TESTING POLICY DOCUMENT

STOCKFELLOW

DEVOPPS

BRIGHTBYTE ENTERPRISES

DEMO 4

Name and Surname	**Student Number**
*Tinotenda Chirozvi	22547747
Diyaana Jadwat	23637252
Dean Ramsey	22599012
Naazneen Khan	22527533
Lubabalo Tshikila	22644106

Contents

1 Introduction

This testing policy establishes comprehensive guidelines and standards for testing the software application for our capstone project. The policy ensures consistent, reliable, and maintainable testing practices across all development phases, with a focus on delivering high-quality software that meets functional and non-functional requirements. It covers the entire software development lifecycle from initial development through production deployment and maintenance.

1.1 Testing Objectives

- Ensure software functionality meets specified requirements
- Identify and resolve defects early in the development cycle
- Maintain code quality and reliability standards
- Provide confidence in software releases
- Support continuous integration and deployment practices
- Minimize production incidents and user-facing issues

1.2 Testing Principles

- **Shift-Left Testing**: Testing activities begin early in the development lifecycle
- **Test Automation**: Prioritize automated testing for efficiency and reliability
- **Risk-Based Testing**: Focus testing efforts on high-risk areas and critical functionality
- **Continuous Testing**: Integrate testing into CI/CD pipelines
- **Test Data Management**: Maintain realistic and secure test data

2 Functional Testing

Functional testing verifies that software components behave according to specified requirements and business logic. All functional tests must validate both positive and negative scenarios, edge cases, and error handling.

2.1 Requirements

- **Test Coverage**: Minimum 80% code coverage for all business logic
- **Test Documentation**: All test cases must include clear descriptions, expected outcomes, and traceability to requirements
- **Data Validation**: Tests must validate input/output data formats, constraints, and business rules
- **Error Handling**: Comprehensive testing of error conditions and exception scenarios

2.2 Automated Testing

2.2.1 Strategy

Automated testing is integrated into the continuous integration pipeline, unfortunately we only have 5 min on to build on our CICD pipeline, so not all our tests are automated because the system is so large that testing takes too long but they are all running locally. Tests run automatically when code is pushed into the main production branch.

2.2.2 Tool Standards

- **Java Applications**: JUnit 5 for unit testing, Mockito for mocking
- **Spring Boot Applications**: Spring Boot Test framework with MockMvc for web layer testing
- **Test Runners**: Maven Surefire Plugin for test execution
- **CI/CD Integration**: Tests must integrate with build pipelines and block deployments on failures

2.3 Unit Testing

Unit tests validate individual components, methods, or classes in isolation. These tests form the foundation of our testing strategy and must be written by developers as part of the development process.

2.3.1 Standards

- **Coverage Requirements**: Minimum 80% line coverage, 70% branch coverage
- **Isolation**: Use mocking frameworks to isolate units under test
- **Test Structure**: Follow Given-When-Then or Arrange-Act-Assert patterns
- **Naming Convention**: Test methods should clearly describe the scenario and expected outcome

2.3.2 Implementation Guidelines

Example structure from UserServiceTest.java:

```
@ExtendWith(MockitoExtension.class)
public class ServiceTest {
    @Mock
    private Repository repository;

    @InjectMocks
    private Service service;

    @Test
    void methodName_ShouldExpectedBehavior_WhenCondition() {
        // Arrange
```

```
        when(repository.method()).thenReturn(expectedValue);

        // Act
        Result result = service.performOperation();

        // Assert
        assertNotNull(result);
        assertEquals(expectedValue, result.getValue());
        verify(repository, times(1)).method();
    }
}
```

2.3.3 Required Test Scenarios

- Valid input processing
- Invalid input handling
- Boundary value testing
- Exception handling

2.4 Integration Testing

Integration tests validate the interaction between multiple components, services, or external systems. These tests ensure that integrated components work correctly together.

