Testing Coroutines

Unit testing a Coroutine

Kotlin coroutines provide an elegant way to write asynchronous code, but sometimes coroutines make it difficult to write unit tests.

The unit tests must be *efficient* and *stable*.

- *How to build a coroutine* from the unit tests

- How to *make unit tests wait until all the jobs* in the coroutine have *finished*.

- How to make unit test run *as fast as possible*, and not sit around waiting for a coroutine delay to finish.

`TestCoroutineDispatcher` allows us to write efficient and stable unit tests for code written with coroutines.

Module `kotlinx-coroutines-test`

- Test utilities for `kotlinx.coroutines`.
- This package provides testing utilities for effectively testing coroutines.

```
Dependencies {
  testImplementation         "org.jetbrains.kotlinx:kotlinx-coroutines-test:$version"
  androidTestImplementation  "org.jetbrains.kotlinx:kotlinx-coroutines-test:$version"
}
```

`kotlinx-coroutines-test` consists of four ingredients

- **TestCoroutineDispatcher**
  - Executes new coroutines immediately
  - Implements a virtual time and gives you fine grained control on it
- **TestCoroutineExceptionHandler**
  - Captures and collects all exceptions, so they can be inspected in tests, and rethrows the first one on cleanup
- **TestCoroutineScope**
  - Provides TestCoroutineDispatcher/TestCoroutineExceptionHandler by default
- **runBlockingTest()**
  - Tie everything up providing above three ingredients. It advances time of the test dispatcher until idle, which effectively makes sure that everything in the test block is run. It also checks for misusage by counting the active jobs before and after the test.

How to build a coroutine from the unit tests

Which options?

To call a suspend function, we need to use a coroutine builder. There's a few to choose from:

- kotlinx.coroutines.**runBlocking**
- kotlinx.coroutines.test.**runBlockingTest**
- kotlinx.coroutines.test.**TestCoroutineDispatcher.runBlockingTest**

runBlocking

- Runs a new coroutine and *blocks the current thread* until its completion.
  - Bridge regular blocking code to libraries written in suspending style
  - To be used in main functions and in tests,
- The default `CoroutineDispatcher` is an internal implementation of event loop that processes continuations in this blocked thread.

Use Case 1: The test doesn't trigger new coroutines and we don't care about delays

```
suspend fun loadData() {
  val articles = networkRequest()
  show(articles)
}

suspend fun networkRequest(): List<Article> {
  return apiService.getArticles()
}

private fun show(
  articles: List<Article>
) {
  _articles.value = articles
}
```

```
@Test fun `test loadData`() = runBlocking {
  coEvery { apiService.getArticles() } coAnswers {
    testArticles
  }

  viewModel.loadData()

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

Use Case 2: The test triggers new coroutines and/or we care about delays

```
class MyViewModel(val apiService:ApiService) : ViewModel() {
  val scope = CoroutineScope(SupervisorJob())

  fun onButtonClicked() {
    scope.launch {
      loadData()
    }
  }
}
```

Dispatchers.Default

```
@Test fun `test onButtonClicked`() = runBlocking {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000)
    testArticles
  }

  viewModel.onButtonClicked()

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

Use Case 2: The test triggers new coroutines and/or we care about delays

```kotlin
class MyViewModel(val apiService:ApiService) : ViewModel() {
  val scope = CoroutineScope(SupervisorJob())
  fun onButtonClicked() {
    scope.launch {
      loadData()
    }
  }
}
```

```kotlin
@Test fun `test onButtonClicked`() = runBlocking {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000)
    testArticles
  }

  viewModel.onButtonClicked()

  delay(3000)

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```
9

`runBlockingTest` to the rescue?

To test suspend functions or coroutines started with `launch` or `async` use the `runBlockingTest` that provides extra test control to coroutines.

• *Eager execution* of `launch` or `async` code blocks

• *Explicit time control* for testing multiple coroutines

• *Auto-advancing* of time for suspend functions

• *Pause*, manually *advance*, and *resume* the execution of coroutines in a test

• *Report uncaught exceptions* as test failures

10

Auto-advance time

```
⇥ coEvery { apiService.getArticles() } coAnswers {
⇥     delay(3000)
      testArticles
  }
```

```
@Test
fun `test loadData`() = runBlocking {
  val duration = measureTimeMillis {
⇥   viewModel.loadData()
    viewModel.articles.getValueForTest()
  }
  println("time elapsed = $duration")
}
```

