Trivago android app source code

I'm not robot

reCAPTCHA

Continue

I'm not robot

reCAPTCHA

Build your own vacation booking site in a few hours We take care of the hassle you face with installation procedures. We provide free installation services to our customers. You get a fully responsive design where you can access the final product through any device with easy interactive navigation and experience. You can create multiple admins that provide roles and permissions to child administrators in your script. Find lists easily through an efficient search system. Simple search mechanisms help reduce search complexity by offering a variety of sorting and filtering options. When Trivago updates a new feature, you can easily get it with our software. Login through social media such as Facebook, Twitter contributes towards user convenience and improved user registrationYou can use multiple languages based on your country. We can easily manage the languages on your admin page. You can use more than 19 languages based on your country. Trivago clone scenario offers multi-currency system. You can use 31 currencies based on your country. You can easily manage the currency on your admin page. Easy to add details to your room and to live. We can easily change list details and images. Google-Plus CSS-3 HTML-5 CodeIgniter MySQL PHP-7 Bootstrap-3 Goola Map Apache Facebook Twitter Is this Trivago script that ensures better responsiveness for all devices? We've developed this scenario with a more compatible scenario for all devices like Smartphones, iPhones, iPads, Tablets, PCs, and so on. Do you offer any custom design support? Yes, we offer free custom support just for design. But for any functional changes or adding new features, we can do at a separate cost. Does this support multinational users and languages? Yes it supports all national and language users. Can I integrate my own website with api keys on your trivago script? There's definitely if you've got the website for your hotel then you just need to give the main API to integrate and then will integrate it. Still wondering? Contact us: support@bootsgrid.com #reportoire Reportoire is a simple set of layers to apply inventory templates to your Android app. For further discussion and information please see our blog post. #Installation Simply add corresponding gradle dependency to build.gradle your application: dependency { // Core compilation 'com.trivago.reportoire:reportoire-core:{latest_version}' // rxV1 compilation 'com.trivago.reportoire:reportoire-rxV1:{latest_version}' // rxV2 'com.trivago.reportoire:reportoire-rxV2:{latest_version}' } ##Sample This is an example of implementation that will first require a memory source and then a network. /** * Simple sources use an API client to query data. */ MyNetworkSource class: CachedObservableSource<Model, string=>() { // CachedObservableSource overrides pleasure onCreateResultObservable(input: </Model,> </Model,> { return mApiClient.getModel(input) } } /** * Repository that will try to get the cached data first before trying a network request. */ class RxMemoryNetworkRepository : Repository<Model>() { // Members private val mMemorySource = MemorySource<Model, string=>() private val mNetworkSource = MyNetworkSource() // Public API fun getModel(input: String): Observable <> <Model>> { // Get the cached result val cachedResult = mMemorySource.getResult(input) // Check if it is a success if (cachedResult is Source.Result.Success) { // If so simply emit the result Observable.just(cachedResult) } // Else subscribe to the network observable return mNetworkSource.resultObservable(input). // And save the result in the memory source if it is a success if (result is Source.Result.Success) { mMemorySource.setModel(input, result.model) } } } // Repository override fun allSources(): List <> <Model, *=> > { return listOf(mNetworkSource, mMemorySource) // Return a list of the used sources } } ##core ##Repository Base class for every repository you build. Here we combine all sources. See above for an example. ###Source This is the base source interface. Each source must perform the following method: /** * Tell the source that its ongoing mission must be *canceled. */fun cancel() /** * Tell the source that everything that has been *stored so far should be re-set. */happy reset() /** * Returns correctly if the source is currently receiving results*. If not wrong. */ fun isGettingResult(): Boolean ###SyncSource A source immediately returns results for a given input. /** * Returns the result immediately. */ fun getResult(input: TInput?) : Source.Result ###AsyncSource A source recalls when there is a result<TModel> fruit. /** * Pass in a call back will return the result once * made or if an error occurs. */ fun fetchResult(input: TInput?, recall: