Kaunas technology university

Faculty of informatics

Project: “GymBuddy”

Software systems testing

LAB2 - Unit testing

Aistis Jakutonis IFF 3/1
Tautrimas Ramančionis IFF 3/1
Nojus Birmanas IFF 3/1
Juozas Balčikonis IFF 3/1

Students

Eligijus Kiudys

Lecturer

Kaunas, 2025

Content

Introduction

The GymBuddy project is a mobile application designed to help fitness enthusiasts manage and enhance their workout routines. It provides predefined training plans, custom workout creation, exercise guides, a workout calendar, and a gym partner-matching feature. The app is developed using Android Studio after an initial plan to use .NET MAUI was changed. Due to better platform stability, library support, and native Android capabilities, the development environment was switched from .NET MAUI to Android Studio early in the project.

To ensure the reliability and correctness of the application's database operations and UI logic, unit testing was performed. The main goals of unit testing in this project were:

- To catch bugs early during the development process.
- To guarantee the correctness of core features such as workout creation, scheduling, and data management.
- To improve overall code quality and maintainability.

The testing setup includes:

- **Robolectric –** To simulate the Android runtime for fast, emulator-free unit tests.
- **JUnit 4 –** The foundational framework used for structuring and running tests.
- **Mockito (with Kotlin extensions) –** For mocking dependencies such as DAOs in logic model tests.
- **Kotlin Coroutines Test –** To properly test coroutine-based asynchronous logic.
- **Room In-Memory Database –** Used in database-related tests to simulate real read/write operations without persistent storage.

All unit tests were written and managed within **Android Studio**, which provides a streamlined interface for running and analysing test results.

Currently, the project focuses on **unit testing** only, without additional integration or UI tests.

Test Overview

To ensure the integrity and reliability of the GymBuddy application, unit tests were developed for the most critical parts of the system, including database interactions, utility functions, and UI logic models. These tests play a vital role in identifying issues early, maintaining consistent functionality, and supporting future feature development.

The main areas of testing include:

- **Database operations** – verifying that reading from and writing to the database behaves as expected.
- **Data transformation** – ensuring string-to-list conversions and vice versa are reliable.
- **Utility logic** – testing custom extensions like conditional Boolean operations.
- **UI state management** – validating the logic that handles user inputs, workout planning, and reactive state updates.

All tests were executed in a fully isolated environment using in-memory databases and mocked dependencies. This approach avoids side effects, ensures speed, and provides consistent, repeatable results.

Tested Classes and Their Purposes

Class	Purpose
ConvertersTest	Verifies the conversion between comma-separated strings and list data structures.
ExerciseTest	Tests that exercise definitions and categories are correctly initialized and linked.
RecommendedWorkoutsTest	Confirms that all predefined workout routines are present and correctly configured.
SettingsDaoTest	Checks insert and update operations in the settings database.
WorkoutDaoTest	Ensures that workout data can be properly stored and updated in the database.
BooleanTest	Validates custom Kotlin Boolean extension functions like then and otherwise.
ExercisesModelTest	Tests the UI logic for selecting and filtering exercise categories.
WorkoutModelTest	Verifies workout creation logic, item management, and validation behavior.
SettingsModelTest	Confirms settings are loaded, modified, and observed correctly through the ViewModel.

Detailed Test Documentation

Database Tests

ConvertersTest

Test Purpose:
The ConvertersTest class ensures that conversion functions correctly transform data between a comma-separated String and a List<String>. These conversions are essential for storing complex data structures in a relational database format using Room.

Tested Methods:

- toItems(String?): Converts a comma-separated string into a list of strings.
- toString(List<String>): Converts a list of strings into a single comma-separated string.

Example Cases Tested:

- Converting "item1,item2,item3" to a list.
- Returning an empty list when input is empty or null.
- Converting a list back into a comma-separated string.
- Returning an empty string from an empty list.

Why It Matters:
Proper conversion ensures data consistency between UI models and database representation. Incorrect conversions could lead to broken workout item displays or save/load failures.

Test Result: All test cases passed successfully.

ConvertersTest.kt:

```
package org.aj.gymbuddy.db

import org.junit.Assert.assertEquals
import org.junit.Test

class ConvertersTest {

    private val converters = Converters()

    @Test
    fun `toItems should convert a comma-separated string to a list of strings`() {
        val input = "item1,item2,item3"
        val expected = listOf("item1", "item2", "item3")
        assertEquals(expected, converters.toItems(input))
    }

    @Test
    fun `toItems should return an empty list for an empty string`() {
        val input = ""
        val expected = emptyList<String>()
        assertEquals(expected, converters.toItems(input))
    }

    @Test
    fun `toItems should return an empty list for a null string`() {
        val input: String? = null
        val expected = emptyList<String>()
        assertEquals(expected, converters.toItems(input ?: ""))
    }

    @Test
    fun `toString should convert a list of strings to a comma-separated string`() {
        val input = listOf("item1", "item2", "item3")
        val expected = "item1,item2,item3"
        assertEquals(expected, converters.toString(input))
    }
```

```
    @Test
    fun `toString should return an empty string for an empty list`() {
        val input = emptyList<String>()
        val expected = ""
        assertEquals(expected, converters.toString(input))
    }
}
```

SettingsDaoTest

Test Purpose:
This test class verifies that the DAO (Data Access Object) for user settings correctly handles insert and update operations.

