## Guided Learning (QLC‑5): Async in Action

### Objective:

Extend the existing MoodSelectorScreen ViewModel (now wired for testable dependencies from QLC4) to load its mood list asynchronously. Drive the implementation using TDD’s red-green-refactor cycle, control virtual time in tests with runTest and advanceUntilIdle, and refactor for dispatcher injection.

### Scenario :

Starting from your retrofitted QLC4 setup, where MoodSelectorScreen accepts a MoodDataSource fake, uses testTags, and has basic selection tests. You need to change the ViewModel so that:

1. It fetches getMoods() asynchronously (simulating a network or database call).
2. Exposes a StateFlow<UiState> with Loading, Success, and Error states.
3. Emits those states in the correct order when loadMoods() is called.

You’ll write failing tests first, control coroutine timing, then implement just enough code to make them pass.

### Step 1 — Define the Feature Behavior and Requirements

In QLC‑5, we are extending the Mood Tag Selector feature to support asynchronous loading of moods into the UI, reflecting a more realistic network/database fetch.  
 The focus now is on driving the asynchronous loading logic using TDD, so that the ViewModel exposes a clear loading, success, and error state for the UI.

This feature must:

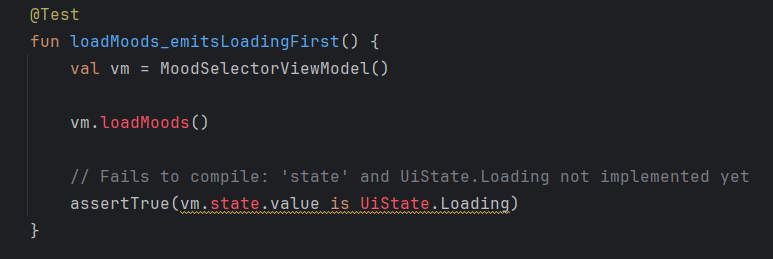
* **Fetch the mood list asynchronously** (as if from a network/database), triggered by calling loadMoods().
* **Expose the UI state** via a StateFlow<UiState>, where UiState is a sealed class with Loading, Success, and Error states.
* **Emit Loading immediately** when loading starts.
* **Emit Success** with data if fetching succeeds.
* **Emit Error** with a message if fetching fails.
* **Be testable and controllable in unit tests** (by injecting a coroutine dispatcher and a fake data source).

### Step 2 — Define the Behavior with a Failing Unit Test (RED)

Start with the simplest case: verify that when loadMoods() is called, the state immediately becomes Loading.

**Expected Behavior:** When the ViewModel’s loadMoods() is called, it should immediately emit UiState.Loading, even if the fetch is still in progress.

**Test Example:**



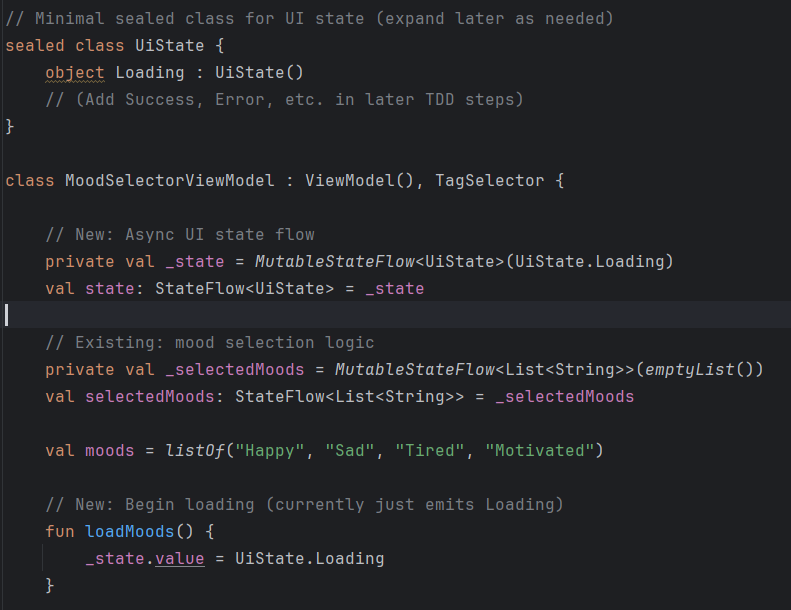
### Step 3 — Make the Test Pass (GREEN)

Now, update the ViewModel so the test passes.