```
@Test
fun `test loadData`() = runBlockingTest {
  val duration = measureTimeMillis {
⇥   viewModel.loadData()
    viewModel.articles.getValueForTest()
  }
  println("time elapsed = $duration")
}
```

time elapsed = 3044   time elapsed = 69

Use Case 2: The test triggers new coroutines and/or we care about delays

```
class MyViewModel(val apiService:ApiService) : ViewModel() {
  val scope = CoroutineScope(SupervisorJob())
  fun onButtonClicked() {
    scope.launch {
⇥     loadData()
    }
  }
}
```

Dispatchers.Default

```
@Test fun `test onButtonClicked`() = runBlockingTest {
⇥ coEvery { apiService.getArticles() } coAnswers {
⇥   delay(3000)
    testArticles
  }

  viewModel.onButtonClicked()

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```
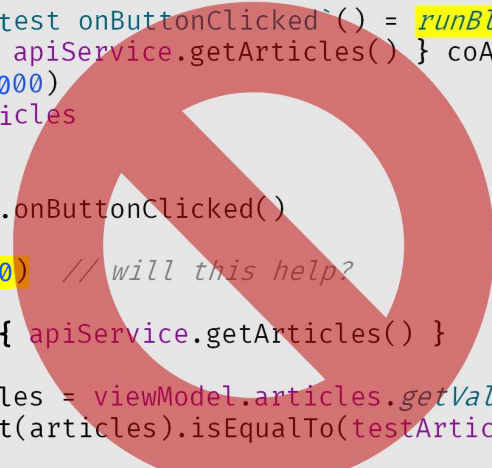
❗ expected: [Article(id=T001, …
   but was : null

Use Case 2: The test triggers new coroutines and/or we care about delays

```kotlin
class MyViewModel(val apiService:ApiService) : ViewModel() {
  val scope = CoroutineScope(SupervisorJob())

  fun onButtonClicked() {
    scope.launch {
      loadData()
    }
  }
}
```

Dispatchers.Default

```kotlin
@Test fun `test onButtonClicked`() = runBlockingTest {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000)
    testArticles
  }

  viewModel.onButtonClicked()

  delay(3000)   // will this help?

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

13

⚠ expected: [Article(id=T001, …
but was : null

Inject CoroutineDispatcher for Testing

```kotlin
class ArticleViewModel(
    val apiService: ApiService
) : ViewModel() {
  val scope = CoroutineScope(SupervisorJob())

  fun onButtonClicked() {
    scope.launch {
      loadData()
    }
  }
}
```

```kotlin
class ArticleViewModel(
    val apiService: ApiService,
    val dispatcher: CoroutineDispatcher
) : ViewModel() {
  val scope = CoroutineScope(SupervisorJob())

  fun onButtonClicked() {
    scope.launch(dispatcher) {
      loadData()
    }
  }
}
```

14

Use Case 2: The test triggers new coroutines …

```kotlin
@Test fun `test onButtonClicked`() = runBlockingTest {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000); testArticles
  }

  val testDispatcher =
    coroutineContext[ContinuationInterceptor] as? TestCoroutineDispatcher

  viewModel = ArticleViewModel(apiService, testDispatcher)

  viewModel.onButtonClicked()

  delay(3000)    // unfortunately, we still needed this delay ☹

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

👍 passes

Better Solution?

```kotlin
@Test fun `test onButtonClicked`() = runBlockingTest {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000); testArticles
  }

  val testDispatcher = ...

  viewModel = ArticleViewModel(apiService, testDispatcher)
  pauseDispatcher()
  viewModel.onButtonClicked()
  resumeDispatcher()

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

👍 passes

Better Solution?

```kotlin
@Test fun `test onButtonClicked`() = runBlockingTest {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000); testArticles
  }

  val testDispatcher = ...

  viewModel = ArticleViewModel(apiService, testDispatcher)

  pauseDispatcher()
  viewModel.onButtonClicked()
  advanceUntilIdle() // this is also OK

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

👍 passes

Better Solution?

```kotlin
@Test fun `test onButtonClicked`() = runBlockingTest {
  coEvery { apiService.getArticles() } coAnswers {
    delay(3000); testArticles
  }

  val testDispatcher = ...

  viewModel = ArticleViewModel(apiService, testDispatcher)

  pauseDispatcher()
  viewModel.onButtonClicked()
  advanceTimeBy(3000) // this is also OK

  coVerify { apiService.getArticles() }

  val articles = viewModel.articles.getValueForTest()
  assertThat(articles).isEqualTo(testArticles)
}
```

