


Dinger



[stoyicker/dinger](https://github.com/stoyicker/dinger) 

What does it do?

Pretends to be Tinder and has a mirrored login flow

Self-triggered, automated swiping

Saves the profiles of whoever it swipes onto Firebase so I can play with the data

Stores them locally too for future offline goodness

High-level overview

Clean architecture

app and *data* start use cases as defined in *domain*

Everything is injected for loose coupling (A/B, unit testing)

Interfaces are used to perform **cross-module** injections

ContentProviders are used to plug in library modules to the application lifecycle for **initialization**, thus avoiding coupling with *app*

crash-reporter and *event-tracker* are injected as **sealed classes** to potentially support choosing from different underlying providers

domain

 interactor

 auth, alarm, autoswipe, like, recommendation

 *domain*  interactor

Defines use cases as reactive entities

Several base classes for different characteristics (use cases with and without result)

Disposability is enforced as anything can be interrupted at any point in time



Result (and, optionally, execution) contexts must be determined by the specific implementation

 *domain*  auth, alarm, autoswipe, like, recommendation






Define features

- One or more use cases
- Interfaces defining actions that need to be executed by the use cases
- Optionally, one or more models, if parameters/result are necessary

domain

-  Architecture - interactor (structure and requirements for a use case)
-  Functionalities - auth, alarm, autoswipe, like, recommendation (models and use cases which execute actual actions)

data

-  RootModule (provides context, database)
-  InitializationComponent (injects into the ContentProvider)
-  crash, event
-  network
-  account, alarm, autoswipe, tinder.auth,
tinder.like, tinder.recommendation

 *data*  crash, event

Providers for the corresponding service as extracted from the corresponding module

Implementation is provided through a sealed class, which hides whatever service is underneath

```
@Module
internal class FirebaseCrashReporterModule {
    @Provides
    @Singleton
    // The injected instance sees a CrashReporter, not Firebase/Fabric/whatnot
    fun instance() = CrashReporter.firebase()
}
```

 *data*  network

Client

Interceptors

Base classes for:






- Abstracted caching (RequestSource)
- Supporting parsing between business entities and transport objects (RequestFacade)

 *data*  account, alarm, autoswipe, tinder.*

Implement features

- Models for request/response if required (e.g. most network requests)
- Entities and additional models and database resolvers as required by Room if database is involved
- Components and modules for feature-scoped dependency injection
- Object mappers to and from *domain* classes
- RequestFacade and RequestSource implementations if network is necessary
- Optionally, passive use-case triggerers (like a scheduled JobIntentService)
- Optionally, an interface defining what needs to be tracked within the package, plus object mappers for its required data


data

-  RootModule (provides context, database)
-  InitializationComponent (injects into the ContentProvider)
-  Services - crash, event (corresponding trackers)
-  Architecture - network (client and configuration)
-  Features - account, alarm, autoswipe, tinder.*
(models, domain mappers, execution classes)



 MainApplication (dependency injection root)

 crash (same as in *data*)




 alarmbanner, home, login, splash

 *app*  alarmbanner, home, login, splash

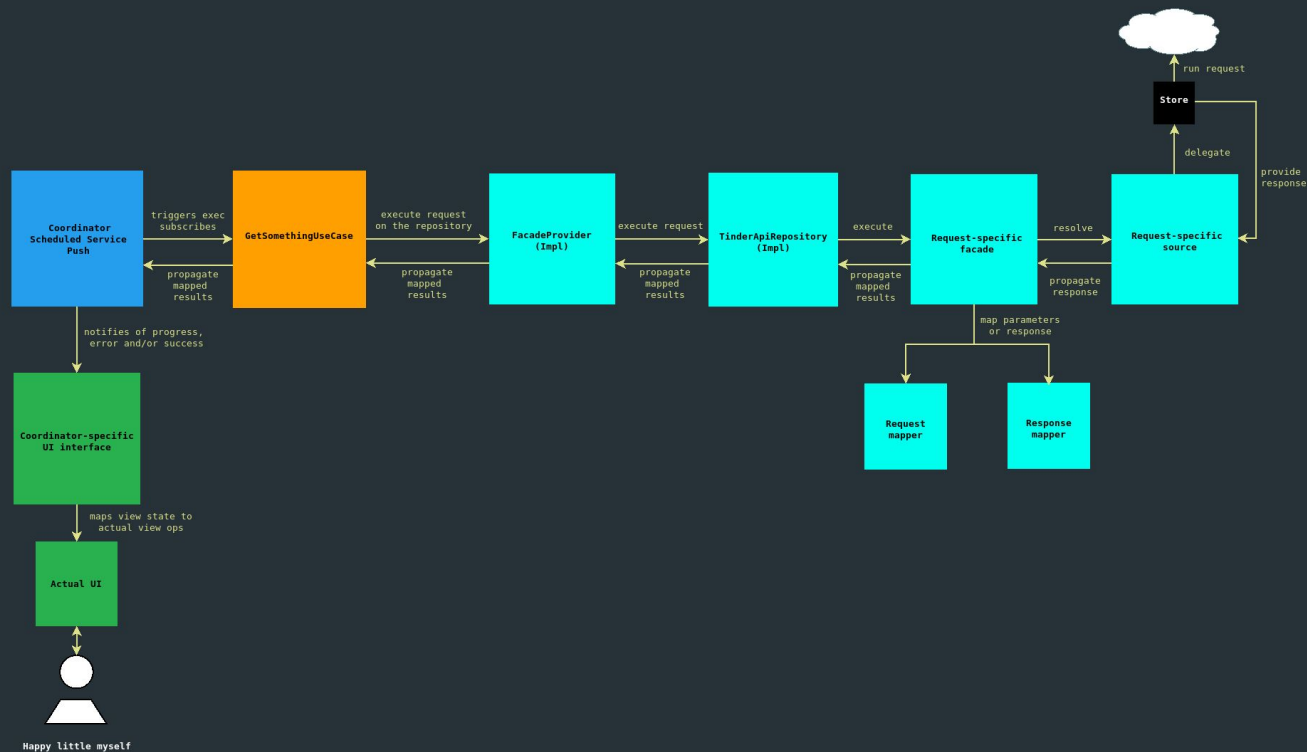
Bind UI to functionality

- DI classes for injecting what the package needs
- Activities are only used as routers between themselves and delegate other things to coordinators/features
- A coordinator acts as a bridge between the use case and updating the UI through an interface defining methods based on the possible states
- The implementation of such interface is what operates on the actual view
- A feature performs the role of a coordinator, but is named differently to state the fact that its functionality is handled by an external agent



-  MainApplication (fires dependency injection)
-  Service - crash (same as in *data*)
-  Features - alarmbanner, home, login, splash (di, coordinator/feature, opt. view interface + impl.)

Typical flow



crash-reporter

Defines what it can do through a sealed class

Offers instances to the available services (currently Firebase only)

```
interface CrashReporter {  
    fun report(throwable: Throwable)  
}  
  
abstract class CrashReporters {  
    companion object {  
        fun firebase(): CrashReporter = CrashReporterImpl.Firebase  
    }  
}  
  
internal sealed class CrashReporterImpl : CrashReporter {  
    /**  
     * Firebase is a singleton, but other crash reporters which can be instantiated  
     * with different ids would not be.  
     */  
    object Firebase : CrashReporterImpl() {  
        override fun report(throwable: Throwable) = FirebaseCrash.report(throwable)  
    }  
}
```

event-tracker is similar

Dinger



[stoyicker/dinger](https://github.com/stoyicker/dinger) 