Tested Methods:

- upsert(SettingsEntity): Inserts a new setting or updates an existing one.
- selectAll(): Retrieves all stored settings.

Example Cases Tested:

- Inserting a new settings row.
- Updating an existing setting and confirming changes were saved using assertEquals.

Why It Matters:
The settings feature stores critical toggles such as "Enable recommended workouts". These must be saved and retrieved reliably to avoid misconfigured behavior or user frustration.

Test Result: All tests passed, confirming correct insert/update behavior.

SettingsDaoTest.kt

```
package org.aj.gymbuddy.db

import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import org.junit.After
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner
import java.util.UUID

@RunWith(RobolectricTestRunner::class)
class SettingsDaoTest {
    private lateinit var database: Database

    @Before
    fun setup() {
        database = Room.inMemoryDatabaseBuilder(
            ApplicationProvider.getApplicationContext(),
            Database::class.java
        ).allowMainThreadQueries().build()
    }

    @After
    fun tearDown() {
        database.close()
    }

    @Test
    fun `upsert inserts new row`() = runBlocking {
        val row = SettingsEntity(
            id = UUID.randomUUID(),
            name = "Test Settings",
            value = true
        )

        database.settings().upsert(row)

        assertEquals(1, database.settings().selectAll().first().size)
    }
```

```
    @Test
    fun `upsert updates old row`() = runBlocking {
        val old = SettingsEntity(
            id = UUID.randomUUID(),
            name = "Test Settings",
            value = true
        )

        database.settings().insert(old)

        val new = old.copy(
            name = "Updated Settings",
            value = false
        )

        database.settings().upsert(new)

        assertEquals(1, database.settings().selectAll().first().size)
        val result = database.settings().selectAll().first().first()

        assertEquals(old.id, result.id)
        assertEquals(new.name, result.name)
        assertEquals(new.value, result.value)
    }
}
```

WorkoutDaoTest

Test Purpose:
This test class validates that workout data is correctly managed by the Room database.
Tested Methods:

- insert(WorkoutEntity)
- upsert(WorkoutEntity)
- selectAll()

Example Cases Tested:

- Verifying a new workout gets inserted and is retrievable.
- Updating an existing workout by ID and ensuring its fields are correctly updated.
- Checking the workout count remains correct after update (ensuring no duplicates are created).

Why It Matters:
User-generated workout plans are a core feature of GymBuddy. Any failure in persisting or updating this data could directly impact user experience.
Test Result: All test cases passed successfully.

WorkoutDaoTest.kt

```
package org.aj.gymbuddy.db

import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import org.junit.After
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner
import java.util.UUID

@RunWith(RobolectricTestRunner::class)
class WorkoutDaoTest {

    private lateinit var database: Database

    @Before
    fun setup() {
        database = Room.inMemoryDatabaseBuilder(
            ApplicationProvider.getApplicationContext(),
            Database::class.java
        ).allowMainThreadQueries().build()
    }

    @After
    fun tearDown() {
        database.close()
    }

    @Test
    fun `upsert inserts new row`() = runBlocking {
        val row = WorkoutEntity(
            id = UUID.randomUUID(),
            name = "Test Workout",
            items = listOf("Item 1", "Item 2")
        )

        database.workouts().upsert(row)
```

```
        assertEquals(1, database.workouts().selectAll().first().size)
    }

    @Test
    fun `upsert updates old row`() = runBlocking {
        val old = WorkoutEntity(
            id = UUID.randomUUID(),
            name = "Test Workout",
            items = listOf("Item 1", "Item 2")
        )

        database.workouts().insert(old)

        val new = old.copy(
            name = "Updated Workout",
            items = listOf("Item 3", "Item 4")
        )

        database.workouts().upsert(new)

        assertEquals(1, database.workouts().selectAll().first().size)
        val result = database.workouts().selectAll().first().first()

        assertEquals(old.id, result.id)
        assertEquals(new.name, result.name)
        assertEquals(new.items, result.items)
    }
}
```

Exercise & Recommendation Logic Tests

ExerciseTest

Test Purpose:
The ExerciseTest class validates the structure and consistency of exercise categories and their entries. It ensures that each exercise is correctly linked to its category and contains complete metadata.
Tested Aspects:

- Correct collection of exercises under each category.
- Proper initialization of category and exercise properties (ID, title, description).
- Correct behavior of lazy category initialization.
- String representation of exercises returns the title.
- All exercise IDs are unique.

