**App Testing Documentation Report**

**I. Test Type: Manual Testing**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Scenario TID** | **Scenario Description** | **Test Case ID** | **Pre Condition** | **Steps to Execute** | **Expected Result** | **Actual Result** | **Status** |
| **Welcome Page** | | | | | | | |
| TS-01 | Verify the presence of three slidable pages | TC-01 | None | Open the app | Three slideable views should displayed with corresponding picture | Three slideable views is displayed with corresponding picture | **PASS** |
| TS-02 | Confirm the presence of a "Start" button on the last page | TC-02 | Page 3 is active | Click "Next" twice and check the presence of "Start" button | "Start" button should be visible | "Start" button is visible | **PASS** |
| **Mood View** | | | | | | | |
| TS-03 | Check if all buttons with emoji images are clickable | TC-03 | Mood View is active | Click each button with emoji image on the mood view | Clicking each button should take user to the journal view of that emoji image | Clicking each button takes user to the journal view of that emoji image | **PASS** |
| TS-04 | Verify that text on the buttons matches the sound played after clicking | TC-04 | Mood View is active | Click each button with text on the mood view | Text and sound should be the same for each button | Text and sound is the same for each button | **PASS** |
| TS-05 | Test navigation buttons at the bottom | TC-05 | Mood View is active | Click on Mood View, Memory View, and Map View buttons | Should Navigate to Mood View, Memory View, and Map View as expected | Navigates to Mood View, Memory View, and Map View as expected | **PASS** |
| **Journal View** | | | | | | | |
| TS-06 | Confirm that clicking on an emoji button navigates to the Journal View | TC-06 | Mood View is active | Click on an emoji button | Journal View should display articles related to the selected emoji | Journal View displays articles related to the selected emoji | **PASS** |
| TS-07 | Check if Emoji Journal View heading and location is consistent with the journal | TC-07 | Journal is selected | Open a specific journal | Information should show accurately for each journal | Information is shown accurately for each journal | **PASS** |
| **Memory View** | | | | | | | |
| TS-08 | Verify the ability to create a new journal | TC-08 | App is open | Click on the Memory View button | Journal creation form should open with relevant fields | Journal creation form opens with relevant fields | **PASS** |
| TS-09 | Test the publish button functionality | TC-09 | New journal is created | Fill out the form and click "Publish" | Journal should be published and associated with the selected emoji | Journal is published and associated with the selected emoji | **PASS** |
| TS-10 | Confirm the appearance of a pin on the world map | TC-10 | Journal is published | Navigate to Map View | A pin should be visible on the map for the published journal | A pin is visible on the map for the published journal | **PASS** |
| **Map View** | | | | | | | |
| TS-11 | Confirm the presence of the world map | TC-11 | App is open | Map View is clicked in navigation bar | World map should be displayed | World map is displayed | **PASS** |
| TS-12 | Test the visibility of pins representing journals | TC-12 | Journals are published | Navigate to Map View | Pins should be visible on the map corresponding to published journals | Pins are visible on the map corresponding to published journals | **PASS** |
| **Pin Manager View** | | | | | | | |
| TS-13 | Verify the functionality of the Pin Manager button | TC-13 | Map View is active | Click on Pin Manager button | Pin Manager view should open successfully | Pin Manager view opens successfully | **PASS** |
| TS-14 | Test the addition of pins through Pin Manager | TC-14 | Pin Manager view is active | Add a pin using Pin Manager | Pins should be successfully added to the selected location on the map | Pins are successfully added to the selected location on the map | **PASS** |
| TS-15 | Confirm the addition of pin categories through Pin Manager | TC-15 | Pin Manager view is active | Add a pin category using Pin Manager | Pin categories should be added successfully with names and symbols | Pin categories are added successfully with names and symbols | **PASS** |

**II. Test Type: UI Testing**

UI testing, or User Interface testing, is a software testing technique that focuses on evaluating the graphical user interface (GUI) of a software application to ensure that it functions correctly and provides a positive user experience. The primary goal of UI testing is to verify that the application's user interface behaves as expected and meets specified requirements.

