



Testing Coroutines

1

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.

2

Module `kotlinx-coroutines-test`

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

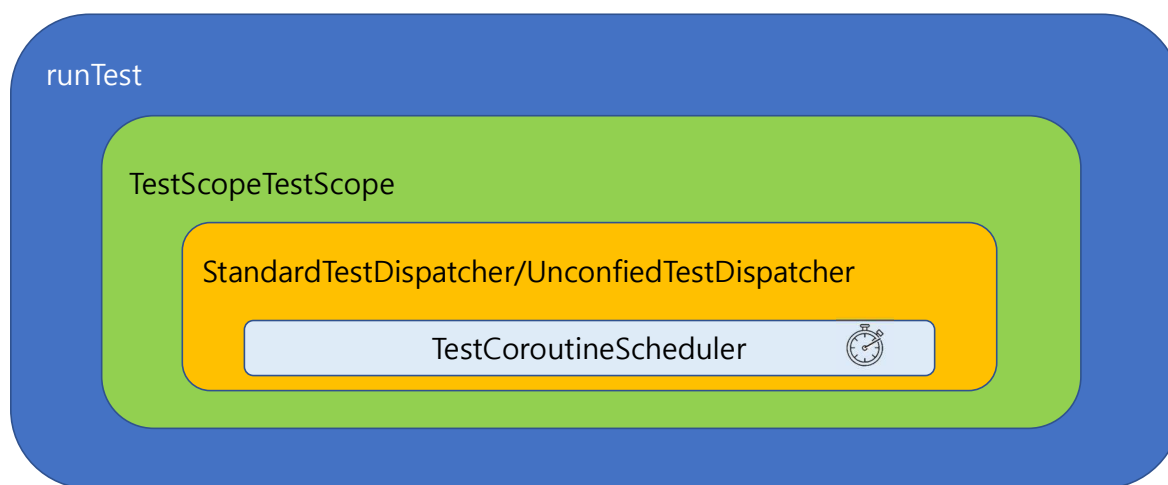
```
Dependencies {  
    testImplementation      "org.jetbrains.kotlinx:kotlinx-coroutines-test:$version"  
    androidTestImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:$version"  
}
```



As of version 1.6.0, the API of the test module changed significantly

3

`kotlinx-coroutines-test` consists of four ingredients



4

runTest() (was runBlockingTest())

Similar to `runBlocking()`, but automatically skips delays and handles uncaught exceptions

```
public fun runTest(  
    context: CoroutineContext = EmptyCoroutineContext,  
    timeout: Duration = DEFAULT_TIMEOUT.getOrThrow(),  
    testBody: suspend TestScope.() -> Unit  
): TestResult
```

5

TestScope

A `CoroutineScope` that integrates with `runTest`, providing access to `TestCoroutineScheduler`

```
public fun runTest(  
    context: CoroutineContext = EmptyCoroutineContext,  
    timeout: Duration = DEFAULT_TIMEOUT.getOrThrow(),  
    testBody: suspend TestScope.() -> Unit  
): TestResult
```

6

Virtual Time Control

```
fun TestScope.advanceUntilIdle() = testScheduler.advanceUntilIdle()
fun TestScope.advanceTimeBy(ms: Long) = testScheduler.advanceTimeBy(ms)
fun TestScope.runCurrent() = testScheduler.runCurrent()
val TestScope.currentTime: Long get() = testScheduler.currentTime
```