Example Cases Tested:

- Checking that each category contains the right exercises.
- Asserting that no exercise has a blank or missing title/description.
- Ensuring that lazy-loaded exercises match the static list of entries.
- Verifying that toString() on BenchPress returns "Bench press".

Why It Matters:
This ensures data integrity in the UI and database. Broken exercise references or duplicates could lead to incorrect display or crashes during workout selection.
Test Result: All tests passed, ensuring structural and content integrity.

ExerciseTest.kt

```kotlin
package org.aj.gymbuddy.db

import org.junit.Assert.assertEquals
import org.junit.Assert.assertTrue
import org.junit.Test
import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner

@RunWith(RobolectricTestRunner::class)
class ExerciseTest {
    @Test
    fun `test category collection`() {
        val warmup = ExerciseCategory.Exercise.entries
            .filter { it.category == ExerciseCategory.Warmup }

        assertTrue(warmup.containsAll(ExerciseCategory.Warmup.exercises))
    }

    @Test
    fun `test exercise category properties`() {
        ExerciseCategory.values().forEach { category ->
            assertTrue(category.id.toString().isNotEmpty())
            assertTrue(category.title.isNotBlank())
        }
    }

    @Test
    fun `test exercise properties`() {
        ExerciseCategory.Exercise.entries.forEach { exercise ->
            assertTrue(exercise.id.toString().isNotEmpty())
            assertTrue(exercise.title.isNotBlank())
            assertTrue(exercise.description.isNotBlank())
        }
    }
```

```
    @Test
    fun `test lazy initialization of exercises`() {
        ExerciseCategory.values().forEach { category ->
            assertEquals(
                ExerciseCategory.Exercise.entries.filter { it.category == category },
                category.exercises
            )
        }
    }

    @Test
    fun `test exercise toString returns title`() {
        val exercise = ExerciseCategory.Exercise.BenchPress
        assertEquals("Bench press", exercise.toString())
    }

    @Test
    fun `test unique exercise IDs`() {
        val allExercises = ExerciseCategory.Exercise.entries
        val uniqueIds = allExercises.map { it.id }.toSet()
        assertEquals(allExercises.size, uniqueIds.size)
    }
}
```

RecommendedWorkoutsTest

Test Purpose:
This test ensures that all predefined recommended workouts are properly defined and accessible via the RecommendedWorkouts enum or object.

Tested Aspects:

- That exactly 4 recommended workout plans are defined.
- That all expected workout types (PushAbsDay, PullHIITDay, LegsDay, ArmsAbsDay) are included.

Why It Matters:
Predefined workouts are shown to users as ready-made plans. Ensuring they are present prevents blank UI sections or logic failures when trying to access them.

Test Result: All assertions passed, confirming the expected recommended workouts are defined.

RecommendedWorkoutsTest.kt

```kotlin
package org.aj.gymbuddy.db

import org.junit.Assert.assertEquals
import org.junit.Assert.assertTrue
import org.junit.Test
import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner

@RunWith(RobolectricTestRunner::class)
class RecommendedWorkoutsTest {

    @Test
    fun `test all recommended workouts are defined`() {
        val allWorkouts = RecommendedWorkouts.values()
        assertEquals(4, allWorkouts.size)
        assertTrue(allWorkouts.contains(RecommendedWorkouts.PushAbsDay))
        assertTrue(allWorkouts.contains(RecommendedWorkouts.PullHIITDay))
        assertTrue(allWorkouts.contains(RecommendedWorkouts.LegsDay))
        assertTrue(allWorkouts.contains(RecommendedWorkouts.ArmsAbsDay))
    }
}
```

Utility Logic Tests

BooleanTest

Test Purpose:
The BooleanTest class verifies the behavior of custom Kotlin Boolean extensions: then {} and otherwise {}. These functions are syntactic utilities used to conditionally execute blocks of code based on a Boolean value, enhancing code clarity in view models and business logic.
Tested Extensions:

- Boolean.then { block }: Executes the block only if the value is true.
- Boolean.otherwise { block }: Executes the block only if the value is false.

Example Cases Tested:

- A block is executed when the value is true, and skipped when false.
- The functions return the original Boolean value (useful for chaining).
- otherwise {} works as an inverse of then {}.
- Both functions behave correctly even when not used for control flow chaining.

Why It Matters:
While these are small utilities, they are frequently used in the UI logic to write clean, readable conditions (e.g., when showing errors or toggling UI states). Ensuring their correctness helps prevent subtle bugs in state transitions.
Test Result: All utility functions behaved as expected in every test case.

