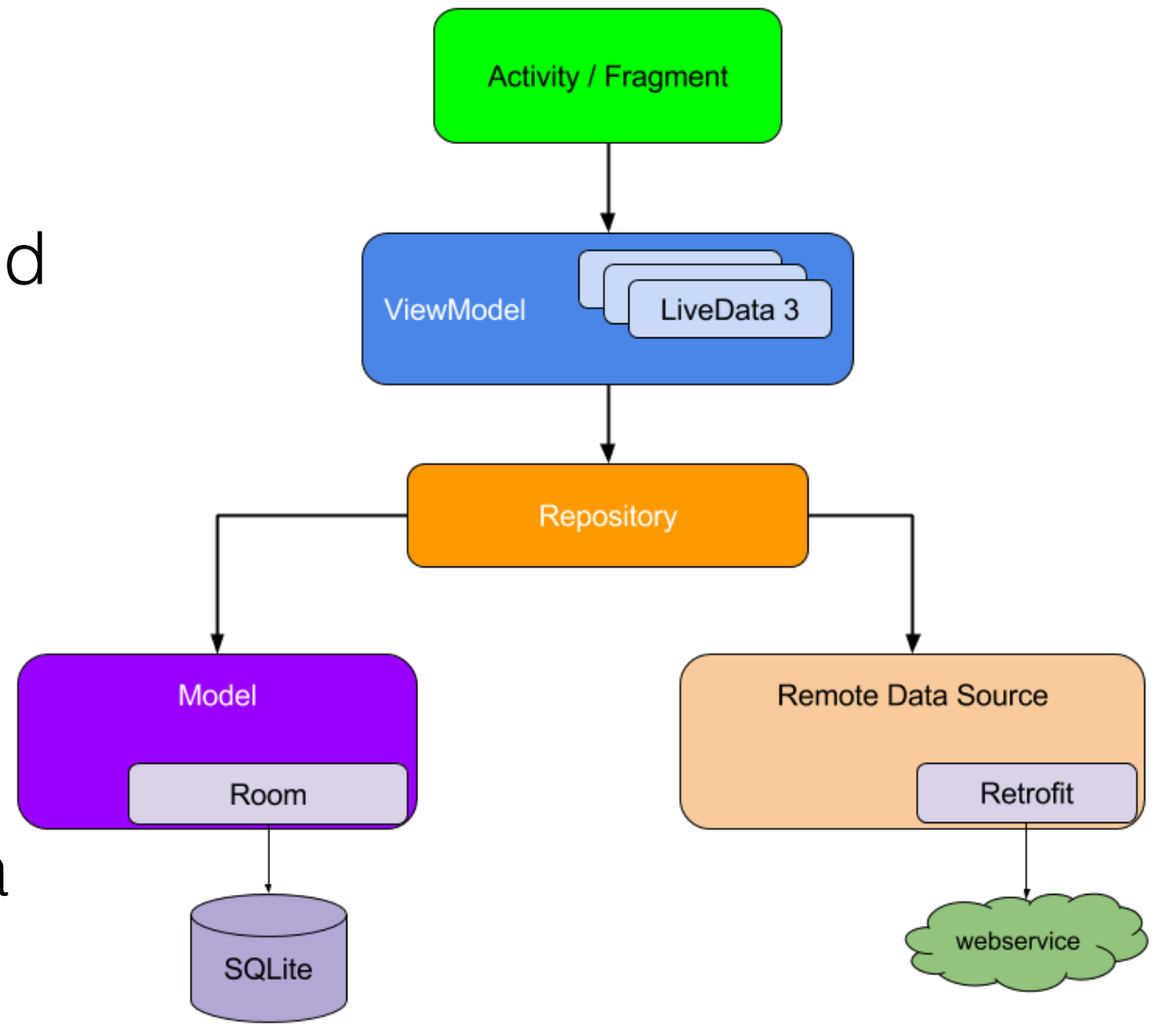
Another Kind of Model

Mobile Computing - Android

Android Architecture Components

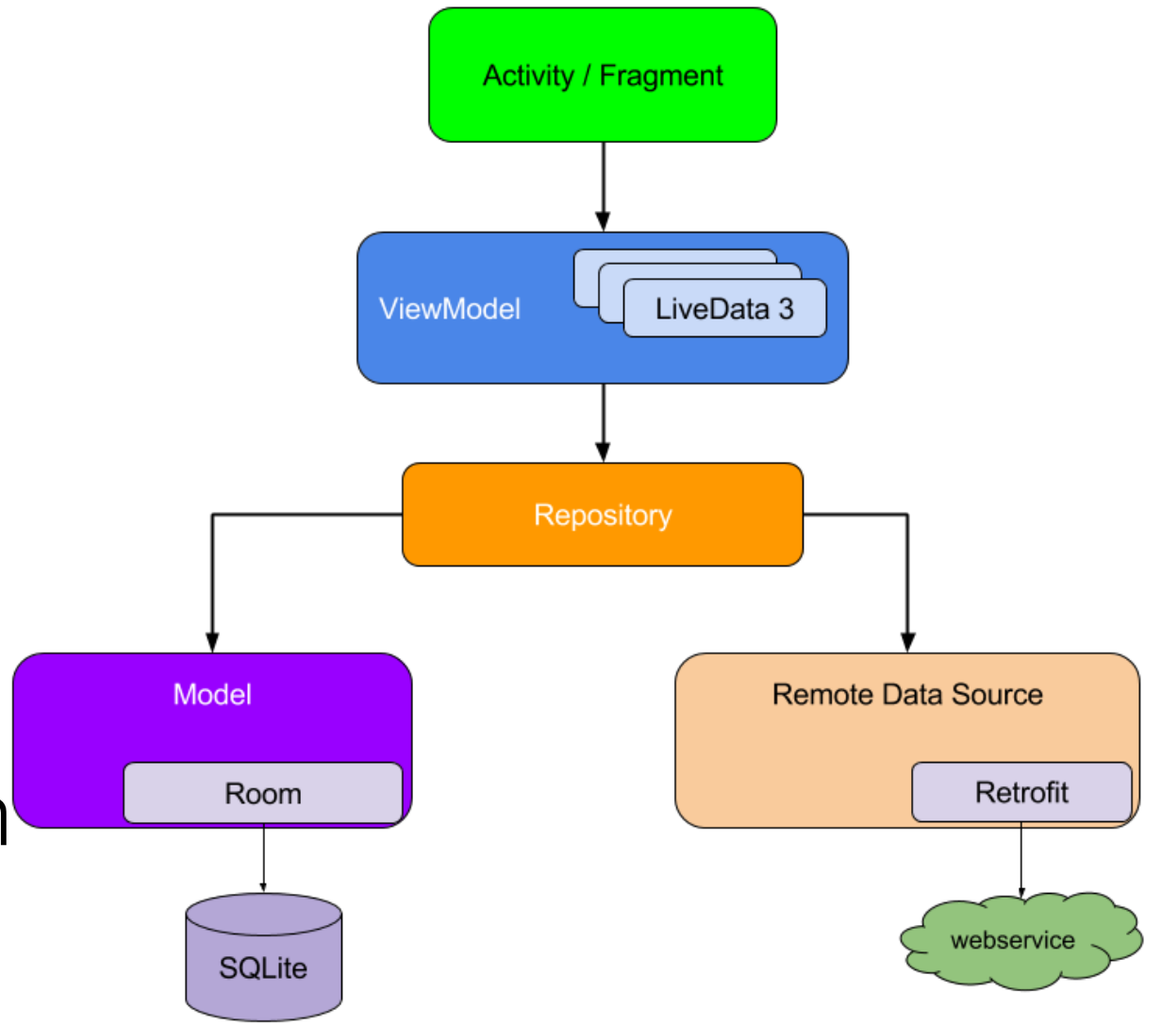
- A relatively new feature are Android Architecture components that can be used to implement a standardized architecture.
- ViewModel hold data that affect views.
- Repositories coordinate data sources.
- Room abstracts access to SQL database



Android Architecture Components

- Were part of the arch components package, but have been moved into the androidx package.

- Mostly the same, but in a different location. Make sure to import from the right package.



Publish-Subscribe

- One pattern that can help decouple components is publish-subscribe. Observer is a variant of this.

- A component can provide a named service. When there are changes, the component will publish the change.

- Other components will subscribe to a service. They register with a call-back that will be invoked when a change is published.

ViewModel

- A way to create and manage instance data that will be used in a fragment or activity. The view model will persist even if the associated activity or fragment is destroyed in a lifecycle event. On recreation, the view model will reconnect.

- The view model should have no awareness of the UI for the activity or fragment it is associated with.

- The activity or fragment will ***observe*** the data in the view model and respond to changes.

- This is a place where business logic can live.

- View models can be used as a communication layer between fragments.

