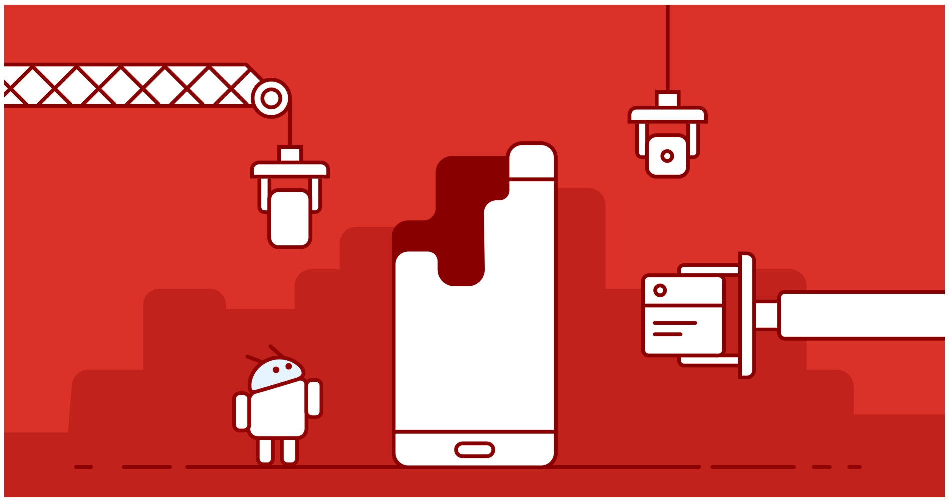
Android Architecture Part 4: Applying Clean Architecture on Android, Hands on (source code included)

June 29, 2017 Mihael Francekorić



In the last part of our Android Architecture series, we adjusted Clean Architecture a bit to the Android platform. We separated Android and the real world from our business logic, satisfied stakeholders and made everything easily testable.

The theory is great, but where do we start when we create a new Android project? Let's get our hands dirty with clean code and turn that blank canvas into an architecture.

Foundations

We will lay down the foundations first – create modules and establish dependencies among them to be on par with the dependency rule.

Those will be our modules, in order from the most abstract one to the concrete implementations:

1. domain

Entities, use cases, repositories interfaces, and device interfaces go into the domain module.

Ideally, entities and business logic should be platform agnostic. To feel safe,

and to prevent us from placing some android stuff in here, we will make it a pure **java** module

2. data

The data module should hold everything related to data persistence and manipulation. Here we will find DAOs, ORMs, SharedPreferences, network related stuff like Retrofit services and similar.

3. device

The device module should have everything related to Android that's not data persistence and UI. In example, wrapper classes for ConnectivityManager, NotificationManager and misc sensors.

We will make both Data and Device modules android modules, as they must know about Android and cannot be pure java.

4. The easiest part, app module (UI module)

This module is already created for you by the Android studio when you create a project.

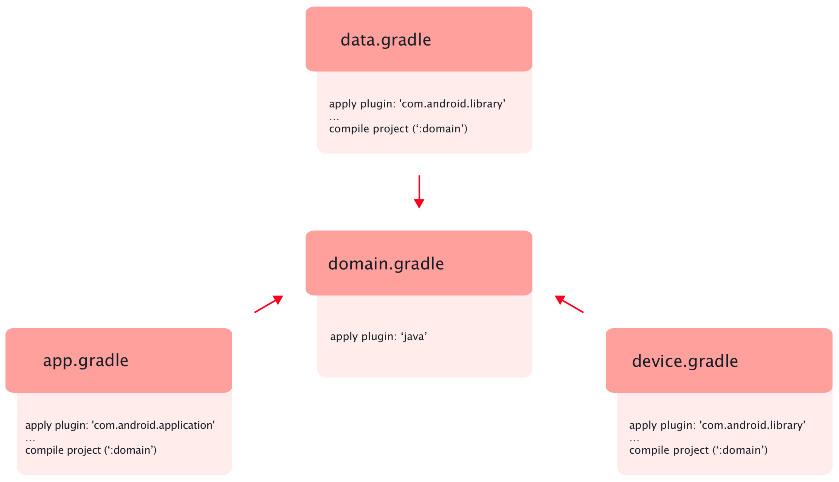
Here you can place all the classes related to the Android UI such as presenters, controllers, view models, adapters and views.

Dependencies

Dependency rule defines that concrete modules depend on the more abstract ones.

