Integration Testing Documentation

1. Testing Strategy Overview

The Tic-Tac-Toe integration testing focuses on critical component interactions and end-to-end workflows using Google Test framework on Ubuntu Linux environment.

Key Integration Areas:

- Database coordination between UserAuth and GameHistory components
- Complete game workflows from authentication through completion
- GUI integration with backend services
- Cross-component error handling and data consistency

Coverage Goals:

- 100% coverage for critical integration paths
- All multi-component workflows tested
- Error scenarios validated across component boundaries

2. Database Integration Tests

2.1 Core Database Operations

Test Case: DatabaseCreationAndInitialization

```
TEST_F(DatabaseIntegrationTest, DatabaseCreationAndInitialization) {
    EXPECT_TRUE(std::filesystem::exists(authDbPath));
    EXPECT_TRUE(std::filesystem::exists(historyDbPath));
    EXPECT_TRUE(userAuth->registerUser("testuser", "test123"));
    int gameId = gameHistory->initializeGame(1001, 1002);
    EXPECT_GT(gameId, 0);
    EXPECT_TRUE(userAuth->login("testuser", "test123"));
    auto game = gameHistory->getGameById(gameId);
    EXPECT_EQ(game.playerX_id.value(), 1001);
}
```

- **Purpose**: Validates simultaneous database initialization without conflicts
- **Expected Outcome**: Both databases created, authentication and game operations succeed

- **Critical Path**: Application startup with dual database system

Test Case: ConcurrentDatabaseAccess

```
TEST_F(DatabaseIntegrationTest, ConcurrentDatabaseAccess) {
    std::vector<std::thread> threads;
    std::vector<int> gameIds(10, -1);
    std::vector<bool> authResults(10, false);

    for (int i = 0; i < 10; i++) {
        threads.emplace_back([&, i]() {
            if (i % 2 == 0) {
                std::string username = "concurrent_user_" + std::to_string(i);
                authResults[i] = userAuth->registerUser(username, "pass123");
            } else {
                gameIds[i] = gameHistory->initializeGame(2000 + i, 3000 + i);
            }
        });
    }
}
```

- **Purpose**: Tests thread-safe concurrent access to separate databases
- **Expected Outcome**: All 10 concurrent operations complete successfully without data corruption
- **Performance Validation**: Ensures database isolation under concurrent load

Test Case: DatabasePersistenceAcrossRecreation

```
EXPECT_TRUE(userAuth->registerUser("persistent_user", "persist123"));
int gameId = gameHistory->initializeGame(5001, 5002);
    EXPECT_GT(gameId, 0);
    EXPECT_TRUE(gameHistory->recordMove(gameId, 4));
    EXPECT_TRUE(gameHistory->recordMove(gameId, 0));
    EXPECT_TRUE(gameHistory->setWinner(gameId, 5001));
// Destroy and recreate components
userAuth.reset();
```

```
gameHistory.reset();
userAuth = std::make_unique<UserAuth>(authDbPath);
gameHistory = std::make_unique<GameHistory>(historyDbPath);
// Verify data persisted
EXPECT_TRUE(userAuth->login("persistent_user", "persist123"));
```

- **Purpose**: Validates data persistence across component lifecycle
- **Implementation**: Creates data, destroys components, recreates them, verifies data remains
- **Expected Outcome**: All data persists after component destruction and recreation
- **Reliability Check**: Ensures robust data handling across application restarts

2.2 Data Consistency

Test Case: CrossComponentDataConsistency

```
TEST_F(DatabaseIntegrationTest, CrossComponentDataConsistency) {
  EXPECT_TRUE(userAuth->registerUser("alice", "alice123"));
  EXPECT_TRUE(userAuth->registerUser("bob", "bob123"));
  int aliceId = qHash(QString("alice"));
  int bobId = qHash(QString("bob"));
  int gameId = gameHistory->initializeGame(aliceId, bobId);
  gameHistory->setWinner(gameId, aliceId);
  auto aliceGames = gameHistory->getPlayerGames(aliceId);
  auto bobGames = gameHistory->getPlayerGames(bobId);
  EXPECT_EQ(aliceGames.size(), 1);
  EXPECT_EQ(bobGames.size(), 1);
  EXPECT_EQ(aliceGames[0].winner_id.value(), aliceId);
}
```

- **Purpose**: Ensures data consistency when player IDs span both databases
- **Expected Outcome**: Player data remains consistent between authentication and game systems
- **Data Integrity**: Validates referential consistency across database boundaries

3. Game Workflow Integration Tests

3.1 Complete Game Scenarios

Test Case: CompletePlayerVsAIWorkflow

```
TEST_F(GameWorkflowIntegrationTest, CompletePlayerVsAIWorkflow) {
    EXPECT_TRUE(simulateLogin("alice", "alice123"));
    int aliceId = qHash(QString("alice"));
    int gameId = gameHistory->initializeGame(aliceId, std::nullopt);
    std::vector<int> moves = {4, 0, 1, 3, 7};
    for (size_t i = 0; i < moves.size(); i++) {
        EXPECT_TRUE(gameHistory->recordMove(gameId, moves[i]));
        auto game = gameHistory->getGameById(gameId);
        EXPECT_EQ(game.moves.size(), i + 1);
    }
    EXPECT_TRUE(gameHistory->setWinner(gameId, aliceId));
}
```

