboard)
- Database performance under load
- API response times under stress

#### 2. Stress Testing:

- Find breaking points for user capacity
- Memory and CPU usage monitoring
- Database connection pool testing
- Redis cache performance

#### 3. Endurance Testing:

- Long-running sessions (24-48 hours)
- Memory leak detection
- Performance degradation over time
- Session management stability

#### 4. Specific Performance Scenarios:

- New user registration during peak times
- Complex profile calculations with many users
- Dashboard loading with large datasets
- Mobile device performance simulation

#### 5. Performance Metrics Collection:

- Response time measurements
- Throughput analysis
- Resource utilization tracking
- Error rate monitoring

#### 6. Realistic Test Data:

- 1000+ synthetic user profiles
- Realistic usage patterns for target demographic
- Geographic distribution across major metro areas
- Various subscription tiers and usage levels

Include automated performance regression testing and detailed reporting with recommendations for optimization.

## **Cursor Prompt 4: Security and Vulnerability Testing**

Create a comprehensive security testing suite for my Flask financial wellness application handling sensitive financial and personal data.

Application Security Context:

- Financial data for African American professionals
- Personal health and relationship information
- Income, debt, and savings data
- Payment processing for subscriptions
- User authentication and session management

Create security tests for:

1. Authentication and Authorization:

- Password strength validation
- Session management security
- Token-based authentication testing
- Role-based access control
- Multi-factor authentication (if implemented)

2. Input Validation and Injection Prevention:

- SQL injection testing across all endpoints
- XSS (Cross-Site Scripting) prevention
- CSRF (Cross-Site Request Forgery) protection
- Input sanitization for all form fields
- File upload security (if applicable)

3. Data Protection:

- Sensitive data encryption verification
- Personal information masking
- Database security configuration
- API endpoint security
- Data transmission security (HTTPS)

4. Financial Data Security:

- Payment information protection
- Financial calculation data security
- Subscription management security
- Audit trail verification

5. Privacy and Compliance Testing:

- User data privacy validation
- Data retention policy compliance
- User consent management

- Data deletion/anonymization testing

#### 6. Application-Specific Security:

- Profile completion security
- Health data protection
- Cultural data sensitivity
- Geographic data protection

Include automated vulnerability scanning, security best practices validation, and detailed security assessment reporting with prioritized recommendations.

---

## Part 3: Implementation Strategy

### Week 1: Manual Implementation (Recommended Start)

#### Days 1-2: Basic Setup

1. Install dependencies
2. Create directory structure
3. Set up conftest.py manually
4. Create basic API tests

#### Days 3-4: Core Testing

1. User registration tests
2. Profile completion tests
3. Basic UI testing with Selenium

#### Days 5-7: Integration

1. Test runner script
2. Generate first test reports
3. Fix obvious issues found

### Week 2: Cursor Enhancement

#### Days 1-3: Use Cursor Prompts





1. Run Cursor prompts to generate additional tests
2. Compare with manual implementation
3. Merge best features from both approaches

## **Days 4-7: Optimization**





1. Add performance testing
2. Security testing implementation
3. Comprehensive reporting setup

## **Expected Outcomes:**

### **After Manual Implementation:**

-  Basic functionality verified
-  Major bugs identified and fixed
-  60-70% test coverage
-  Clear understanding of what works

### **After Cursor Enhancement:**

-  Comprehensive test suite
-  Advanced testing scenarios
-  80-90% test coverage
-  Professional-grade testing infrastructure

## **Cost/Time Savings:**

### **Manual + Cursor Approach:**

- **Your Time:** 10-14 days
- **Developer Time Saved:** 5-7 days
- **Cost Savings:** \$2,000-3,500

**This approach gives you the best of both worlds - understanding from manual work plus comprehensive coverage from Cursor automation.**