Best practices for coroutines in Android

This page presents several best practices that have a positive impact by making your app more scalable and testable when using coroutines.

Note: These tips can be applied to a broad spectrum of apps. However, you should treat them as guidelines and adapt them to your requirements as needed.

Inject Dispatchers

Don't hardcode `Dispatchers` when creating new coroutines or calling `withContext`.

```
// DO inject Dispatchers
class NewsRepository(
    private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
    suspend fun loadNews() = withContext(defaultDispatcher) { /* ... */ }
}

// DO NOT hardcode Dispatchers
class NewsRepository {
    // DO NOT use Dispatchers.Default directly, inject it instead
    suspend fun loadNews() = withContext(Dispatchers.Default) { /* ... */ }
}
```

This dependency injection pattern makes testing easier as you can replace those dispatchers in unit and instrumentation tests with a test dispatcher (#test-coroutine-dispatcher) to make your tests more deterministic.

Note: The **`viewModelScope`** (https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471) property of **`ViewModel`** (/topic/libraries/architecture/viewmodel) classes is hardcoded to

`Dispatchers.Main`. Replace it in tests by calling **`Dispatchers.setMain`** and passing in a test dispatcher.

Note: While you might have seen hardcoded dispatchers in code snippets across this site and our codelabs, this is only to keep the sample code concise and simple. In your application, you should inject dispatchers.

Suspend functions should be safe to call from the main thread

Suspend functions should be main-safe, meaning they're safe to call from the main thread. If a class is doing long-running blocking operations in a coroutine, it's in charge of moving the execution off the main thread using `withContext`. This applies to all classes in your app, regardless of the part of the architecture the class is in.

```
class NewsRepository(private val ioDispatcher: CoroutineDispatcher) {

    // As this operation is manually retrieving the news from the server
    // using a blocking HttpURLConnection, it needs to move the execution
    // to an IO dispatcher to make it main-safe
    suspend fun fetchLatestNews(): List<Article> {
        withContext(ioDispatcher) { /* ... implementation ... */ }
    }
}

// This use case fetches the latest news and the associated author.
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
    // This method doesn't need to worry about moving the execution of the
    // coroutine to a different thread as newsRepository is main-safe.
    // The work done in the coroutine is lightweight as it only creates
    // a list and add elements to it
    suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()

        val response: List<ArticleWithAuthor> = mutableEmptyList()
        for (article in news) {
```

```
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

This pattern makes your app more scalable, as classes calling suspend functions don't have to worry about what `Dispatcher` to use for what type of work. This responsibility lies in the class that does the work.

The ViewModel should create coroutines

`ViewModel` (/topic/libraries/architecture/viewmodel) classes should prefer creating coroutines instead of exposing suspend functions to perform business logic. Suspend functions in the `ViewModel` can be useful if instead of exposing state using a stream of data, only a single value needs to be emitted.

```
// DO create coroutines in the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiSt
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}

// Prefer observable state rather than suspend functions from the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {
    // DO NOT do this. News would probably need to be refreshed as well.
    // Instead of exposing a single value with a suspend function, news shoul
    // be exposed using a stream of data as in the code snippet above.
```

```
    suspend fun loadNews() = getLatestNewsWithAuthors()
}
```

Views shouldn't directly trigger any coroutines to perform business logic. Instead, defer that responsibility to the `ViewModel`. This makes your business logic easier to test as `ViewModel` objects can be unit tested, instead of using instrumentation tests that are required to test views.

In addition to that, your coroutines will survive configuration changes automatically if the work is started in the `viewModelScope`. If you create coroutines using `lifecycleScope` instead, you'd have to handle that manually. If the coroutine needs to outlive the `ViewModel`'s scope, check out the Creating coroutines in the business and data layer section (#create-coroutines-data-layer).

Note: Views should trigger coroutines for UI-related logic. For example, fetching an image from the Internet or formatting a String.

Don't expose mutable types

Prefer exposing immutable types to other classes. In this way, all changes to the mutable type is centralized in one class making it easier to debug when something goes wrong.

```
// DO expose immutable types
class LatestNewsViewModel : ViewModel() {

    private val _uiState = MutableStateFlow(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    /* ... */
}

class LatestNewsViewModel : ViewModel() {

    // DO NOT expose mutable types
    val uiState = MutableStateFlow(LatestNewsUiState.Loading)

    /* ... */
```

```
}
```

The data and business layer should expose suspend functions and Flows

Classes in the data and business layers generally expose functions to perform one-shot calls or to be notified of data changes over time. Classes in those layers should expose **suspend functions for one-shot calls** and **Flow to notify about data changes**.

