


Dinger



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

High-level overview

Clean architecture

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

Everything is injected for loose coupling

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

 DomainHolder (proxy to actual action implementations)

 interactor

 repository

 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



Interfaces which encompass functionality groups

```
interface TinderApiRepository {  
    fun login(parameters: DomainAuthRequestParameters): Single<DomainAuthenticatedUser>  
  
    fun getRecommendations(): Single<Collection<DomainRecommendationUser>>  
  
    fun likeRecommendation(recommendation: DomainRecommendationUser): Single<DomainLikedRecommendationAnswer>  
}
```

Additional interface to group all functionality groups

```
interface FacadeProvider {  
    fun tinderApiRepository(): TinderApiRepository  
  
    // ...  
}
```





// TODO Break down into the feature packages instead!

 *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

-  DomainHolder (proxy to actual action implementations)
-  Architecture - interactor (structure and requirements for a use case)
-  Architecture - repository (interface with required actions)
-  Functionalities - auth, alarm, autoswipe, like, recommendation (models and use cases which execute actual actions)


data

 RootModule (provides context, database)

 ComponentHolder (proxy to components)

 crash, event

 network

 account, alarm, autoswipe, tinder.auth,
tinder.like, tinder.recommendation

// TODO Break down into the feature packages as well!

 *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:

- Defining request targets (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
- Object mappers to and from *domain* classes
- RequestFacade and RequestSource implementations if network is necessary; else something else (like a schedulable JobIntentService for auto-swiping)
- Optionally, an interface defining what needs to be tracked within the package, plus object mappers for its required data


data


 RootModule (provides context, database)

 ComponentHolder (proxy to components)

// TODO Break down into the feature packages as well!

 crash, event (provide corresponding trackers)


 network (client and its configuration)

 account, alarm, autoswipe, tinder.auth,
tinder.like, tinder.recommendation (features)



 MainApplication (fires dependency injection)

 crash (same as in *data*)




 alarmbanner, home, login, splash

 *app*  alarmbanner, home, login, splash

Bind UI to functionality

- Define a component that is responsible for injecting what the package needs and is initialized from `ApplicationComponent`
- 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)
-  crash (same as in *data*)
-  alarmbanner, home, login, splash (features)

crash-reporter

Defines what it can do through a sealed class

Offers instances to the available services (currently Firebase only)

```
sealed class CrashReporter {  
    abstract fun report(throwable: Throwable)  
  
    private class Firebase : CrashReporter() {  
        override fun report(throwable: Throwable) {  
            FirebaseCrash.report(throwable)  
        }  
    }  
  
    companion object {  
        // The explicit return hides implementation details  
        fun firebase(): CrashReporter = CrashReporter.Firebase()  
    }  
}
```

event-tracker is similar

Future work

Split module-wide components into the corresponding feature packages

Decouple the capabilities of event trackers and crash reporters from the sealed classes encompassing the services into interfaces which they will implement

Decoupled Gradle project

Consider Docker for CI

Product development

Dinger



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