You might remember from the third part of this series that UI (*app*), DB – API (*data*) and Device (*device*) stuff is together in the outer ring. Meaning that they are on the same abstraction level. How do we connect them together then?

Ideally, these modules would depend only on the *domain* module. In that case, dependencies would look somewhat like a star:



```
apply plugin: 'com.android.library'
...
compile project (':domain')
```

```
apply plugin: 'java'
```

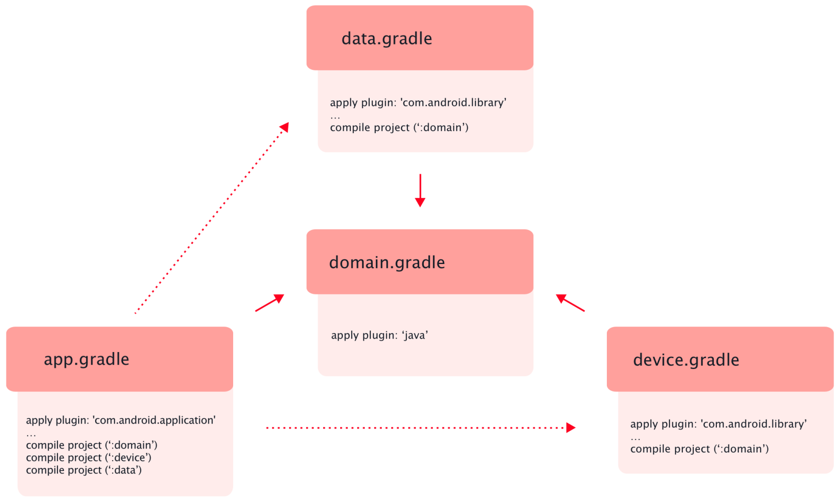
```
apply plugin: 'com.android.application'
...
compile project (':domain')
```

```
apply plugin: 'com.android.library'
...
compile project (':domain')
```

But, we are dealing with Android here and things just cannot be perfect. Because we need to create our object graph and initialize things, modules sometimes depend on an another module other than the domain.

For example, we are creating object graph for dependency injection in the *app* module. That forces *app* module to know about all of the other modules.

Our adjusted dependencies graph:


data.gradle
apply plugin: 'com.android.library'
...
compile project (':domain')
domain.gradle
apply plugin: 'java'
app.gradle
apply plugin: 'com.android.application'
...
compile project (':domain')
compile project (':device')
compile project (':data')
device.gradle
apply plugin: 'com.android.library'
...
compile project (':domain')

Bricks, a lot of bricks

Finally, it's time to write some code. To make things easier, we will take an RSS Reader app as an example. Our users should be able to manage their RSS feed subscriptions, fetch articles from a feed and read them.

Domain

let’s start with the domain layer and create our core business models and logic.

Our business models are pretty straightforward:

– *Feed* – holds RSS feed related data like the url, thumbnail URL, title and description

– *Article* – holds article related data like the article title, url and publication date

And for our logic, we will use UseCases, aka. Interactors. They encapsulate small parts of business logic in concise classes. All of them will implement general UseCase contract:

```
public interface UseCase<P> {

   interface Callback {

      void onSuccess();
      void onError(Throwable throwable);
    }

   void execute(P parameter, Callback callback);
 }
```

The very first thing our users will do when they open our app is add a new RSS feed subscription. So to start with our interactors, we will create *AddNewFeedUseCase* and its helpers to handle feed addition and validation logic.

AddNewFeedUseCase will use *FeedValidator* to check feed URL validity, and we will also create the *FeedRepository* contract which will provide our business logic some basic CRUD capabilities to manage feed data:

```
public interface FeedRepository {

    int createNewFeed(String feedUrl);

    List<Feed> getUserFeeds();

    List<Article> getFeedArticles(int feedId);

    boolean deleteFeed(int feedId);
}
```

Note how our **naming** in the domain layer clearly propagates the idea of what our app is doing.

