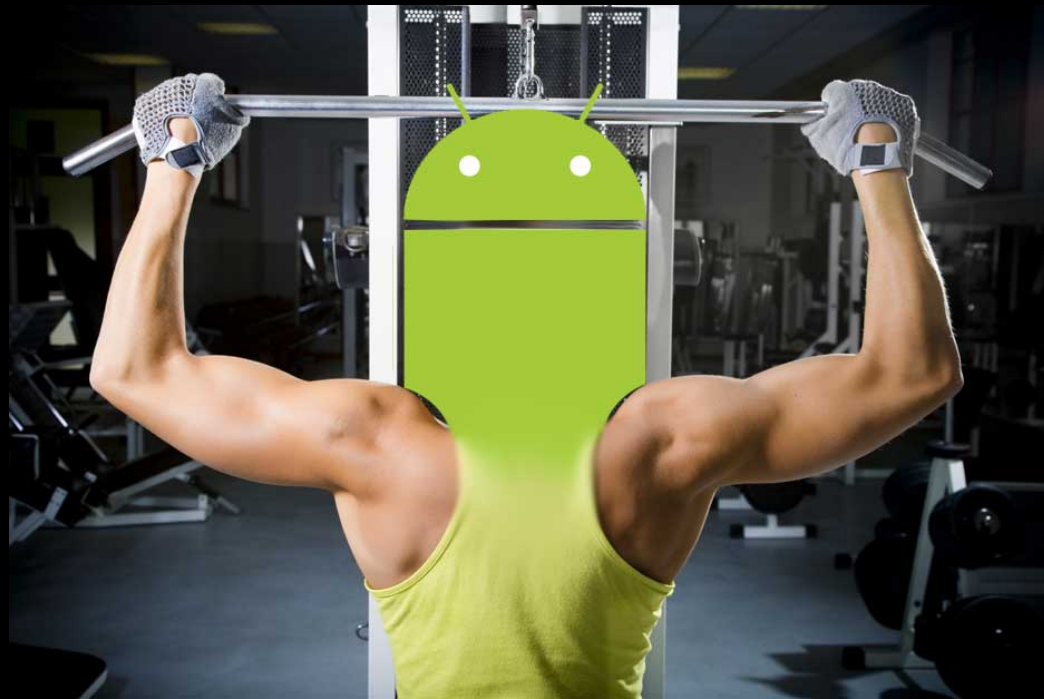


MVRX: MVVM with steroids



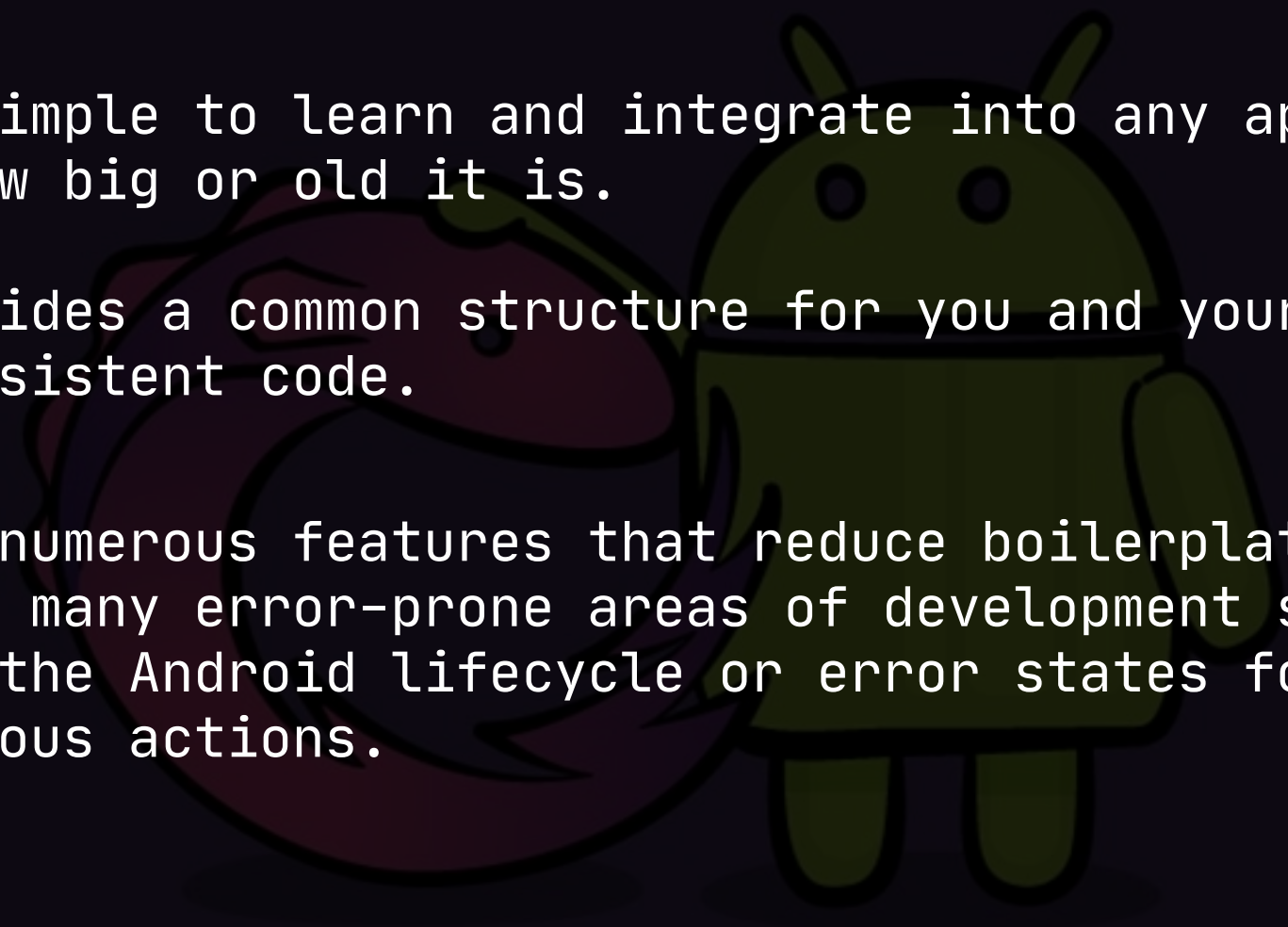
What is MVRX? (Mavericks)



About



- MvRx is simple to learn and integrate into any app no matter how big or old it is.
- MvRx provides a common structure for you and your team to write consistent code.
- MvRx has numerous features that reduce boilerplate and eliminate many error-prone areas of development such as managing the Android lifecycle or error states for asynchronous actions.



We will



- Learn how to integrate MvRx into an existing app
- Learn how Jetpack's ViewModel architecture solves key lifecycle and data challenges
- Learn how to model complex state with simple immutable data classes
- Write unit tests without worrying about lifecycles or robolectric



Appendix

- MvRx is Kotlin first and Kotlin only
- data classes, and receiver types

MvRx is built on top of the following existing technologies and concepts:

- Kotlin
- Android Architecture Components
- RxJava
- React (conceptually)
- [Epoxy](#) (optional)

Review

Data class

```
data class User(val name:  
String, val age: Int)
```

Receiver types

```
val sum: Int.(Int) → Int =  
other → plus(other) }
```

```
5.sum(5)  
10
```



This is what it looks like:

```
data class HelloWorldState(val title: String = "Hello World") : MvRxState

/**
 * Refer to the wiki for how to set up your base ViewModel.
 */
class HelloWorldViewModel(initialState: HelloWorldState) :
    MyBaseMvRxViewModel<HelloWorldState>(initialState, debugMode = BuildConfig.DEBUG) {
    fun getMoreExcited() = setState { copy(title = "$title!") }
}

class HelloWorldFragment : BaseFragment() {
    private val viewModel: HelloWorldViewModel by fragmentViewModel()

    override fun EpoxyController.buildModels() = withState(viewModel) { state →
        header {
            title(state.title)
        }
        basicRow {
            onClick { viewModel.getMoreExcited() }
        }
    }
}
```

Core Concepts



Core concepts



State

MvRxState is an interface that you should extend with an immutable Kotlin data class. Your ViewModel will be generic on this state class. State can only be modified inside of a MvRxViewModel but you can observe it anywhere.