7

TestCoroutineScheduler

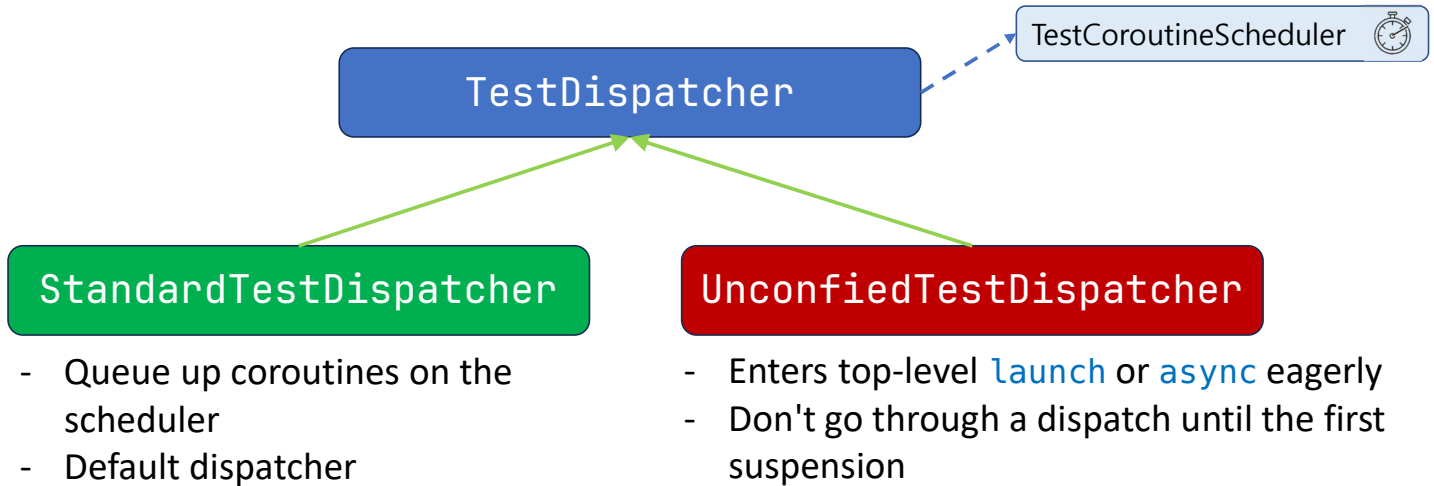
The source of *virtual time*, used for controlling execution order and skipping delays

- `advanceUntilIdle()`
 - runs all enqueued tasks until no more
- `advanceTimeBy(ms)`
 - runs the enqueued tasks until the current virtual time advances by ms
- `runCurrent()`
 - runs the tasks that are scheduled at this point of virtual time
- `currentTime`
 - gets the current virtual time

8

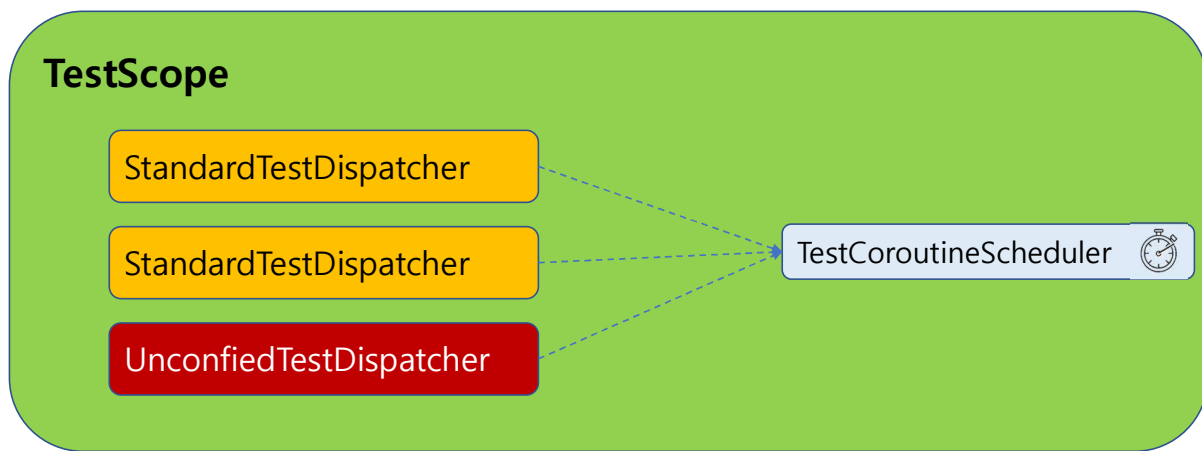
TestDispatcher

- Gives you fine grained control on the virtual time using `TestCoroutineScheduler`



9

All `TestDispatchers` must share the same `TestCoroutineScheduler`



10

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.runTest`
- `kotlinx.coroutines.test.runBlockingTest` (*deprecated*)

11

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.

