Android Templates 3.0

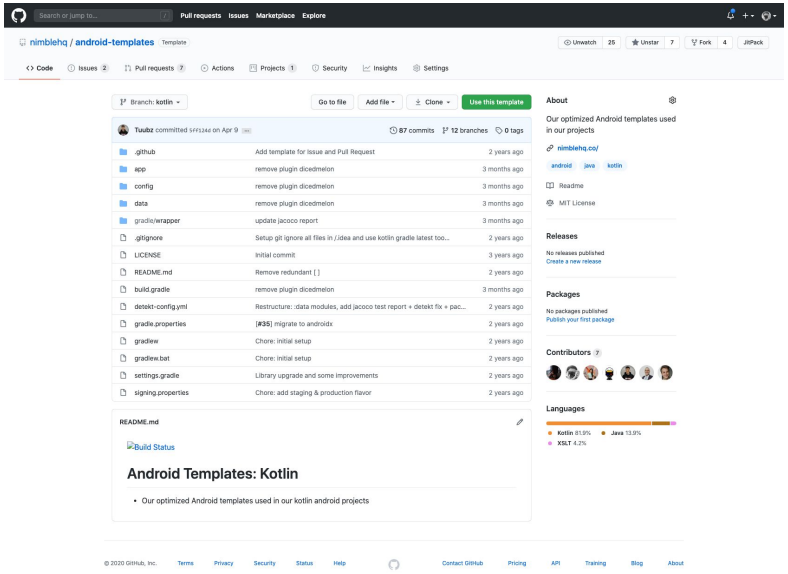
Lucas - Huey - Manh

Growth Session #29 - June 26 2020

Again, what is Android Templates?

A code template that includes all the **base architecture** components:

- Easy & convenient whenever we want to **start a new Android Project** 🔥
- Align projects and developers by using the same coding environment 🚀



What's planned for Android Template 3.0?

The idea is to **improve** our existing base template, as it was quite outdated:

Setup the right **code coverage tool**, by upgrading **Jacoco** ✅

→ Shows which parts of the code have not been or have been tested already, to increase test coverage

Setup an **Android bootstrap** functionality ✅

→ Align all our developers on the same code style, plugins, ...

Setup basic fastlane components with **Firebase app distribution** ✅

→ Make all our lives a lot more easy, as the app distribution part will all be automated

Update our **module structure** 🚫

→ Separate components in to multiple modules, to improve building time

Update our **MVVM** Architecture with **UseCase** 🚫

→ Enhance a cleaner code architecture and improve testability

Update our **Dependency Injection** according to a **single activity architecture** 🚫

→ Keep us up to date with the newest technologies: Navigation Component

Update our module structure

The advantages of modularisation?

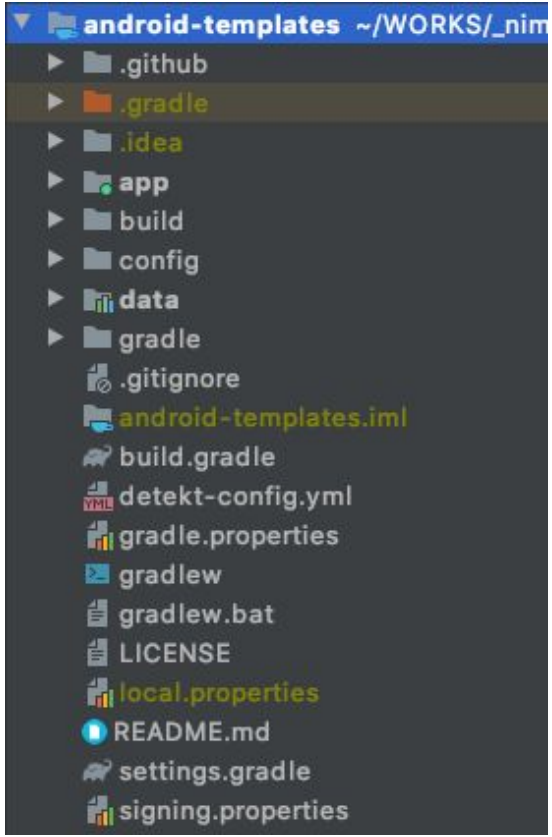
- Speeds up builds 🚀
- Enable on demand delivery 📦
- Simplify development 💻
- Reuse modules across apps 🧩
- Experiment with new technologies ✨
- Scale development teams 👨‍👨‍👧‍👧
- Enables refactoring 🧹
- Simplifies test automation 👮‍♀️

Update our module structure

Why do we need to update our module structure?