Immutability

MvRxState is forced to be immutable. If debug mode is on in your ViewModel, your state properties will be explicitly checked for immutability and will throw an exception if they are not. State should be mutated with Kotlin data class [copy](#)



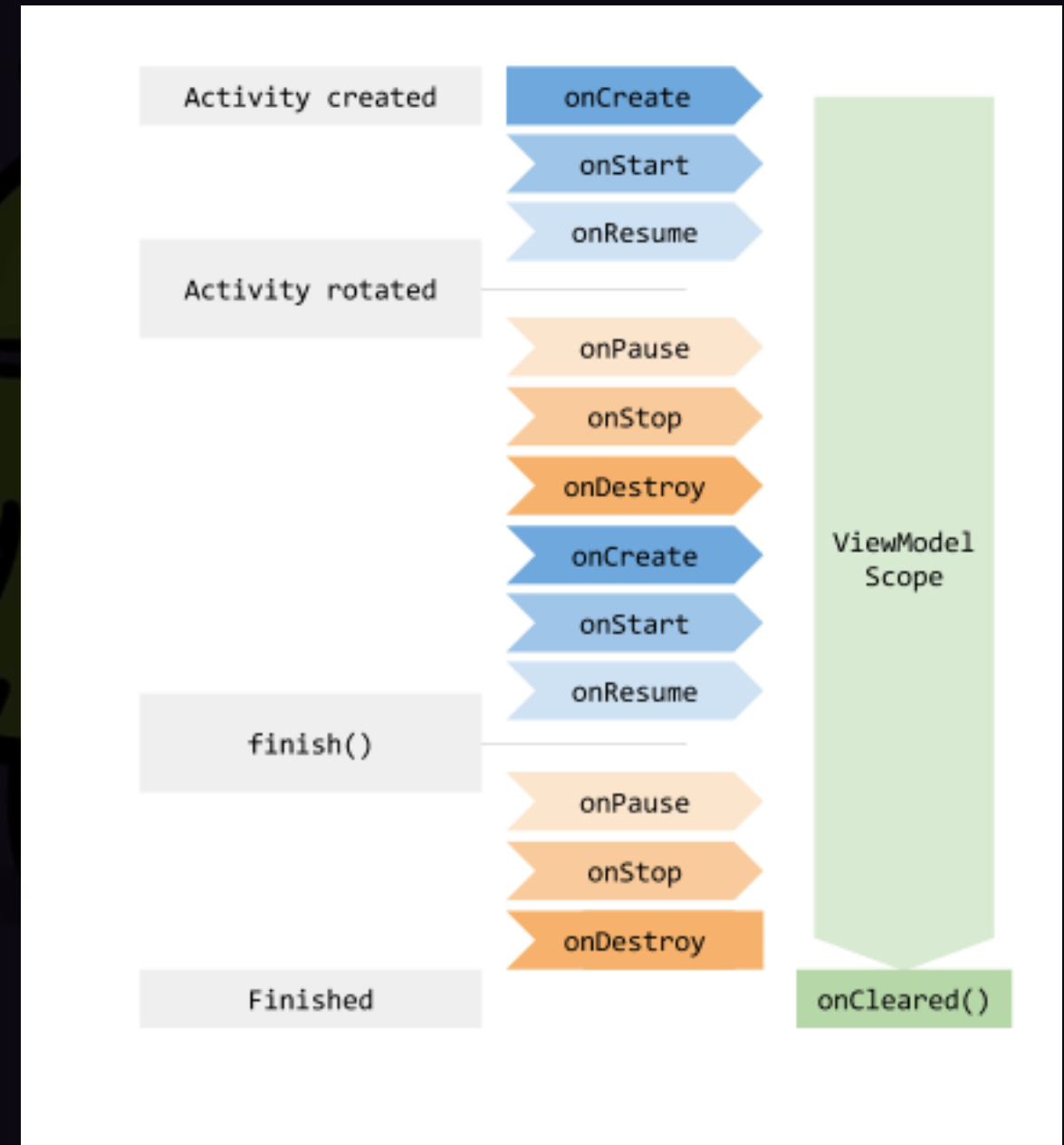
State creation

MvRx will create the initial state for your ViewModel automatically. This will use the default constructor if possible. However, if you need to pass some default state such as an id, you can create a secondary constructor that takes a *parcelable args class*. MvRx will automatically look for the key `MvRx.KEY_ARG` in the arguments in the Fragment that created the ViewModel and call the secondary constructor of the state class like this:

```
data class MyState(val foo: Int) : MvRxState {  
    constructor(args: MyArgs) : this(args.foo)  
}
```

ViewModel

A `MvRxViewModel` extends Google's own `ViewModel`. ViewModels make Android development dramatically simpler because they are tied to the "logical lifecycle" of your screen rather than the "view lifecycle". While Fragments, Views, and Activities get recreated on configuration changes, ViewModels do not. MvRx uses Kotlin delegates to either create a new ViewModel or return the existing instance that was already created. This lifecycle diagram clearly illustrates how ViewModels simplify lifecycles on Android:



ViewModel creation

A ViewModel can be created in two ways:

1. If your ViewModel requires no external dependencies, create a single argument constructor with just state:
2.

```
class MyViewModel(initialState: MyState) :
    BaseMvRxViewModel(initialState, debugMode =
    true)
```


If your ViewModel requires external dependencies, or any constructor arguments besides initial state, have the ViewModel's companion object implement `MvRxViewModelFactory`:

```
class MyViewModel(initialState: MyState, dataStore:
    DataStore) : BaseMvRxViewModel(initialState,
    debugMode = true) {
    companion object :
        MvRxViewModelFactory<MyViewModel, MyState> {
        override fun create(viewModelContext:
            ViewModelContext, state: MyState): MyViewModel {
            val dataStore = if (viewModelContext is
                FragmentViewModelContext) {
                // If the ViewModel has a fragment
                scope it will be a FragmentViewModelContext, and you
                can access the fragment.
                viewModelContext.fragment.inject()
            }
            return MyViewModel(state, dataStore)
        }
    }
}
```

State factory

An alternative way of creating initial state is to override `initialState` in your `ViewModelFactory`.

1. If the `initialState` function returns a non-null state, it will be used over the state's default, or secondary arg constructor.



```
class MyViewModel(initialState: MyState, datastore:
DataStore) : BaseMvRxViewModel(initialState, debugMode =
true) {
    companion object : MvRxViewModelFactory<MyViewModel,
MyState> {

        override fun initialState(viewModelContext:
ViewModelContext): MyState? {
            // Args are accessible from the context.
            // val foo = viewModelContext.args<MyArgs>.foo

            // The owner is available too, if your state needs a value
            // stored in a DI component, for example.
            val foo = viewModelContext.activity.inject()
            return MyState(foo)
        }
    }
}
```

