Testing Strategy for Affiliate Matrix

This document outlines the testing strategy for the remaining modules of the Affiliate Matrix system. It provides guidance on test types, test coverage requirements, and best practices for ensuring the quality and reliability of the implementation.

Testing Principles

1. **Test-Driven Development (TDD)**: Write tests before implementing functionality to ensure requirements are met.
2. **Comprehensive Coverage**: Aim for at least 80% code coverage across all modules.
3. **Isolation**: Test components in isolation using mocks and stubs for dependencies.
4. **Integration**: Test how components work together in realistic scenarios.
5. **Automation**: Automate tests to run on every code change.
6. **Performance**: Include performance tests for critical paths.

Test Types

1. Unit Tests

Unit tests verify that individual functions, methods, and classes work as expected in isolation.

Key Focus Areas: - Core business logic in each module - Edge cases and error handling - Data transformations and validations

Tools: - pytest for Python backend - Jest for JavaScript/TypeScript frontend

Example Unit Test for Key Management:

```
# test_key_manager.py
import pytest
from datetime import datetime, timedelta
from app.core.key_management.key_manager import KeyManager,
ApiCredential

def test_credential_expiration():
    # Test that credential expiration is correctly detected
    now = datetime.utcnow()

    # Create a credential that is not expired
```

```python
    valid_credential = ApiCredential(
        service_name="test_service",
        api_key="valid_key",
        created_at=now,
        expires_at=now + timedelta(days=1)
    )
    assert not valid_credential.is_expired

    # Create a credential that is expired
    expired_credential = ApiCredential(
        service_name="test_service",
        api_key="expired_key",
        created_at=now - timedelta(days=2),
        expires_at=now - timedelta(days=1)
    )
    assert expired_credential.is_expired

    # Create a credential with no expiration
    no_expiry_credential = ApiCredential(
        service_name="test_service",
        api_key="no_expiry_key",
        created_at=now
    )
    assert not no_expiry_credential.is_expired

def test_get_credential(mocker):
    # Test retrieving a credential using a mocked Vault client
    mock_vault_client = mocker.Mock()
    mock_vault_client.secrets.kv.v2.read_secret_version.return_value = {
        "data": {
            "data": {
                "service_name": "test_service",
                "api_key": "test_key",
                "created_at": datetime.utcnow().isoformat(),
                "expires_at": None
            }
        }
    }

    # Create KeyManager with mocked Vault client
    key_manager = KeyManager(vault_url="http://localhost:8200",
vault_token="test_token")
    key_manager.vault_client = mock_vault_client

    # Get credential
    credential = key_manager.get_credential("test_service")

    # Verify the result
    assert credential is not None
    assert credential.service_name == "test_service"
    assert credential.api_key == "test_key"
    # Verify Vault client was called correctly
    mock_vault_client.secrets.kv.v2.read_secret_version.assert_called_once_with(
        path="aggregators/test_service"
    )
```

2. Integration Tests

Integration tests verify that different components work together correctly.

Key Focus Areas: - Interactions between modules - Data flow through multiple components - API endpoints and database operations

Tools: - pytest for backend integration tests - Supertest for API testing

Example Integration Test for Master Index and API Integration:

```
# test_master_index_integration.py
import pytest
from app.core.master_index import MasterIndex
from app.services.aggregator_api import AggregatorApiService

@pytest.fixture
def master_index():
    return MasterIndex()

@pytest.fixture
def api_service(mocker):
    # Mock the API service to return test data
    service = AggregatorApiService()
    mocker.patch.object(
        service,
        'get_programs',
        return_value=[
            {
                "id": "test-1",
                "name": "Test Program 1",
                "website": "https://test1.com",
                "commission_type": "percentage",
                "commission_value": 10.0
            },
            {
                "id": "test-2",
                "name": "Test Program 2",
                "website": "https://test2.com",
                "commission_type": "flat",
                "commission_value": 50.0
            }
        ]
    )
    return service

def test_import_from_api(master_index, api_service):
    # Test importing data from the API service to the master index

    # Get programs from API
    programs = api_service.get_programs()

    # Import to master index
    result = master_index.import_from_aggregator("test_aggregator", programs)

    # Verify import results
    assert result["total"] == 2
    assert result["added"] == 2
    assert result["updated"] == 0
    assert result["failed"] == 0

    # Verify programs were added to the index
    program1 = master_index.get_program("test-1")
    assert program1 is not None
    assert program1.name == "Test Program 1"
    assert program1.commission_type == "percentage"

    program2 = master_index.get_program("test-2")
    assert program2 is not None
    assert program2.name == "Test Program 2"
    assert program2.commission_type == "flat"
```

3. API Tests

API tests verify that the REST API endpoints work correctly.