* testLaunchPerformance: Measures the time it takes to launch the application. Uses the measure block to gather performance metrics, specifically XCTApplicationLaunchMetric().
* testMoodViewNavigation: Navigates through the application to test the navigation to the "Mood" tab. Launches the application and taps on buttons to navigate to the desired view. Verifies the existence of the "Mood" tab within a timeout period.
* test\_TappingHappyButtonNavigatesToEmojiJournalView: Tests the navigation from the "Mood" view to the "EmojiJournalView" upon tapping the "Happy" button. Launches the application, navigates to the "Mood" view, taps the "Happy" button, and verifies the appearance of the "EmojiJournalView." Taps on the first journal cell within the view.
* test\_TappingMapTabNavigatesToMapView: Tests the navigation from the "Mood" view to the "MapView" upon tapping the "Map" tab. Launches the application, navigates to the "Mood" view, taps the "Map" tab, and verifies the appearance of the "MapView."

**The code:**

import XCTest

final class MappedJournalUITests: XCTestCase {

override func setUpWithError() throws {

continueAfterFailure = false

}

override func tearDownWithError() throws {

// Put teardown code here. This method is called after the invocation of each test method in the class.

}

// performance testing

func testLaunchPerformance() throws {

if #available(macOS 10.15, iOS 13.0, tvOS 13.0, watchOS 7.0, \*) {

// This measures how long it takes to launch your application.

measure(metrics: [XCTApplicationLaunchMetric()]) {

XCUIApplication().launch()

}

}

}

//Test for opening page

func testMoodViewNavigation() throws {

let app = XCUIApplication()

app.launch()

// Navigate to MoodView

app.buttons["Next"].tap()

app.buttons["Next"].tap()

app.buttons["Start"].tap()

// Navigate to MoodView

let tabView = app.tabBars.buttons["Mood"]

print(tabView)

XCTAssertTrue(tabView.waitForExistence(timeout: 5), "Mood tab bar item not found")

}

func test\_TappingHappyButtonNavigatesToEmojiJournalView() {

let app = XCUIApplication()

app.launch()

// Navigate to MoodView

app.buttons["Next"].tap()

app.buttons["Next"].tap()

app.buttons["Start"].tap()

// Tap on the "Happy" button (replace "Mood-Button-Happy" with the actual identifier)

let happyButton = app.buttons["Mood-Image-Happy"]

XCTAssertTrue(happyButton.waitForExistence(timeout: 5), "Happy Image not found")

happyButton.tap()

// Verify that EmojiJournalView is presented

let emojiNavBar = app.navigationBars["Happy"]

XCTAssertTrue(emojiNavBar.waitForExistence(timeout: 5), "EmojiJournalView not displayed")

// Find and tap on the first journal in the list

let firstJournalCell = app.cells.firstMatch

XCTAssertTrue(firstJournalCell.waitForExistence(timeout: 5), "First journal cell not found")

firstJournalCell.tap()

}

func test\_TappingMapTabNavigatesToMapView() {

let app = XCUIApplication()

app.launch()

// Navigate to MoodView

app.buttons["Next"].tap()

app.buttons["Next"].tap()

app.buttons["Start"].tap()

// Navigate to MapView

let mapTab = app.tabBars.buttons["Map"]

XCTAssertTrue(mapTab.waitForExistence(timeout: 5), "Map tab not found")

mapTab.tap()

}

}

**III. Test Type: Integration Testing**

Integration testing is a software testing technique that focuses on verifying the interactions between different components or modules of a software application. The primary goal of integration testing is to ensure that the individual components, which may have been tested in isolation, work together as intended when integrated into a larger system.

Test 1: Pin

* This test is focused on saving and fetching a Pin entity from the Core Data store.
* Setup: The setUp method initializes a Core Data stack for testing by creating an in-memory persistent store coordinator and a managed object context. The managed object context is associated with the in-memory store coordinator.
* Test Method: testSaveAndFetchPin
* Code:

Given: A Pin object is created with specific latitude and longitude values.

When: The Pin object is saved to the managed object context.

Then: A fetch request retrieves Pin objects from the managed object context. Assertions are made to check if the saved Pin is present in the database and if its attributes (latitude and longitude) match the values set during creation.

