
STOCARD

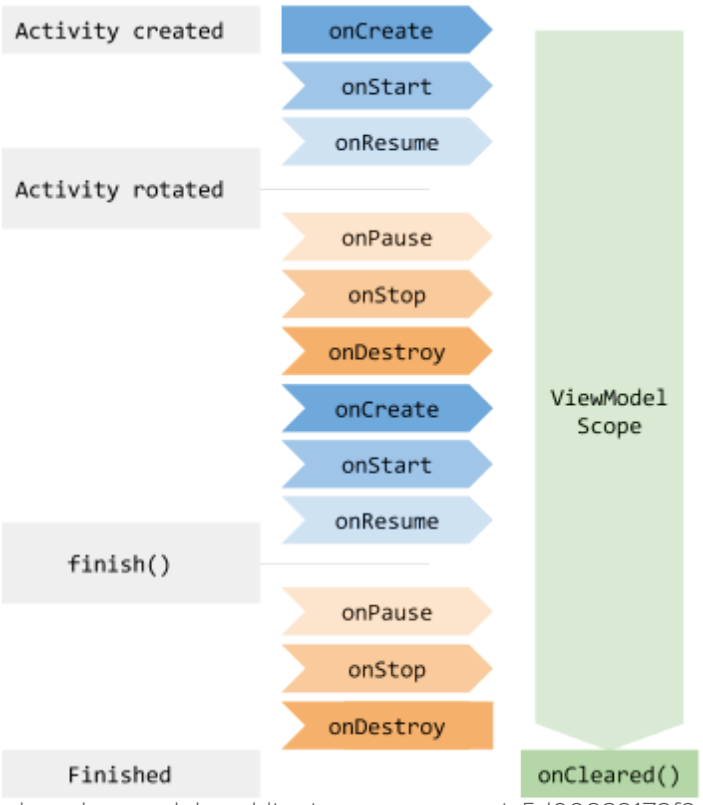
LiveData (+ Architecture)

LiveData

- Observable
- Lifecycle aware
  - Regarding the Observer (Activity/Fragment/Service)
  - Updates only when active
  - Cleanup after destroyed
- No more logic in the Activities/Fragments

ViewModel

- Provides LiveData to the Consumers
- Survives configuration changes
- Keeps the Activitities/Fragments lean

