Part A :Manual Test Scenarios - Add Item Process

Scenario 1: Add Item Success with Gallery Image

Preconditions:

- The user is logged in.
- The user is on the Add Item screen.
- The device has access to the gallery.

Steps:

1. Tap Pick Image (Gallery).
2. Select an image from the device gallery.
3. Enter a valid value in the Name field.
4. Enter a valid value in the Type field , Color, Season, and Tags.
5. Tap Save Item.

Expected Result:

- The selected image is displayed in the image preview.
- The image is uploaded successfully to Firebase Storage.
- A new item document is created in Firebase Firestore under the user's items collection.
- The item contains the correct details and image URL.
- A success message is shown and the user returns to the previous screen.

Postconditions:

- A new clothing item is stored in the user's digital closet.

Scenario 2: Add Item Failure – Missing Required Fields

Preconditions:

- The user is logged in.
- The user is on the Add Item screen.

Steps:

1. Leave the Name field empty.
2. Enter a valid value in the Type field
   (or alternatively: fill Name and leave Type empty).
3. Tap Save Item.

Expected Result:

- An error message is displayed indicating that Name and Type are mandatory.
- No image is uploaded to Firebase Storage.
- No item document is created in Firebase Firestore.
- The user remains on the Add Item screen.

Postconditions:

- No changes are made to the database.

Scenario 3: Add Item Failure – Camera Capture Canceled

Preconditions:

- The user is logged in.
- The user is on the Add Item screen.
- The device has a camera application available.

Steps:

1. Tap Take Photo (Camera).
2. The camera application opens.
3. The user cancels the camera operation without taking a photo.
4. The application returns to the Add Item screen.

Expected Result:

- No image is displayed in the image preview.
- No image is uploaded to Firebase Storage.
- The user remains on the Add Item screen.
- The user can continue adding the item using another method (e.g., Gallery).

Postconditions:

- No image and no item are saved.

Part B – Unit Tests

Test 1: uploadImage returns URL from data source (with Mockito)

Purpose:
Verify that uploadImage() delegates to the data source and returns the URL

Test Steps:

1. A mock instance of FirebaseItemsDataSource is created using Mockito.
2. The mocked data source is configured to return a fake image URL.
3. uploadImage() is called on the repository.
4. The returned URL is asserted.
5. Mockito verifies that the correct data source method was invoked.

Expected Result: Returned URL equals mocked URL .

```
12 Usages
private val userId = "user123"

@Test
fun uploadImage_returnsUrlFromDataSource() = runTest {
    // Setup
    val dataSource = mock<FirebaseItemsDataSource>()
    val repo = ItemsRepository(dataSource)
    val fakeUri = mock<Uri>()
    val expectedUrl = "https://fake.storage/image.jpg"

    whenever( methodCall = dataSource.uploadItemImage(userId,  localUri = fakeUri)).thenReturn( value = expectedUrl)

    // Call
    val actualUrl = repo.uploadImage(userId, fakeUri)

    // Assertions
    assertEquals(expectedUrl, actualUrl)
    verify( mock = dataSource).uploadItemImage(userId,  localUri = fakeUri)
}
```

Explanation: This test verifies that uploadImage() delegates the upload operation to the data source and returns the same image URL provided by the mocked data source. Using a mock keeps the test isolated from Firebase Storage and focuses only on repository behavior.

Test 2: addItem returns item ID from data source

Purpose:
To verify that addItem() correctly returns the document ID generated by the data source.

Test Steps:

1. A mock data source is created.
2. A valid Item object is prepared.
3. The data source is configured to return a predefined document ID.
4. addItem() is invoked.
5. The returned ID is asserted and the interaction is verified.

Expected Result: Returned item ID equals the mocked document ID, and addItem(userId, item) is called exactly once on the data source.

```kotlin
@Test
fun addItem_returnsIdFromDataSource() = runTest {
    // Setup
    val dataSource = mock<FirebaseItemsDataSource>()
    val repo = ItemsRepository(dataSource)

    val item = Item(
        ownerUid = userId,
        closetId = "default",
        name = "Pink sweater",
        type = "shirt"
    )

    val expectedId = "doc_123"
    whenever( methodCall = dataSource.addItem(userId, item)).thenReturn( value = expectedId)

    // Call
    val actualId = repo.addItem(userId, item)

    // Assertions
    assertEquals(expectedId, actualId)
    verify( mock = dataSource).addItem(userId, item)
}
```

Explanation: This test verifies that addItem() delegates the call to the data source and returns the document ID generated by it. Using a mock keeps the test isolated from Firebase/Firestore and focuses only on repository behavior.

