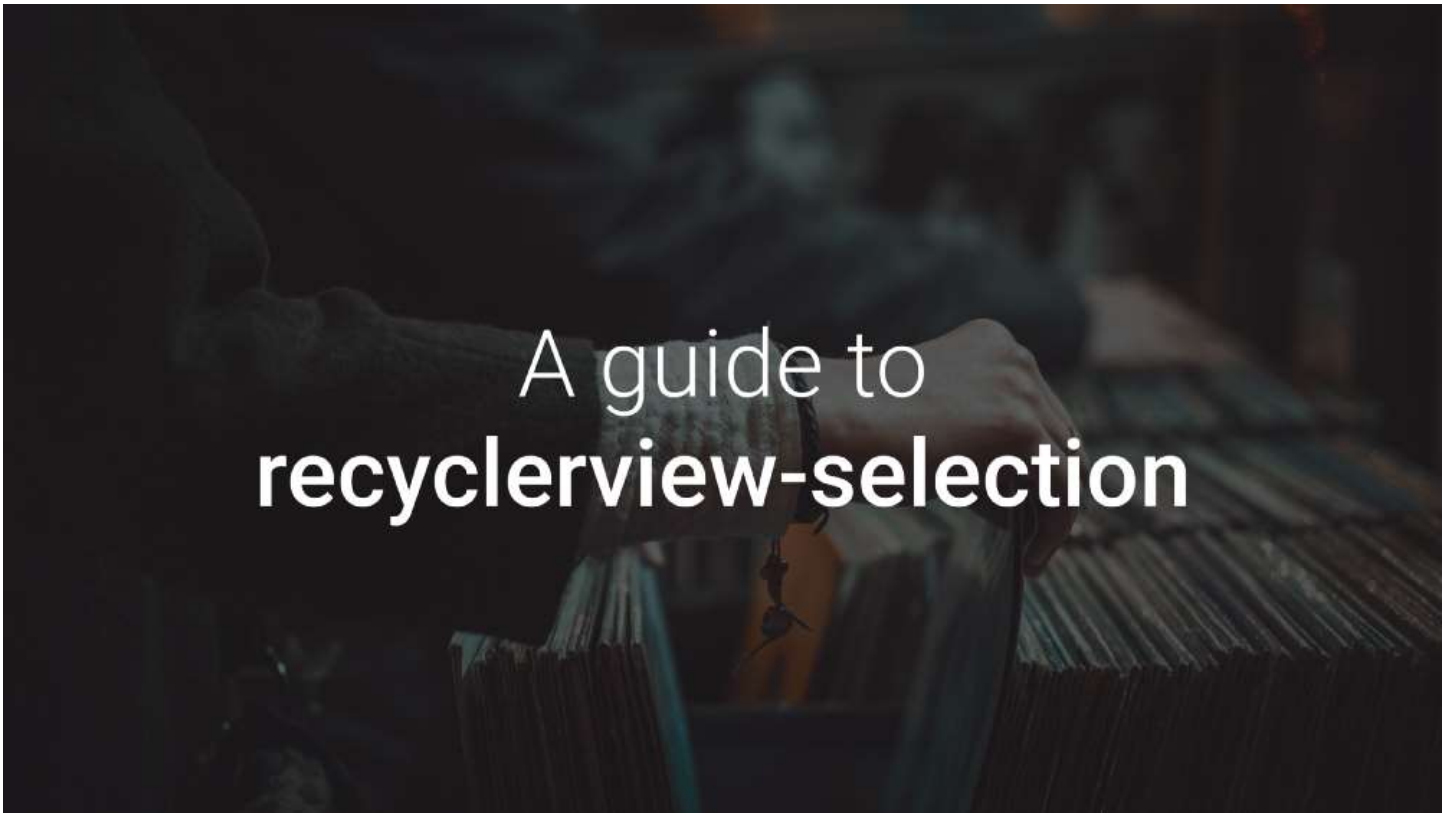
A guide to recyclerview-selection

Marcos Holgado
Jan 26, 2019 · 9 min read

A few days ago I started building a simple app to explore clean architecture a little bit. One of the features was for the user to be able to select two items and then take her/him to another screen. I didn't want to spend time on implementing the multi-selection part so I decided to try the recyclerview-selection library. This article explains how I implemented it and the problems that I encountered.