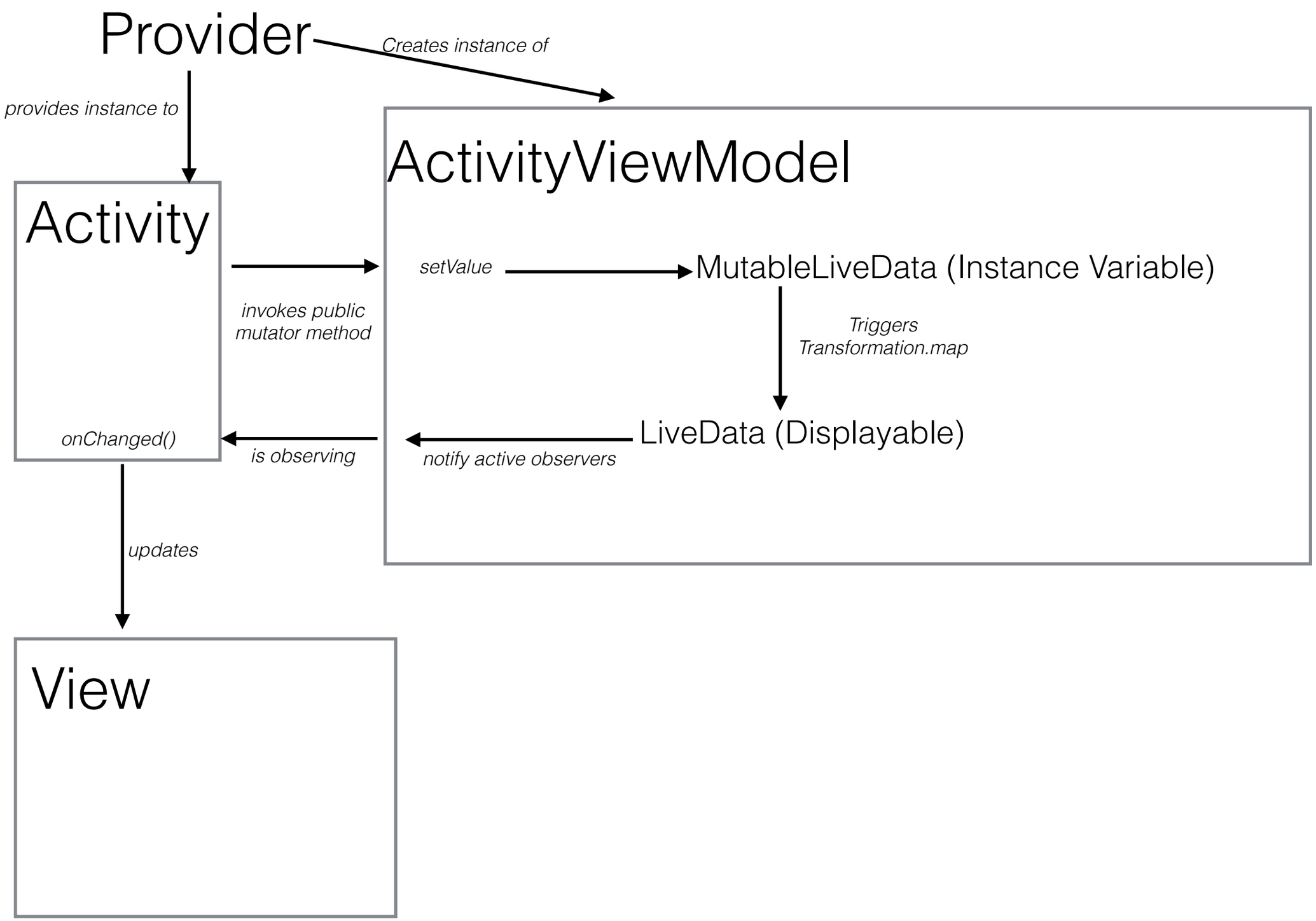
LiveData

- A wrapper that mediates access to data. Live data may change and an activity/fragment can have an observer that will listen and respond to change. When the observer is registered, the onChange method will be invoked to access the current value.

- Designed to work with UI related data and not things like data base request. (See documentation about Room.)

- Live Data is life-cycle aware and will not notify observers that are in an inactive state.
- Observers are notified on becoming active or after a configuration change.
    - getValue() - get the value held.
- Changes to live data will only happen inside the view model.
- MutableLiveData allows outside agents to change the data using the methods
    - setValue(value) - change the value now. Only allowed in the main thread.
    - postValue(value) - Create a task that will run on the main thread eventually. If multiple posts happen before the task is executed, only the last dispatch will happen.

How it fits together



PageViewModel

Our PlaceHolderFragment will have two instance variables: mIndex, mText

```
public class PageViewModel extends ViewModel {

    private MutableLiveData<Integer> mIndex = new MutableLiveData<>();
    private LiveData<String> mText =
        Transformations.map(mIndex, new Function<Integer, String>() {
            @Override
            public String apply(Integer input) {
                return "Hello world from section: " + input;
            }
    });

    public void setIndex(int index) {
        mIndex.setValue(index);
    }

    public LiveData<String> getText() {
        return mText;
    }
}
```

mText depends on mIndex

mIndex is mutable, we provide a method where an outside agent can change it.