Put everything together, our *AddNewFeedUseCase* looks like this:

```
public final class AddNewFeedUseCase implements UseCase

    private final FeedValidator feedValidator;
    private final FeedRepository feedRepository;

    @Override
    public void execute(final String feedUrl, final Call
        if (feedValidator.isValid(feedUrl)) {
            onValidFeedUrl(feedUrl, callback);
        } else {
            callback.onError(new InvalidFeedUrlException
        }
    }

    private void onValidFeedUrl(final String feedUrl, fi
        try {
            feedRepository.createNewFeed(feedUrl);
            callback.onSuccess();
        } catch (final Throwable throwable) {
            callback.onError(throwable);
        }
    }
}
```

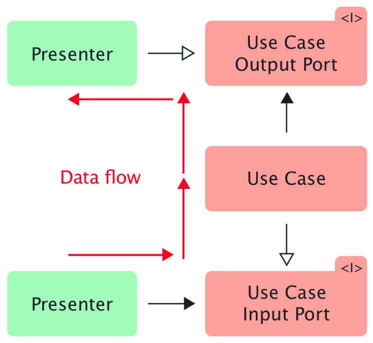
*Constructors are omitted for the sake of brevity.

Now, you might be wondering, why is our use case, as well as our callback, an interface?

To demonstrate our next problem better, let’s investigate *GetFeedArticlesUseCase.*

It takes a feedId -> fetches feed articles via *FeedRespository* -> returns feed articles

Here comes the data flow problem, the use case is in between presentation and data layer. How do we establish communication between layers? Remember those input and output ports?



Our use case must implement the input port (interface). Presenter calls the method on the use case, and the data flows to the use case (feedId). Use case maps feedId to feed articles and wants to send them back to the presentation layer. It has a reference to the output port (Callback), as the output port is defined in the same layer, so it calls a method on it. Hence, data goes to the output port – presenter.

We will tweak our UseCase contracts a bit:

```
public interface UseCase<P, R> {

   interface Callback<R> {
       void onSuccess(R return);
       void onError(Throwable throwable);
   }

   void execute(P parameter, Callback<R> callback);
}
```

```
public interface C

   interface Callb
       void onSucc
       void onErro
   }

   void execute(P 
}
```

UseCase interfaces are Input ports, and *Callback* interfaces are output ports.

GetFeedArticlesUseCase implementation is as follows:

```
class GetFeedArticlesUseCase implements UseCase<Integer

   private final FeedRepository feedRepository;

   @Override
   public void execute(final Integer feedId, final Call
       try {
           callback.onSuccess(feedRepository.getFeedArt
       } catch (final Throwable throwable) {
           callback.onError(throwable);
       }
   }
 }
```

One last thing to note in the domain layer is that Interactors should only contain business logic. And in doing so, they can use repositories, combine other interactors and use some utility objects like *FeedValidator* in our example.

UI

Awesome, we can fetch articles, let's show them to the user now.

Our view has a simple contract:

```
interface View {

    void showArticles(List<ArticleViewModel> feedArticle

    void showErrorMessage();

    void showLoadingIndicator();
}
```

}

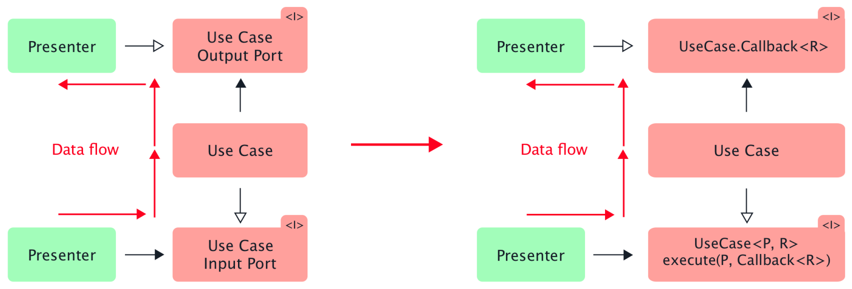
Presenter for that view has a very simple presentation logic. It fetches articles, maps them to the view models and passes on to the view, simple, right?

Simple presenters are yet another feat of the clean architecture and presentation-business logic separation.

Here is our *FeedArticlesPresenter* :

```
class FeedArticlesPresenter implements UseCase.Callback

    private final GetFeedArticlesUseCase getFeedArticles
    private final ViewModeMapper viewModelMapper;

    public void fetchFeedItems(final int feedId) {
        getFeedArticlesUseCase.execute(feedId, this);
    }

    @Override
    public void onSuccess(final List<Article> articles)
        getView().showArticles(viewModelMapper.mapArticl
    }

    @Override
    public void onError(final Throwable throwable) {
        getView().showErrorMessage();
    }
  }
```

Note that *FeedArticlesPresenter* implements Callback interface and it passes itself to the use case, it is actually an output port for the use case and in that way it closes the data flow. This is the concrete example of the data flow that we mentioned earlier, we can tweak labels on the flow diagram to match this example:



Where our parameter P is integer feedId, and return type R is a list of the Articles.