**Behavior to Implement:** Set the state to UiState.Loading at the start of loadMoods().

#### Action:

* **Start off basic:** We need the following minimal code to make the test pass:  
  + The UiState sealed class (with just Loading for now)
  + The state property and backing MutableStateFlow
  + The loadMoods() method that sets the state to Loading



* **Run your test again.** The test should now pass, as the ViewModel emits Loading when loadMoods() is called.

#### What's next?

Now that your first test is green, you're ready to:

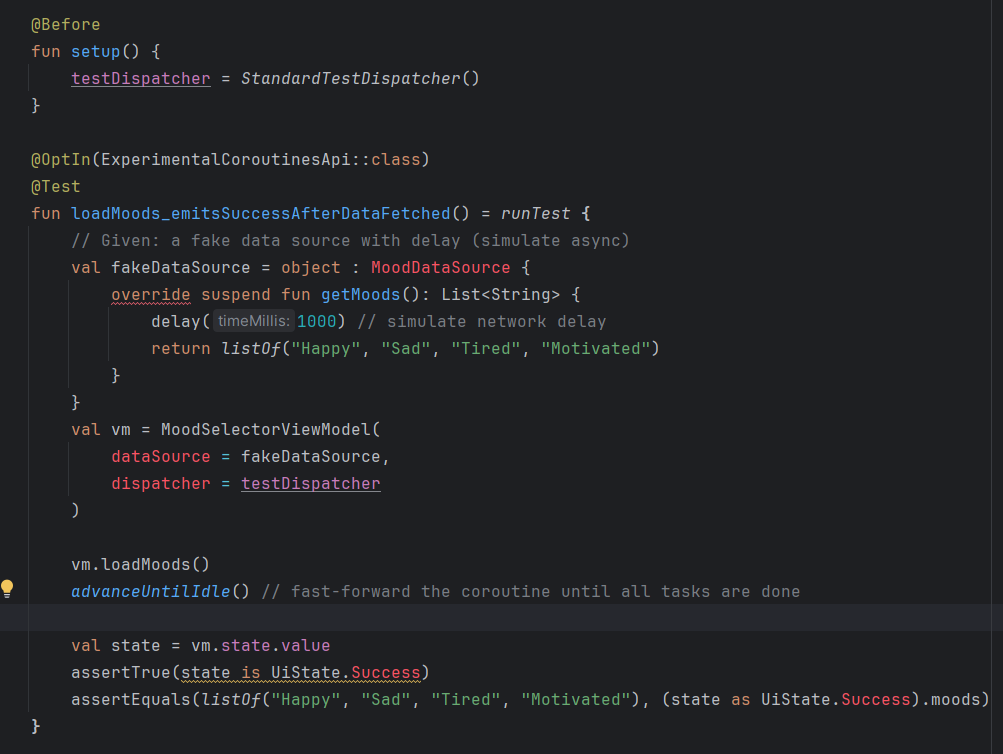
* Write the **next failing test** (RED) for when loading succeeds and you want to emit a UiState.Success containing the loaded moods.
* Continue TDD by adding just enough code to pass that test, and so on.

### Step 4 — Add a New Failing Test for Success (RED)

#### Define the next behavior:

When loading completes successfully (after an async fetch), the ViewModel’s state should become UiState.Success with the loaded moods.