ResultCallback<TModel>) ##rxV1 & rxV2 ###RXSource A simple rxjava-based source. /** * Returns an observable result. */ interesting resultsObservable (input: TInput?): <> <TModel>Observable > ###CachedObservable An abstract RxSource source will use the specified observation source or re-register to the previous one if possible. Internally it uses the cache operator. /** * Overwrite this method and provide a new observable that will be used to obtain results. */ fun summary onCreateResultObservable (input: TInput?): Observed > Problem If you have any running issues <> <TModel>applications after including this library, concider to check for this RxRelay issue and update RxRelay or 1.2.0 or 2.0.0 About Reportoire was built by trivago 🏨 License Reportoire licensed under Apache Version 2.0. Page 2#reportoire Reportoire is a simple layers to apply storage templates to your Android app. For more details</TModel> </TModel> </TModel> </TModel> </TModel> </TModel> and information please see our blog post. #Installation Simply add corresponding gradle dependency to build.gradle your application: dependency { // Core compilation 'com.trivago.reportoire:reportoire-core:{latest_version}' // rxV1 compilation 'com.trivago.reportoire:reportoire-rxV1:{latest_version}' // rxV2 compilation 'com.trivago.reportoire:reportoire-rxV2:{latest_version}' } ##Sample This is an example of implementation that will first require a memory source and then a network. /** * Simple sources use an API client to query data. */ class RxMemoryNetworkRepository : Repository<Model>() { // Members private val mMemorySource = MemorySource<Model, string=>() private val mNetworkSource = MyNetworkSource() // Public API fun getModel(input: String): Observable <> <Model>> { // Get the cached result val cachedResult = mMemorySource.getResult(input) // Check if it is a success if (cachedResult is Source.Result.Success) { // If so simply emit the result Observable.just(cachedResult) } // Else subscribe to the network observable return mNetworkSource.resultObservable(input). // And save the result in the memory source if it is a success if (result is Source.Result.Success) { mMemorySource.setModel(input, result.model) } } } // Repository override fun allSources(): List <> <Model, *=> > { return listOf(mNetworkSource, mMemorySource) // Return a list of the used sources } } ##core ##Repository Base class for every repository you build. Here we combine all sources. See above for an example. ###Source This is the base source interface. Each source must perform the following method: /** * Tell the source that its ongoing mission must be *canceled. */fun cancel() /** * Tell the source that everything that has been *stored so far should be re-set. */happy reset() /** * Returns correctly if the source is currently receiving results*. If not wrong. */ fun isGettingResult(): Boolean ###SyncSource A source immediately returns results for a given input. /** * Returns the result immediately. */ fun getResult(input: TInput?) : Source.Result ###AsyncSource A recalled source gave when the result was & TModel> fruit. /** * Pass in a call back will return the result once * made or if an error occurs. */ fun fetchResult(input: TInput?, recall: ResultCallback<TModel>) ##rxV1 & rxV2 ###RXSource A simple rxjava-based source. /** * Returns an observable result. */ interesting resultsObservable (input: Observable <> <TModel>and) ###CachedObservable Một tóm tắt</TModel> </TModel> </Model,> </Model,> </Model,> </Model,> </Model,> the source will be able to observe or re-register earlier if possible. Internally it uses the cache operator. /** * Overwrite this method and provide a new observable that will be used to obtain results. */ fun summary onCreateResultObservable (input: TInput?): Observed > Problem If you have any running issues <> <TModel>applications after including this library, concider to check for this RxRelay issue and update RxRelay or 1.2.0 or 2.0.0 About Reportoire was built by trivago 🏨 License Reportoire licensed under Apache Version 2.0. Well! We're rewriting our app, absolutely from scratch! Interesting times were in front of us, but as a wise man once said: great power carries great responsibility. We were there, faced with what could be a once-in-a-lifetime opportunity to build an app from the bottom up. But how can we exceed the expectations of so many users with the first release on Day 1? Obviously, the first thing you think in this situation is: Finally! Bye bye legacy code, no more refactoring, everything brand new and shiny standing exactly how we want and need it. But to achieve this ideal scenario, a series of decisions were made. This is not only a challenge to our architectural goals, but also for everyone else involved, including product owners, designers and project