## 

## Guided Learning (QLC-4): Introduction to Stubs and Dependency Injection

### 

### Objective:

Decouple your TaskListViewModel from hardcoded data by introducing a repository abstraction.  
You’ll define a TaskRepositoryProtocol, write a FakeRepository stub, and inject it into your TaskListViewModel, all driven by failing tests first.

This promotes:

* Separation of concerns
* Easier unit testing with predictable data
* Cleaner architecture that matches production patterns

### Step 1 — Define the Problem and Requirements

We need to move data loading out of TaskListViewModel into a **repository**, so:

* The ViewModel focuses on presentation logic only
* Data access is handled by a TaskRepositoryProtocol
* Tests can inject a stub (FakeRepository) for controlled test data

This is critical in SwiftUI apps where testability is often impacted by tightly coupled view models.

### 

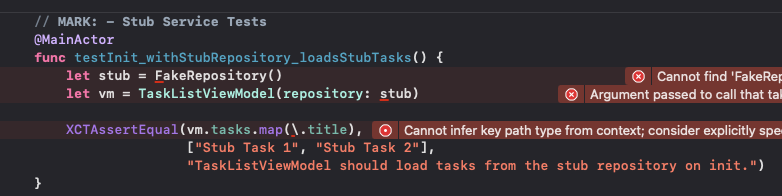
### 

### 

### 

### Step 2 — Write the Initial Test (RED)

In TaskListViewModelTests.swift, write a test like this:

****

## Independent Task (QLC‑4.1): Getting the GREEN light with the Repository Protocol.

**Objective:**Ensure that TaskListViewModel no longer manages its own data directly but instead relies on an injected repository conforming to a protocol.

**Steps:**

1. **Write a Unit Test for Dependency Injection:**
   * In **TaskListViewModelTests.swift**, write a test that initializes the view model with a StubRepository.
   * Verify that vm.tasks contains the expected stubbed data after initialization.
2. **Run the Test and Confirm It Fails:**
   * The test should fail because TaskListViewModel does not yet support dependency injection.
3. **Create the Repository Protocol:**
   * Inside **repositories/RepositoryProtocol.swift**, define a protocol with the required methods for fetching tasks (e.g., fetchTasks()).
4. **Implement the StubRepository:**
   * Create **FakeRepository.swift** inside the **repositories/** folder.
   * Make it conform to the protocol.
   * Ensure it returns a predictable set of tasks.
5. **Refactor TaskListViewModel:**
   * Add a repository property.
   * Initialize it with the repository via constructor injection.
   * Update the tasks array to be populated by the repository.
6. **Run the Unit Test and Confirm It Passes:**
   * Your test should now pass with the stub in place.

Check <https://github.com/Edrzapi/IOSTDD/tree/QLC-4.1-ANS> for the solution

In Test-Driven Development, we want our tests to verify the logic of the unit under test in this case, the TaskListViewModel — without being affected by external factors like databases, APIs, or network calls.

A Stub Repository is a simple implementation of a protocol that provides fixed, predictable responses. It does not perform any real logic but instead simulates the behavior of a real service in a controlled way.

By using a stub repository:

* You can control exactly what data your view model receives.
* Your tests become deterministic, they will always behave the same way, no surprises.
* You avoid unintended side effects, making your tests more reliable and faster.
* You reinforce the habit of programming to an abstraction (the protocol), not a concrete implementation.

This pattern ensures your TaskListViewModel stays focused on its business logic while all data handling remains delegated. A key benefit of Dependency Injection.

## Guided Learning (QLC‑4.2): Refactoring with Dependency Injection and Stubbed Repositories

**Objective:**Ensure that all unit tests now work with an injected repository, keeping tests isolated from hardcoded data and supporting true dependency injection.

### Step 1 — Define the Problem and Requirements

We need to refactor TaskListViewModel to delegate all data operations to a **repository**, so that:

* The **ViewModel focuses purely on presentation logic** — its only responsibility is to provide data to the view and respond to user interactions.
* **Data access and storage are handled by RepositoryProtocol**, allowing us to easily swap between real data sources, stubs, or fakes depending on context.
* **Unit tests can inject a controlled EmptyStubRepository**, giving us isolated, predictable test results, without relying on hardcoded data or real storage.