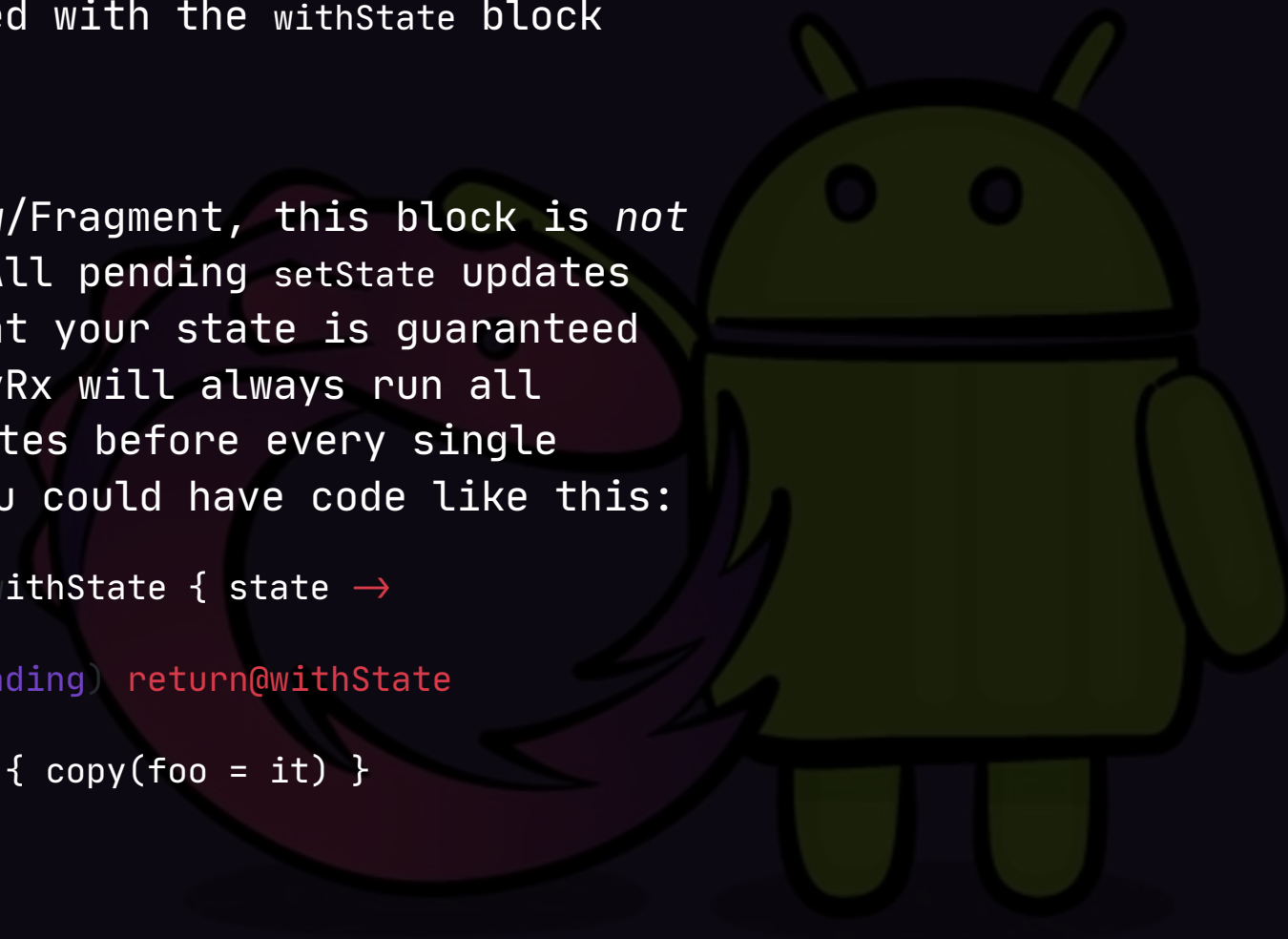
Accessing State

State can be accessed with the `withState` block like:

```
withState { state → }
```

Unlike in a `MvRxView/Fragment`, this block is *not run synchronously*. All pending `setState` updates are run first so that your state is guaranteed to be up to date. `MvRx` will always run all pending `setState` updates before every single `withState` block so you could have code like this:

```
fun fetchSomething() = withState { state →  
    if (state.foo is Loading) return@withState  