BooleanTest.kt

```
package org.aj.gymbuddy.lang

import org.junit.Assert.assertFalse
import org.junit.Assert.assertTrue
import org.junit.Test

class BooleanTest {

    @Test
    fun `then executes block when true`() {
        var executed = false
        true.then { executed = true }
        assertTrue(executed)
    }

    @Test
    fun `then does not execute block when false`() {
        var executed = false
        false.then { executed = true }
        assertFalse(executed)
    }

    @Test
    fun `then returns the original boolean value when true`() {
        val result = true.then {}
        assertTrue(result)
    }

    @Test
    fun `then returns the original boolean value when false`() {
        val result = false.then {}
        assertFalse(result)
    }

    @Test
```

```
    fun `otherwise executes block when false`() {
        var executed = false
        false.otherwise { executed = true }
        assertTrue(executed)
    }

    @Test
    fun `otherwise does not execute block when true`() {
        var executed = false
        true.otherwise { executed = true }
        assertFalse(executed)
    }

    @Test
    fun `otherwise returns the original boolean value when false`() {
        val result = false.otherwise {}
        assertFalse(result)
    }

    @Test
    fun `otherwise returns the original boolean value when true`() {
        val result = true.otherwise {}
        assertTrue(result)
    }
}
```

UI State Model Tests

ExercisesModelTest

Test Purpose:
ExercisesModelTest validates the logic behind exercise selection, category expansion, filtering, and interaction modes. It ensures that the state updates consistently in both normal and selection modes.
Tested Behaviors:

- Expanding/collapsing categories in simple and selection modes.
- Lazy selection and clearing of exercises.
- Filter updates and corresponding state resets.
- Managing the expanded and selected exercise state lists.

Why It Matters:
A mismanaged state here could lead to exercises being shown/hidden incorrectly, filters not applying, or selection behavior breaking in the UI.
Test Result: All test cases passed, confirming consistent and predictable state behavior.

ExercisesModelTest.kt

```
package org.aj.gymbuddy.ui

import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import org.aj.gymbuddy.db.ExerciseCategory
import org.aj.gymbuddy.ui.screen.exercises.ExercisesModel
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner

@RunWith(RobolectricTestRunner::class)
class ExercisesModelTest {

    private lateinit var model: ExercisesModel

    @Before
    fun setup() {
        model = ExercisesModel()
    }

    @Test
    fun `expand adds category to expanded list in simple mode`() = runBlocking {
        val category = ExerciseCategory.Warmup

        model.expand(category)

        val state = model.state.first()
        assertEquals(listOf(category), state.expanded)
    }

    @Test
    fun `expand removes category from expanded list in simple mode`() = runBlocking {
        val category = ExerciseCategory.Warmup

        model.expand(category) // Add first

        model.expand(category) // Remove

        val state = model.state.first()
        assertEquals(emptyList<ExerciseCategory>(), state.expanded)
    }

    @Test
```

```
    fun `expand adds category to expanded list in selection mode`() = runBlocking {
        val category1 = ExerciseCategory.Warmup
        val category2 = ExerciseCategory.Back
        val exercise = category1.exercises.first()

        model.select(exercise) // Enter selection mode
        model.expand(category2)

        val state = model.state.first()
        assertEquals(listOf(category2), state.expanded)
    }

    @Test
    fun `expand removes category from expanded list in selection mode`() =
runBlocking {
        val category1 = ExerciseCategory.Warmup
        val category2 = ExerciseCategory.Back
        val exercise = category1.exercises.first()

        model.expand(category1) // Add first
        model.select(exercise) // Enter selection mode
        model.expand(category2) // Add
        model.expand(category2) // Remove

        val state = model.state.first()
        assertEquals(listOf(category1), state.expanded)
    }

    @Test
    fun `expand does not remove category from expanded list in selection mode`() =
runBlocking {
        val category1 = ExerciseCategory.Warmup
        val category2 = ExerciseCategory.Back
        val exercise1 = category1.exercises.first()
        val exercise2 = category2.exercises.first()

        model.expand(category1) // Add first
        model.select(exercise1) // Enter selection mode
        model.expand(category2) // Add
        model.select(exercise2) // Select another exercise
        model.expand(category2) // Not able to close

        val state = model.state.first()
        assertEquals(listOf(category1,category2), state.expanded)
    }

    @Test
    fun `setFilter updates filter and clears expanded and selected lists`() =
runBlocking {
        val category = ExerciseCategory.Warmup
        val exercise = category.exercises.first()

        model.select(exercise)
        model.expand(category)
        model.setFilter("New Filter")

        val state = model.state.first()
        assertEquals("New Filter", state.filter)
        assertEquals(emptyList<ExerciseCategory>(), state.expanded)
        assertEquals(emptyList<ExerciseCategory.Exercise>(), state.selected)
    }

    @Test
    fun `select adds exercise to selected list`() = runBlocking {
        val category = ExerciseCategory.Warmup
        val exercise = category.exercises.first()

        model.select(exercise)
```

```
        val state = model.state.first()
        assertEquals(listOf(exercise), state.selected)
    }

    @Test
    fun `select removes exercise from selected list`() = runBlocking {
        val category = ExerciseCategory.Warmup
        val exercise = category.exercises.first()

        model.select(exercise) // Add
        model.select(exercise) // Remove

        val state = model.state.first()
        assertEquals(emptyList<ExerciseCategory.Exercise>(), state.selected)
    }

    @Test
    fun `clearSelection clears selected list and updates expanded list`() =
runBlocking {
        val category = ExerciseCategory.Warmup
        val exercise = category.exercises.first()

        model.select(exercise)
        model.clearSelection()

        val state = model.state.first()
        assertEquals(emptyList<ExerciseCategory.Exercise>(), state.selected)
        assertEquals(listOf(category), state.expanded)
    }
}
```