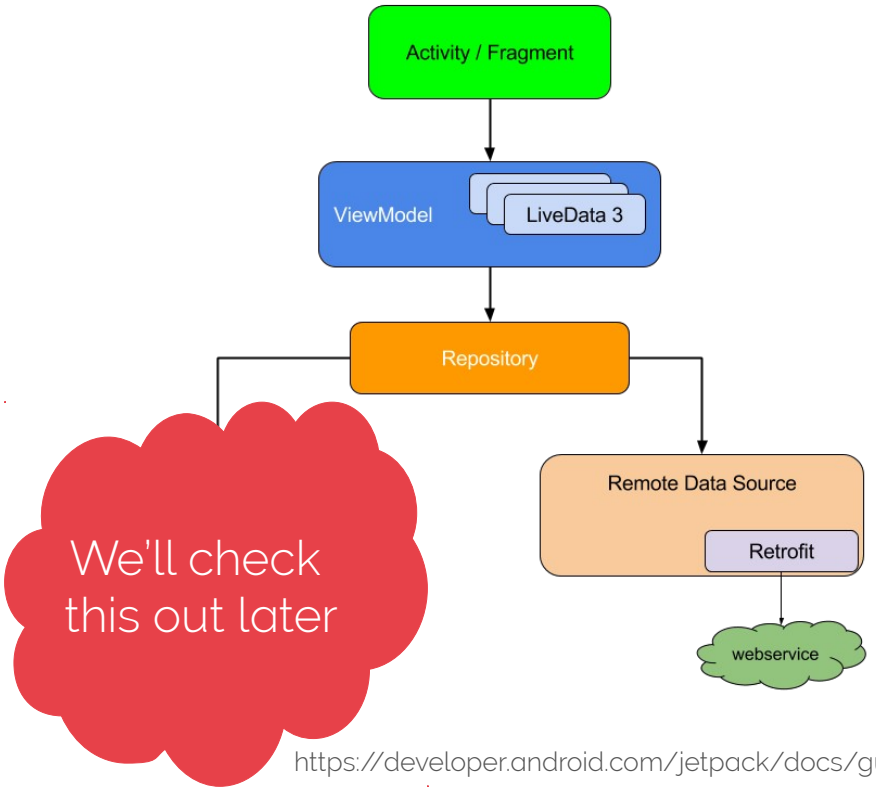


android.jlelse.eu/exploring-viewmodel-architecture-component-5d60828172f9

Repository

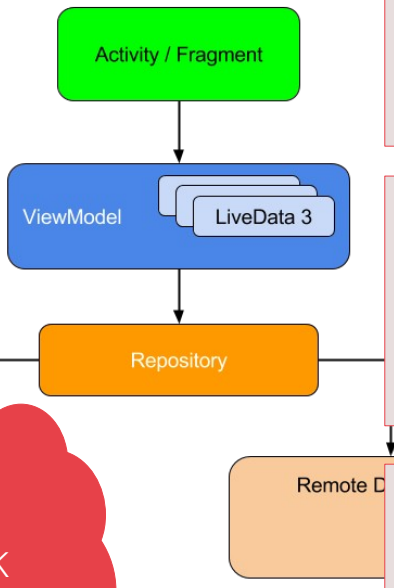
- Manages the Data
- Abstraction over where the data comes from
- Great for testing

Architecture



https://developer.android.com/jetpack/docs/guide

Architecture



We'll check this out later

https://developer.android.com/je

```
data class ChatViewState(
    val networkState: NetworkState,
    val messages: List<Message>
)
```

```
interface ChatViewModel {
    val viewState: LiveData<ChatViewState>
    fun send(text: String): LiveData<Boolean>
}
```

```
interface ChatRepository {
    val chatMessages: LiveData<List<Message>>
    val networkState: LiveData<NetworkState>
    fun send(text: String): LiveData<Boolean>
}
```

MainActivity

```
private val viewModel by lazy {
    ViewModelProviders.of(this).get(ChatViewModel::class.java)
}
override fun onCreate(savedInstanceState: Bundle?) {
    send_button.setOnClickListener {
        viewModel.send(input)
            .observe(this, Observer { success -> ... })
    }
    viewModel.viewState
        .observe(this, Observer { viewState :  -> ... })
}
```

ChatRepository

```
class ChatRepository {
    val messages = MutableLiveData<List<Message>>()
    val networkingState = MutableLiveData<NetworkState>()

    fun refresh() { ... }
    fun send(text: String) { ... }
}
```

ChatRepository - send

```
fun send(text: String): LiveData<Boolean> {
    val message = Message(..., message = text)
    val result = MutableLiveData<Boolean>()
    backend.postMessage(message).enqueue(
            onFailure = { result.postValue(false) },
            onResponse = {
                result.postValue(true)
                refresh()
            }
    )
    return result
}
```

ChatRepository - refresh

```
private fun refresh() {
    networkingState.postValue(NetworkState.REFRESHING)
    backend.fetchMessages().enqueue(
            onFailure = { error ->
                networkingState.postValue(NetworkState.ERROR)
            },
            onResponse = { response ->
                response.body()?.let { messages.postValue(it) }
                networkingState.postValue(NetworkState.DONE)
            }
    )
}
```

ChatViewModel

```
val combined = MediatorLiveData<ChatViewState>()
combined.addSource(chatRepository.messages) { messages ->
    val currentState = combined.value ?: initialState
    if (messages != null)
        combined.value = currentState.copy(messages = messages)
}
combined.addSource(chatRepository.networkState) { state ->
    val currentState = combined.value ?: initialState
    if (state != null)
        combined.value = currentState.copy(networkState = state)
}
```

One more thing – self updating Repository

- LiveData is lifecycle aware
  - Poll while a observer is active
  - Cleanup after the last observer is gone

```
private val messages = object : LiveData<List<Message>>() {
    // start the polling when the LiveData becomes active
    override fun onActive() { ... }
    // stop the polling when the LiveData becomes inactive
    override fun onInactive() { ... }
}
```

One more thing – self updating Repository

- But how to build something that
  - executes network requests periodically (poll the data)
  - executes in the background
  - can be canceled easily (after last subscriber is gone)
  - most importantly: fits on a single slide

One more thing – self updating Repository

```
val updater = GlobalScope.launch {
    while (true) { // just keep looping
        try {
            networkingState.postValue(NetworkState.REFRESHING)
            messages.postValue(backend.fetchMessages().await())
            networkingState.postValue(NetworkState.DONE)
        } catch (ex: Exception) {
            networkingState.postValue(NetworkState.ERROR)
        }
        delay(2, TimeUnit.SECONDS) // let's wait a bit
    }
}
```

```
updater.cancel() // stop the polling
```

One more thing – self updating Repository

- Kotlin Coroutines
  - Launched with `GlobalScope.launch{ }`
  - Function that can be suspended and resumed
  - No thread blocking
  - Suspended at `await()` and `delay()`
- Retrofit CoroutineCallAdapterFactory
  - `backend.fetchMessages().await()`
  - github.com/JakeWharton/retrofit2-kotlin-coroutines-adapter