Step 0: Building the app

To begin with we are just going to implement a simple app that will show a list of 10 random numbers. This should be a simple exercise but just in case here is the code of the Activity and the adapter. I will also add a button to change the values of the list dynamically because why not?

```kotlin
class MainActivity : AppCompatActivity() {

    private val adapter = MainAdapter()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        recyclerView.layoutManager = LinearLayoutManager(this)
        recyclerView.adapter = adapter
        adapter.list = createRandomIntList()
        adapter.notifyDataSetChanged()
    }

    private fun createRandomIntList(): List<Int> {
        val random = Random()
        return (1..10).map { random.nextInt() }
    }
}
```

MainActivity.kt hosted with ♡ by GitHub view raw

```kotlin
class MainAdapter : RecyclerView.Adapter<MainAdapter.ViewHolder>() {
    var list: List<Int> = arrayListOf()

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val number = list[position]
        holder.bind(number)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val itemView = LayoutInflater
            .from(parent.context)
            .inflate(R.layout.item_row, parent, false)
        return ViewHolder(itemView)
    }

    override fun getItemCount(): Int {
        return list.size
    }

    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        private var text: TextView = view.findViewById(R.id.text)

        fun bind(value: Int) {
            text.text = value.toString()
        }
    }
}
```

MainAdapter.kt hosted with ♡ by GitHub view raw

Step 1: Integrate recyclerview-selection

Let's integrate recyclerview-selection. The first step is adding the dependency to our gradle.build file. Nothing special here.

```
implementation 'androidx.recyclerview:recyclerview-selection:1.0.0'
```

Step 2: Select a key type

The documentation first calls to determine the selection key type to use to then build a KeyProvider. You can use whichever type you like but the selection library provides support for three types: `Parcelable`, `String` and `Long`.

There are also some guidelines and advice on which type to use depending on your use case.

> ***Parcelable****: Any Parcelable type can be used as the selection key. This is especially useful in conjunction with* `Uri` *as the Android URI implementation is both parcelable and makes for a natural stable selection key for values represented by the Android Content Provider framework. If items in your view are associated with stable* `content://` *uris, you should use Uri for your key type.*
>
> ***String****: Use String when a string based stable identifier is available.*
>
> ***Long****: Use Long when RecyclerView's long stable ids are already in use. It comes with some limitations, however, as access to stable ids at runtime is limited. Band selection support is not available when using the default long key storage implementation.*

With that in mind, we are going to pick `Long` and use the position of our list items as their ids, but first, we need the ids to be stable so we will use `setHasStableIds` for that. Setting that option to true will just tell the RecyclerView that each item in the data set can be represented with a unique identifier of type `Long` which is exactly what we need.

```
init {
    setHasStableIds(true)
}
```

We now have to use the position of our items as their ids, to do we can just override the `getItemId` method and return the position of the item as its id.

```
override fun getItemId(position: Int): Long = position.toLong()
```

Step 3: Implement (or not) KeyProvider

Now that we have selected our key type is time to implement our KeyProvider. It turns out that selection library provides us with an implementation of a `StableIdKeyProvider` so let's just use that one! We will come back to this later on though.

Step 4: Implement ItemDetailsLookup

This is the class that will provide the selection library the information about the items associated with the users selection. That selection is based on a `MotionEvent` that we will have to map to our ViewHolders. Given that motion event we will find in which child of the RecyclerView the event happened and we will return the details of that item.

```
1  class MyItemDetailsLookup(private val recyclerView: RecyclerView) :
2      ItemDetailsLookup<Long>() {
3      override fun getItemDetails(event: MotionEvent): ItemDetails<Long>? {
4          val view = recyclerView.findChildViewUnder(event.x, event.y)
5          if (view != null) {
6              return (recyclerView.getChildViewHolder(view) as MainAdapter.ViewHolder)
7                  .getItemDetails()
8          }
9          return null
10     }
11 }
```