👍 passes

```
@ExperimentalCoroutinesApi
fun runBlockingTest(context: CoroutineContext = EmptyCoroutineContext,
                    testBody: suspend TestCoroutineScope.() -> Unit) {

    val scope = TestCoroutineScope(safeContext)                        Setup

    val deferred = scope.async {
        scope.testBody()
    }
    dispatcher.advanceUntilIdle()                                      Run Test
    deferred.getCompletionExceptionOrNull()?.let { throw it }

    scope.cleanupTestCoroutines()
    val endingJobs = safeContext.activeJobs()
    if (activeJobs().isNotEmpty()) {                                   Cleanup
        throw UncompletedCoroutinesError("Test finished with active jobs")
    }

}
```

More Realistic Demo for Timing Control

```
class ArticleViewModel(apiService…, dispatcher…): ViewModel() {
  fun onButtonClicked() {
      _articles.value = Resource.Loading
      scope.launch(dispatcher) {
          loadData() // Let's inline this method call for illustration purpose
      }
  }
  suspend fun loadData() {
      _articles.value = networkRequest()
  }
  suspend fun networkRequest(): Resource<List<Article>> {
      return apiService.getArticles()
  }
}
```

More Realistic Demo for Timing Control

```kotlin
class ArticleViewModel(apiService…, dispatcher…): ViewModel() {
  fun onButtonClicked() {
    _articles.value = Resource.Loading
    scope.launch(dispatcher) {
        _articles.value = apiService.getArticles()
    }
  }
}
```

How to test the value of the livedata _articles
when onButtonClicked() is called?

```kotlin
@Test
fun `test using pause-resume dispatcher`() = testDispatcher.runBlockingTest {
    // Given
    viewModel = ArticleViewModel(apiService, testDispatcher)

    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        Resource.Success(testArticles)
    }

    // When
    pauseDispatcher()
    viewModel.onButtonClicked()

    var articles = viewModel.articles.getValueForTest()
    assertThat(articles).isEqualTo(Resource.Loading)

    resumeDispatcher()

    // Then
    coVerify { apiService.getArticles() }

    articles = viewModel.articles.getValueForTest()
    assertThat(articles).isEqualTo(Resource.Success(testArticles))
}
```

More on `runBlockingTest`

- `runBlockingTest` relies on `TestCoroutineDispatcher` and `TestCoroutineScope`.
- When `runBlockingTest` calls a suspend function or launches a new coroutine, it *executes it immediately* by default.
  - This has the effect of making the coroutines single threaded and offers the ability to explicitly control all coroutines in tests.
- `TestCoroutineDispatcher` implements a virtual time and gives you fine grained control on it.
  - pause/resume execution and control its virtual clock.

More on `runBlockingTest` (Cont'd)

- In addition, `runBlockingTest` will *rethrow uncaught exceptions*.
  - This makes it easier to test when a coroutine is throwing an exception.
- One of the features of `runBlockingTest` is that it *won't let you leak coroutines* after the test completes.
  - If there are any unfinished coroutines, like our launch coroutine, at the end of the test, it will fail the test.

Caution!

- `runBlockingTest` is experimental, and currently has *a bug that makes it fail the test* if a coroutine switches to a dispatcher that executes a coroutine on another thread.

Manuel Vivo
@manuelvicnt

The expectation is right, it should work. There's an open issue about this:

Kotlin/kotlinx.coroutines

#1204 **runBlockingTest fails with "This job has not completed yet"**

💬 49 comments

PaulWoitaschek opened on May 17, 2019

runBlockingTest fails with "This job has not completed yet" · Issue #1204...
This simple block: runBlockingTest { suspendCancellableCoroutine { cont -> thread { Thread.sleep(1000) cont.resume(Unit) } } } Fails with the ...
🔗 github.com

Important!!

- The function `runBlockingTest` will always block the caller, just like a regular function call. The coroutine will *run synchronously on the same thread*.

- You should avoid `runBlocking` and `runBlockingTest` in your application code and prefer `launch` which returns immediately.

- `runBlockingTest` should only be used from tests as it executes coroutines in a test-controlled manner.

- `runBlocking` can be used to provide blocking interfaces to coroutines.