    MyRequest().execute { copy(foo = it) }  
}
```



Updating State

ViewModels are the only object that can modify state. To do so, they can call `setState`. The `setState` block is called a reducer and is called with the current state as the receiver type of the block and returns the new state like this:

```
setState { copy(changed = true) }
```

For clarity, a verbose version of the API would look like:

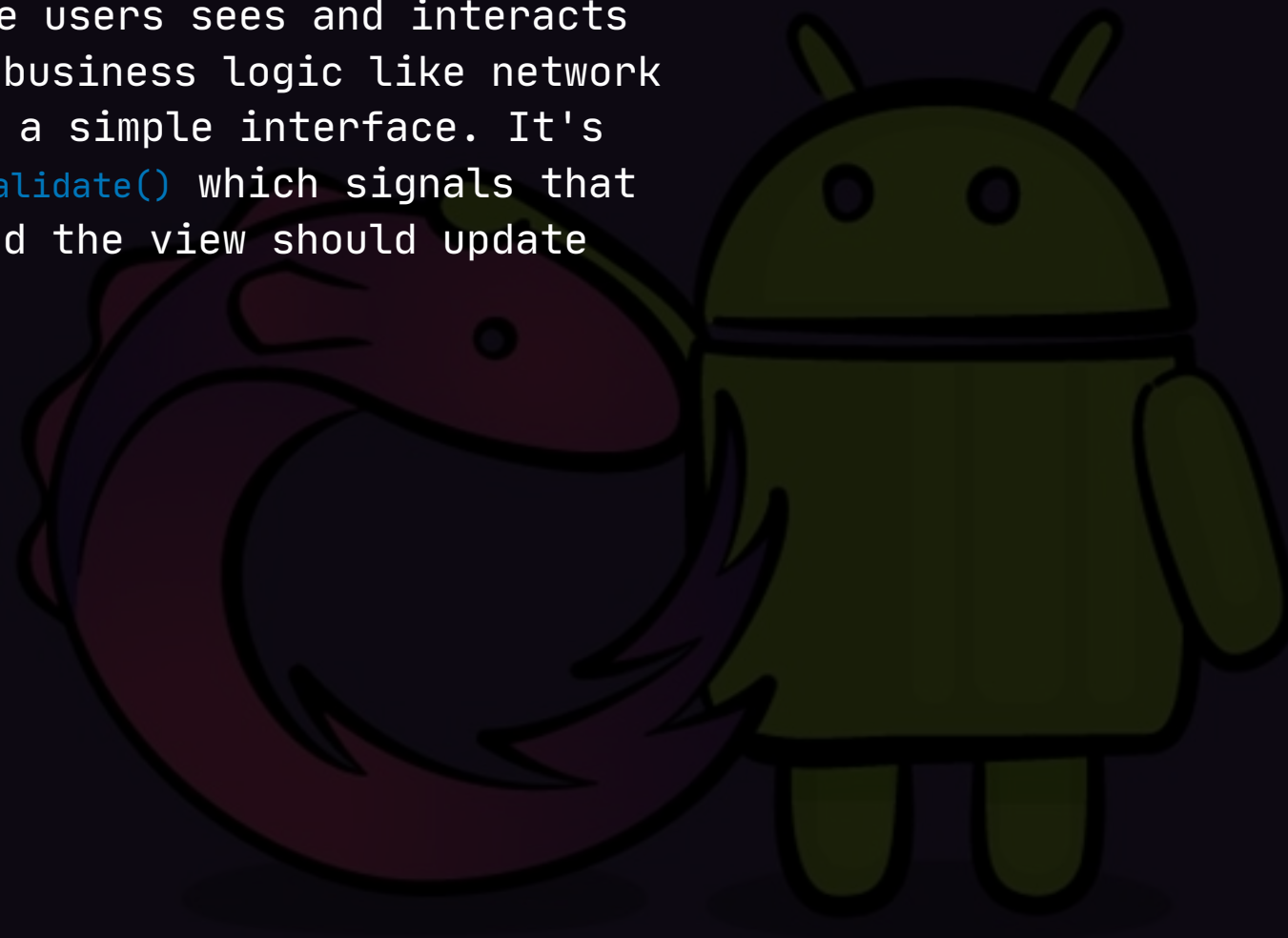
```
setState { currentState →  
    return currentState.copy(changed = true)  
}
```