```
// Classes in the data and business layer expose
// either suspend functions or Flows
class ExampleRepository {
    suspend fun makeNetworkRequest() { /* ... */ }

    fun getExamples(): Flow<Example> { /* ... */ }
}
```

This best practice makes the caller, generally the presentation layer, able to control the execution and lifecycle of the work happening in those layers, and cancel when needed.

Creating coroutines in the business and data layer

For classes in the data or business layer that need to create coroutines for different reasons, there are different options.

If the work to be done in those coroutines is relevant only when the user is present on the current screen, it should follow the caller's lifecycle. In most cases, the caller will be the ViewModel, and the call will be cancelled when the user navigates away from the screen and the ViewModel is cleared. In this case, `coroutineScope` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) or `supervisorScope` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/supervisor-scope.html) should be used.

```
class GetAllBooksAndAuthorsUseCase(
    private val booksRepository: BooksRepository,
    private val authorsRepository: AuthorsRepository,
) {
    suspend fun getBookAndAuthors(): BookAndAuthors {
        // In parallel, fetch books and authors and return when both requests
        // complete and the data is ready
        return coroutineScope {
            val books = async { booksRepository.getAllBooks() }
            val authors = async { authorsRepository.getAllAuthors() }
            BookAndAuthors(books.await(), authors.await())
        }
    }
}
```

If the work to be done is relevant as long as the app is opened, and the work is not bound to a particular screen, then the work should outlive the caller's lifecycle. For this scenario, an external `CoroutineScope` should be used as explained in the Coroutines & Patterns for work that shouldn't be cancelled blog post (https://medium.com/androiddevelopers/coroutines-patterns-for-work-that-shouldnt-be-cancelled-e26c40f142ad)

.

```
class ArticlesRepository(
    private val articlesDataSource: ArticlesDataSource,
    private val externalScope: CoroutineScope,
) {
    // As we want to complete bookmarking the article even if the user moves
    // away from the screen, the work is done creating a new coroutine
    // from an external scope
    suspend fun bookmarkArticle(article: Article) {
        externalScope.launch { articlesDataSource.bookmarkArticle(article) }
            .join() // Wait for the coroutine to complete
    }
}
```

`externalScope` should be created and managed by a class that lives longer than the current screen, it could be managed by the `Application` class or a `ViewModel` scoped to a navigation graph.

Inject TestDispatchers in tests

An instance of `TestDispatcher` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-test-dispatcher/index.html) should be injected into your classes in tests. There are two available implementations in the `kotlinx-coroutines-test` library (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/):

- `StandardTestDispatcher` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-standard-test-dispatcher.html): Queues up coroutines started on it with a scheduler, and executes them when the test thread is not busy. You can suspend the test thread to let other queued coroutines run using methods such as `advanceUntilIdle` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/advance-until-idle.html).
- `UnconfinedTestDispatcher` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-unconfined-test-dispatcher.html): Runs new coroutines eagerly, in a blocking way. This generally makes writing tests easier, but gives you less control over how coroutines are executed during the test.

See the documentation of each dispatcher implementation for additional details.

To test coroutines, use the `runTest` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/run-test.html) coroutine builder. `runTest` uses a `TestCoroutineScheduler` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-test-coroutine-scheduler/index.html) to skip delays in tests and to allow you to control virtual time. You can also use this scheduler to create additional test dispatchers as needed.

```
class ArticlesRepositoryTest {

    @Test
    fun testBookmarkArticle() = runTest {
        // Pass the testScheduler provided by runTest's coroutine scope to
        // the test dispatcher
        val testDispatcher = UnconfinedTestDispatcher(testScheduler)
```