Test 3: getMyItems returns list of items from data source

Purpose:
To verify that getMyItems() returns the same list of items provided by the data source.

Test Steps:

1. A mock data source is created.
2. A predefined list of items is prepared.
3. The data source is configured to return this list.
4. getMyItems() is called.
5. The returned list is compared with the expected list.

Expected Result:

- The returned list matches the mocked list.
- The data source method is called once.

```
@Test
fun getMyItems_returnsItemsFromDataSource() = runTest {
    // Setup
    val dataSource = mock<FirebaseItemsDataSource>()
    val repo = ItemsRepository(dataSource)

    val expected = listOf(
        Item(id = "1", ownerUid = userId, closetId = "default", name = "Jeans", type = "pants"),
        Item(id = "2", ownerUid = userId, closetId = "default", name = "Jacket", type = "coat")
    )

    whenever( methodCall = dataSource.getItemsByOwner(userId)).thenReturn( value = expected)

    // Call
    val actual = repo.getMyItems(userId)

    // Assertions
    assertEquals(expected, actual)
    verify( mock = dataSource).getItemsByOwner(userId)
}
```

Explanation:This test verifies that the repository correctly retrieves and returns the user's items from the data source.

Part D – Code Review

Code Review Summary Document

שם הסודנט: מוחמד עמורי . תז:323936567.

Module Reviewed: CustomerMyProfileScreen.kt

Type: Jetpack Compose UI screen .
Project Reviewed: Workit

1. Short Architecture / Flow Description

This screen implements a user profile page with two main modes: View Mode and Edit Mode, and it loads/saves data through a repository.

Flow:

1. Load User Data :
    - On first composition (LaunchedEffect(Unit)), the screen retrieves the current user ID using FirebaseAuth.getInstance().currentUser?.uid.
    - If uid exists, it calls userRepository.getUserById(uid) and fills the UI state variables (fullName, gender, bio).
    - isLoading is set to false after the loading attempt.
2. Loading UI:
    - While isLoading == true, the screen displays a CircularProgressIndicator.
3. View Mode (Read-only):
    - Shows profile fields in a Card.
    - “Edit Profile” button switches to edit mode (isEditing = true).
4. Edit Mode:
    - Allows editing:
        - Full name (text field)
        - Gender (dropdown menu)
        - Bio (multi-line text field)
    - “Save changes” calls userRepository.saveCustomerProfile(...).
        - On success → exits edit mode (isEditing = false)
        - On error → prints stack trace

- “Cancel” discards edits (UI-wise) and exits edit mode.

2. 3 Key Comments

Key Comment 1 — Improve Separation of Concerns (Structure / Modularity)

Issue: The composable handles both UI rendering and data logic (FirebaseAuth uid retrieval, repository calls, loading state management).
Suggestion: Move data operations to a ViewModel (MVVM approach):

- ViewModel exposes uiState (loading/data/error)
- UI only observes state and sends user actions (edit/save/cancel)
  Why it helps: Better readability, maintainability, and significantly easier unit testing (logic can be tested without UI).

Key Comment 2 — Add Proper Error Handling + User Feedback (Bug Detection / UX)

Issue: Error cases are not user-friendly:

- If uid == null, the screen quietly stops loading without clear user guidance.
- If getUserById fails or returns null, there is no error message.
- Save errors only call printStackTrace() (no UI feedback).
  Suggestion:
- Add an errorMessage state and show a Snackbar/Toast on:
  - missing uid (user not logged in)
  - failed load
  - failed save
- Consider navigating back or redirecting to Login when uid is null.
  Why it helps: Prevents silent failures and improves usability and reliability.

Key Comment 3 — Improve Loading UI Organization (Readability)

Issue: The code uses return@Column inside the UI tree when loading, which works but reduces clarity and scalability.
Suggestion: Use a clearer structure:

- If loading → show a centered progress indicator (e.g., inside a Box)
- Else → render the screen content
  Why it helps: Cleaner code, easier future changes, fewer UI flow mistakes.