WorkoutModelTest

Test Purpose:
This class tests the ViewModel used for creating and managing individual workout plans. It handles field validation, list manipulation, and communication with the DAO.
Tested Behaviors:

- Validating workout name uniqueness and non-emptiness.
- Adding/removing/editing individual workout items.
- Clearing item input states.
- Handling duplicate items and whitespace trimming.
- Applying selections from exercise pickers.
- Submitting the workout to the database.

Why It Matters:
The workout creation flow is central to user engagement. Any validation issue or logic error could lead to saved workouts being incorrect or the UI rejecting valid input.
Test Result: All logic executed correctly across a wide range of inputs.

WorkoutModelTest.kt

```
package org.aj.gymbuddy.ui

import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.test.runTest
import org.aj.gymbuddy.db.ExerciseCategory
import org.aj.gymbuddy.db.WorkoutDao
import org.aj.gymbuddy.ui.screen.workout.WorkoutModel
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.kotlin.mock
import org.mockito.kotlin.whenever
import org.robolectric.RobolectricTestRunner
import org.mockito.kotlin.any
import org.mockito.kotlin.verify

@RunWith(RobolectricTestRunner::class)
class WorkoutModelTest {

    private lateinit var dao: WorkoutDao
    private lateinit var model: WorkoutModel

    @Before
    fun setup() {
        dao = mock()
        model = WorkoutModel(dao)
    }

    @Test
    fun `save with blank name sets nameError to true`() = runTest {
        model.setName("")
        model.save()

        val state = model.state.first()
        assertEquals(true, state.nameError)
    }

    @Test
    fun `save with existing name sets nameError to true`() = runTest {
        whenever(dao.existsByName("Duplicate Workout")).thenReturn(true)

        model.setName("Duplicate Workout")
```

```
        model.save()

        val state = model.state.first()
        assertEquals(true, state.nameError)
    }

    @Test
    fun `save with valid name saves workout and resets state`() = runBlocking {
        whenever(dao.existsByName(any())).thenReturn(false)

        model.setName("New Workout")
        model.save()

        val state = model.state.first()
        assertEquals("", state.workout.name)
        verify(dao).upsert(any())
    }

    @Test
    fun `validateName sets nameError to true for blank name`() = runTest {
        model.setName("")
        model.validateName()

        val state = model.state.first()
        assertEquals(true, state.nameError)
    }

    @Test
    fun `setName updates workout name and clears nameError`() = runTest {
        model.setName("Workout Name")

        val state = model.state.first()
        assertEquals("Workout Name", state.workout.name)
        assertEquals(false, state.nameError)
    }

    @Test
    fun `setItem updates item and clears itemError`() = runTest {
        model.setItem("New Item")

        val state = model.state.first()
        assertEquals("New Item", state.item)
        assertEquals(false, state.itemError)
    }

    @Test
    fun `validateItem sets itemError to true if item exists in workout`() = runTest {
        model.setItem("Duplicate Item")
        model.addItem()
        model.setItem("Duplicate Item")

        val isValid = model.validateItem()
        val state = model.state.first()

        assertEquals(true, state.itemError)
        assertEquals(true, isValid)
    }

    @Test
    fun `validateItem returns false for unique item`() = runTest {
        model.setItem("Unique Item")

        val isValid = model.validateItem()
        val state = model.state.first()

        assertEquals(false, state.itemError)
        assertEquals(false, isValid)
    }
```

```
    @Test
    fun `resetItem clears item and itemError`() = runTest {
        model.setItem("Item")
        model.resetItem()

        val state = model.state.first()
        assertEquals("", state.item)
        assertEquals(false, state.itemError)
    }

    @Test
    fun `addItem adds item to workout and clears item state`() = runTest {
        model.setItem("New Item")
        model.addItem()

        val state = model.state.first()
        assertEquals(listOf("New Item"), state.workout.items)
        assertEquals("", state.item)
        assertEquals(false, state.itemError)
    }

    @Test
    fun `addItem trims whitespace from item before adding to workout`() = runTest {
        model.setItem("  Trimmed Item  ")
        model.addItem()

        val state = model.state.first()
        assertEquals(listOf("Trimmed Item"), state.workout.items)
        assertEquals("", state.item)
        assertEquals(false, state.itemError)
    }

    @Test
    fun `deleteItem removes item from workout`() = runTest {
        model.setItem("Item to Delete")
        model.addItem()
        model.deleteItem("Item to Delete")

        val state = model.state.first()
        assertEquals(emptyList<String>(), state.workout.items)
    }

    @Test
    fun `editItem sets item for editing and removes it from workout`() = runTest {
        model.setItem("Item to Edit")
        model.addItem()
        model.editItem("Item to Edit")

        val state = model.state.first()
        assertEquals("Item to Edit", state.item)
        assertEquals(emptyList<String>(), state.workout.items)
    }

    @Test
    fun `editItem does nothing if item field is already populated`() = runTest {
        model.setItem("Existing Item")
        model.addItem()
        model.setItem("New Item")

        model.editItem("Existing Item")

        val state = model.state.first()
        assertEquals(listOf("Existing Item"), state.workout.items)
        assertEquals("New Item", state.item)
    }

    @Test
```

```
    fun `setSelected updates workout with selected exercises`() = runTest {
        val exercises = listOf(
            ExerciseCategory.Exercise.Rowing,
            ExerciseCategory.Exercise.PullUps
        )

        model.setSelected(exercises)

        val state = model.state.first()
        assertEquals(listOf("Rowing", "PullUps"), state.workout.items)
    }

    @Test
    fun `setSelected clears workout items when given an empty list`() = runTest {
        val exercises = emptyList<ExerciseCategory.Exercise>()

        model.setSelected(exercises)

        val state = model.state.first()
        assertEquals(emptyList<String>(), state.workout.items)
    }
}
```