**Why this matters:**

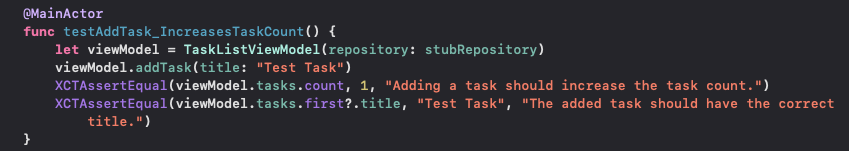
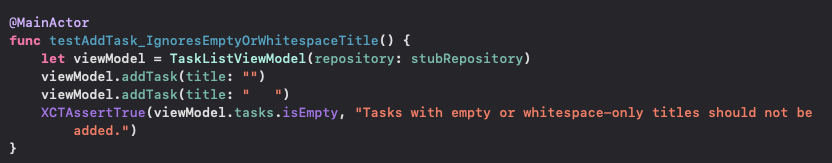
In SwiftUI apps, tightly coupled view models can quickly become hard to test and maintain.By pushing data access into a repository layer, we gain:

* **Improved testability** — no more hardcoded data in the ViewModel.
* **Flexibility for future features** — like persisting data, syncing with APIs, or introducing caching layers.

### Step 2 — Write the Unit Tests First (RED)

Write unit tests that verify the TaskListViewModel works with an injected RepositoryProtocol.  
Your tests should **focus on ViewModel logic only**, assuming the repository works correctly.

**Tests to write in TaskListViewModelTests.swift:**

* testTasks\_AreEmptyOnInit() — The ViewModel should start with the tasks provided by the repository (which is empty in this case).****
* testAddTask\_IncreasesTaskCount() — Adding a task should increase the count in the ViewModel’s tasksarray.****
* testRemoveTask\_DecreasesTaskCount() — Removing a task by ID should reduce the task count.****
* testAddTask\_IgnoresEmptyOrWhitespaceTitle() — Adding empty or whitespace-only titles should not create a task.
* 

These tests will **fail initially** because the TaskListViewModel does not yet delegate to the repository.

### 

### 

### 

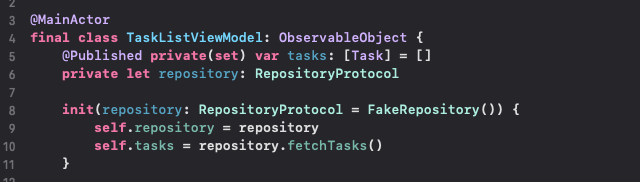
### 

### 

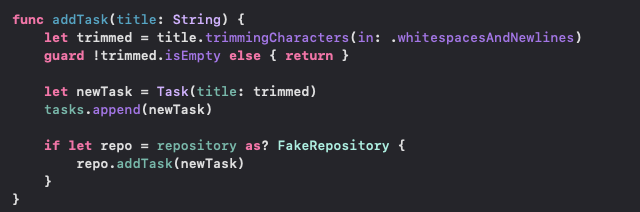
### 

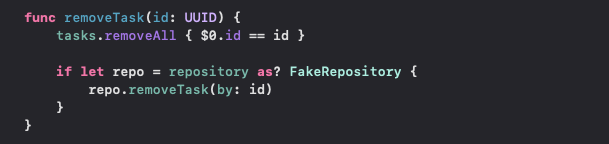
### Step 3 — Refactor the Production Code (GREEN)

* **Start with the easiest win — create an EmptyStubRepository:**Inside your repositories/ folder:****
* This guarantees your unit tests will inject a working stub with no side effects.
* Refactor the TaskListViewModel:
  + Update your TaskListViewModel to accept a RepositoryProtocol in its initializer and fetch tasks from it.

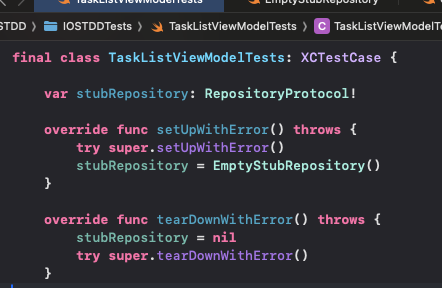


* + Ensure the addTask and removeTask methods also update the repository if required.