You do not have to use presenters to handle presentation logic, we could say that Clean architecture is “frontend” agnostic – meaning you can use MVP, MVC, MVVM or anything else on top of it.

Let’s throw some rx in the mix

Now, if you were wondering why there is such hype about RxJava, we will take a look at the reactive implementation of our use cases:

```
public interface UseCase<P, R> {

   Single<R> execute(P parameter);
}
```

```
public interface Compl

   Completable execute
}
```

Callback interfaces are now gone and we use rxJava *Single/Completable* interface as our output port.

Reactive implementation of the *GetFeedArticlesUseCase*:

```
class GetFeedArticlesUseCase implements UseCase<Integer

   private final FeedRepository feedRepository;

   @Override
   public Single<List<Article>> execute(final Integer f
       return feedRepository.getFeedArticles(feedId);
   }
}
```

And reactive *FeedArticlePresenter* is as follows:

```
class FeedArticlesPresenter {

   private final GetFeedArticlesUseCase getFeedArticles
   private final ViewModeMapper viewModelMapper;

   public void fetchFeedItems(final int feedId) {
       getFeedItemsUseCase.execute(feedId)
                    .map(feedViewModeMapper::mapFeedItems
                    .subscribeOn(Schedulers.io())
                    .observeOn(AndroidSchedulers.mainThre
                    .subscribe(this::onSuccess, this::onE
   }

   private void onSuccess(final List articleViewModels)
      getView().showArticles(articleViewModels);
   }

   private void onError(final Throwable throwable) {
      getView().showErrorMessage();
   }
}
```

Although it's a bit hidden, the same data flow inversion principle still holds, because without RxJava presenters were implementing the callback, and with RxJava subscribers are also contained in the outer layer – somewhere in the presenter.

Data and Device

Data and Device modules contain all of the implementation details that business logic does not care about. It only cares about the contracts, allowing you to easily test it and swap out implementations without touching the business logic.

Here you can use your favorite ORMs or DAOs to store data locally and network services to fetch data from the network. We will implement *FeedService* to fetch articles, and use *FeedDao* to store articles data on the device.

Each data source, both network, and local storage, will have its own models to work with.

In our example, they are *ApiFeed* – *ApiArticle* and *DbFeed* – *DbArticle.*

Concrete implementation of the *FeedRepository* is found in the Data module as well.

Device module will hold implementation of the *Notifications* contract that is a wrapper around the *NotificationManager* class. We could perhaps use *Notifications* from our business logic to show the user a notification when there are new articles published in which the user might be interested in and drive engagement.

Models, models everywhere.

You might have noticed that we mentioned more models than just entities or business models.

In reality, we also have db models, API models, view models and of course, business models.

It is a good practice for every layer to have its own model that it works with, so your concrete details, such as views, do not depend on the specific details of your lower layer implementations. This way, you won't have to break unrelated code if you, for example, decide to change from one ORM to another.

To make that possible, it is necessary to use object mappers in each layer. In the example, we used *ViewModelMapper* to map domain *Article* model to the *ArticleViewModel.*

Conclusion

Following these guidelines, we created a robust and versatile architecture. At first, it may seem like a lot of code, and it kind of is, but remember that we are building our architecture for the future changes and features. And if you do it correctly, future you will be thankful.

In the next part, we will cover maybe the most important part of this architecture, its testability and how to test it out.

So, in the meantime, what part of the architecture implementation did you find interesting the most?

Part I

Part II

Part III

Update

You can view the source code here.

Do you like our work?

GET IN TOUCH

Comments Community 1 Login

Recommend 10 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Luna Vulpo • a month ago

ow in such architecture places a external triggers like FCM notifications or LocationFence. In my app I need to show screen (start activity) when I get notification from FCM and when user enter to zone (base on location fences)?

1 • Reply • Share ›

Francis Mariano • 4 months ago

Hello Mihael. Nice article. Thank you so much by code.
I am not understanding the function of the ViewActionQueue which is injected into BasePresenter.
Can anyone help me with this doubt?

Best regards.

1 • Reply • Share ›

Mihael Francekovic → Francis Mariano • 3 months ago

Hi Francis,
ViewActionQueue is "frontend" implementation detail and it is not really important for the Clean architecture.

ViewActionQueue is a middle man in between a Presenter and a UseCase.
It subscribes to a given chain that results in a ViewAction, awaits for completion of that chain and passes result ViewAction to the Presenter.
We are not retaining our Presenters when configuration change occurs.
ViewActionQueue will ensure that if one Presenter invokes a UseCase, the second Presenter that we use after the Configuration change will get the result ViewAction of a UseCase that the first Presenter invoked.