The MvRx syntax leverages Kotlin [receiver types](#) and copy from [Kotlin data classes](#).

Conceptually, this is extremely similar to React's `setState` function including the fact that it isn't run synchronously

View

A `MvRxView` is what the users sees and interacts with. It is free of business logic like network requests. `MvRxView` is a simple interface. It's main function is `invalidate()` which signals that state has changed and the view should update



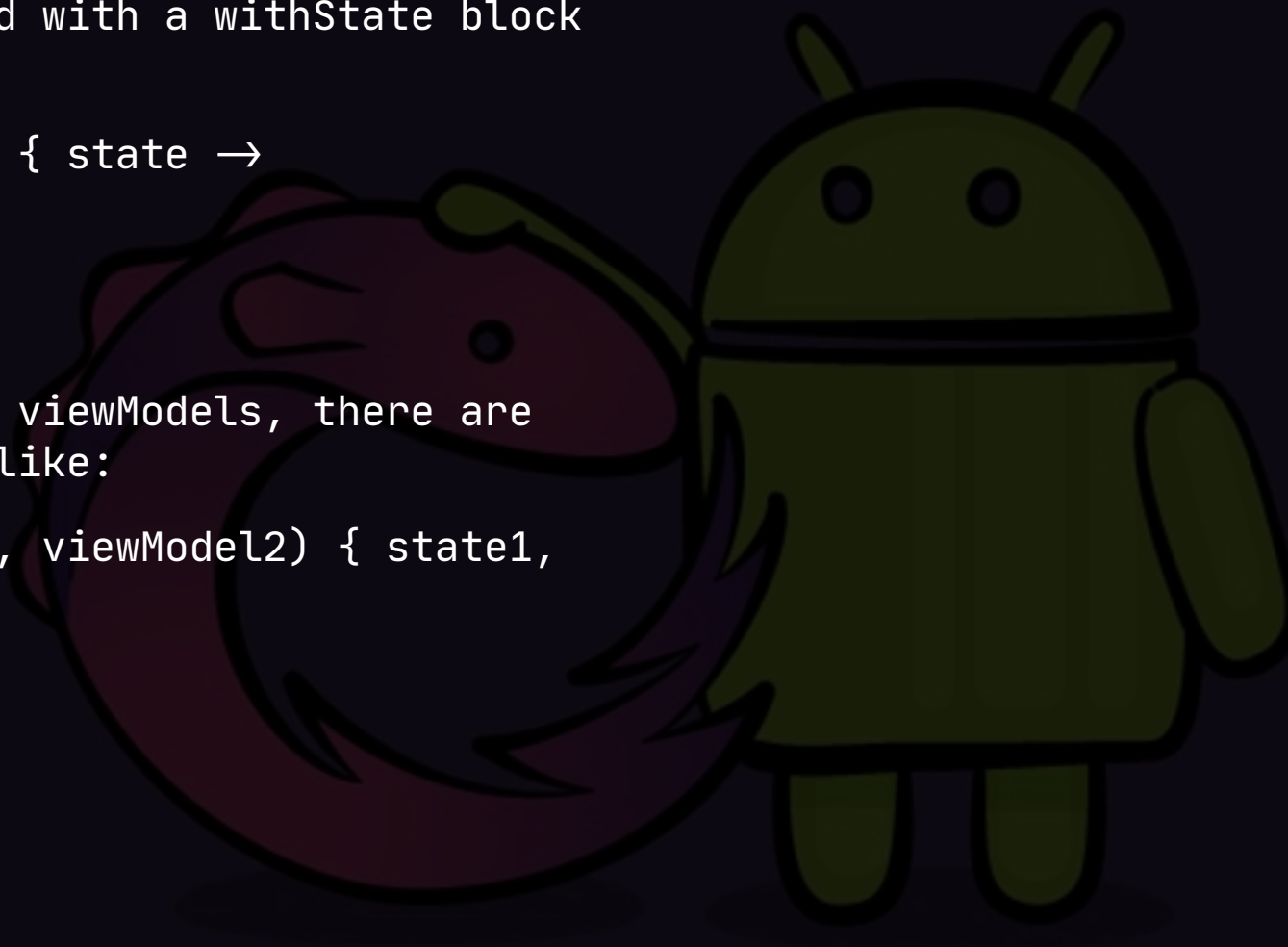
Accessing state

State can be accessed with a `withState` block such as:

```
withState(viewModel) { state →  
    ...  
}
```

If you have multiple `viewModels`, there are overloaded versions like:

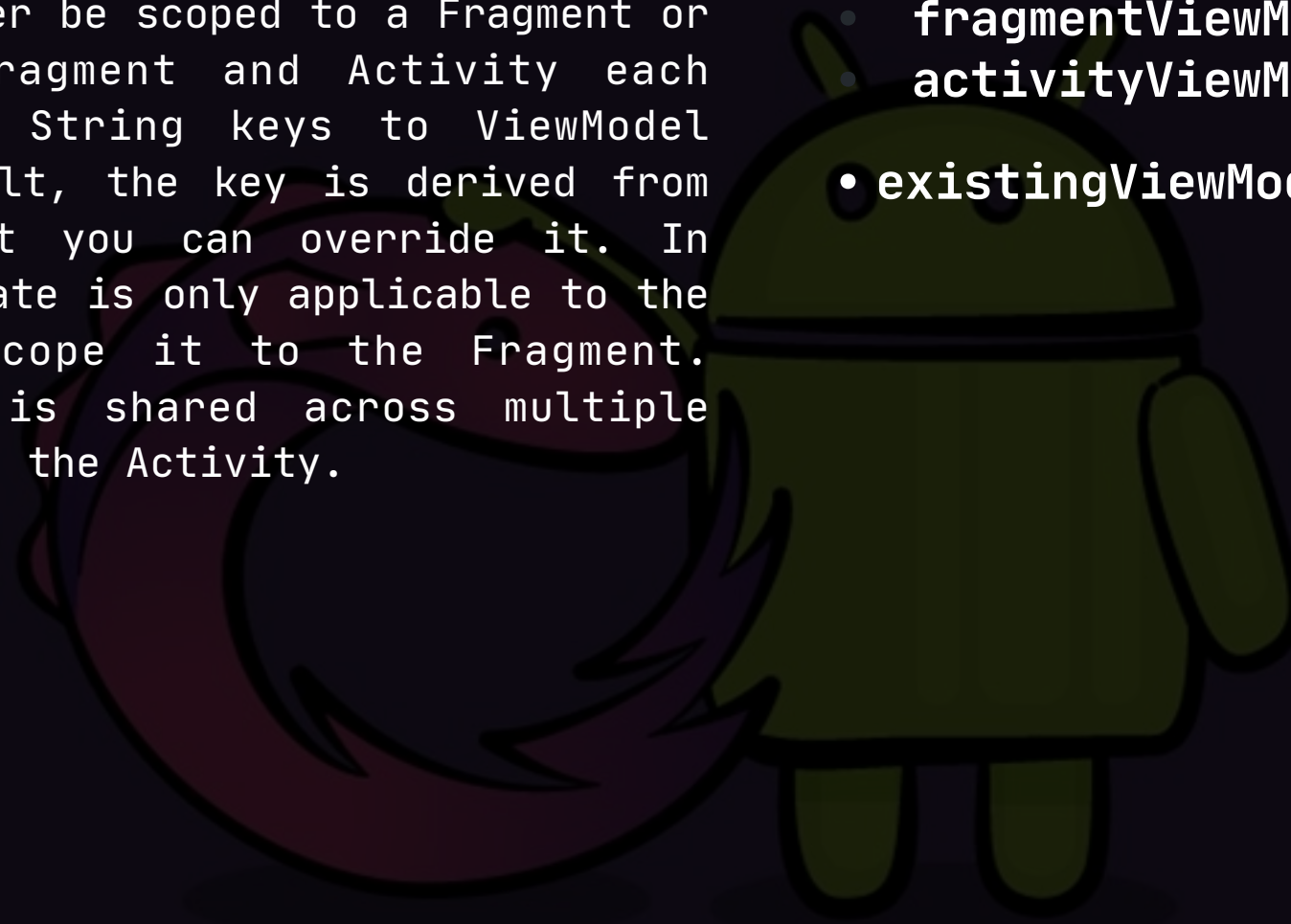
```
withState(viewModel1, viewModel2) { state1,  
    state2 → }
```