* Don’t forget to update the test setup and teardown with the following:

****

This guarantees your tests always use the clean stub on every run.

**Guided Learning (QLC‑4.3): Adding Async/Await Support to the App**

**Objective:**Introduce Swift Concurrency (async/await) into the repository layer and refactor TaskListViewModel to use asynchronous data access. This ensures the architecture is future‑proof for real network/persistence operations and leverages Swift 6’s safety rules.

### 

### Step 1 — Define the Problem and Requirements

We currently use a synchronous repository abstraction. To better model real-world behavior (remote APIs, disk I/O) and align with Swift 6 concurrency:

* The TaskRepositoryProtocol should expose async methods.
* The TaskListViewModel should call those methods from @MainActor contexts.
* Unit tests must run deterministically using an async stub implementation.
* No production networking is required yet, we simulate async via stubs.

Why this matters:  
Swift Concurrency gives us structured, safer asynchronous code:

* UI updates remain on the main thread via @MainActor.
* Tests can await results without callback expectations.
* Swapping in a real network or persistence implementation later requires no signature changes.

### Step 2 — Write the Failing Tests First (RED)

Add new async unit tests in TaskListViewModelTests.swift targeting the asynchronous behavior. Example tests:

* testLoadTasks\_PopulatesTasksFromRepository() — After calling await vm.loadTasks(), the tasks published by the ViewModel match the stub’s seed data.
* testAddTask\_AppendsToViewModelAndRepository() — Calling await vm.addTask("New") increases both the repository and ViewModel counts.
* testRemoveTask\_RemovesFromViewModelAndRepository() — After seeding, removing a task via await vm.removeTask(id) decreases counts.
* testAddTask\_IgnoresEmptyTitle() — Passing whitespace does not change counts.

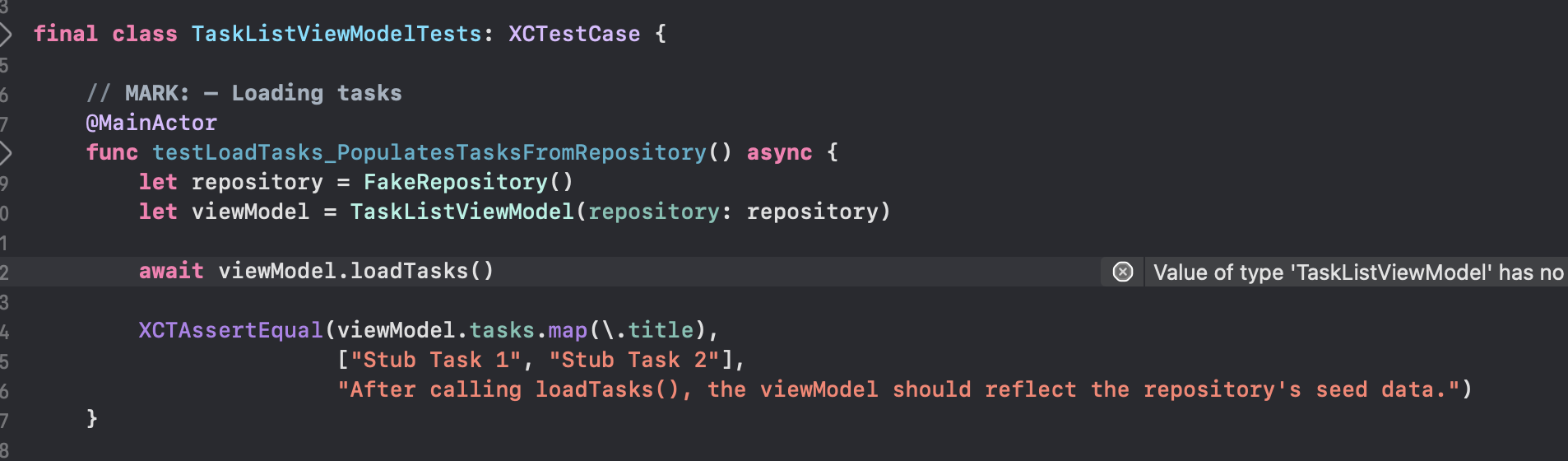
**At this stage, we intentionally use a** FakeRepository **instead of simple stubs.**The fake repository provides *predictable, mutable, and persistent* in-memory data that more closely mirrors real-world repository logic. This allows our tests to:

* **Simulate realistic behavior** for loading, adding, and removing tasks asynchronously.
* **Verify that changes persist** across multiple operations, not just return canned responses.
* **Catch integration and state-related bugs** that would be missed by a stateless stub.