WorkoutsModelTest

Test Purpose:
WorkoutsModelTest verifies the logic for displaying, inserting, and removing multiple workout plans. It tests how the list of saved workouts is managed and retrieved from the DAO.

Tested Behaviors:

- Fetching an empty or populated list of workouts on initialization.
- Inserting and deleting workouts from the stored list.
- Selecting a workout plan by its ID.

Why It Matters:
This model powers the screen where users view and manage all their plans. Ensuring this list is synced with the database is key for reliable access and deletion.

Test Result: All DAO interactions and state updates worked as intended.

WorkoutsModelTest.kt

```
package org.aj.gymbuddy.ui

import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.test.runTest
import org.aj.gymbuddy.db.WorkoutDao
import org.aj.gymbuddy.db.WorkoutEntity
import org.aj.gymbuddy.ui.screen.workouts.WorkoutsModel
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.kotlin.*
import org.robolectric.RobolectricTestRunner
import java.util.UUID

@RunWith(RobolectricTestRunner::class)
class WorkoutsModelTest {

    private lateinit var dao: WorkoutDao
    private lateinit var model: WorkoutsModel

    @Before
    fun setup() {
        dao = mock()
        whenever(dao.selectAll()).thenReturn(flowOf(emptyList()))
        model = WorkoutsModel(dao)
    }

    @Test
    fun `init sets empty workout list from dao`() = runTest {
        val state = model.state.first()
        assertEquals(emptyList<WorkoutEntity>(), state.workouts)
    }

    @Test
    fun `init sets workout list from dao`() = runTest {
        val workouts = listOf(
            WorkoutEntity(name = "Workout A", items = listOf("Pushups")),
            WorkoutEntity(name = "Workout B", items = listOf("Pullups"))
        )
        whenever(dao.selectAll()).thenReturn(flowOf(workouts))
        model = WorkoutsModel(dao)

        val state = model.state.first()
        assertEquals(workouts, state.workouts)
    }
    @Test
    fun `insertWorkoutPlan adds workout`() = runTest {
```

```
        val workout = WorkoutEntity(name = "Workout X", items = listOf("Squats"))
        model.insertWorkoutPlan(workout)
        verify(dao).insert(workout)
    }

    @Test
    fun `deleteWorkoutPlan removes workout`() = runTest {
        val workout = WorkoutEntity(name = "Workout Y", items = listOf("Lunges"))
        model.deleteWorkoutPlan(workout)
        verify(dao).delete(workout)
    }

    @Test
    fun `selectWorkoutPlan calls DAO with correct ID`() = runTest {
        val id = UUID.randomUUID()
        model.selectWorkoutPlan(id)
        verify(dao).select(id)
    }

}
```

SettingsModelTest

Test Purpose:
This test class ensures that the settings ViewModel correctly observes and modifies user preferences. It tests reactive state updates and interaction with the SettingsDao.

Tested Behaviors:

- Initialization with default settings when the database is empty.
- Observing settings changes through Flow.
- Inserting, updating, and deleting settings via DAO calls.
- Fetching setting values or names by ID, including error handling.