Scopes in ViewModels

- A `CoroutineScope` keeps track of all coroutines it creates.
- Cancel a scope cancels all coroutines it created
  - *structured concurrency*
- If your `ViewModel` is getting destroyed, all the asynchronous work that it might be doing must be stopped. Otherwise, you'll waste resources and potentially leaking memory.
- **If you consider that certain asynchronous work should persist after `ViewModel` destruction, it is because it should be done in a lower layer of your app's architecture.**

viewModelScope

- `viewModelScope` is provided as an *extension property* of the `ViewModel` class.
- This scope is bound to `Dispatchers.Main` and will automatically be cancelled when the `ViewModel` is cleared.

```
val ViewModel.viewModelScope: CoroutineScope
  get() {
    …
    CloseableCoroutineScope(SupervisorJob() + Dispatchers.Main.immediate)
    …
  }
```

```
dependencies {
  implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$version"
}
```

Using viewModelScope

```kotlin
class ArticleViewModel(
    val apiService: ApiService
) : ViewModel() {
    val scope =
        CoroutineScope(SupervisorJob())

    fun onButtonClicked() {
        scope.launch {
            loadData()
        }
    }
```

Dispatchers.Default

```kotlin
    override fun onCleared() {
        super.onCleared()
        scope.cancel()
    }
```

```kotlin
class ArticleViewModel(
    val apiService: ApiService
) : ViewModel() {

    fun onButtonClicked() {
        viewModelScope.launch {
            loadData()
        }
    }
}
```

Dispatchers.Main

Will this work?

```kotlin
class ArticleViewModel(
    val apiService: ApiService
) : ViewModel() {

    fun onButtonClicked() {
        viewModelScope.launch {
            loadData()
        }
    }
}
```

Dispatchers.Main

⚠ java.lang.IllegalStateException:
Module with the Main dispatcher had failed to initialize.

```kotlin
@Test fun `test onButtonClicked`() = runBlockingTest {
    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        testArticles
    }

    pauseDispatcher()
    viewModel.onButtonClicked()
    resumeDispatcher()

    coVerify { apiService.getArticles() }

    val articles = viewModel.articles.getValueForTest()
```

Dispatchers.Main uses Looper.getMainLooper() to run code in the UI thread, which is available in Instrumented tests but **not** in Unit tests.

Dispatchers.Main Delegation

- `Dispatchers.setMain` will override the `Main` dispatcher in test situations:
    - To execute a test in situations where the platform Main dispatcher is not available, or
    - To replace `Dispatchers.Main` with a testing dispatcher.

- Once you have this dependency in the runtime, `ServiceLoader` mechanism will overwrite `Dispatchers.Main` with a testable implementation.

Unit Testing viewModelScope

- `viewModelScope` uses the `Dispathers.Main`.

- *Replace the main dispatcher* by calling `Dispatchers.setMain(dispatcher: CoroutineDispatcher)` with a `TestCoroutineDispatcher`.*

- *Should call* `Dispatchers.resetMain()` *and clean up the dispatcher* when the test finishes running, to ensure that a unit test run in isolation and without any side effects.

*Note that `Dispatchers.setMain` is only needed if you use `viewModelScope` or you hardcode `Dispatchers.Main` in your codebase.

Inject CoroutineDispatcher for Testing Again

```kotlin
class ArticleViewModel(
    val apiService: ApiService
) : ViewModel() {

  fun onButtonClicked() {
    viewModelScope.launch {
      loadData()
    }
  }
}
```

```kotlin
class ArticleViewModel(
    val apiService: ApiService,
    val dispatcher: CoroutineDispatcher
) : ViewModel() {

  fun onButtonClicked() {
    viewModelScope.launch(dispatcher) {
      loadData()
    }
  }
}
```

Replace Dispatchers.Main with TestCoroutineDispatcher

```kotlin
val testDispatcher = TestCoroutineDispatcher()

@Before
fun init() {
    Dispatchers.setMain(testDispatcher)
    viewModel = ArticleViewModel(apiService, testDispatcher)
}

@After
fun teardown() {
    Dispatchers.resetMain()
    testDispatcher.cleanupTestCoroutines()
}

@Test
fun `test`() = testDispatcher.runBlockingTest {
    ...
}
```

Dispatchers should be injected into your *ViewModels* so you can properly test. Passing the *Dispatcher* via the constructor would make sure that your test and production code use the same dispatcher.