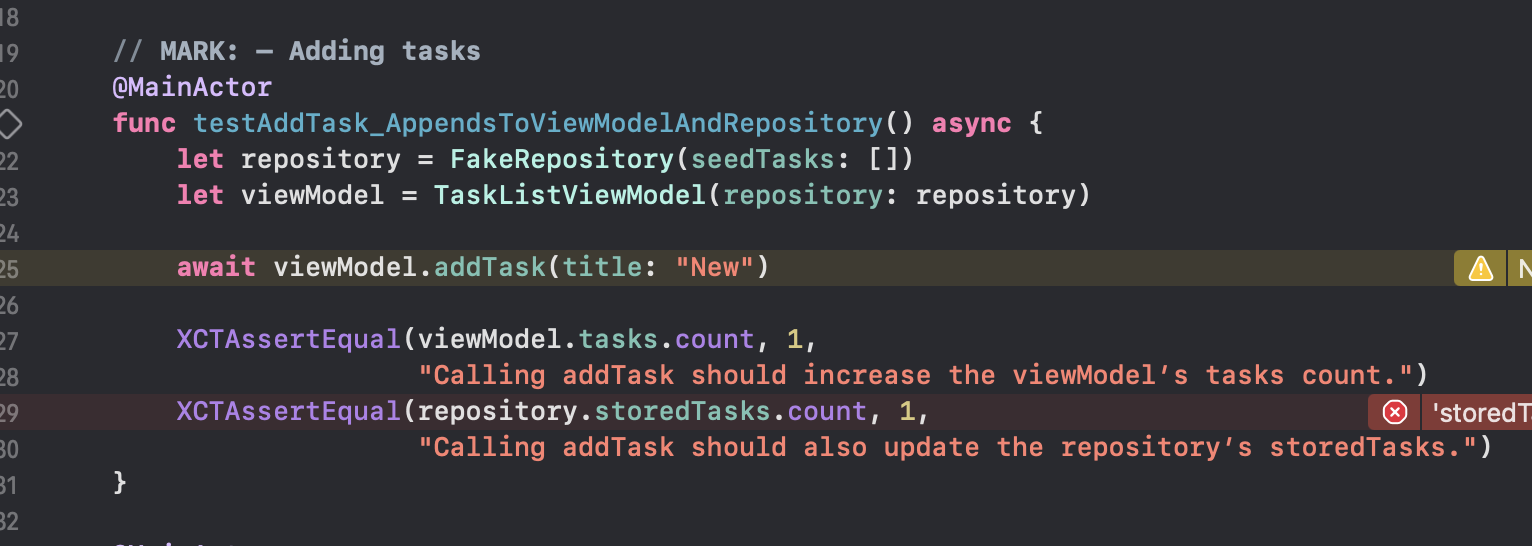
By making the FakeRepository async-capable and using it in tests, we’re integrating with application logic in a way that matches actual app use, while keeping tests fast and isolated from production data or external systems.

**Our tests should now look like the following**, we’ve removed any redundant setup/teardown boilerplate because each test instantiates its own FakeRepository with explicit seed data.