Why It Matters:
Settings impact the app's behavior globally. Missing or incorrect settings could disable core features like recommended workouts or user preferences.
Test Result: All state changes and error cases were handled as expected.

SettingsModelTest.kt

```
package org.aj.gymbuddy.ui.screen.settings

import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.test.advanceUntilIdle
import kotlinx.coroutines.test.runTest
import org.aj.gymbuddy.db.SettingsConstants
import org.aj.gymbuddy.db.SettingsDao
import org.aj.gymbuddy.db.SettingsEntity
import org.junit.Assert.assertEquals
import org.junit.Assert.assertNull
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.kotlin.*
import org.robolectric.RobolectricTestRunner
import java.util.UUID

@OptIn(ExperimentalCoroutinesApi::class)
@RunWith(RobolectricTestRunner::class)
class SettingsModelTest {

    private lateinit var mockDao: SettingsDao

    private lateinit var settingsModel: SettingsModel

    @Before
    fun setup() {
        mockDao = mock()

        whenever(mockDao.selectAll()).thenReturn(
            flowOf(emptyList(), listOf(
                SettingsEntity(id = SettingsConstants.ENABLE_RECOMMENDED_WORKOUTS,
name = "Enable recommended workouts", value = true),
                SettingsEntity(id = SettingsConstants.ENABLE_TEST, name = "Enable
setting1", value = true),
                SettingsEntity(id = SettingsConstants.ENABLE_TEST2, "Enable
setting2", value = false)
            ))
        )
    }

    @Test
    fun `init inserts default settings if database is empty`() = runTest {
        settingsModel = SettingsModel(mockDao)
```

```
        advanceUntilIdle()

        verify(mockDao).upsert(SettingsEntity(id =
SettingsConstants.ENABLE_RECOMMENDED_WORKOUTS, name = "Enable recommended workouts",
value = true))
        verify(mockDao).upsert(SettingsEntity(id = SettingsConstants.ENABLE_TEST,
name = "Enable setting1", value = true))
        verify(mockDao).upsert(SettingsEntity(id = SettingsConstants.ENABLE_TEST2,
"Enable setting2", value = false))

        verify(mockDao, atLeastOnce()).selectAll()
    }

    @Test
    fun `init observes settings and updates state`() = runTest {
        whenever(mockDao.selectAll()).thenReturn(
            flowOf(listOf(
                SettingsEntity(id = UUID.randomUUID(), name = "Setting 1", value =
true),
                SettingsEntity(id = UUID.randomUUID(), name = "Setting 2", value =
false)
            ))
        )

        settingsModel = SettingsModel(mockDao)

        advanceUntilIdle()

        val state = settingsModel.state.value

        assertEquals(2, state.settings.size)
        assertEquals("Setting 1", state.settings.first().name)
        assertEquals(true, state.settings.first().value)
        assertEquals("Setting 2", state.settings.last().name)
        assertEquals(false, state.settings.last().value)

        verify(mockDao, atLeastOnce()).selectAll()
    }


    @Test
    fun `insertSetting calls dao insert`() = runTest {
        settingsModel = SettingsModel(mockDao)

        val settingToInsert = SettingsEntity(id = UUID.randomUUID(), name = "New
Setting", value = true)

        settingsModel.insertSetting(settingToInsert)

        verify(mockDao).insert(settingToInsert)
    }

    @Test
    fun `upsertSetting calls dao upsert`() = runTest {
        settingsModel = SettingsModel(mockDao)

        val settingToUpsert = SettingsEntity(id = UUID.randomUUID(), name = "Upserted
Setting", value = false)

        settingsModel.upsertSetting(settingToUpsert)

        verify(mockDao).upsert(settingToUpsert)
    }

    @Test
    fun `deleteSetting calls dao delete`() = runTest {
        settingsModel = SettingsModel(mockDao)
```

