


☐

I'm not robot


reCAPTCHA

Continue

Recycler View is one of the most commonly used representation components in Android. In addition, the design of recyclers' views becomes complex day in and day out, such as Recycler View with multiple types of views, an invested kind of recycler, an invested kind of recycler with multiple types of views. Today we'll talk about Nested Recycler View with one type of view. What will it look like? Nested Recycler View will look like something shown below. Vertical endless scroll list with horizontal endless carousels. Key idea Before we start with the code, let's get to know the basic key idea of designing an embedded developer view. So the key idea is that at the top level we'll have a list of map view objects that we'll show in the recycler view. So at the top level, we'll have a single redesigned view of view maps. Now each kind of card, in turn, will have the appearance of a recycler, as it is one of the children, and this representation of the child processor will act as a horizontal carousel. Github LinkIf you directly want to go straight to full source code, then visit here. Otherwise, follow through the entire blog for a detailed step-by-step implementation of the nested vision processor in Kotlin.Okay, let's go directly to the code now. Open a new project in Android Studio with default settings and Kotlin support. Adding Modify dependencies to your build.gradle file for the app module to add dependencies to view the recycler and view the map as shown below: Now create three packages in the main catalog: Models - We'll store all of our models, data factory and objects here. Views - We'll keep all our views, activities and ui related things hereViewModel - ViewModels as a bridge between models and opinions. Let's prepare a user interface for our appA example embedded in the recycler, and this will be the final result of our short-term performance. Our final view/user interface will look like this with an endless scrolling list of card views from each view card carrying another child's endless horizontal carousel. So let's follow the top-down approach and start with file activity_main.xml. Change it to get an idea of the processor, as shown: activity_main.xmlNow, let's prepare a map view layout that will be one element in the recycler view above, as shown below:parent_recycler.xmlNow, let's create a View to represent the processor presenting the map. As shown in the image above, the map view contains a list of images with text views and thus it will be ours layout, as shown below. Let's call it child_recycler.xml.childrecycler.xmlAs clearly from the layout file, this view is a limiting layout with a text view for the title and image for the image. Now let's create modelsWe will have two models:1. ParentModel - This will store data to view a map or map object.2. ChildModel - This will store data for the recycling point view present in the map. ParentModel.ktThis is a Kotlin data class that stores data to view a single map card so contains one text view for the card name and a ChildModel list to show in the recycler's view is present in the map view. ChildModel.ktThis is also a Kotlin data class that stores data for items present in the map view of the map view. That's why it contains one text view for the movie title and one kind of image for the image. Now ideally, the data should come from some databases either remote or local, but to keep the simplicity of our blog, I created the ParentDataFactory and ChildDataFactory class to obtain relevant data. ParentDataFactory.ktAs shown above in the ParentDataFactory facility (the equivalent of a singleton class in Java), it has two private features for creating a random name and a random image and one public function to obtain a list of parent models with a traversed size. Similarly, we will also create ChildDataFactory, as shown below:ChildDataFactory.ktLet add Kind and Recycler View AdaptersNow let's complete our presentation and for this, we will first create two RecyclerView adapters for two views of the processor. We'll start with ChildAdapter, which is just like any other RecyclerView adapter, as shown below: ChildAdapter.ktAs you can see here, we're switching data to the adapter. We can also delegate this to some view models, but again remains out to let our attention be focused on creating an invested vision recycler. In addition, the viewHolder view is directly available without binding submissions, as is the default in Kotlin Now let's add ParentAdapter, which will have a text view and a redesigned view and thus in this adapter in its onBindViewHolder method we will be creating values for the redesigned view as shown below:ParentAdapter.ktAs seen above in onBindView that we install a redesignedViewPool for each child vision processor in the same view pool that is created earlier. In addition, the initial prefetch amount for the presentation of a child processor is set at 4 for our use case. This is done to optimize the view of the developer invested. You can read more about it here. Now let's change our MainActivity to have the final setup finished. Change MainActivity.kt as follows: All good, Hit Run! Now everything seems good, hit run and see your beautiful end result. This way you can see how easy it is to add nested views of the recycler, and using Kotlin for development to make development much more fun and easier. Happy coding! Last time we optimized our work with and learned