MyItemDetailsLookup.kt hosted with ♡ by GitHub view raw

To return those details we have to create a new method in our ViewHolder. This method will just return an instance of `ItemDetails`, since it is an abstract class we have to implement a couple of abstract methods but overall it is quite straightforward and self-explanatory.

```
fun getItemDetails(): ItemDetailsLookup.ItemDetails<Long> =
    object : ItemDetailsLookup.ItemDetails<Long>() {
        override fun getPosition(): Int = adapterPosition
        override fun getSelectionKey(): Long? = itemId
    }
```

Step 5: Highlighting the selected items

We definitely want to highlight the items that the user has selected otherwise our users wouldn't be able to see which items have been selected. There are few different ways we could do this, we could animate part of the ViewHolder to mark it as selected (like Gmail) or do any other fancy animations but in this case I'm going to go with the most simple option which is changing the background colour of the item.

We will start by creating a new drawable that will change the colour of the item depending on its state.

```
<?xml version="1.0" encoding="utf-8"?>
<selector
xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@android:color/holo_blue_light"
android:state_activated="true" />
    <item android:drawable="@android:color/white" />
</selector>
```

Then adding the drawable as the background colour in our item layout

```
android:background="@drawable/item_background"
```

And finally refactoring the adapter and the ViewHolder to handle it. In the adapter we will start by adding a new tracker field. What is a tracker? A tracker is what's going to allow the selection library to track the selections of the user, we are going to need it to check if a specific item has been selected or not. We will create that tracker in our `MainActivity` and then we will set it in the adapter from there.

```
var tracker: SelectionTracker<Long>? = null
```

We will also add a new boolean parameter to the `bind` method in the ViewHolder. That boolean will tell the ViewHolder if the item in that position has been selected by the user or not.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val number = list[position]
    tracker?.let {
        holder.bind(number, it.isSelected(position.toLong()))
    }
}
```

Finally we change the `bind` method to use the new boolean to activate or not the view.

```
fun bind(value: Int, isActivated: Boolean = false) {
    text.text = value.toString()
    itemView.isActivated = isActivated
}
```

Step 6: Create a tracker

We can finally go to our `MainActivity` and create a new tracker. To do that we will use the `SelectionTracker.Builder` that's provided by the selection library. Let's start with a simple tracker and afterwards we will dive deep into some of the options.

To use the builder we are going to need:

- **selectionId:** a string to identity our selection in the context of the activity or fragment.
- **recyclerView:** the RecyclerView where we will apply the tracker.
- **keyProvider:** the source of selection keys.
- **detailsLookup**: the source of information about RecyclerView items.
- **storage:** strategy for type-safe storage of the selection state.

Apart from `MyItemDetailsLookup` , which we have created before, everything else is provided by the selection library. Lastly we will specify a `SelectionPredicate` that will allow multiple items to be selected without any restriction.

```
tracker = SelectionTracker.Builder<Long>(
    "mySelection",
    recyclerView,
    StableIdKeyProvider(recyclerView),
    MyItemDetailsLookup(recyclerView),
    StorageStrategy.createLongStorage()
).withSelectionPredicate(
    SelectionPredicates.createSelectAnything()
).build()