- **Purpose**: Complete Player vs AI workflow from login to completion
- **Expected Outcome**: Game initializes, moves recorded sequentially, winner set, appears in history
- **Integration Scope**: UserAuth + GameHistory + Game Logic

Test Case: CompletePlayerVsPlayerWorkflow

```
TEST_F(GameWorkflowIntegrationTest, CompletePlayerVsPlayerWorkflow) {
    EXPECT_TRUE(simulateLogin("alice", "alice123"));
    EXPECT_TRUE(simulateLogin("bob", "bob123"));
    int aliceId = qHash(QString("alice"));
    int bobId = qHash(QString("bob"));
    int gameId = gameHistory->initializeGame(aliceId, bobId);
    std::vector<int> drawMoves = {4, 0, 8, 2, 6, 3, 5, 1, 7};
    for (int move : drawMoves) {
        EXPECT_TRUE(gameHistory->recordMove(gameId, move));
    }
```

```
    EXPECT_TRUE(gameHistory->setWinner(gameId, -1)); // Draw
    auto aliceGames = gameHistory->getPlayerGames(aliceId);
    auto bobGames = gameHistory->getPlayerGames(bobId);
    EXPECT_EQ(aliceGames.size(), 1);
    EXPECT_EQ(bobGames.size(), 1);
}
```

- **Purpose**: Validates dual authentication and PvP game completion
- **Expected Outcome**: Both players authenticate, game completes as draw, appears in both histories
- **Multi-User Testing**: Ensures proper two-player scenario handling

3.2 Logic and State Consistency

Test Case: GameLogicHistoryConsistency

```
TEST_F(GameWorkflowIntegrationTest, GameLogicHistoryConsistency) {
    Board gameBoard;
    int gameId = gameHistory->initializeGame(aliceId, std::nullopt);
    std::vector<std::pair<int, int>> positions = {{1, 1}, {0, 0}, {0, 1}, {2, 2}, {2, 1}};
    Player currentPlayer = Player::X;
    for (const auto& [row, col] : positions) {
        EXPECT_TRUE(gameBoard.makeMove(row, col, currentPlayer));
        int position = row * 3 + col;
        EXPECT_TRUE(gameHistory->recordMove(gameId, position));
        if (gameBoard.isGameOver()) {
            WinInfo result = gameBoard.checkWinner();
            std::optional<int> winnerId = (result.winner == Player::X) ? aliceId : -2;
            EXPECT_TRUE(gameHistory->setWinner(gameId, winnerId));
            break;
        }
        currentPlayer = (currentPlayer == Player::X) ? Player::O : Player::X;
    }
    // Verify consistency by recreating board from history
    Board verifyBoard;
```

```
    auto game = gameHistory->getGameById(gameId);
    Player verifyPlayer = Player::X;
    for (const auto& move : game.moves) {
        int row = move.position / 3;
        int col = move.position % 3;
        EXPECT_TRUE(verifyBoard.makeMove(row, col, verifyPlayer));
        verifyPlayer = (verifyPlayer == Player::X) ? Player::O : Player::X;
    }
    EXPECT_EQ(gameBoard.isGameOver(), verifyBoard.isGameOver());
}
```

- **Purpose**: Ensures perfect consistency between real-time game logic and stored history
- **Expected Outcome**: Board state recreated from history matches original exactly
- **Critical Feature**: Essential for game replay and analysis capabilities

4. GUI Integration Tests

4.1 Complete User Interface Workflows

Test Case: CompleteLoginToGameWorkflow

```
TEST_F(GUIIntegrationTest, CompleteLoginToGameWorkflow) {
    QStackedWidget* stackedWidget = mainWindow->getStackedWidget();
    LoginPage* loginPage = mainWindow->getLoginPage();
    GameWindow* gameWindow = mainWindow->getGameWindow();
    EXPECT_EQ(stackedWidget->currentWidget(), loginPage);
    EXPECT_TRUE(performLogin(aliceUsername, "alice123"));
    EXPECT_EQ(stackedWidget->currentWidget(), gameWindow);
    EXPECT_EQ(mainWindow->getCurrentUser(), aliceUsername);
    QList<QPushButton*> buttons = gameWindow->findChildren<QPushButton*>();
    bool foundPvPButton = false, foundPvAIButton = false;
    for (auto* btn : buttons) {
        if (btn->text() == "Player vs Player") foundPvPButton = true;
        if (btn->text() == "Player vs AI") foundPvAIButton = true;
    }
```

```
    EXPECT_TRUE(foundPvPButton && foundPvAIButton);
}
```

- **Purpose**: Tests complete GUI navigation from login to game selection
- **Expected Outcome**: Successful login navigates to game window with proper UI elements
- **User Experience**: Validates primary user workflow through interface