The current template has 2 modules: **app** and **data**.

The need of separating the business logics in **app** module into smaller modules.



Update our module structure

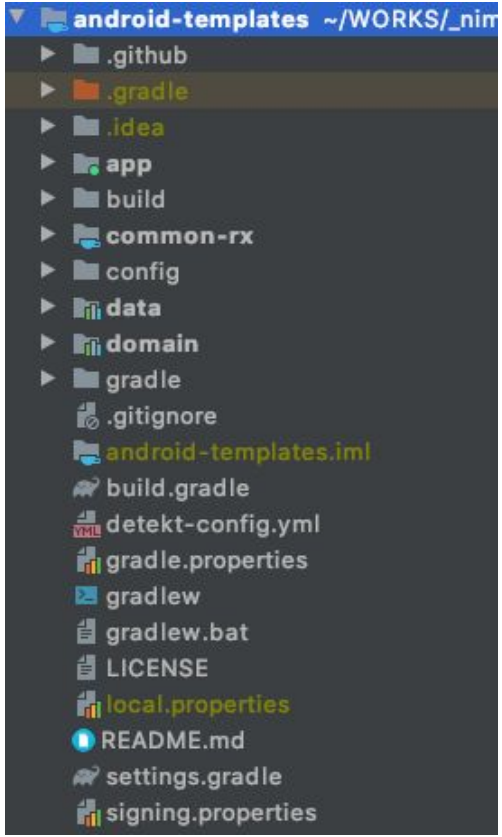
Introduce common-rx and domain modules

common-rx

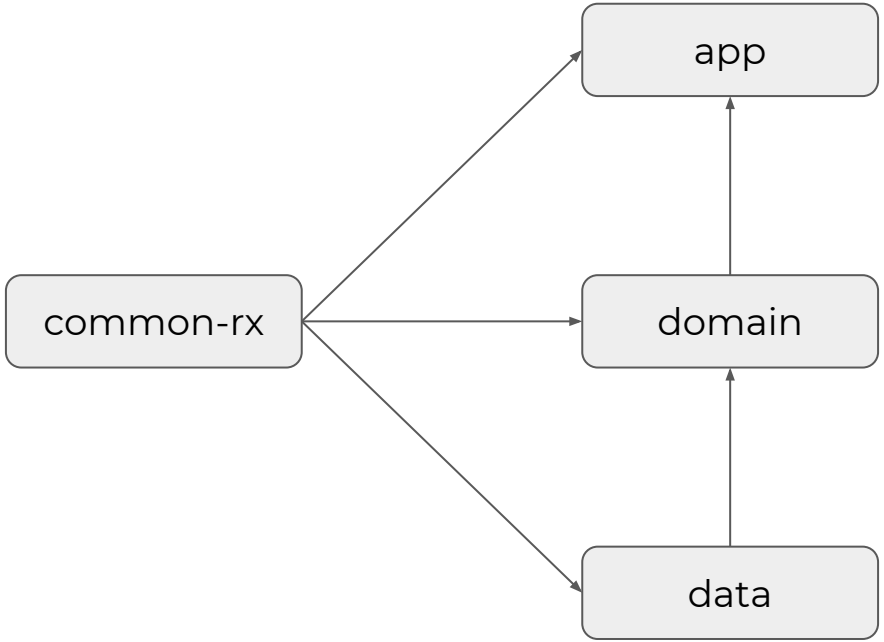
- Contains all common rx operations for all modules

domain

- Manages remote/local repositories, persistences
- Manages **UseCase** for app module



Update our module structure



Update our MVVM Architecture with UseCase

Old implementation, MVVM ready but...

- Use repository directly inside ViewModel along with Schedulers
- Why should NOT we call the repository directly from ViewModel?
  - Because we want to avoid **God ViewModel objects** that deals with both UI logic and dataflow logic.
  - We also want the dataflow logic to be reusable across different VMs.