12

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()
    } returns testArticles

    viewModel.loadData()

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

13

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 onClicked() {
        scope.launch {
            loadData()
        }
    }
}
```

Dispatchers.Default

```
@Test fun `test onClicked`() {
    coEvery { apiService.getArticle() } returns {
        delay(100)
        Article(1, "Title", "Content")
    }
}
```

Dispatchers.Default

```
@Test fun `test onButtonClicked`() = runBlocking {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3_000)  
        testArticles  
    }  
  
    viewModel.onButtonClicked()  
  
    val articles = viewModel.articles.getValueForTest()  
    assertThat(articles).isEqualTo(testArticles)  
}
```

! expected: [Article(id=T001, ...
but was : null

14

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 onClicked() {  
        scope.launch {  
            loadData()  
        }  
    }  
}
```



```
@Test fun `test onClicked`() = runBlocking {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3000)  
        testArticles  
    }  
  
    viewModel.onClicked()  
  
    val articles = viewModel.articles.getValueForTest()  
    assertEquals(testArticles, articles)  
}
```



takes time; unreliable



passes

15

`runTest` to the rescue?

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

- *Eager execution* of *top-level* `launch` or `async` code blocks if used with `UnconfinedTestDispatcher`
 - But, default is `StandardTestDispatcher`
- *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
 - **Rethrows the first uncaught exception** as test failures <= same as `runBlocking`
 - Manual capture is possible if needed

16

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")  
}
```

time elapsed = 3044

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

time elapsed = 69

17

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 onClicked() {  
        scope.launch {  
            loadData()  
        }  
    }  
}
```

Dispatchers.Default



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

```
@Test fun `test onClicked`() = runTest {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3000)  
        testArticles  
    }  
  
    viewModel.onButtonClicked()  
  
    val articles = viewModel.articles.getValueForTest()  
    assertThat(articles).isEqualTo(testArticles)  
}
```

18

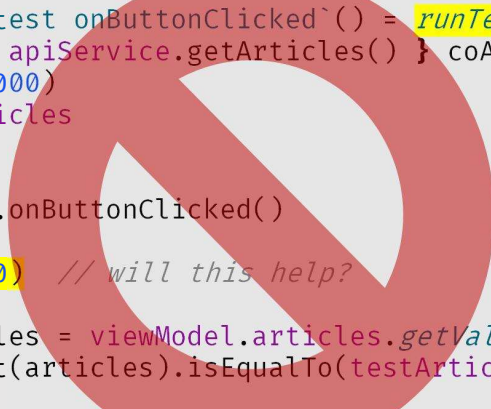
Use Case 2: The test triggers new coroutines and/or we care about delays

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



❗ expected: [Article(id=T001, ...
but was : null

```
@Test fun `test onClicked`() = runTest {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3000)  
        testArticles  
    }  
    viewModel.onClicked()  
    delay(3000) // will this help?  
    val articles = viewModel.articles.getValueForTest()  
    assertEquals(testArticles, articles)  
}
```



19

Inject CoroutineDispatcher for Testing

```
class ArticleViewModel(  
    val apiService: ApiService  
) : ViewModel() {  
    val scope = CoroutineScope(SupervisorJob())  
  
    fun onClicked() {  
        scope.launch {  
            loadData()  
        }  
    }  
}
```



```
class ArticleViewModel(  
    val apiService: ApiService,  
    val dispatcher: CoroutineDispatcher  
) : ViewModel() {  
    val scope = CoroutineScope(SupervisorJob())  
  
    fun onClicked() {  
        scope.launch(dispatcher) {  
            loadData()  
        }  
    }  
}
```

20

Solution (runTest)

```
@Test fun `test onClicked`() = runTest {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3000); testArticles  
    }  
    val testDispatcher =  
        coroutineContext[ContinuationInterceptor] as TestDispatcher  
    viewModel = ArticleViewModel(apiService, testDispatcher)  
  
    viewModel.onButtonClicked()  
    advanceTimeBy(3000); runCurrent()  
  
    val articles = viewModel.articles.getValueForTest()  
    assertThat(articles).isEqualTo(testArticles)  
}
```