3) Example of Improvement (Change Based on Feedback)

Improvement Example: Show a User-Friendly Save Error

Before:
On save failure, the code runs:

- onError = { it.printStackTrace() }
  The user sees no message and doesn't know saving failed.

After (recommended change):

- Add saveErrorMessage state.
- In onError, update the message and show Snackbar/Toast:
  - "Failed to save profile. Please try again."
- Keep the user in Edit Mode so they don't lose their inputs.

Benefit: Better UX and clearer failure handling.

Final Summary: CustomerMyProfileScreen.kt has a good functional flow (load → view → edit → save/cancel) and is generally readable.
The main recommended improvements are:

1. Move data logic to ViewModel (better structure and testability)
2. Add real error handling with user feedback (better reliability)
3. Refactor loading UI flow for clearer readability

Part C : UI Test using Espresso

Test Explanation Side Menu

The test is on the UI of the app's side menu. The side menu pages change if the user is logged in or not, therefore there are before and after functions to ensure the user is not logged in to test the navigation to certain pages like Register and Sign In. in the before function we make the user log out and update right after the side menu drawer to change. In the after function we also make the user log out in case we tested before the pages required login.

To test if navigation works for each page we test if a view that belongs to the correct fragment is displayed. This test is not specific for each fragment and so we made a helper function that input the desired fragment and its view. In the helper function we mimic the user so we force open the side menu and click to the page, we also check to see if after clicking back we are in the home page again.

Test Explanation ItemRepository

The test is on the UI during the display item list process. We work with ItemRepository. The test checks that after login the app displays the item as defined with the correct strings. To test it in isolation we added an injection file that as a default will return to the viewmodel the real repository but in the test we can set the repository as a fake one. The result is to verify the correct display for the empty item list and the description for the existing item.

Code review - UniSmart 331659516

The project uses react native, even without knowledge about it, it is very well structured.
Include enough comments to help its readability without over-explaining.
Variable names are easy to understand, not ambiguous.
Code formatting is consistent.

React native doesn't separate UI and data like in android studio but there is separation within the files between the view and logic.

Each file is small and has one responsibility.
Screens handle UI and navigation.routes and flow are clear.
authentication for users is not inside screen code, handled in a different section.
Onboarding separated from authentication