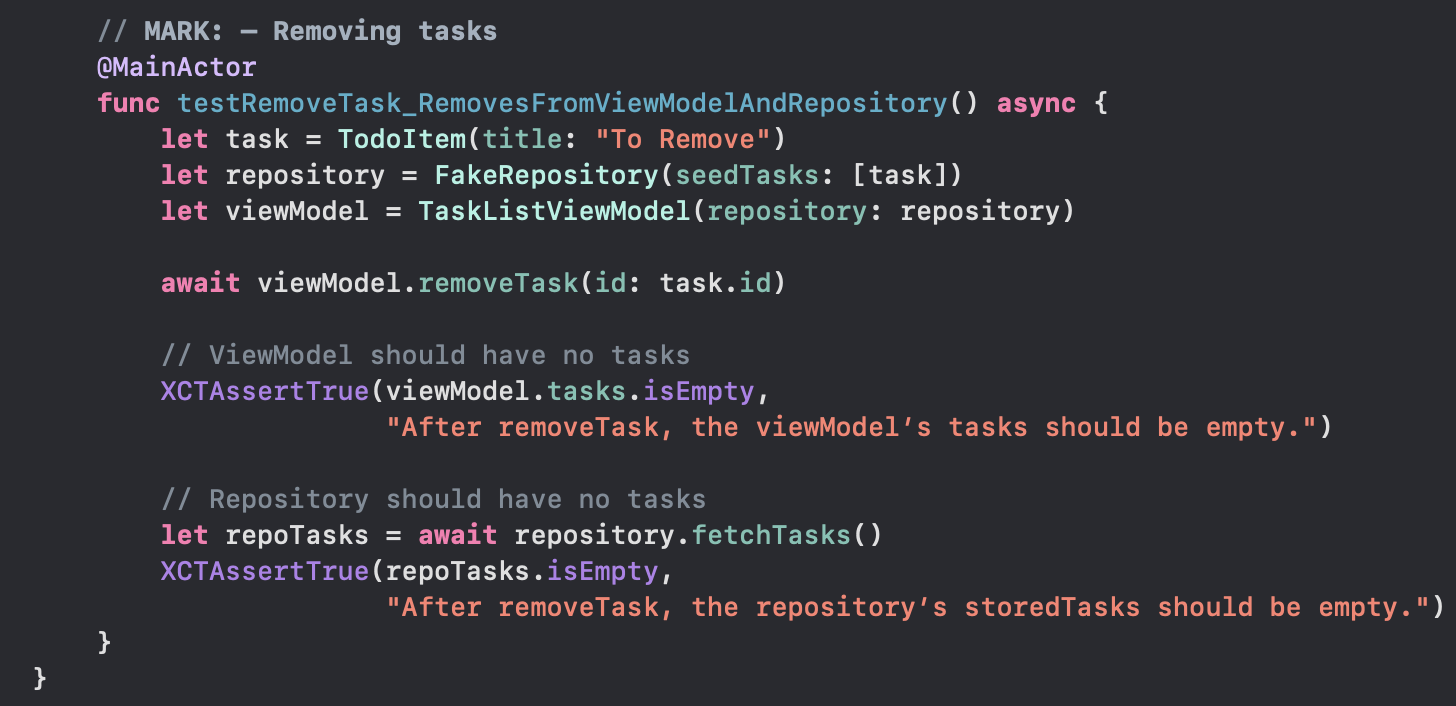
1. **testLoadTasks\_PopulatesTasksFromRepository()**
   * Call await viewModel.loadTasks() and assert the stub’s two seed titles appear.



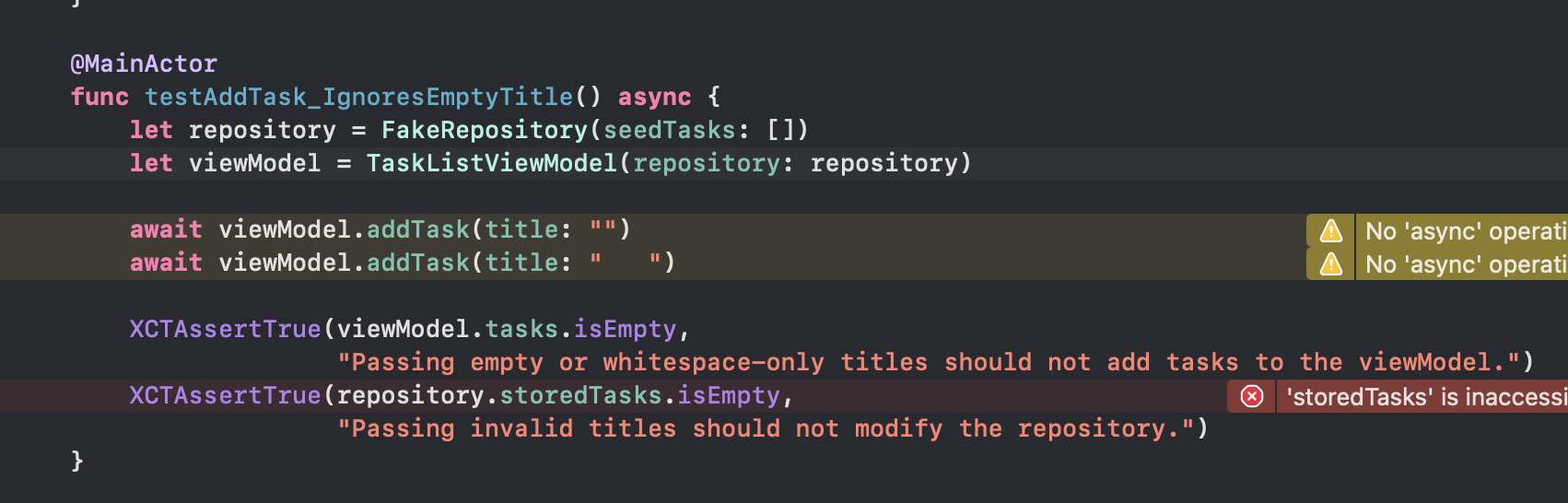
1. **testAddTask\_AppendsToViewModelAndRepository()**
   * With an empty seed, call await viewModel.addTask("New") and expect both viewModel.tasks and repository.storedTasks to contain one item.