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



21

Solution (runTest)

```
@Test fun `test onClicked`() = runTest {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3000); testArticles  
    }  
    val testDispatcher =  
        coroutineContext[ContinuationInterceptor] as TestDispatcher  
    viewModel = ArticleViewModel(apiService, testDispatcher)  
  
    viewModel.onButtonClicked()  
    advanceUntilIdle() // this is also OK  
  
    val articles = viewModel.articles.getValueForTest()  
    assertThat(articles).isEqualTo(testArticles)  
}
```

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



22

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()
    }
}

sealed class Resource<out R>{
    data class Success<out T>(val data: T?): Resource<T>()
    data class Error(val message: String?): Resource<Nothing>()
    object Loading : Resource<Nothing>() }
```

More Realistic Demo for Timing Control

```
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?

```

@Test
fun `realistic test example`() = runTest {
    // Given
    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        Resource.Success(testArticles)
    }

    val testDispatcher =
    coroutineContext[ContinuationInterceptor] as TestDispatcher

    viewModel = ArticleViewModel(apiService, testDispatcher)

    // When
    viewModel.onButtonClicked()

    var articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Loading, articles)

    advanceTimeBy(3000); runCurrent()

    // Then
    articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Success(testArticles), articles)
}

```

```

fun onButtonClicked() {
    _articles.value = Resource.Loading
    scope.launch(dispatcher) {
        _articles.value = apiService.getArticles()
    }
}

```

25

```

@Test
fun `realistic test example`() = runTest {
    // Given
    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        Resource.Success(testArticles)
    }

    val testDispatcher =
    coroutineContext[ContinuationInterceptor] as TestDispatcher

    viewModel = ArticleViewModel(apiService, testDispatcher)

    // When
    viewModel.onButtonClicked()

    var articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Loading, articles)

    advanceTimeBy(3000); runCurrent()

    // Then
    articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Success(testArticles), articles)
}

```

```

fun onButtonClicked() {
    _articles.value = Resource.Loading
    scope.launch(dispatcher) {
        _articles.value = apiService.getArticles()
    }
}

```

26

```

@Test
fun `realistic test example`() = runTest {
    // Given
    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        Resource.Success(testArticles)
    }

    val testDispatcher =
        coroutineContext[ContinuationInterceptor] as? TestDispatcher

    viewModel = ArticleViewModel(apiService, testDispatcher)

    // When
    viewModel.onButtonClicked()

    var articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Loading)

    advanceTimeBy(3000); runCurrent()

    // Then
    articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Success(testArticles))
}

```

```

fun onButtonClicked() {
    _articles.value = Resource.Loading
    scope.launch(dispatcher) {
        _articles.value = apiService.getArticles()
    }
}

```

27

```

@Test
fun `realistic test example`() = runTest {
    // Given
    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        Resource.Success(testArticles)
    }

    val testDispatcher =
        coroutineContext[ContinuationInterceptor] as TestDispatcher

    viewModel = ArticleViewModel(apiService, testDispatcher)

    // When
    viewModel.onButtonClicked()

    var articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Loading)

    advanceTimeBy(3000); runCurrent()

    // Then
    articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Success(testArticles))
}

```

```

fun onButtonClicked() {
    _articles.value = Resource.Loading
    scope.launch(dispatcher) {
        _articles.value = apiService.getArticles()
    }
}

```

28

```

@Test
fun `realistic test example`() = runTest {
    // Given
    coEvery { apiService.getArticles() } coAnswers {
        delay(3000)
        Resource.Success(testArticles)
    }

    val testDispatcher =
        coroutineContext[ContinuationInterceptor] as TestDispatcher
    viewModel = ArticleViewModel(apiService, testDispatcher)

    // When
    viewModel.onButtonClicked()

    var articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Loading, articles)

    advanceTimeBy(3000); runCurrent()

    // Then
    articles = viewModel.articles.getValueForTest()
    assertEquals(Resource.Success(testArticles), articles)
}

```

```

fun onButtonClicked() {
    _articles.value = Resource.Loading
    scope.launch(dispatcher) {
        _articles.value = apiService.getArticles()
    }
}

```

29

Review on `runTest`

- `runTest` relies on `TestDispatcher` and `TestScope`.
- When `runTest` calls a suspend function or launches a new coroutine, it *executes it immediately* if `UnconfinedTestDispatcher` is used.
 - This has the effect of making the coroutines single threaded and offers the ability to explicitly control all coroutines in tests.
- `TestDispatcher` implements a virtual time and gives you fine grained control on it.
 - control its virtual clock.