#### Expected Behavior:

* Call loadMoods()
* Wait for the async fetch to complete
* state.value should be UiState.Success(moods

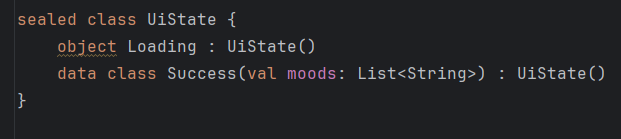
## Step 5 — Make the Success Test Pass (GREEN)

**Goal:**

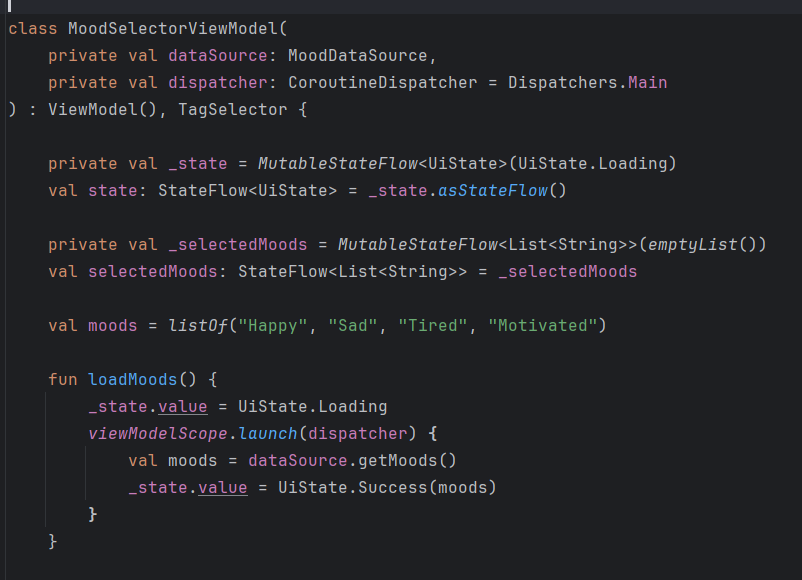
Update your MoodSelectorViewModel so that after loading (asynchronously), it emits UiState.Success with the correct list of moods.

**Refactor the ViewModel for async and testability**

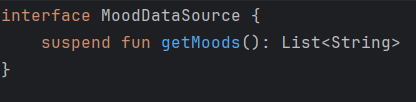
**1. Update your UiState to add Success:**



**2. Change your ViewModel to support async and allow test dispatcher injection:**

****

**3. Implement a DataSource:**

****

**The constructor now accepts a MoodDataSource and a CoroutineDispatcher (with a default of Dispatchers.Main for production). loadMoods() uses a coroutine to fetch asynchronously and then updates the state to UiState.Success.**

**4. Now, re-run your test:**

* **Your test should now pass for both the loading and success behaviors!**

**You now have:**

* **A testable, async ViewModel that emits Loading and, after an async fetch, emits Success with the data.**
* **Tests that can inject a fake data source and test dispatcher for deterministic, instant unit tests.**

**QLC-5.1: Error Handling & Refactor**

### Objective

Extend your MoodSelectorViewModel and tests to support error states. Your goal: write and pass a failing test for error handling, then refactor your implementation to ensure robust, testable async code.

### 

### 

### Task: Add Error Case & Refactor

#### Why?

* **Completeness:** Robust UIs handle error states, not just happy paths.
* **Testability:** Refactoring for async and error handling ensures that your ViewModel can be tested for both success and failure, without real network or database calls.
* **Separation of Concerns:** Keeping error-handling logic in the ViewModel (not the UI) means tests stay fast, focused, and deterministic.

#### Your Steps

1. **RED: Write a new failing test** Write a unit test (in the same style as before) that simulates a data fetch failure by having your MoodDataSource throw an exception.  
   * Call loadMoods()
   * Advance the test dispatcher (advanceUntilIdle())
   * Assert that the ViewModel emits UiState.Error with the correct error message.
2. **GREEN: Make it pass**
   * Refactor your MoodSelectorViewModel to catch exceptions in the loadMoods() coroutine and emit UiState.Error(message).
3. **REFACTOR: Review and tidy up**
   * Ensure your ViewModel's constructor supports dependency injection for dispatcher and data source.
   * Clean up duplicated setup in your tests.
   * Confirm that all tests (loading, success, error) remain green.

Answers a simple solution can be found here - <https://github.com/Edrzapi/Android_tdd_tag_selector/tree/QLC-5.1-ANS>