TL;DR (Too long, didn’t read)

Databinding android hive uses getter setter method of pojo to set values.
Observable updates the ui as soon as the data gets changed but it is not life cycle aware.
ViewModel is life cycle aware but it doesn’t provide context.
AndroidViewModel provides access to context.
LiveData updates the ui as soon as the data gets changed and it is lifecycle aware but it does not provide few getter setters.
MutableLiveData Updates the ui as soon as data gets changed, it is life cycle aware and gives few useful getter setter methods.

Databinding

How?

1. Enable data binding in app/build.gradle -> android block:

```
android {
    dataBinding {
        enabled = true
    }
```

2. We wrap our xml file with <layout> </layout> tag.
3. <layout> tag will be the parent tag which will contain <data> tag. <data> </data> and our xml content.
4. <data> tag will contain variable tag. <variable></>.
5. Variable tag has two attributes: name and type. We can give any value in name and the full path of our pojo to type variable.
6.

https://www.journaldev.com/20292/android-mvvm-design-pattern

There will be three things:
1. Model : pojo, modal, data… whatever you call it
2. View: Activity, fragment, dialog etc…
3. ViewModel : Link between View and Model

https://www.journaldev.com/11780/android-data-binding
https://www.androidhive.info/android-working-with-databinding/

Official Documentation:
https://developer.android.com/topic/libraries/data-binding/start

https://google-developer-training.gitbooks.io/android-developer-advanced-course-practicals/unit-6-working-with-architecture-components/lesson-14-room,-livedata,-viewmodel/14-1-a-room-livedata-viewmodel/14-1-a-room-livedata-viewmodel.html

(Note the root in index there: topic/libraries/ArchitectureComponents)

Data binding:

What does it do?

1. No findViewById boiler plate codes
2. Directly set method that is setting up the value for view in xml view
   https://www.androidhive.info/android-databinding-in-recyclerview-profile-screen/
3. Extend BaseObservable in Model class
4. Bindable annotation for getters (Accessors)
5. Two way binding (@={}) in xml
6. Notify call in setter method (notifyPropertyChanged(BR.field))
7. Get pojo, call setter method, call binder.set method.
8. Source: https://developer.android.com/topic/libraries/data-binding/observability
   There are total 3 types those can be observable which are: objects, fields and collections.
9. https://developer.android.com/topic/libraries/data-binding/two-way
   We can also perform Two-Way Binding, that is whenever there is a change in whether view or data, both will be notified.

   Binding Adapter References:

   Focus on CustomAdapter class

   https://stackoverflow.com/questions/32520759/android-data-binding-errorexecution-failed-java-lang-runtimeexception

   https://stackoverflow.com/questions/42380056/android-data-binding-using-bindingadapter-with-the-android-namespace-does-no

   Two-way Data binding

   https://www.bignerdranch.com/blog/two-way-data-binding-on-android-observing-your-view-with-xml/

Live Data:

What does it do?

1. https://developer.android.com/topic/libraries/data-binding/architecture#livedata
   Unlike objects that implement `Observable`—such as observable fields—**`LiveData` objects know about the lifecycle** of the observers subscribed to the data changes. All we need to get benefits is just the replacement of *ObservableField* by *LiveData<T>*. LiveData is used in conjunction with ViewModel.
2. https://developer.android.com/topic/libraries/architecture/livedata#the_advantages_of_using_livedata
   However, **LiveData has no public methods. Hence, we use MutableLiveData** that gives us useful methods such as setValue(T) and postValue(T). All we need to get benefits is just the replacement of *LiveData<T> by MutableLiveData<T>.*
3. **Sagra note:** If we are using LiveData or MutableLiveData, we do not bind values in xml?
4. No inheritance
5. No bindable Annotation
6. No notifyPropertyChanged call in setter method
7. No xml link
8. No need to make pojo object, setter method call and then binder.set call. Replacing by single set method call on ViewModel object.
9. Receive response in observer method and do whatever you want to do. So the flexibility if you want to do something different or something additional instead of simply updating the ui.
10.

ViewModel:

Simplest tutorial I have seen so far:
https://www.journaldev.com/21168/android-livedata

Sagar note:
getter setter methods for data that we set on our views

https://developer.android.com/topic/libraries/data-binding/architecture#livedata
ViewModel is also an observable which means, it can notify the changes to ui if it is bound using Data binding library.

1. The sole purpose of ViewModel is to survive with data across configuration changes of view.
2. Instead of observing observable field, view will observe ViewModel.
3. View will get data from ViewModel and ViewModel will get data from MutableLiveData, LiveData or from your pojo class. How?