```
        val articlesDataSource = FakeArticlesDataSource()
        val repository = ArticlesRepository(
            articlesDataSource,
            testDispatcher
        )
        val article = Article()
        repository.bookmarkArticle(article)
        assertThat(articlesDataSource.isBookmarked(article)).isTrue()
    }
}
```

All `TestDispatchers` should share the same scheduler. This allows you to run all your coroutine code on the single test thread to make your tests deterministic. `runTest` will wait for all coroutines that are on the same scheduler or are children of the test coroutine to complete before returning.

Note: The above works best if no other `Dispatchers` are used in the code under test. This is why it's not recommended to hardcode `Dispatchers` in your classes.

Note: The coroutine testing APIs changed significantly in kotlinx.coroutines 1.6.0. See the migration guide (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-test/MIGRATION.md) if you need to migrate from the previous testing APIs.

Avoid GlobalScope

This is similar to the *Inject Dispatchers* best practice. By using `GlobalScope` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html), you're hardcoding the `CoroutineScope` that a class uses bringing some downsides with it:

- Promotes hard-coding values. If you hardcode `GlobalScope`, you might be hard-coding `Dispatchers` as well.

- Makes testing very hard as your code is executed in an uncontrolled scope, you won't be able to control its execution.
- You can't have a common `CoroutineContext` to execute for all coroutines built into the scope itself.

Instead, consider injecting a `CoroutineScope` for work that needs to outlive the current scope. Check out the Creating coroutines in the business and data layer section (#create-coroutines-data-layer) to learn more about this topic.

```
// DO inject an external scope instead of using GlobalScope.
// GlobalScope can be used indirectly. Here as a default parameter makes sens
class ArticlesRepository(
    private val articlesDataSource: ArticlesDataSource,
    private val externalScope: CoroutineScope = GlobalScope,
    private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
    // As we want to complete bookmarking the article even if the user moves
    // away from the screen, the work is done creating a new coroutine
    // from an external scope
    suspend fun bookmarkArticle(article: Article) {
        externalScope.launch(defaultDispatcher) {
            articlesDataSource.bookmarkArticle(article)
        }
            .join() // Wait for the coroutine to complete
    }
}

// DO NOT use GlobalScope directly
class ArticlesRepository(
    private val articlesDataSource: ArticlesDataSource,
) {
    // As we want to complete bookmarking the article even if the user moves
    // from the screen, the work is done creating a new coroutine with Global
    suspend fun bookmarkArticle(article: Article) {
        GlobalScope.launch {
            articlesDataSource.bookmarkArticle(article)
        }
            .join() // Wait for the coroutine to complete
    }
}
```

Learn more about `GlobalScope` and its alternatives in the Coroutines & Patterns for work that shouldn't be cancelled blog post (https://medium.com/androiddevelopers/coroutines-patterns-for-work-that-shouldnt-be-cancelled-e26c40f142ad).

Make your coroutine cancellable

Cancellation in coroutines is cooperative, which means that when a coroutine's `Job` is cancelled, the coroutine isn't cancelled until it suspends or checks for cancellation. If you do blocking operations in a coroutine, make sure that the coroutine is *cancellable*.

For example, if you're reading multiple files from disk, before you start reading each file, check whether the coroutine was cancelled. One way to check for cancellation is by calling the `ensureActive` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/ensure-active.html) function.

```
someScope.launch {
    for(file in files) {
        ensureActive() // Check for cancellation
        readFile(file)
    }
}
```

All suspend functions from `kotlinx.coroutines` such as `withContext` and `delay` are cancellable. If your coroutine calls them, you shouldn't need to do any additional work.

For more information about cancellation in coroutines, check out the Cancellation in coroutines blog post (https://medium.com/androiddevelopers/cancellation-in-coroutines-aa6b90163629).

Watch out for exceptions

Unhandled exceptions thrown in coroutines can make your app crash. If exceptions are likely to happen, catch them in the body of any coroutines created with `viewModelScope`

or `lifecycleScope`.

```
class LoginViewModel(
    private val loginRepository: LoginRepository
) : ViewModel() {

    fun login(username: String, token: String) {
        viewModelScope.launch {
            try {
                loginRepository.login(username, token)
                // Notify view user logged in successfully
            } catch (exception: IOException) {
                // Notify view login attempt failed
            }
        }
    }
}
```

Caution: To enable coroutine cancellation, don't consume exceptions of type **`CancellationException`** (don't catch them, or always rethrow them if caught). Prefer catching specific exception types like **`IOException`** over generic types like **`Exception`** or **`Throwable`**.

For more information, check out the blog post Exceptions in coroutines (https://medium.com/androiddevelopers/exceptions-in-coroutines-ce8da1ec060c), or Coroutine exceptions handling (https://kotlinlang.org/docs/exception-handling.html) in the Kotlin documentation.

Learn more about coroutines

For more coroutines resources, see the Additional resources for Kotlin coroutines and flow (/kotlin/coroutines/additional-resources) page.

Last updated 2023-03-01 UTC.