## ViewModel

App components like Activities and Fragments have a lifecycle managed by the Android Framework. The Framework may decide to destroy or recreate them based on some user actions or device events which are completely out of your control.

Since these objects might be destructed or re-created by the operating system, any data you hold in them will be lost. For instance, if you have a list of users in your Activity, when the Activity is re-created for a configuration change, the new Activity will have to re-fetch the list of users. For simple data, it can use the *onSaveInstanceState* method and restore it from the bundle in *onCreate* but this is just suitable for small information like UI state, not for potentially big data like a list of users.

Another problem is that, these UI Controllers (activities, fragments etc) frequently need to make some asynchronous calls which may take some time to return. The UI Controller needs to manage these calls and clean them up when it is destroyed to avoid potential memory leaks.

This requires a lot of maintenance and in the case of re-create for a configuration change, it is wasted resources since it will need to re-issue the same call.

Last but not least, these UI Controllers already have a lot of responsibility to react to user actions or handle the operating system communication. When they also need to handle their resources manually, it bloats the class, creating god activities / fragments. This also makes testing a lot harder.

It would be easier and more efficient to separate out view data ownership from UI controller logic. Lifecycles provides a new class called *ViewModel* which is a helper class for the UI Controller that is responsible to prepare the data for the UI. The ViewModel is automatically retained during configuration changes so the data it holds is immediate available to the next Activity instance. For the example we’ve mentioned above, it would be the ViewHolder’s responsibility to acquire and keep the list of users, not the Activity or the Fragment.

|  |
| --- |
| public class MyViewModel extends ViewModel {  private final MutableLiveData<List<User>> users;  public LiveData<List<User>> getUsers() {  if (users == null) {  users = new MutableLiveData<List<Users>());  loadUsers();  }  return users;  }   private void loadUsers() {  // do async operation to fetch users  } } |

Now the Activity can access this list as follows:

|  |
| --- |
| public class MyActivity extends AppCompatActivity {  public void onCreate(Bundle savedInstanceState) {  MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);  model.getUsers().observe(this, users -> {   // update UI  });  } } |

If Activity is re-created, it will receive the same *MyViewModel* instance that was created by the previous Activity.

When the owner Activity is finished, the Framework will call ViewModel’s *onCleared* method so that it can cleanup resources.

Since the ViewModel outlives specific Activity and Fragment instantiations, it should never ever reference a View or any class that may hold a reference to the Activity context.

If the ViewModel needs the Application context (e.g. to find a system service), it can extend the *AndroidViewModel* class and have a constructor that receives the Application in the constructor (Application class extends Context).

### Sharing Data Between Fragments

It is very common that 2 or more fragments in an activity needs to communicate with each-other. This is never trivial as both fragments needs to define some interface description and the owner Activity must bind the two together. Moreover, both fragments must handle the case where the other fragment is not created yet or not visible.

This is a common pain point which can be addressed using ViewModels.

Imagine a common case of master detail fragment where we have a Fragment in which user selects an item from a list and another fragment that displays the contents of the selected item.

These Fragments can share a ViewModel using their Activity scope to handle this communication.

|  |
| --- |
| public class SharedViewModel extends ViewModel {  private final MutableLiveData<Item> selected = new MutableLiveData<Item>();   public void select(Item item) {  selected.setValue(item);  }   public LiveData<Item> getSelected() {  return selected;  } } |

## 

|  |
| --- |
| public class MasterFragment extends Fragment {  private SharedViewModel model;  public void onActivityCreated() {  model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);  itemSelector.setOnClickListener(item -> {  model.select(item);  });  } } |

## 

|  |
| --- |
| public class DetailFragment extends LifecycleFragment {  public void onActivityCreated() {  SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);  model.getSelected().observe(this, { item ->  // update UI  });  } } |

Notice that both fragments are using *getActivity()* while getting the *ViewModelProvider.* This means both of them will receive the same *SharedViewModel* instance which is scoped to the Activity.

Below are the benefits of this approach:

* The Activity does not need to do anything, nor know anything about this communication.
* Fragments don’t need to know about each-other besides the *SharedViewModel* contract. If one of them disappears, the other one will keep working as usual.
* Each fragment have their own Lifecycle and is not affected by the lifecycle of the other one. In fact, you may have a UI where one fragment replaces the other one and it will work without any problems.

### The lifecycle of a ViewModel

ViewModels are scoped to the Lifecycle passed to the ViewModelProvider when getting the ViewModel. The ViewModel will stay in memory until the Lifecycle it’s scoped to goes away permanently - in the case of an Activity, once it finishes; in the case of a Fragment, once it’s detached.