**The code:**

import XCTest

import CoreData

@testable import MappedJournal

class PinIntegrationTests: XCTestCase {

var managedObjectContext: NSManagedObjectContext!

override func setUp() {

super.setUp()

// Set up the Core Data stack for testing

let managedObjectModel = NSManagedObjectModel.mergedModel(from: [Bundle(for: Pin.self)])

let persistentStoreCoordinator = NSPersistentStoreCoordinator(managedObjectModel: managedObjectModel!)

do {

try persistentStoreCoordinator.addPersistentStore(ofType: NSInMemoryStoreType, configurationName: nil, at: nil, options: nil)

} catch {

fatalError("Failed to initialize in-memory store coordinator: \(error)")

}

managedObjectContext = NSManagedObjectContext(concurrencyType: .mainQueueConcurrencyType)

managedObjectContext.persistentStoreCoordinator = persistentStoreCoordinator

}

override func tearDown() {

managedObjectContext = nil

super.tearDown()

}

func testSaveAndFetchPin() {

// Given

let pin = Pin(context: managedObjectContext)

pin.pinLatitude = 37.7749

pin.pinLongtitude = -122.4194

// Set other properties as needed

// When

do {

try managedObjectContext.save()

} catch {

XCTFail("Failed to save context: \(error)")

return

}

// Then

let fetchRequest: NSFetchRequest<Pin> = Pin.fetchRequest()

do {

let pins = try managedObjectContext.fetch(fetchRequest)

XCTAssertEqual(pins.count, 1, "There should be one pin in the database")

XCTAssertEqual(pins.first?.pinLatitude, 37.7749, "Latitude should match")

XCTAssertEqual(pins.first?.pinLongtitude, -122.4194, "Longitude should match")

// Add other assertions as needed

} catch {

XCTFail("Failed to fetch pins: \(error)")

}

}

}

Test 2: Delete Pin

* Setup: Similar to the first test, the setUp method initializes a Core Data stack for testing with an in-memory persistent store coordinator and a managed object context.
* Test Method: testDeletePin
* Code:

Given: A Pin object is created with specific latitude and longitude values. The Pin object is saved to the managed object context.

When: The Pin object is deleted from the managed object context. The managed object context is saved after the deletion.

Then: A fetch request retrieves Pin objects from the managed object context. An assertion is made to check if there are no Pin objects in the database after deletion.

**The code:**

import XCTest

import CoreData

@testable import MappedJournal

class DeletePinIntegrationTests: XCTestCase {

var managedObjectContext: NSManagedObjectContext!

override func setUp() {

super.setUp()

// Set up the Core Data stack for testing

let managedObjectModel = NSManagedObjectModel.mergedModel(from: [Bundle(for: Pin.self)])

let persistentStoreCoordinator = NSPersistentStoreCoordinator(managedObjectModel: managedObjectModel!)

do {

try persistentStoreCoordinator.addPersistentStore(ofType: NSInMemoryStoreType, configurationName: nil, at: nil, options: nil)

} catch {

fatalError("Failed to initialize in-memory store coordinator: \(error)")

}

managedObjectContext = NSManagedObjectContext(concurrencyType: .mainQueueConcurrencyType)

managedObjectContext.persistentStoreCoordinator = persistentStoreCoordinator

}

override func tearDown() {

managedObjectContext = nil

super.tearDown()

}

func testDeletePin() {

// Given

let pin = Pin(context: managedObjectContext)

pin.pinLatitude = 37.7749

pin.pinLongtitude = -122.4194

// Set other properties as needed

do {

try managedObjectContext.save()

} catch {

XCTFail("Failed to save context: \(error)")

return

}

// When

managedObjectContext.delete(pin)

do {

try managedObjectContext.save()

} catch {

XCTFail("Failed to save context after deleting pin: \(error)")

return

}

// Then

let fetchRequest: NSFetchRequest<Pin> = Pin.fetchRequest()

do {

let pins = try managedObjectContext.fetch(fetchRequest)

XCTAssertEqual(pins.count, 0, "There should be no pins in the database after deletion")

} catch {

XCTFail("Failed to fetch pins: \(error)")

}

}

}