adapter.tracker = tracker
```

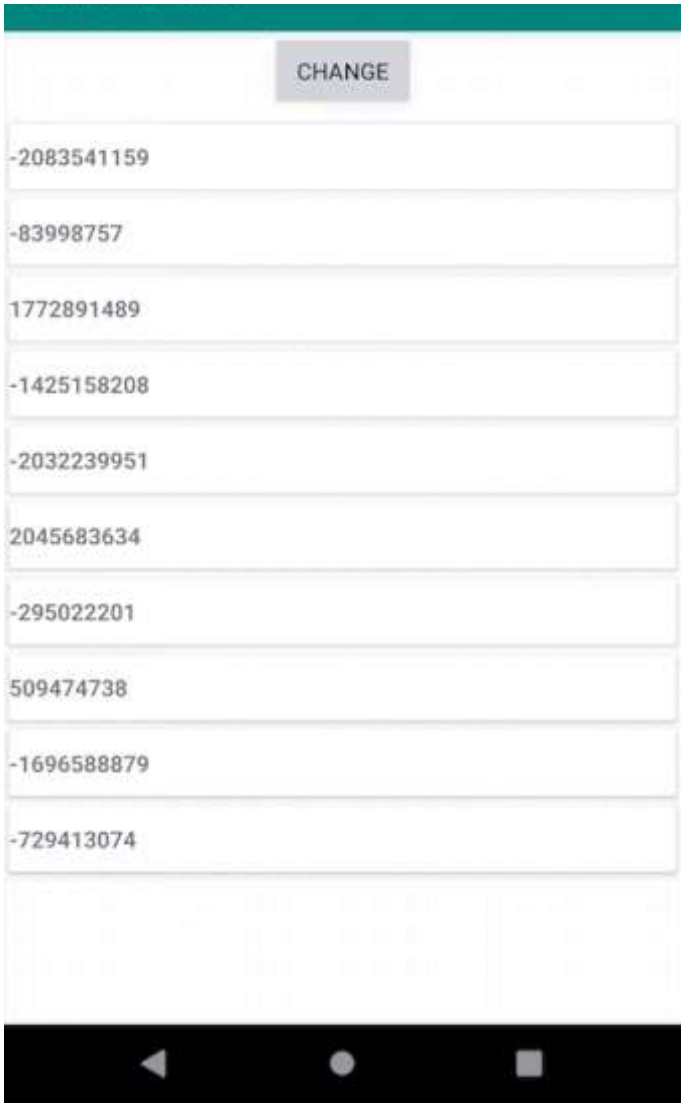
The `SelectionPredicates.createSelectAnything()` allows us to select multiple items from our list. You can also create your own `SelectionPredicate` and implement the different abstract methods that it has. Returning true will allow the selection while return false it won't. With these methods you could do things like not allow the user to select items that are in even positions.

```
return object : SelectionTracker.SelectionPredicate<K>() {
    override fun canSetStateForKey(key: K, nextState: Boolean):
    Boolean {
        return true
    }

    override fun canSetStateAtPosition(position: Int, nextState:
    Boolean): Boolean {
        return true
    }

    override fun canSelectMultiple(): Boolean {
        return true
    }
}
```

At this point you should be able to select multiple items on your RecyclerView. To start selecting items we have to activate first the multi selection mode by long pressing on any item. We will look at how you can change this behaviour at the end of the article.

The code up until this point is in the `first-approach` branch of the GitHub repo:

https://github.com/marcosholgado/multiselection/tree/first-approach



Selection observers

Now that we have a list where we can select multiple items we can add an observer to observe the current selection and do something with it.

We are going to do something really simple, once the user has selected two items we will go to another screen where we will show the sum of both elements.

I will let you do the implementation of that new Activity (or fragment if you want) so I'm just going to focus on the observer implementation.

Below you can see the observer code. When the selection changes, it will check the size of the selection and if it's equal to two it will then launch the new activity.