Key Focus Areas: - Request validation - Response format and status codes - Authentication and authorization - Rate limiting and error handling

Tools: - pytest with requests or httpx for Python - Postman for manual API testing - Newman for automated Postman collections

Example API Test for Dorking Endpoints:

```
 # test_dorking_api.py
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_trigger_dorking():
    # Test triggering a dorking operation via API
    response = client.post(
        "/api/dorking/trigger",
        json={
            "query": "affiliate program commission",
            "strategy": "affiliate_programs",
            "max_results": 10
        },
        headers={"X-API-Key": "test_api_key"}
    )

    # Verify response
    assert response.status_code == 202
    data = response.json()
    assert "task_id" in data
    assert "estimated_completion" in data

def test_get_dorking_results():
    # Test retrieving dorking results
    # First create a task
    trigger_response = client.post(
        "/api/dorking/trigger",
        json={
            "query": "affiliate program commission",
            "strategy": "affiliate_programs",
            "max_results": 10
        },
        headers={"X-API-Key": "test_api_key"}
    )
    task_id = trigger_response.json()["task_id"]

    # Then get the results
    response = client.get(
        f"/api/dorking/results/{task_id}",
        headers={"X-API-Key": "test_api_key"}
    )

    # Verify response
    assert response.status_code == 200
    data = response.json()
    assert "status" in data
    assert "results" in data

    # Status could be "pending", "processing", or "completed"
    assert data["status"] in ["pending", "processing", "completed"]
```

4. Frontend Tests

Frontend tests verify that the UI components and interactions work correctly.

Key Focus Areas: - Component rendering - User interactions - State management - API integration

Tools: - Vue Test Utils for component testing - Jest for JavaScript unit testing - Cypress for end-to-end testing

Example Vue Component Test:

```
 // BudgetAllocation.spec.js
import { mount } from '@vue/test-utils'
import BudgetAllocation from '@/components/BudgetAllocation.vue'

describe('BudgetAllocation', () => {
  test('renders the component with campaign data', () => {
    const campaigns = [
      { id: '1', name: 'Campaign 1', current_budget: 1000, spent_budget: 500 },
      { id: '2', name: 'Campaign 2', current_budget: 2000, spent_budget: 1000 }
    ]

    const wrapper = mount(BudgetAllocation, {
      props: {
        campaigns,
        totalBudget: 5000
      }
    })

    // Check that campaign names are displayed
    expect(wrapper.text()).toContain('Campaign 1')
    expect(wrapper.text()).toContain('Campaign 2')

    // Check that budget values are displayed
    expect(wrapper.text()).toContain('$1,000')
    expect(wrapper.text()).toContain('$2,000')

    // Check that remaining budget is calculated correctly
    expect(wrapper.text()).toContain('$2,000') // Total remaining
  })

  test('allocates budget when allocation button is clicked', async () => {
    const campaigns = [
      { id: '1', name: 'Campaign 1', current_budget: 1000, spent_budget: 500 },
      { id: '2', name: 'Campaign 2', current_budget: 2000, spent_budget: 1000 }
    ]

    const wrapper = mount(BudgetAllocation, {
      props: {
        campaigns,
        totalBudget: 5000
      }
    })

    // Find and click the allocate button
    const allocateButton = wrapper.find('[data-test="allocate-button"]')
    await allocateButton.trigger('click')
    // Check that the allocate event was emitted
    expect(wrapper.emitted('allocate')).toBeTruthy()
    expect(wrapper.emitted('allocate')[0]).toEqual([{
      strategy: 'proportional',
      metric: 'roi'
    }])
  })
})
```

5. Performance Tests

Performance tests verify that the system meets performance requirements under load.

Key Focus Areas: - Response time for critical operations - Throughput under load - Resource utilization - Caching effectiveness

Tools: - Locust for load testing - pytest-benchmark for function performance - Chrome DevTools for frontend performance

Example Locust Test for Master Index API:

```
 # locustfile.py
from locust import HttpUser, task, between

class MasterIndexUser(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        # Login to get API key
        response = self.client.post(
            "/api/auth/login",
            json={"username": "test_user", "password": "test_password"}
        )
        self.api_key = response.json()["api_key"]

    @task(3)
    def search_programs(self):
        # Search for programs with different queries
        queries = [
            "affiliate marketing",
            "ecommerce commission",
            "digital products"
        ]
        query = random.choice(queries)

        self.client.get(
            f"/api/programs/search?query={query}&limit=20",
            headers={"X-API-Key": self.api_key}
        )

    @task(1)
    def get_program_details(self):
        # Get details for a specific program
        program_ids = ["amz-001", "ebay-002", "etsy-003"]
        program_id = random.choice(program_ids)

        self.client.get(
            f"/api/programs/{program_id}",
            headers={"X-API-Key": self.api_key}
        )
```

6. Security Tests

Security tests verify that the system is protected against common vulnerabilities.