An observer can react to a change in mText

PlaceHolderFragment

```
* A placeholder fragment containing a simple view.
 */
public class PlaceholderFragment extends Fragment {

    private static final String ARG_SECTION_NUMBER = "section_number";

    private PageViewModel pageViewModel;

    public static PlaceholderFragment newInstance(int index) {
        PlaceholderFragment fragment = new PlaceholderFragment();
        Bundle bundle = new Bundle();
        bundle.putInt(ARG_SECTION_NUMBER, index);
        fragment.setArguments(bundle);
        return fragment;
    }
```

Our PlaceHolderFragment will have a view model

PlaceHolderFragment

Use the basic default factory.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    pageViewModel = new ViewModelProvider(this).get(PageViewModel.class);
```

Make or reacquire a ViewModel associated with this fragment instance

```
    int index = 1;
    if (getArguments() != null) {
        index = getArguments().getInt(ARG_SECTION_NUMBER);
    }
    pageViewModel.setIndex(index);
}
```

Mutate the index

PlaceHolderFragment

```
@Override
public View onCreateView(
        @NonNull LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View root = inflater.inflate(R.layout.fragment_main, container, false);
        final TextView textView = root.findViewById(R.id.section_label);
        pageViewModel.getText().observe( getViewLifecycleOwner(),
          new Observer<String>() {
            @Override
            public void onChanged(@Nullable String s) {
                textView.setText(s);
            }
        });
        return root;
    }
}
```

Our PlaceHolderFragment will register as an observer for the LiveData mText from the view model

Connecting to the Activity

```
    private FirstListener myActivity;
    private MainViewModel mainViewModel;

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        myActivity=(FirstListener)context;

        MainActivity mainActivity = (MainActivity) context;

        // Reconnect to the MainViewModel
        mainViewModel = new
            ViewModelProvider(myActivity()).get(MainViewModel.class);

}
```

Must wait until we have attached so we know our activity

Now we can use the public mutator methods of the MainViewModel class as well as any methods that the Listener promises

Transformations

- Provides the ability to functionally create a LiveData object from another.
- map applies a function to a LiveData source and returns a new dependent LiveData object from it. If the source changes, the dependent objects will use the map and change as well.

```
MutableLiveData userLiveData = ...;
LiveData userName = Transformations.map(userLiveData, user -> {
     return user.firstName + " " + user.lastName; // Returns String
});
```

If userLiveData changes, we use that to update userName

```
void setUser(String user) {
     this.userLiveData.setValue(user);
}
```

Public mutator that triggers the change.

Transformations

- switchMap - similar, but used to connect to a backing source in a repository.

userLiveData depends on userIdLiveData

```
MutableLiveData userIdLiveData = ...;
LiveData userLiveData = Transformations.switchMap(userIdLiveData,
    id -> repository.getUserById(id)); // Returns LiveData
```

If userIdLiveData changes, we switch to the LiveData we got from the backing repository

```
void setUserId(String userId) {
     this.userIdLiveData.setValue(userId);
}
```

Public mutator that triggers the change.

Transformations

If you want to use a lambda expression like: `(x) -> 2*x`

You will need to make sure that you are using Java 1.8. To fix this in Android Studio, go to project structure and select Modules. Change both the source and target compatibility to be at least Java 1.8.

MediatorLiveData

- A specialized LiveData class that can merge value updates from a number of different LiveData sources.

```
LiveData liveData1 = …;
LiveData liveData2 = …;
```

```
MediatorLiveData liveDataMerger = new MediatorLiveData<>();
 liveDataMerger.addSource(liveData1, value -> liveDataMerger.setValue(value));
 liveDataMerger.addSource(liveData2, value -> liveDataMerger.setValue(value));
```

if the source emits a value the liveDataMerger will be updated

MediatorLiveData

- Can use an observer to specialize the behavior for a source.

Use an observer on liveData1

```
liveDataMerger.addSource(liveData1, new Observer() {
     private int count = 1;

     @Override public void onChanged(@Nullable Integer s) {
         count++;
         liveDataMerger.setValue(s);
         if (count > 10) {
             liveDataMerger.removeSource(liveData1);
         }
     }
});
```

Use the first 10 values we get from the source. Then remove it

Backing Data

- PagedList - a class that can connect with a data source and provide a chunk of data at a time. Can be used by a RecyclerView with a PagedListAdapter.

- Room - An observable persistent data base. Provides an abstraction layer that hides the SQL database.

- RetroFit - An API that will allow an observable data source that fetches information from a backend server.

Resources

- Android Architecture Components
- Android Developer Live Data
- Mediator Live Data
- Transformations
- Android Developer LiveModel overview
- Setting up a Room database