Inspiration: https://android.jlelse.eu/android-architecture-components-viewmodel-e74faddf5b94

https://stackoverflow.com/questions/44148966/androidviewmodel-vs-viewmodel

```
public class UsersViewModel extends ViewModel {

    private List<User> userList;

    public List<User> getUserList() {
       if (userList == null) {
            usersList = loadUsers();
        }
        return userList;
    }

    private List<User> loadUsers() {
        // do something to load users
    }
}
```

We can see here that we are just extending the *ViewModel* class and our *ViewModel* class will have all methods related to data such as setData, getData, updateData etc… which will be called by our *View* through of course our *ViewModel* class. How?

Inspiration: https://android.jlelse.eu/android-architecture-components-viewmodel-e74faddf5b94

```
UsersViewModel usersViewModel =
            ViewModelProviders.of(this).get(UsersViewModel.class);

        showUsers(usersViewModel.getUserList());
```

Above code will be written in our *View* class.

4. Now, as we know the benefits of *MutableLiveData*, is there any way so that we can use *MutableLiveData* to notify about changes and to get our ui automatic update in that way in conjunction with *ViewModel* so that we can survive the data through configuration changes of our *View?*
5. Fortunately, Yes! Google Android Team has already thought about it and to implement our purpose, all we need to do is using *MutableLiveData* in *ViewModel*. How?
Inspiration: https://android.jlelse.eu/android-architecture-components-livedata-1ce4ab3c0466

```
public class UsersViewModel extends ViewModel {
   private MutableLiveData<List<User>> userLiveData =
       new MutableLiveData<>();

   //Notifies changes to view
   private MutableLiveData<User> userMutableLiveData;

    public LiveData<List<User>> getUserList() {
        return userLiveData;
    }

    //This is how view will get data and will set observer on it to observe any changes
    LiveData<User> getUser() {
       if (userMutableLiveData == null) {
          userMutableLiveData = new MutableLiveData<>();
       }
       return userMutableLiveData;
    }

}
```

6. *View* will get data from *ViewModel* and will set the *observer* on it to be notified about any changes on that data. Like below:

   Inspiration: https://www.journaldev.com/22561/android-mvvm-livedata-data-binding#viewmodel

```
usersViewModel.getUser().observe(this, new Observer<User>() {
   @Override
   public void onChanged(@Nullable User user) {
       if (!user.isValid() && (user.getEmail().length() > 0 ||
user.getPassword().length() > 0)) {
           Toast.makeText(getApplicationContext(), "email : " + user.getEmail() + "
password " + user.getPassword(), Toast.LENGTH_SHORT).show();
       }
   }
});
```

7. More: https://www.journaldev.com/22561/android-mvvm-livedata-data-binding
8.
9.

How?

1. Implement dependency:
2. In layout file, have layout as root tag.
3. Data binding example: https://www.androidhive.info/android-working-with-databinding/
4. https://github.com/ravi8x/Android-DataBindng-RecyclerView

5. <data> tag inside <layout> tag will have <variable> tag.
6. So, it will be like: layout -> 1. data -> 1.1 variable, 2. normal layout
7. Variable tag will have two attributes: Name and Type
8. For name attribute, give alias (any variable name) (A setter method will be generated on the base of alias you give here) (You will use the same alias to access field of class that you specify as *Type* under *Data* tag)
9. For type, we will refer our pojo class for that layout
10. As a value, bind your view (controller, widget or whatever you call it) with the method or field like: `"@={alias.field}"`
11. Java 8 Lambda expression: http://tutorials.jenkov.com/java/lambda-expressions.html
12. How to use java 8 in Android: https://android.jlelse.eu/features-of-revolutionary-release-java-8-for-android-d8abe06c34c5
13. More on Java 8 and Android: https://code.tutsplus.com/tutorials/java-8-for-android-cleaner-code-with-lambda-expressions--cms-29661
14.

Few libraries and terminologies:
1. Data binding
2. Arch Lifecycle components
3. Android Observable
4. ViewModel
5. LiveData

1. Every screen, especially wherever there is some kind of data (which alway will have mostly everywhere), there will be a ViewModel.
   For example, login screen will have LogInViewModel.

Model:
1. Getters are Accessors and Setters are Mutators.
2. Pojo class can have accessors and mutators.
3. Accessors can have @Bindable annotation.
4. When accessor has @Bindable annotation, associated binding expression gets refreshed on change of that accessor value who has @Bindable annotation.
5. Don't forget to make your accessors null proof.