Update our MVVM Architecture with UseCase

Old implementation, MVVM ready but...

```
private fun fetchApi(): Observable<List<String>> =
    repository
        .getExampleData()
        .subscribeOn(schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .map { exampleResponse : ExampleResponse ->
            exampleResponse.data.children.map { it.data.author }
        }
        .doOnSubscribe { isLoading.onNext( t: true) }
        .toObservable()
```

Update our MVVM Architecture with UseCase

New implementation

- Move dataflow logic into another layer: **Use Cases**
- Define a base which contains general **errors** and handle specific **exceptions**: IOException and StreamResetException
- Handle threads inside use cases
- Define some general returned objects for use cases based on using technology. In this case: **SingleUseCase**, **CompletableUseCase**, **FlowableUseCase** from Rxjava2

Update our MVVM Architecture with UseCase

Our UseCase:

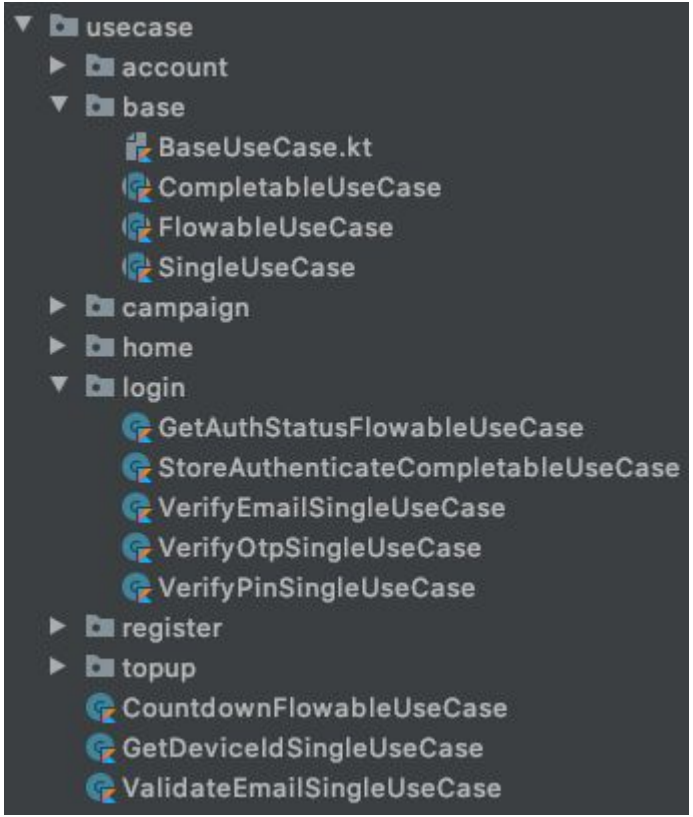
```
class GetAuthorsSingleUseCase @Inject constructor(
    rxSchedulersProvider: RxSchedulerProvider,
    private val repository: UserRepository
) : SingleUseCase<Unit, List<String>>(
    rxSchedulersProvider.io(),
    rxSchedulersProvider.main(),
    ::Ignored
) {

    override fun create(input: Unit): Single<List<String>> {
        return repository.getExampleDate()
            .map { exampleResponse : ExampleResponse ->
                exampleResponse.children.map { it.author }
            }
    }
}
```

Refactor VM:

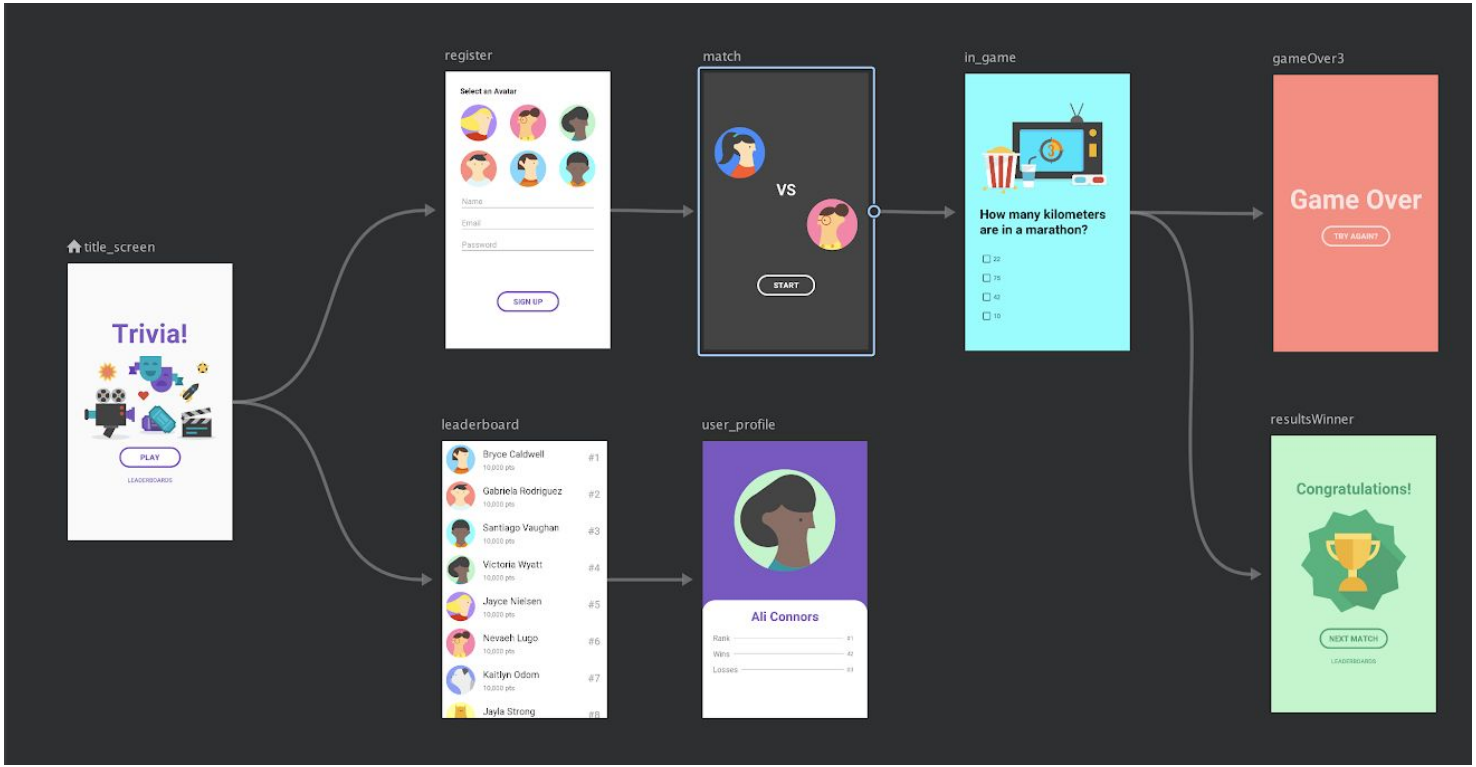
```
private fun fetchApi(): Observable<List<String>> =
    getAuthorsSingleUseCase.execute(Unit)
        .doOnSubscribe { it: Disposable!
            // TODO Integration - Show loading
        }
        .toObservable()
```

Update our MVVM Architecture with UseCase



Update to Single Activity Architecture

Why?



Update to Single Activity Architecture

How?

- @Scope: Scoping object instances
    - @ActivityScope (*for Activities*)
    - @FragmentScope (*for Fragments*)

- @Module - provide dependency instances:
    - Activity Module
    - Fragment Modules

- Navigator - defines a mechanism for navigating:
    - Navigate in app via NavController
    - @Binds navigator from Activity Module

Conclusion

- Many pull requests are waiting for review: https://github.com/nimblehq/android-templates/pulls
- Worked in an interesting and satisfying **pair programming** style with 3 people ⇒ Resulted in a better quality standard

What’s next?

- Update tests for all necessary components and modules
- Add CI integration: Bitrise, CircleCI ✅


Get more value with less effort

Thanks!

Contact Nimble

nimblehq.co

hello@nimblehq.co

Bangkok

399 Interchange 21 Sukhumvit Road, Unit #2402-03, Klong Toei, Wattana, Bangkok 10110, Thailand

Singapore

28C Stanley St, Singapore 068737

Hong Kong

20th Floor, Central Tower

28 Queen's Road, Central, Hong Kong

nimble