Key Focus Areas: - Authentication and authorization - Input validation and sanitization - Secure storage of sensitive data - Protection against common attacks (SQL injection, XSS, CSRF)

Tools: - OWASP ZAP for automated security testing - pytest with security-focused test cases - Bandit for Python code security analysis

Example Security Test for Key Management:

```
# test_key_manager_security.py
import pytest
from app.core.key_management.key_manager import KeyManager

def test_credential_encryption(mocker):
    # Test that credentials are encrypted before storage

    # Mock Vault client
    mock_vault_client = mocker.Mock()

    # Create KeyManager with mocked Vault client
    key_manager = KeyManager(vault_url="http://localhost:8200",
vault_token="test_token")
    key_manager.vault_client = mock_vault_client

    # Create and store a credential
    credential = ApiCredential(
        service_name="test_service",
        api_key="sensitive_api_key",
        secret="very_sensitive_secret",
        created_at=datetime.utcnow()
    )
    key_manager.store_credential(credential)
    # Verify that Vault was called with encrypted data
    # The actual implementation would encrypt sensitive fields
    mock_vault_client.secrets.kv.v2.create_or_update_secret.assert_called_once()
    call_args = mock_vault_client.secrets.kv.v2.create_or_update_secret.call_args[1]

    # Verify path is correct
    assert call_args["path"] == "aggregators/test_service"

    # In a real implementation, we would verify that the secret was encrypted
    # For this test, we're just checking that the method was called
```

Test Coverage Requirements

Each module should have the following minimum test coverage:

Module	Unit Tests	Integration Tests	API Tests	UI Tests
Aggregator Connection	80%	70%	90%	N/A
API Integration	80%	70%	90%	N/A
Key Management	90%	70%	N/A	N/A
Master Index	80%	70%	90%	70%
Dynamic Indexing & Caching	80%	70%	N/A	N/A
Trigger System	80%	70%	80%	70%
Budgeting System	80%	70%	90%	80%
Apex Optimizations	70%	60%	N/A	N/A
Monitoring System	80%	70%	80%	80%

Test Data Management

Test Data Sources

1. **Mock Data**: Generate realistic mock data for affiliate programs, campaigns, and performance metrics.
2. **Anonymized Production Data**: Use anonymized data from production for realistic testing scenarios.

3. **Synthetic Data**: Create synthetic data that tests edge cases and boundary conditions.

Test Database Setup

1. Use a separate test database for integration tests.
2. Reset the database to a known state before each test run.
3. Use database migrations to ensure schema consistency.

Example Test Database Setup:

```python
# conftest.py
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from app.db.base import Base
from app.core.config import settings

@pytest.fixture(scope="session")
def engine():
    return create_engine(settings.TEST_DATABASE_URL)

@pytest.fixture(scope="session")
def tables(engine):
    Base.metadata.create_all(engine)
    yield
    Base.metadata.drop_all(engine)

@pytest.fixture
def db_session(engine, tables):
    connection = engine.connect()
    transaction = connection.begin()
    session = sessionmaker(bind=connection)()

    yield session

    session.close()
    transaction.rollback()
    connection.close()
```

Mocking External Dependencies

When testing components that depend on external services, use mocks to isolate the component under test.

Example Mocking External API:

```python

# test_aggregator_api.py

import pytest import requests from app.services.aggregator_api import AggregatorApiService

def test_get_programs(mocker): # Mock the requests.get method mock_response = mocker.Mock() mock_response.status_code = 200 mock_response.json.return_value = { "programs": [ { "id": "test-1", "name": "Test Program 1", "website": "https://test1.com", "commission_type": "percentage", "commission_value": 10.0 } ] } mocker.patch('requests.get', return_value=mock_response)

```
 # Create service and call method
service = AggregatorApiService(api_url="https://api.example.com")
programs = service.get_programs()

# Verify result
assert len(programs) == 1
assert programs[0]["name"] == "Test Program 1"

# Verify requests.get was
```

(Content truncated due to size limit. Use line ranges to read in chunks)
```