Does that make sense?

Does that make sense?

1 ^ | v • Reply • Share ›

listen • 5 months ago

Nice job. Where should we put the app state, and how to subscribe the state stream if we uses RxJava?

1 ^ | v • Reply • Share ›

Dmitriy Khaver • 5 months ago

Thank you! Source code ?(:

1 ^ | v • Reply • Share ›

Mihael Francekovic ➦ Dmitriy Khaver • 5 months ago

You are welcome Dmitriy! We will share source code in a week.

^ | v • Reply • Share ›

Grisha Zaripov ➦ Mihael Francekovic • 5 months ago

Source code?

^ | v • Reply • Share ›

Mihael Francekovic ➦ Grisha Zaripov • 5 months ago

Source code is finally here! You can find it on the bottom of the article. Thank you for waiting :)

^ | v • Reply • Share ›

Dmitriy Khaver ➦ Mihael Francekovic • 4 months ago

Stable version of Android Studio doesn't configure either.

^ | v • Reply • Share ›

Dmitriy Khaver ➦ Mihael Francekovic • 4 months ago

Hi, have troubles with configuring this project on Android Studio 3.0 Beta 2 (retro lambda + gradle doesn't work together)
I am not sure how to fix it , cause I tried Googles recommendations - it didn't work.

Reply • Share ›

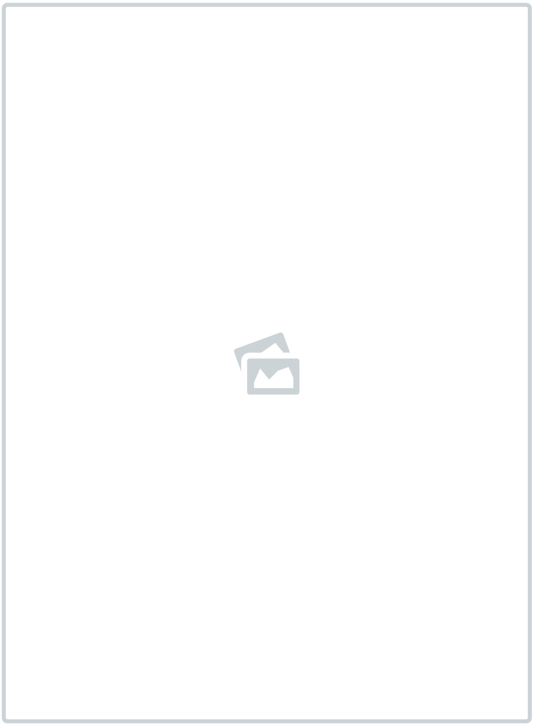
Tomislav Novoselec → Dmitriy Khaver • 3 months ago

Hey **@Dmitriy Khaver** , can you verify you have Java 8 installed on your machine?
The screenshot you provided is just a warning, it should be breaking the app configuration.

Reply • Share ›

Dmitriy Khaver → Tomislav Novoselec • 3 months ago

Hi, this is the problem I am dealing with, my friend has tried to open this app on a different PC (unstable Android Studio) he had same result, and I have Java 8
Is it a bug in gradle ?



Reply • Share ›

Mihael Francekovic → Grisha Zaripov • 5 months ago

Source code is finally here! You can find it on the bottom of the article.
Thank you for waiting :)

Thank you for waiting :)

Reply • Share ›

Zeeshan Shabbir • 6 months ago

Finally the long waited post is here :D Please share the repo link here so that i can get my hand dirty with code

1 • Reply • Share ›

Mihael Francekovic → Zeeshan Shabbir • 5 months ago

Hi Zeeshan, there are some finishing touches left on the source code. We will share the repo in a week, stay tuned :)

1 • Reply • Share ›

Zeeshan Shabbir → Mihael Francekovic • 5 months ago

that's great :)

Reply • Share ›

Mihael Francekovic → Zeeshan Shabbir • 5 months ago

You can find the source code at the bottom of the article :)

Reply • Share ›

Knowell Cedrich Riego • a month ago

Thanks for this great article about Clean Architecture, hope I found it earlier.

Reply • Share ›

Roel van Dun • a month ago

Oh, 1 more question (sorry for the many questions, but I think it's clearer to answer them separately): Is it possible to use a DI module? Then that module will know all other modules, instead of the App module.