Test Case: ErrorHandlingIntegration

```
TEST_F(GUIIntegrationTest, ErrorHandlingIntegration) {
    LoginPage* loginPage = mainWindow->getLoginPage();
    QLineEdit* usernameEdit = loginPage->findChild<QLineEdit*>("m_usernameEdit");
    QLineEdit* passwordEdit = loginPage->findChild<QLineEdit*>("m_passwordEdit");
    QPushButton* registerButton = loginPage->findChild<QPushButton*>("m_registerButton");
    QLabel* statusLabel = loginPage->findChild<QLabel*>("m_statusLabel");
    QString newUsername = "newuser_" + QString::number(reinterpret_cast<uintptr_t>(this));
    // Test password too short
    usernameEdit->setText(newUsername);
    passwordEdit->setText("123");
    QTest::mouseClick(registerButton, Qt::LeftButton);
    QCoreApplication::processEvents();
    EXPECT_TRUE(statusLabel->text().contains("5 characters", Qt::CaseInsensitive));
    // Test valid registration
    passwordEdit->setText("test123");
    QTest::mouseClick(registerButton, Qt::LeftButton);
    QCoreApplication::processEvents();
    EXPECT_TRUE(statusLabel->text().contains("success", Qt::CaseInsensitive));
}
```

- **Purpose**: Tests GUI error handling integration with backend validation
- **Expected Outcome**: GUI displays appropriate error messages and handles failures gracefully
- **User Experience**: Ensures clear feedback for invalid inputs

Test Case: RealTimeGameHistoryUpdates

```
TEST_F(GUIIntegrationTest, RealTimeGameHistoryUpdates) {
    EXPECT_TRUE(performLogin(aliceUsername, "alice123"));
    GameHistory* gameHistory = mainWindow->getGameHistory();
    GameHistoryGUI historyWindow(gameHistory, aliceUsername);
    historyWindow.show();
    QSignalSpy initSpy(gameHistory, SIGNAL(gameInitialized(int)));
    QSignalSpy moveSpy(gameHistory, SIGNAL(moveRecorded(int, int)));
    QSignalSpy completeSpy(gameHistory, SIGNAL(gameCompleted(int, std::optional<int>)));
    int aliceId = qHash(aliceUsername);
    int gameId = gameHistory->initializeGame(aliceId, std::nullopt);
    gameHistory->recordMove(gameId, 4);
    gameHistory->recordMove(gameId, 0);
    gameHistory->setWinner(gameId, aliceId);
    QCoreApplication::processEvents();
    EXPECT_EQ(initSpy.count(), 1);
    EXPECT_EQ(moveSpy.count(), 2);
    EXPECT_EQ(completeSpy.count(), 1);
    QTreeWidget* gamesTree = historyWindow.findChild<QTreeWidget*>();
    EXPECT_GT(gamesTree->topLevelItemCount(), 0);
}
```

- **Purpose**: Validates real-time GUI updates as game events occur
- **Expected Outcome**: GUI immediately reflects game initialization, moves, and completion
- **Real-Time Updates**: Critical for responsive user interface

5. Cross-Component Error Handling

Test Case: DatabaseErrorHandling

```
TEST_F(DatabaseIntegrationTest, DatabaseErrorHandling) {
    // Test invalid game operations
    EXPECT_FALSE(gameHistory->recordMove(99999, 0)); // Non-existent game
    EXPECT_FALSE(gameHistory->setWinner(99999, 1001)); // Non-existent game
    // Test auth error scenarios
```

```
    EXPECT_FALSE(userAuth->login("nonexistent", "password"));
    EXPECT_FALSE(userAuth->registerUser("", "")); // Empty credentials
    EXPECT_FALSE(userAuth->registerUser("user", "123")); // Too short
    EXPECT_FALSE(userAuth->registerUser("user", "12345")); // No letters
    EXPECT_FALSE(userAuth->registerUser("user", "abcde")); // No digits
}
```

- **Purpose**: Tests error handling consistency across database operations
- **Expected Outcome**: Invalid operations fail gracefully without affecting valid operations
- **Robustness**: Ensures system stability under error conditions

6. Test Infrastructure

6.1 Database Cleanup Strategy

The testing framework uses QTemporaryDir for automatic cleanup and unique filename generation:

```
QString authDbPath = tempPath + "/test_users_" +
    QString::number(reinterpret_cast<uintptr_t>(this)) + ".db";
QString historyDbPath = tempPath + "/test_history_" +
    QString::number(reinterpret_cast<uintptr_t>(this)) + ".db";
```

6.2 Test Isolation

Each test generates unique database instances using:

- QTemporaryDir for automatic cleanup
- Unique identifiers based on test instance pointers
- Separate database files per test to prevent conflicts

7. Coverage Summary

7.1 Critical Integration Paths Covered

- **Authentication + Database**: User registration, login, persistence across restarts
- **Game Logic + History**: Move recording, winner tracking, game state consistency
- **GUI + Backend**: Complete user workflows, error handling, real-time updates
- **Multi-Component**: Cross-database consistency, concurrent access, error propagation