1. **testRemoveTask\_RemovesFromViewModelAndRepository()**
   * Seed a single task, call await viewModel.removeTask(id:), and assert both the view model and repository are empty.

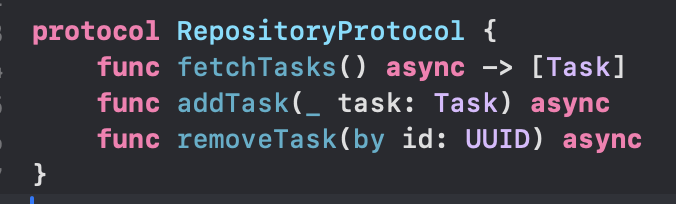


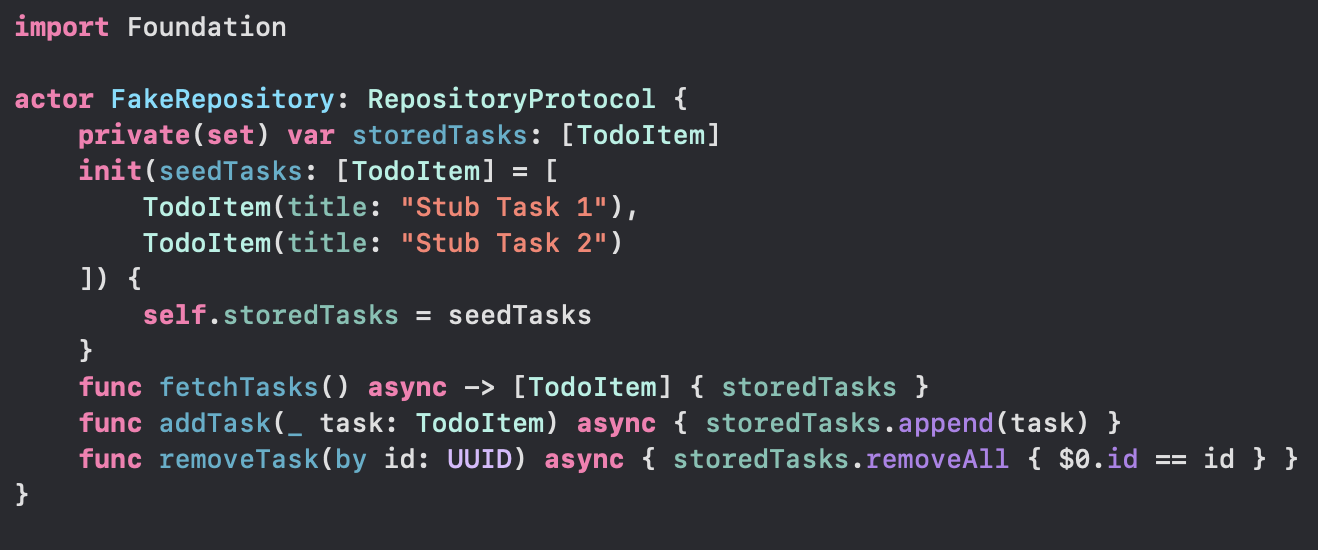
1. **testAddTask\_IgnoresEmptyTitle()**
   * Call await viewModel.addTask("") and await
   * viewModel.addTask(" "), then assert no mutations occur.