```
tracker?.addObserver(
    object : SelectionTracker.SelectionObserver<Long>() {
        override fun onSelectionChanged() {
            super.onSelectionChanged()
            val items = tracker?.selection!!.size()
```

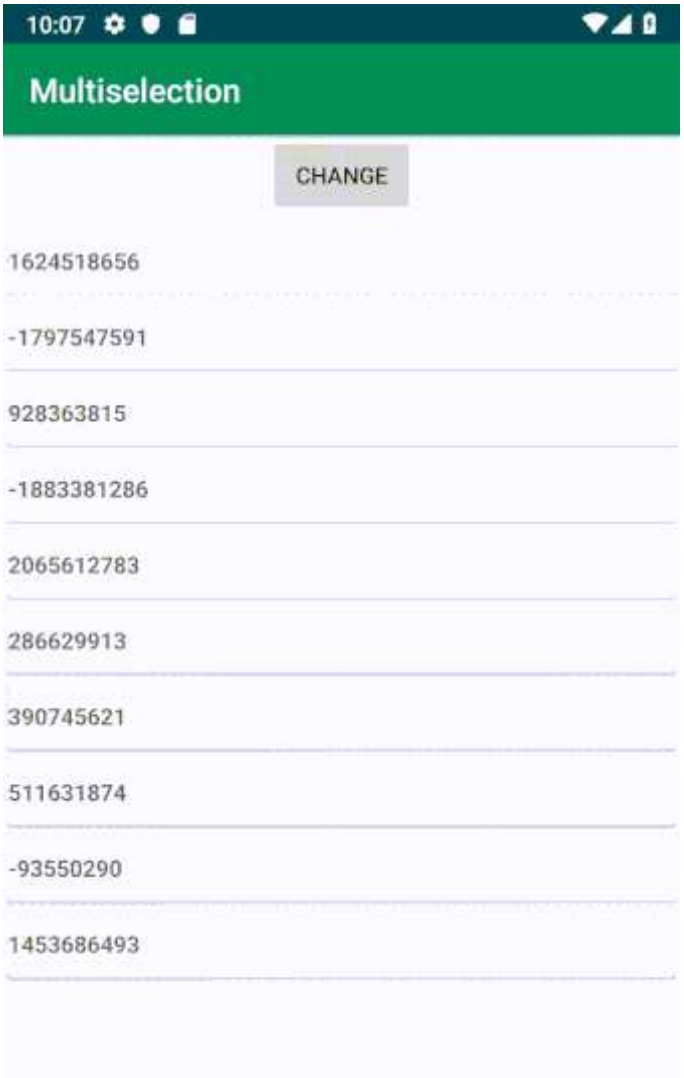
```
                if (items == 2) {
                    launchSum(tracker?.selection!!)
                }
            }
        })
```

For the `launchSum` function we map the selection to an ArrayList<Int> that we will pass to the new activity so it can do the sum.

```
private fun launchSum(selection: Selection<Long>) {
    val list = selection.map {
        adapter.list[it.toInt()]
    }.toList()
    SumActivity.launch(this, list as ArrayList<Int>)
}
```

You can get the code, including the new activity, from the `selection-sum` branch:

https://github.com/marcosholgado/multiselection/tree/selection-sum



Multi-selection with the new SumActivity

Retaining the state across lifecycle events

Up until this point we are not persisting the state across the different Android lifecycle events. If for instance we rotate the app we will lose the current selection. In fact if you try to rotate the app it will crash with:

```
java.lang.NullPointerException: Attempt to invoke virtual method 'int
androidx.recyclerview.widget.RecyclerView$ViewHolder.getAdapterPosition()' on a
null object reference
```

This crash happens in the `StableIdKeyProvider` that we get for free from the selection library when it tries to remove items that are no longer selected while the view is being detached. Rather than keep using `StableIdKeyProvider` we are going to go ahead and replace it with our own `ItemKeyProvider`. The implementation, as you can see below, it's quite simple.

```
class MyItemKeyProvider(private val recyclerView: RecyclerView) :
    ItemKeyProvider<Long>(ItemKeyProvider.SCOPE_MAPPED) {

    override fun getKey(position: Int): Long? {
        return recyclerView.adapter?.getItemId(position)
    }