ViewModel Scope

ViewModels can either be scoped to a Fragment or an Activity. A Fragment and Activity each contain a map of String keys to ViewModel instances. By default, the key is derived from the class name but you can override it. In general, if your state is only applicable to the current screen, scope it to the Fragment. However, if data is shared across multiple screens, scope it to the Activity.

- `fragmentViewMode`
- `activityViewModel`
- `existingViewModel`



Async<T>

MvRx makes dealing with async requests like fetching from a network, database, or anything that can be mapped to an observable incredibly easy. MvRx includes Async<T>. Async is a **sealed class** with four subclasses:

- Uninitialized
- Loading
- Success
- Fail
 - Fail has an error property to retrieve the failure.

You can directly call an Async object and it will return either the value if it is Success or null otherwise thanks to Kotlin's **invoke operator**. For example:

```
var foo = Loading()
println(foo()) // null
foo = Success<Int>(5)
println(foo()) // 5
foo = Fail(IllegalStateException("bar"))
println(foo()) // null
```

Observable.execute {...}



MvRxViewModel ships with the following type extension: `Observable<T>.execute(reducer: S. (Async<V>) → S)`. This looks simple but its implications are powerful. When you call `execute` on an observable, it will subscribe to it, immediately emit `Loading`, and then will emit `Success` or `Fail` for `onNext` and `onError` events on the observable stream.

MvRx will automatically dispose of the subscription in `onCleared` of the `ViewModel` so you *never have to manage the lifecycle or unsubscribing*.

```
MyRequest.create(123).execute { copy(response = it) }
```

In this case:

- `MyRequest.create(123)` returns `Observable<MyResponse>`
- `execute` subscribes to the stream and immediately emits `Loading<MyResponse>`. It will then emit either `Success<MyResponse>` or `Fail<MyResponse>` depending on the result.
- For each emission, it will call the reducer. This is simple to `setState` in the `UpdatingStateSection`. Inside the block:
 - it is the `Async` value emitted.
 - `copy` is the `copy` function from Kotlin data classes.
 - The block implicitly returns the result of the `copy` function which is the updated state.

Testing

Every MvRx release ships with a mvr-x-testing artifact as well. The artifact ships with a MvRxTestRule which will run all MvRx reducers synchronously. This enables you to call ViewModel actions that update state and make assertions on the state right after. A simple test that increments a counter would look like this:

```
class CounterViewModelTest {  
    @Test  
    fun testIncrementCount() {  
        // Create your ViewModel in your test. The  
        // rule won't apply to a field-instantiated  
        // ViewModel.  
        val viewModel =  
            CounterViewModel(CounterState())  
        viewModel.incrementCount()  
        withState(viewModel) { state ->  
            assertEquals(1, state.count)  
        }  
    }  
  
    companion object {  
        @JvmField  
        @ClassRule  
        val mvrXTestRule = MvRxTestRule()  
    }  
}
```