At this point, all four tests will **fail** because neither the protocol nor the view model yet support any async methods or loadTasks().

### Step 3 — Implement the Production Code to Align with the Async tests (GREEN)

* **Rename your Task model to TodoItem.** This avoids naming collisions with SwiftUI’s .task { … } modifier and the Task { … } concurrency builder.
* Tests expect: await repository.fetchTasks(), so the contract must declare async methods before anything else will compile.
  + **Update your stubs/fakes to conform.** The tests drive through FakeRepository, so it must implement the new async methods to satisfy the protocol and preserve seed data behavior. (Optionally delete stubs)



* + **Add loadTasks() and wire up async in TaskListViewModel.** The VM must call the repository’s async methods, update its published state, and satisfy the tests’ await vm.loadTasks() and subsequent calls to addTask/removeTask. (Simplified by turning our fake into an actor)



### Step 4 — Refactor & Cleanup (REFACTOR)

1. Remove Stubs You Don’t Need  
   • If you’ve replaced your old EmptyStubRepository with the actor-based FakeRepository, delete the unused stub files and tests.
2. Rename Core Types (Optional)  
   • If you decide to switch from Task → ToDoItem (to avoid clashes with SwiftUI’s Task), update all model, protocol and test references.
3. Solidify Your Protocols  
   • Ensure RepositoryProtocol: Sendable and that your actor stub conforms.  
   • Audit any other services/coordinators to confirm they’re concurrency-safe.
4. DRY Up Your ViewModel  
   • Extract common logic (e.g. trimming + guard) into small helpers if it reads better.  
   • Consider moving loadTasks() into init via a Task { await loadTasks() } so the view never has to .taskitself.
5. Simplify Your SwiftUI  
   • Replace manual ForEach(indices) hacks with a keyed List(viewModel.tasks, id: \.id) once your model renames don’t collide.  
   • Remove leftover .buttonStyle or .accessibilityIdentifier calls you no longer need.
6. Audit & Re-run All Tests  
   • Confirm your changes haven’t broken any of the async tests or new coordinator/mocking specs.  
   • Clean up test setup/teardown now that your fakes are actors.

### Why Asynchronous Repositories & ViewModels in QLC‑4.3

**Real-World Data Is Async**• Networking, database reads/writes, even file I/O all use async/await in Swift 6.  
• Modeling your protocol with async means your tests and production code share the same signature.

**Cleaner Separation of Concerns**

* **ViewModel**• Presentation logic only: holds published state (e.g. [TodoItem], loading flags).  
  • Runs under @MainActor—never touches I/O or background threads itself.
* **Service / Repository**• Business logic only: create/update/delete, persistence, logging.  
  • Stateless and concurrency-safe when implemented as an **actor** (or otherwise Sendable).

**Update Your Stub to an Actor**• Convert FakeRepository into an **actor** so its in-memory state (storedTasks) is data-race-safe.  
• Implement the new async protocol methods and preserve your seed data behavior.  
• **Ensure RepositoryProtocol: Sendable** and that your model (TodoItem) also conforms to Sendable so Swift can safely pass your repository and data across concurrency domains.

**Enter the TaskCoordinator**• **Orchestration**: Receives UI intents (e.g. addNewTask(title:)), calls the async service, then notifies via delegate.  
• **Decoupling**: UI/ViewModel doesn’t need to know how tasks are created—only that they appeared.  
• **Testability**: Stub the async service & inject a mock delegate to assert calls (e.g. “didAdd” notifications).  
• **Future Growth**: Perfect place to insert navigation, analytics, permission checks—without bloating other layers.

**Putting It All Together in QLC-4.3**

* **Async Protocol & Actor-Stub** → Model real data flows in tests & prod
* **Async ViewModel** → Safely updates UI under @MainActor
* **TaskCoordinator** → Workflow glue, sequencing async calls + delegation