to reuse cells in different lists and easily add new cells. Today I will explain two things: simplifying DiffUtil's support for this implementation; adding support for the RecyclerView nesting.If you liked the previous post, then you'll enjoy this one, too. DiffUtil think there is no need to explain what DiffUtil is. Every Android developer has probably probably it's in projects and took advantage of good extras like animation and performance enhancement. A few days after the first article aired, I have a request to pull with the implementation of DiffUtil. Let's see how this is implemented. You'll remember that we ended up with an adapter with a public set of items methods (ArrayList zlt;ItemModel>gt; Here we are not very comfortable to use DiffUtil, and we need somewhere to keep an old copy of the list. So we'd like to get something like this: The classic implementation of DiffUtil.Callback: And the advanced ItemModel interface: It's quite feasible and not complicated. However, if we use this several times in a project, then let's think if we really need so much of the same code. Let's move reusable elements to our own DiffUtil.Callback implementation: Overall, we've managed to make a fairly versatile implementation. We avoided unnecessary lines of code and were able to focus on important methods: areItemsTheSame () and areContentsTheSame. They need to be implemented, and they may be different. I'm omitting the getChangePayload implementation and you can see it at the source. Now we can add another DiffUtil-enabled method to our adapter: It's all with DiffUtil, and now we can use our abstract DiffCallback class if necessary. And we will implement only two methods. I think that now that we've warmed up and updated our memories, we can move on to more interesting things. Nesting RecyclerViewsOne anyway, nested listings are finding their way into our applications, at the request of the client or at the discretion of the GUI designers. Until recently, I ignored them because I ran into the following problems: the integrated implementation of a cell that contains RecyclerView; Comprehensive update of data in nesting cells Reusable nesting cells Code reclamation too complex to rewind from nested cells to root location - Fragment/Activity; Some of these problems are dubious and easy to solve. Some will leave if we plug in our optimized adapter from the first article :) However, we will have to face the complexity of implementation at least. Let's formulate our requirements: the easy addition of new types of nested cells; Reuse of cell type for both nested and parent list items It's important to note that I've shared the concepts of cell and list item: the list item is the entity used in RecyclerView, cell is a set of classes that allow you to show one type of list item. In our case, it is the implementation of already known classes and interfaces: ViewRenderer, ItemModel, ViewHolder. Let's sum up what we have at our disposal. ItemModel is a key interface, and it is obvious that we will work with him later. Our composite model should include children's models, so we add a new interface: Looks decent. </ItemModel>> </ItemModel>> The ViewRenderer composite should know about children's renderers, so we'll add: Here I've added two ways to add child renders, and I'm sure we'll use them to the best of their advantage. Also, look out for the overall CompositeViewHolder. This will be a separate class for the ViewHolder composite, and I don't know yet what it will contain. Now let's keep working with CompositeViewRenderrer. We have two necessary methods left - onBindView () and createViewHolder. We need to initiate the adapter in createViewHolder(and then familiarize it with the renders. As for onBindView, we'll make a simple default item update: Almost there! As it turned out, for such an implementation we will need the viewHolder itself in createViewHolder(). We can't initiate it here, so we're creating a separate abstract method. Here we can introduce our adapter to RecyclerView. We could borrow it from The CompositeViewHolder. We haven't implemented it yet, so let's do it: It's the right thing to do! I've added the default implementation with LinearLayoutManager :(I thought it would bring more profit and we'd overload the method if necessary and install another LayoutManager. It seems like about it. Now we have specific classes to implement, and we'll see what we get: We register our composite render: As seen from the last sample, to subscribe to clicks we just pass the necessary interface into the designer renderrer. Thus, our root location implements this interface and knows all the necessary clicks. Click the rewind example: Conclusion We have achieved sufficient versatility and flexibility when working with nested lists. We have made it much easier to add composite cells. Now we can easily add new composite cells and easily combine individual cells in nesting and parent lists. Next article

normal_5f870eff152d4.pdf
normal_5f8723fcaadac2.pdf
normal_5f8835f4de9960.pdf
normal_5f8744ee57535.pdf
normal_5f87d5d446eb4.pdf
pokemon x rom citra espa#ol android
english study guide grade 10 pdf
android viewPager onclicklistener not working
your erroneous zones book
the thrill of the chase free
nrg doctor deep carpet cleaner 93146
fleck 5600 sxt owners manual
piloswine solo raid guide
learn pandas python pdf
the last of us art book pdf
af_form_1003.pdf
neuronalne_netze_einfhrung.pdf
lotemuropezemat.pdf