Custom JUnit4 Rule for Coroutine Testing

```kotlin
class MainCoroutineRule(
    val testDispatcher: TestCoroutineDispatcher = TestCoroutineDispatcher()
) : TestWatcher(), TestCoroutineScope by TestCoroutineScope(testDispatcher){

    override fun starting(description: Description?) {
        super.starting(description)

        Dispatchers.setMain(testDispatcher)
    }

    override fun finished(description: Description?) {
        super.finished(description)

        Dispatchers.resetMain()
        cleanupTestCoroutines()
    }
}
```

Recommended Way to Use Coroutines in Android

- On Android, you can use a scope to cancel all running coroutines when, for example, the user navigates away from an **Activity** or **Fragment**.
- For coroutines started by the UI, it is typically correct to start them on `Dispatchers.Main` (main thread on Android).
- A coroutine started on `Dispatchers.Main` won't block the main thread while suspended.
- Since a `ViewModel` coroutine almost always updates the UI on the main thread, starting coroutines on the main thread saves you extra thread switches.
- A coroutine started on the `Main` thread can switch dispatchers any time after it's started.
  - Ex) it can use another dispatcher to parse a large JSON result off the main thread.

Calling heavy-lifting suspend function from coroutines

```
suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T
```

- To switch between any dispatcher, use `withContext`.
  - Calling `withContext` switches to the other dispatcher just for the lambda then comes back to the dispatcher that called it with the result of that lambda.
- By default, Kotlin coroutines provides three Dispatchers: `Main`, `IO`, and `Default`.
  - The IO dispatcher is optimized for IO work like reading from the network or disk, while the Default dispatcher is optimized for CPU intensive tasks.

Calling heavy-lifting suspend function from coroutines

```kotlin
class ArticleViewModel(
    val apiService: ApiService,
    val dispatchers: DispatcherProvider = DefaultDispatcherProvider()
) : ViewModel() {
    fun onButtonClicked() {
        viewModelScope.launch(dispatchers.main) {
            loadData()
        }
    }
    suspend fun loadData() {
        val articles = networkRequest()
        show(articles)
    }
    suspend fun networkRequest(): List<Article> {
        return withContext(dispatchers.io) {
            apiService.getArticles()
        }
    }
    private fun show(articles: List<Article>) { … }
}
```

Main-safety

Calling heavy-lifting suspend function from coroutines

```kotlin
class ArticleViewModel(
    val apiService: ApiService,
    val dispatchers: DispatcherProvider = DefaultDispatcherProvider()
) : ViewModel() {
    fun onButtonClick
        viewModelScop
            loadData(
        }
    }
```

```kotlin
interface DispatcherProvider {
    val main: CoroutineDispatcher
    val mainImmediate: CoroutineDispatcher
    val default: CoroutineDispatcher
    val io: CoroutineDispatcher
}
```

```kotlin
class DefaultDispatcherProvider(
    override val main: CoroutineDispatcher = Dispatchers.Main,
    override val mainImmediate: CoroutineDispatcher = Dispatchers.Main.immediate,
    override val default: CoroutineDispatcher = Dispatchers.Default,
    override val io: CoroutineDispatcher = Dispatchers.IO,
) : DispatcherProvider
```

```kotlin
            apiService.getArticles()
        }
    }
    private fun show(articles: List<Article>) { … }
}
```

Suspend functions should guarantee **main-safety**

- It's a good idea to start UI-related coroutines on the Main thread.

- By convention, you should ***ensure that suspend functions are main-safe***.

- Then it is safe to call them from any dispatcher, even Dispatchers.Main.

```
suspend fun networkRequest(): List<Article> {
  return withContext(dispatchers.io) {
    apiService.getArticles()
  }
}
```

- You do not need to use withContext to call main-safe suspending functions.

  *Libraries like **Room** and **Retrofit** offer **main-safety** out of the box.*

Write a timeout test

```
interface Api {
    suspend fun fetch(): String
}

class SuspendingFakeApi : Api {
    val deferred = CompletableDeferred<String>()

    override suspend fun fetch(): String {
        return deferred.await() // wait forever ...
    }
}

suspend fun loadData(api: Api): String {
    return withTimeout(5_000) {
        api.fetch()
    }
}
```

Write a timeout test (Cont'd)

```kotlin
@Test(expected = TimeoutCancellationException::class)
fun `bad test`() = runBlockingTest{
    val api = SuspendingFakeApi()

    println("result = ${loadData(api)}") // always timeout ...
}
```

```kotlin
@Test fun `test timeout expired`() = runBlockingTest{
    val api = SuspendingFakeApi()

    launch {
        loadData(api)
    }

    advanceTimeBy(5_000)

    api.deferred.complete("Hello")
}
```

Test launches a separate coroutine to call loadData. **This is a key part of testing timeouts**, the timeout should happen in a different coroutine than the one runBlockingTest creates. By doing so, we can call the next line, advanceTimeBy(5_000) which will advance time by 5 seconds and cause the other coroutine to timeout.