managers. Having a diverse team with great knowledge, allows us to leverage our expertise when addressing hot topics such as multi-functional teams, self-organizing work processes, MVP goal definitions, and new and modern UX practices. I'm not going to dive into the details of the architecture we ended up using, but rather try to focus on the decisions we make, the problems we face down the road, and what we've learned from them. ArchitectureNow that I described the situation, I can explain more how we approach this opportunity. Our goal is clear: don't let the architecture get in the way. Architecture must be something transparent that allows developers to build anything seamless (stressed from anything because at the moment, we don't have any idea what features have come). We call it in software, scalable. Of course, it's not just about building new features on top of each other, these features will also stay there for a while and you want to make sure they still work no matter how many other interesting things come. There is an easy way to do this: check. Therefore, the architecture should also be maintained and inspectable. Last but not least, it's also used and owned by all team members. Therefore, it needs to be easy to understand and easy to use. This is where balance comes in and the challenge of finding a balance between good practice and not on techniques needs to be done. Clean architectureWith these </TModel> this</TModel> in mind, we started evaluating the different models out there that accomplished our goals and we came across clean architecture or in other words... keep those exact details, details. We don't want to look at the whole thing with mission statement We are building an Android app. But we want to change the way we think about building a customer app to allow users to explore and book their ideal hotel. The fact that it runs on Android is a specification made. That's how we tried to think. Try to focus more on the problem itself and not on the tools or methods we use to solve it. If you are interested and would like to know more about this approach, we spoke at GDG Google I/O Extension 2017 in Heraklion on this topic, you can check out the slides. So one general thinking is established, let's dive into what we've learned from trying to apply it to a real scenario like our Android app:The first and general lesson will be: don't be afraid to spend as much time as necessary in discussions. This will likely save tons of time later in evaluating code, refactoring and struggling to keep your system healthy. Moreover, making decisions as a group will increase the sense of ownership of everyone involved, allowing the entire group to remember the reasons behind the conclusions reached. Other lessons we learned from this process are:Case management may not be trivialWhen you think about domain logic, it's likely that you just need to create an expression of a use case for every operation your application will take. However, this must be thought t smoothly. Try to answer these questions:What happens to use cases after the operation is over? Can I have different cases of a use case? Is my dependency management system ready for that? When should a use case perform an operation and when should it only return the final value? What will be provided by the use case? It's just a success? Should it handle errors, or just pass it to the upper layer? Any changes in the parameters? How long should use cases that remain in memory? Who is responsible for cleaning them up when they are no longer needed? How should use cases work during rotation? In general, even if the use cases are concise and simple, you really need to look at all the situations because they will be everywhere and do every important operation that your application needs to perform. For us, that means allowing our dependency to store different versions of a use case to make them available both after rotating and creating two different sublayer of them. One that will act as a BehaviorSubject, which means that it will ring play out results as soon as anyone starts listening to it and another one (based on a PublishRelay that will play out the result only after it is done. Possible integration testing valuable as unit testingWith this architecture, one of the effects we have is many smaller and more basic layers, with mediocre methods consisting only of a single call to another dependency. Example:/** * Use case to check if email is registered at trivago*/ CheckIfAccountExistsUseCase class(initialState: BaseUseCase.State<String, boolean=>?, private val mAccountsRepository: IAccountsRepository: PublishUseCase<String, boolean=>{initialState) { // BaseUseCase overrides pleasure onExecute(email: String?): <> <Boolean> </Boolean> </Boolean> { if (email == null || !email.isEmail()) { Observable.just(Result.Error(InvalidParameterException(you should provide an email))) } else { mAccountsRepository.doesAccountExist(email) } } As you can see, this use case basically returns the This is a very simple case but not uncommon to go through concise use cases like this. Because of this, unit testing becomes so simple that developers often get the feeling that it doesn't bring any value to the code base. Although one can understand that view, this is not always true and unit testing cannot just be dropped. Because even if it's just a call, we have to make sure it happens, and the results can be properly mapped into what the upper layer expects. Something like this:class CheckIfAccountExistsUseCaseSpecs : FeatureSpec({ feature(Emits an error if no email was provided) { val repository: IAccountsRepository = mock() val useCase = CheckIfAccountExistsUseCase(null, repository) val testObserver = TestObserver <> <Boolean>>() useCase.result().subscribe(testObserver) scenario (an error is ingought if the email is null) { useCase.execute(null, true) testObserver.assertValue { (it is<Boolean>Result.Error).throwable!!. message.equals (you should provide email) } } scenario (an error is issued if the email is invalid) { useCase.execute(asdf, true) testObserver.assertValue { (it is Result.Error <Boolean>throwable!!. message.equals (you should provide email) } } scenario (repository result is played out if a valid email is given) { val result = Result.Success(true) 'when'(repository.doesAccountExist(valid@email.com)) .thenReturn(Observable.just(result)) useCase.execute(valid@email.com , true) testObserver.assertValue(result) } } }) What we can do to increase confidence in code for the most relevant developers is to test how these layers interact with each other. In fact, since we are using this new approach, the complexity of classes is reduced, but the number of them may be higher, and a good part of our logic remains in the interactions of the classes. So it is a good idea to also test this interaction: here where integrated tests come to the rescue:class DealDetailsViewModelIntegrationSpecs: FeatureSpec({</Boolean> </Boolean> </Boolean> </Boolean> </Boolean> </String,> </String,> transaction details) { scenario(error will be played if no hotel details can be download) { // Prepare the case for use of the transaction 'when'(mockDealsMemorySource.loadDeal(Mockito.anyString())).thenReturn(Observable.never()) // Prepare guests hotel details use case 'when'(mockHotelDetailsNetworkSource.performHotelDetailsSearch(Mockito.anyString()) .thenReturn(Observable.error(RuntimeException(Some network errors!))) // Download viewModel.loadDealDetails(ru, barb) // Get the latest error val error = viewModel.error().blockingFirst() // Error checking.shouldBe(Some network errors!) } } In this test, we didn't exactly test one method, but the whole interaction between all components worked as expected. We have a use case that has no storage value (we ridicule memory sources to never inging out anything) and a network request does not. As a result, we don't know exactly which method will deliver the error or in which form. What concerns us, is that the viewing pattern inging out an error through its observations. This adds a new dimension to the quality of our testing. Data flow is not the same as dependency flow (domain data conflict)This is, in my opinion, a really interesting discussion and we've had it in a lot of contexts, also with iOS people. The question is: if the domain class needs data from the data layer, does it have to rely on it (is it dependent on it)?NO. Where the data comes from does not mean nothing to the domain. That's why we don't directly require data supplies, but archives. So even if the data stream looks like this: There is also a long discussion on this topic that I find particularly enlightening: Its short version is that for each part of your application must be completely independent. It should only expose interfaces that determine how it can get whatever it takes to implement business logic and then other modules will implement these interfaces using any framework that exceeds our core visibility. TakeawaysIn this process, it is important that we listen to the needs of our colleagues, they will be the builders of features that affect the user and create the wow-effects we are looking for. Architecture should only be there to provide basic ground so they can shine. If you have any suggestions, other thoughts on this, or you are simply interested in knowing more about any particular aspect. Do not hesitate to leave a comment here or drop me a message! Of course, especially thanks to all the Android development teams that have contributed to making this possible. Thanks for reading. Read.