Examples:
1. Ordinary Vs Data binding example
2. Ordinary Vs ViewModel example
3. Ordinary Vs LiveData example
4. Data binding with ViewModel
5. Data binding, ViewModel and LiveData

Data binding:

BaseObservable:

Source: https://www.androidhive.info/android-databinding-in-recyclerview-profile-screen/

1. Extend BaseObservable
2. Bindable annotation for getters (Accessors)
3. Two way binding (@={})
4. Notify call in setter method (notifyPropertyChanged(BR.field))
5. Get pojo, call setter method, call binder.set method.

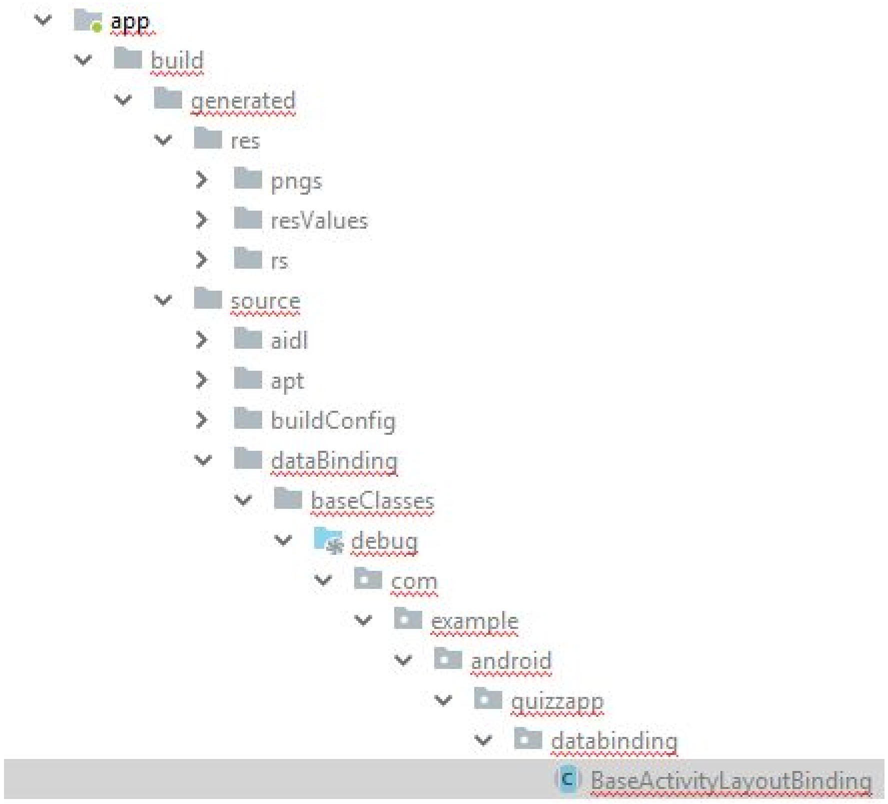
MutableLiveData field:

Inspiration: https://www.journaldev.com/22561/android-mvvm-livedata-data-binding

1. No inheritance
2. No bindable Annotation
3. No notifyPropertyChanged call in setter method
4. No xml link
5. No need to make pojo object, setter method call and then binder.set call. Replacing by single set method call on ViewModel object.
6. Receive response in observer method and do whatever you want to do. So the flexibility if you want to do something different or something additional instead of simply updating the ui.

Data binding (Next to introduction level)

1. You cannot have same name space in xml that uses data binding
2. Location



```
app
  build
    generated
      res
        pngs
        resValues
        rs
      source
        aidl
        apt
        buildConfig
        dataBinding
          baseClasses
            debug
              com
                example
                  android
                    quizzapp
                      databinding
                        BaseActivityLayoutBinding
```

3. Unable to find data binding class?

https://medium.com/@douglas.iacovelli/how-to-fix-data-binding-errors-after-android-studio-3-1-update-b922173b36c5

4. Common guide, problems and solutions of data binding: https://guides.codepath.com/android/Applying-Data-Binding-for-Views
5. If you are binding the method for fragment, note that the method will be looked into activity and not in that fragment.
6. Take care of infinite loop in two-way data binding
7. Data binding with view stub https://stackoverflow.com/questions/34712952/android-data-binding-how-to-use-viewstub-with-data-binding/34713054
8. Communication between fragments through shared viewmodel https://developer.android.com/training/basics/fragments/communicating

9. Databinding bindingAdapter EditText Cursor Method https://stackoverflow.com/questions/32321440/edittext-cursor-resetting-to-left-after-android-data-binding-update
10.
11.

Questions:

1. Where to put method that opens up dialog to set image options: in view or in vm?
2. Common view model and multiple observers
3.