Suggestion / improvement
Onboarding boundary clarification
add comment when start onboarding on files and function that leads to it.
Input validation
Add limit of character and length for password, and popup if invalid
Add adjustable hour slot to planner and include Saturday - can help the user plan

```
 1 package com.example.mycloset
 2
 3 import androidx.test.espresso.Espresso.onView
 4 import androidx.test.espresso.assertion.
   ViewAssertions.matches
 5 import androidx.test.espresso.contrib.DrawerActions
 6 import androidx.test.espresso.contrib.
   NavigationViewActions
 7 import androidx.test.espresso.matcher.ViewMatchers.
   isDisplayed
 8 import androidx.test.espresso.matcher.ViewMatchers.
   withId
 9 import androidx.test.ext.junit.rules.
   ActivityScenarioRule
10 import androidx.test.ext.junit.runners.AndroidJUnit4
11 import com.example.mycloset.ui.MainActivity
12 import org.junit.After
13 import org.junit.Rule
14 import org.junit.Test
15 import org.junit.runner.RunWith
16 import org.junit.Before
17
18 @RunWith(AndroidJUnit4::class)
19 class SIdeBarMenuNavigationTest {
20     @get:Rule
21     val activityRule = ActivityScenarioRule(
   MainActivity::class.java)
22
23     @Before
24     fun confirmedLogout() {
25         // Force Logout so that 'group_logged_out' is
    always visible
26         com.google.firebase.auth.FirebaseAuth.
   getInstance().signOut()
27         activityRule.scenario.onActivity { activity
    ->
28             // This forces the menu to update
   immediately for the test
29             (activity as MainActivity).
   updateDrawerMenuVisibility()
30         }
```

```
31      }
32      @After
33      fun cleanupLoggedUser(){
34          com.google.firebase.auth.FirebaseAuth.
    getInstance().signOut()
35          activityRule.scenario.onActivity { activity
     ->
36              // This forces the menu to update
    immediately for the test
37              (activity as MainActivity).
    updateDrawerMenuVisibility()
38          }
39      }
40      @Test
41      fun testRegisterNav(){
42          testNavigationFlow(R.id.nav_register, R.id.
    tvRegisterTitle)
43      }
44      @Test
45      fun testLoginNav(){
46          testNavigationFlow(R.id.nav_login, R.id.
    btnLogin)
47      }
48      @Test
49      fun testProfileNav(){
50          testNavigationFlow(R.id.nav_profile, R.id.
    tvProfileTitle)
51      }
52      @Test
53      fun testSettingNav(){
54          testNavigationFlow(R.id.nav_setting, R.id.
    tvSettingTitle)
55      }
56
57      private fun testNavigationFlow(menuFragmentId:
    Int, uniqueViewId: Int){
58          onView(withId(R.id.drawer_layout)).perform(
    DrawerActions.open())
59          onView(withId(R.id.nav_view)).perform(
    NavigationViewActions.navigateTo(menuFragmentId))
60
```

```
61          onView(withId(uniqueViewId)).check(matches(
    isDisplayed()))
62
63          activityRule.scenario.onActivity { it.
    onBackPressedDispatcher.onBackPressed() }
64
65          onView(withId(R.id.tvHomeTitle)).check(
    matches(isDisplayed()))
66      }
67  }
```

```
 1 package com.example.mycloset
 2
 3 import androidx.fragment.app.testing.
   launchFragmentInContainer
 4 import androidx.lifecycle.Lifecycle
 5 import androidx.test.espresso.Espresso.onView
 6 import androidx.test.espresso.assertion.
   ViewAssertions.matches
 7 import androidx.test.espresso.matcher.ViewMatchers
 8 import androidx.test.espresso.matcher.ViewMatchers.
   isDisplayed
 9 import androidx.test.espresso.matcher.ViewMatchers.
   withEffectiveVisibility
10 import androidx.test.espresso.matcher.ViewMatchers.
   withId
11 import androidx.test.espresso.matcher.ViewMatchers.
   withText
12 import androidx.test.espresso.matcher.ViewMatchers.
   Visibility
13 import androidx.test.ext.junit.rules.
   ActivityScenarioRule
14 import com.example.mycloset.data.model.Item
15 import com.example.mycloset.ui.MainActivity
16 import com.example.mycloset.ui.item.ItemListFragment
17 import com.example.mycloset.util.Injection
18 import org.junit.After
19 import org.junit.Before
20 import org.junit.Rule
21 import org.junit.Test
22
23 class ItemFuncUITest {
24     @get:Rule
25     val activityRule = ActivityScenarioRule(
   MainActivity::class.java)
26     private lateinit var fakeRepo :
   FakeItemRepository
27
28     @Before
29     fun setup(){
30         fakeRepo = FakeItemRepository()
31         fakeRepo.itemsToReturn = listOf(Item(name = "
```

```
31 Test Item", type = "Test type",
32             color = "Test", season = "Test"))
33         Injection.setRepository(fakeRepo)
34
35     }
36
37     @After
38     fun repoReset(){
39         Injection.reset()
40     }
41
42     @Test
43     fun testItemRendersInViewHolder(){
44         val scenario = launchFragmentInContainer<
   ItemListFragment>(
45             themeResId = R.style.Theme_MyCloset
46         )
47         scenario.moveToState(Lifecycle.State.RESUMED)
48         scenario.onFragment { fragment ->
49             fragment.viewModel.loadItems("test_user")
50         }
51         onView(withText("Test Item")).check(matches(
   isDisplayed()))
52         onView(withText("Test type • Test • Test")).
   check(matches(isDisplayed()))
53     }
54      @Test
55      fun testDisplayEmptyList(){
56          val fakeRepo = FakeItemRepository()
57          fakeRepo.itemsToReturn = emptyList()
58
59          val scenario = launchFragmentInContainer<
   ItemListFragment>(
60              themeResId = R.style.Theme_MyCloset
61          )
62          scenario.moveToState(Lifecycle.State.RESUMED
   )
63          Thread.sleep(1000)
64          onView(withId(R.id.tvEmpty))
65              .check(matches(withEffectiveVisibility(
   ViewMatchers.Visibility.VISIBLE)))
```

```
66          }
67
68  }
```