30

Important!!

- The function `runTest` 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 `runTest` in your application code and prefer `launch` which returns immediately.
- `runTest` 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.

31

Scopes in ViewModels

- A `CoroutineScope` keeps track of all coroutines it creates.
- Cancelling 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.**

32

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"
}
```

33

Using viewModelScope

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

    fun onClicked() {
        scope.launch {
            loadData()
        }
    }

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

Dispatchers.Default



```
class ArticleViewModel(
    val apiService: ApiService
) : ViewModel() {
    fun onClicked() {
        viewModelScope.launch {
            loadData()
        }
    }
}
```

Dispatchers.Main

34

Will this work?

```
class ArticleViewModel(  
    val apiService: ApiService  
) : ViewModel() {  
  
    fun onClicked() {  
        viewModelScope.launch {  
            loadData()  
        }  
    }  
}
```

Dispatchers.Main



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

```
@Test fun `test onClicked`() = runTest {  
    coEvery { apiService.getArticles() } coAnswers {  
        delay(3000)  
        testArticles  
    }  
  
    viewModel.onClicked()  
    advanceUntilIdle()  
  
    val articles = viewModel.articles.getValueForTest()  
    assertThat(articles).isEqualTo(testArticles)  
}
```

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 done this, `ServiceLoader` mechanism will overwrite `Dispatchers.Main` with a testable implementation.
- If `Main` is overridden with a `TestDispatcher`, then its `TestCoroutineScheduler` is used when new `TestDispatcher` or `TestScope` instances are created without `TestCoroutineScheduler` being passed as an argument.

Unit Testing viewModelScope

- `viewModelScope` uses the `Dispatchers.Main`.
- *Replace the main dispatcher* by calling `Dispatchers.setMain(dispatcher: CoroutineDispatcher)` with a `TestDispatcher`.*
- *Should call* `Dispatchers.resetMain()` 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.*

37

Replace Dispatchers.Main with TestDispatcher

```
val testDispatcher = StandardTestDispatcher()

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

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

@Test
fun `test`() = runTest {
    ...
}
```

38



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.

39

Custom JUnit4 Rule for runTest

```
class CoroutineRule(  
    val testDispatcher: TestDispatcher = StandardTestDispatcher()  
) : TestWatcher() {  
    override fun starting(description: Description?) {  
        super.starting(description)  
  
        Dispatchers.setMain(testDispatcher)  
    }  
    override fun finished(description: Description?) {  
        super.finished(description)  
  
        Dispatchers.resetMain()  
    }  
}
```

40

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` shouldn'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.

41

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.

42

Calling heavy-lifting suspend function from coroutines

```
class ArticleViewModel(  
    val apiService: ApiService,  
    val dispatchers: DispatcherProvider = DefaultDispatcherProvider()  
) : ViewModel() {  
    fun onClicked() {  
        viewModelScope.launch {  
            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

43

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

```
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
```

44

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.*



45

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()  
    }  
}
```

46

Write a timeout test (Cont'd)

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

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

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

    launch {
        loadData(api)
    }

    advanceTimeBy(5_000); runCurrent()
    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 `runTest` 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.

47

Running background work

- `runTest` blocks until all child coroutines have finished.
 - This may cause `runTest` to fail with timeout exception.
- Coroutines launched in `backgroundScope` will be cancelled at the end of the test.

```
@Test fun testExampleBackgroundJob() = runTest {
    val channel = Channel<Int>()
    backgroundScope.launch {
        var i = 0
        while (true) {
            channel.send(i++)
        }
    }
    repeat(100) {
        assertEquals(it, channel.receive())
    }
}
```

48