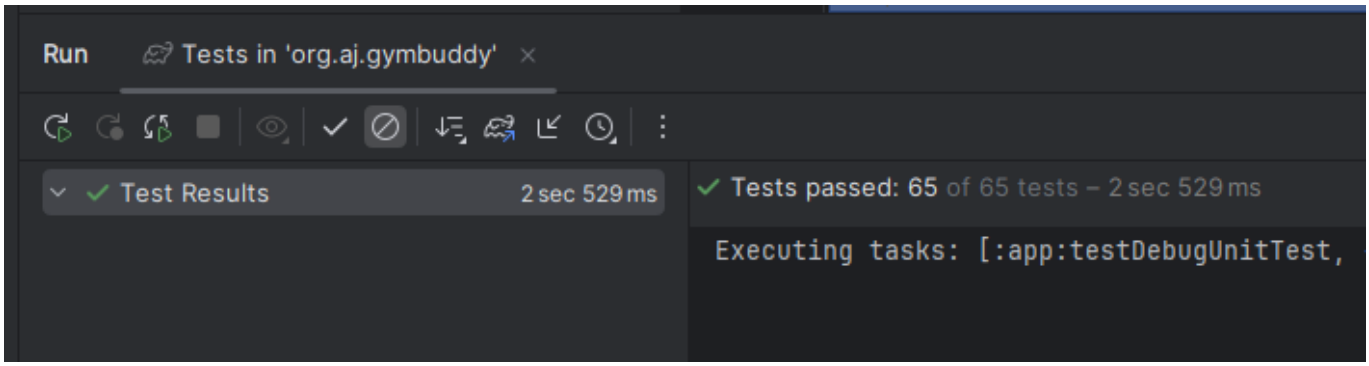
```
        val settingToDelete = SettingsEntity(id = UUID.randomUUID(), name = "Setting
to Delete", value = true)

        settingsModel.deleteSetting(settingToDelete)

        verify(mockDao).delete(settingToDelete)
    }

    @Test
    fun `getSettingValueById returns value when setting exists`() = runTest {
        settingsModel = SettingsModel(mockDao)

        val settingId = UUID.randomUUID()
        val settingEntity = SettingsEntity(id = settingId, name = "My Setting", value
= true)

        whenever(mockDao.select(settingId)).thenReturn(flowOf(settingEntity))

        val value = settingsModel.getSettingValueById(settingId)

        assertEquals(true, value)
        verify(mockDao).select(settingId)
    }

    @Test
    fun `getSettingValueById returns null when setting does not exist`() = runTest {
        settingsModel = SettingsModel(mockDao)

        val settingId = UUID.randomUUID()

        whenever(mockDao.select(settingId)).thenReturn(flowOf()) // Empty flow

        val value = settingsModel.getSettingValueById(settingId)

        assertNull(value)
        verify(mockDao).select(settingId)
    }

    @Test
    fun `getSettingValueById returns null when dao select throws exception`() =
runTest {
        settingsModel = SettingsModel(mockDao)

        val settingId = UUID.randomUUID()

        whenever(mockDao.select(settingId)).thenThrow(RuntimeException("DB Error"))

        val value = settingsModel.getSettingValueById(settingId)

        assertNull(value)
        verify(mockDao).select(settingId)
    }

    @Test
    fun `getNameValueById returns name when setting exists`() = runTest {
        settingsModel = SettingsModel(mockDao)

        val settingId = UUID.randomUUID()
        val settingEntity = SettingsEntity(id = settingId, name = "My Setting Name",
value = true)

        whenever(mockDao.select(settingId)).thenReturn(flowOf(settingEntity))

        val name = settingsModel.getNameValueById(settingId)

        assertEquals("My Setting Name", name)
        verify(mockDao).select(settingId)
    }
```

```
    @Test
    fun `getNameValueById returns null when setting does not exist`() = runTest {
        settingsModel = SettingsModel(mockDao)

        val settingId = UUID.randomUUID()

        whenever(mockDao.select(settingId)).thenReturn(flowOf()) // Empty flow

        val name = settingsModel.getNameValueById(settingId)

        assertNull(name)
        verify(mockDao).select(settingId)
    }

    @Test
    fun `getNameValueById returns null when dao select throws exception`() = runTest
{
        settingsModel = SettingsModel(mockDao)

        val settingId = UUID.randomUUID()

        whenever(mockDao.select(settingId)).thenThrow(RuntimeException("DB Error"))

        val name = settingsModel.getNameValueById(settingId)

        assertNull(name)
        verify(mockDao).select(settingId)
    }
}
```

Test Execution and Results

All unit tests in the GymBuddy project were executed using **Android Studio's built-in test runner**. The test suite includes logic tests, data conversion, model validation, and reactive state handling. Tests were grouped and run across multiple packages, including *db*, *lang*, and *ui*.

Tests were triggered using the **Gradle task** *:app:testDebugUnitTest* within the IDE, which executed all test classes under the *org.aj.gymbuddy* package.



Result Summary:
Total tests executed: 65
Tests passed: 65
Failures: 0
Execution time: ~2.5 seconds

Conclusion

The unit testing phase of the GymBuddy project played a critical role in verifying the correctness, stability, and maintainability of the application's core logic. By covering database operations, utility functions, and reactive UI models, the test suite ensured that both backend processes and user-facing logic perform reliably under expected conditions.
All 65 unit tests passed successfully, confirming the robustness of implemented features such as:

- Workout creation and management.
- Settings storage and retrieval.
- Exercise selection and filtering.
- Custom utility logic.

The test-driven approach adopted in this phase allows the team to move forward with greater confidence as new features are added or existing ones are refined. While the current test suite focuses on unit testing, future work may involve:

- **Integration testing** (e.g., Firebase interactions).
- **UI testing** (e.g., Espresso for click flows and navigation).
- **Code coverage analysis** to ensure completeness.

In summary, GymBuddy's unit testing effort has established a solid foundation for ongoing development and long-term project reliability.