    override fun getPosition(key: Long): Int {
        val viewHolder = recyclerView.findViewHolderForItemId(key)
        return
            viewHolder?.layoutPosition ?: RecyclerView.NO_POSITION
    }
}
```

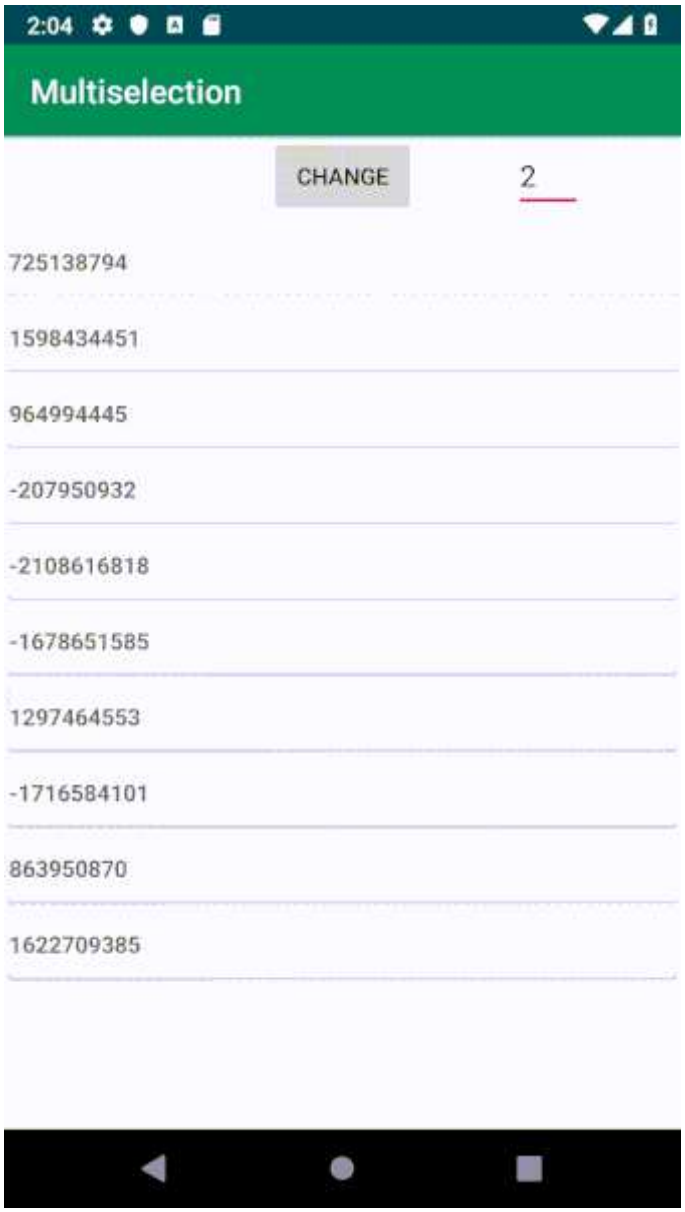
Other crashes and solutions

There is another way we can recreate the crash that we saw before. Let's say that we want to change on runtime the amount of items the user can select. To do that we could create an `EditText` with the amount of rows that are selectable so the user can change that number whenever he wants to select more than two items.

If we were still using the `StableIdKeyProvider`, after launching the app and tapping on the EditText, the app would crash with the same error that we got earlier. The solution as we saw before is to create our own `ItemKeyProvider`.

The final code can be found here:

https://github.com/marcosholgado/multiselection



Caveats and final thoughts

I decided to use the selection library because I thought it would give me a quick solution for what I needed in a quickly manner. Unfortunately **I ended up spending more time than what I wanted dealing with issues** like the crashes mentioned above.

Another problem is the fact that the only way you can “enable” the multi-selection mode is by long pressing on the first item that you want to select. I ideally wanted to have a button to enable that mode but doing it just wasn’t worth the effort. In case you want to give it a go this is how it works:

There is an `TouchInputHandler` class that handles the single tap and long press events. On the single tap you will find this line of code:

```
if (mSelectionTracker.hasSelection()) { ... }
```

You need that if statement to return true in order to be able to select something. The long press event doesn’t check that and directly adds the item to the selection list. The `hasSelection()` method comes from the `DefaultSelectionTracker` which is the one that’s used by default when we create a new tracker using the `SelectionTracker.Builder`. This is the implementation of that method:

```
@Override
public boolean hasSelection() {
    return !mSelection.isEmpty();
}
```

Ideally you should be able to override the single tap method in `TouchInputHandler` but the class is final so you would have to extend from `MotionInputHandler` instead and re-implement all the methods. But that doesn’t really matter because we can’t provide our tracker with a specific `TouchInputHandler` since it is created and assigned when we call `build()` on the `SelectionTracker`. What we really have to do is create our own tracker and not use the `SelectionTracker.Build` which create a `DefaultSelectionTracker` by default. As you can see that’s probably too much effort and writing your own solution should probably be easier.

Overall, if you are looking for a quick solution the selection library should work just fine, specially after reading this article and dealing with the crashes that I mentioned. There is room for some customization but the overhead may be a little too much.

That was everything for today, If you have any questions please reach out on Twitter or leave a comment.

Marcos Holgado (@Orbycius) | Twitter

The latest Tweets from Marcos Holgado (@Orbycius). Señor Android Developer for @SkySports and @SkyNews. Public speaker...

www.twitter.com

Android Android App Development AndroidDev Android Development Android Apps