Reply • Share ›

Roel van Dun • a month ago

Btw, I don't think that `void onSuccess(R return);` will compile ;)

Reply • Share ›

Roel van Dun • a month ago

Hi Tomo, thanks for this nice article! One question: what if a presenter uses multiple use cases (interactors), and thus implements multiple Interactor.Callback interfaces? They all have the same method signature, so how can you handle them separately? Only by implementing them as anonymous classes?

• Reply • Share ›

Sarthak Mittal • a month ago

Hello Five Agency, awesome articles on Android Clean Architecture!
Cannot understand one thing though, shouldn't we aim to reuse domain models, maybe declare them as interfaces and use concrete models (present in data module) to return them from repository?

• Reply • Share ›

narendra techguy • 2 months ago

After many users demands final part us here
Worlds best article on android clean architecture
many many thanks
Just one question how to handle threading for usecase if not using rxjava

• Reply • Share ›

Tomo narendra techguy • 2 months ago

There are many approaches I guess. The one that ran across my mind now is for example using a decorator pattern to wrap your use case with something like ThreadPooledUseCase and inject it decorated like that into presenters. They wouldn't know the difference, it would see the same interface and call the same execute method, but what would happen under the hood is that your wrapper can switch to another thread from the thread pool (or simply spawning new one) and then calling inner execute on the use case it wraps. You can use any Android / Java mechanism to switch to another thread. You can play with that and combine stuff, switch back to UI thread for example when triggering use case callback (since you're not using RxJava observable/observer), etc...

1 • Reply • Share ›

narendra techguy Tomo • 2 months ago

Thanks for reply and 5 starts for decorator pattern suggestion

• Reply • Share ›

Ivan Katić • 3 months ago

Greetings :)

Nice article series. Still, I have one problem and I didn't find an answer here.

So, I assume that you are using constructor injection (manual, or using Dagger, doesn't matter) to pass an use case instance to a presenter, right?

How do you handle situation where one use case should be called multiple times from one presenter? For example, you have a list of items, where each item has delete button. Each click on that button should execute DeleteItemUseCase. How do you solve that? I have few ideas on my mind, but none of them seems very clean to me.
1. Instead of use case, you pass use case factory to presenter, and then factory creates new use case for every single call? (This would imply that I have two types of use cases - first ones are made by DI, another ones are made by factories, and it "smells" bad to me, for some reason)
2. Use one use case instance for all calls? How? How do you handle callbacks which are async (for example, i call delete id1 and than id2, and I get one onSuccess() and one onError() call - how do I know which one has failed?)
3. Do you have any third way of handling such situations, by any chance?

^ | v • Reply • Share ›

Mihael Francekovic ➦ Ivan Katić • 2 months ago

Hi Ivan

You are right, use case factory smells bad.
Not only that it introduces additional boilerplate, but it also comes with an additional cost of use case allocation for every required use case.

UseCases should be stateless and reusable, meaning that we must be able to call an instance of a use case multiple times and that it will work as expected.

Let's take FavouriteArticleUseCase as an example.
We call it repeatedly in the ArticlesPresenter.
This snippet will help me to answer your question regarding callbacks:
https://gist.github.com/0xi...

Hope that this helps.
What do you think about this approach?

• Reply • Share ›

Ivan Katić Mihael Francekovic
• 2 months ago

Actually, you've already answered on my question: "UseCases should be stateless and reusable, meaning that we must be able to call an instance of a use case multiple times and that it will work as expected."

Great, thanks, you've helped me :)

• Reply • Share ›

Ivan Katić Mihael Francekovic
• 2 months ago

Hi,

Thanks for the answer. Nice trick, looks like it should work. I'm afraid, though, that using one use case instance would cause some bugs if the use case object had some internal state. I don't see that happening often, but it could, couldn't it?

• Reply • Share ›

Filipe Oliveira • 3 months ago

Very good article !

Also , very nice to follow with the code example which is very well structured. The only thing missing are the unit tests :)

Thank you and Best regards !

Related

BLOG

20 Tools to Consider When Doing Qualitative Remote

How to Decide on The Right Tool in Less Than 5 Minutes? User research is

Gabrijela Šitum

SEARCH ENGINE OPTIMIZATION, SEO

Complete SEO Guide for Web Developers

Complete SEO guide for meta tags, URLs, robots, sitemaps, social tags,

Ozren Lapčević

BLOG

Android A Part 5: Ho Clean Arc

Why shoul about test Programm

David Geček