2.4.1 Types

- **Component Integration**: Testing interactions between application layers
- **Service Integration**: Testing API endpoints and service contracts
- **Database Integration**: Testing data persistence and retrieval operations
- **External Service Integration**: Testing third-party service integrations

2.4.2 Implementation Standards

Example from UserControllerTest.java :

```
@WebMvcTest(value = Controller.class, excludeAutoConfiguration = {
    SecurityAutoConfiguration.class,
    SecurityFilterAutoConfiguration.class
})
public class ControllerIntegrationTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private Service service;
```

```
    @Test
    void endpoint_ShouldReturnExpectedResponse() throws Exception
       {
        mockMvc.perform(get("/api/endpoint")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.field").value("
                   expectedValue"));
    }
}
```

2.4.3 Requirements

- Test realistic data flows and business scenarios
- Validate API contracts and response formats
- Test authentication and authorization mechanisms
- Verify error handling and response codes
- Test configuration and environment-specific behavior

2.5 End-to-End Testing

End-to-end tests validate complete user workflows and business processes across the entire application stack, from user interface to data persistence.

2.5.1 Implementation Strategy

- **User Journey Testing**: Test critical user paths and business workflows
- **Cross-Service Testing**: Validate interactions between microservices
- **Data Flow Testing**: Ensure data integrity across system boundaries
- **Performance Validation**: Verify acceptable response times for user interactions

2.5.2 Execution Environment

- **Test Environment**: Dedicated environment that mirrors production
- **Test Data**: Realistic data sets that represent production scenarios
- **External Dependencies**: Mock or stub external services appropriately
- **Monitoring**: Implement test execution monitoring and reporting

3 Non-Functional Testing

Non-functional testing validates system attributes such as performance, security, usability, and reliability. These tests ensure the application meets quality standards beyond functional requirements.

3.1 Types and Requirements

- **Performance Testing**: Response time, throughput, resource utilization
- **Security Testing**: Authentication, authorization, data protection
- **Usability Testing**: User experience, accessibility, interface design
- **Reliability Testing**: System availability, error recovery, fault tolerance
- **Scalability Testing**: Load handling, resource scaling, capacity planning

3.2 Usability Testing

Ensure applications provide intuitive, accessible, and efficient user experiences across different user types and scenarios.

3.2.1 Testing Areas

- **User Interface Design**: Layout, navigation, visual consistency
- **Accessibility**:
- **User Workflow**: Task completion efficiency, error recovery
- **Cross-Platform Compatibility**: Different browsers, devices, screen sizes

3.2.2 Implementation

- Conduct user acceptance testing with representative users
- Gather user feedback through surveys and usability sessions

3.3 Security Testing

All applications must undergo comprehensive security testing to identify vulnerabilities and ensure data protection compliance.

3.3.1 Testing Areas

- **Authentication and Authorization**: User access controls and permissions
- **Data Protection**: Encryption, data handling, privacy compliance
- **Input Validation**: SQL injection, XSS, CSRF protection
- **API Security**: Rate limiting, token validation, secure communications
- **Infrastructure Security**: Network security, configuration management

3.3.2 Implementation Standards

Based on the security exclusions in test configurations:

- Test security configurations and access controls
- Validate input sanitization and output encoding
- Test session management and token handling
- Perform penetration testing for critical applications
- Implement security scanning in CI/CD pipelines

3.4 Scalability Testing

3.4.1 Performance Requirements

Applications must handle expected load volumes and scale appropriately under varying conditions.

3.4.2 Testing Types

- **Load Testing**: Normal expected traffic patterns
- **Stress Testing**: Beyond normal capacity limits
- **Volume Testing**: Large amounts of data processing
- **Spike Testing**: Sudden load increases
- **Endurance Testing**: Extended period performance

3.4.3 Implementation

- Define performance benchmarks and acceptance criteria
- Test database performance and query optimization
- Validate caching mechanisms and effectiveness
- Test auto-scaling capabilities and resource management
- Monitor system resources during load testing

3.4.4 Metrics and Monitoring

- Response time percentiles (95th, 99th percentile)
- Throughput (requests per second)
- Resource utilization (CPU, memory, disk, network)
- Error rates under load conditions
- System recovery time after load reduction

4 Conclusion

This testing policy provides the framework for comprehensive software testing that ensures quality, reliability, and maintainability of our applications. Regular review and updates of this policy ensure it remains relevant and effective as technology and requirements evolve.