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

Two-Way Data Binding on Android: Observing Your View with XML



[Andrew Bailey](https://www.bignerdranch.com/about/the-team/andrew-bailey)

Aug 9, 2017 • [Android](https://www.bignerdranch.com/blog/categories/android)

If you’ve used Data Binding in an Android app before, you’ll know how it makes your life easier by simplifying the problems you face when building your UI. Not only does it give you a type-safe, compile-time verified replacement to the standard findViewById method, but can also take care of all the heavy lifting in keeping your views up-to-date by seamlessly integrating your Java/Kotlin code with your XML layouts.

It’s even the backbone of the Model View ViewModel (MVVM) pattern on Android. If you haven’t tried out Data Binding yet, you can read more about it in [one of our other blog posts](https://www.bignerdranch.com/blog/descent-into-databinding/), or on Google’s [Data Binding documentation](https://developer.android.com/topic/libraries/data-binding/index.html).

Data Binding is built around the idea of using data from a regular Java/Kotlin object to set attributes on your layouts, and that’s the extent to which most people use it. This is great because it allows you to define your view logic independently from the Android Framework (a boon for unit testing), but what if your view needs to set attributes on your object?

This is where two-way Data Binding comes in. Two-way Data Binding is a technique of binding your objects to your XML layouts so that both the object can send data to the layout, and the layout can send data to the object. You’ll see a suboptimal way of setting this up first, and then take a look at the built-in two-way binding (@={variable}) syntax. The examples here are made with the MVVM architecture in mind, but they apply to any object that’s being attached to a view with Data Binding.

Simple Two-Way Data Binding with Listeners: A Quick and Dirty Approach

A quick way to achieve two-way Data Binding works in the same way as regular, “one-way” Data Binding. You probably wouldn’t want to use this in your app (as you’ll see later on), but instead as an intermediate step towards achieving your first two-way Data Binding setup. Consider that you’re making a password creation screen, and you want to show a password strength indicator as they type. To start, your layout looks like this:

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<data>

<variable

name="viewModel"

type="com.example.PasswordViewModel" />

</data>

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" />

<TextView

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="@{viewModel.passwordQuality}"/>

</LinearLayout>

</layout>

And your view model looks like this:

**public** **class** **PasswordViewModel** **extends** BaseObservable **{**

**private** String password**;**

@Bindable

**public** String **getPasswordQuality()** **{**

**if** **(**password **==** **null** **||** password**.**isEmpty**())** **{**

**return** "Enter a password"**;**

**}** **else** **if** **(**password**.**equals**(**"password"**))** **{**

**return** "Very bad"**;**

**}** **else** **if** **(**password**.**length**()** **<** 6**)** **{**

**return** "Short"**;**

**}** **else** **{**

**return** "Okay"**;**

**}**

**}**

**public** **void** **setPassword(**String password**)** **{**

**this.**password **=** password**;**

notifyPropertyChanged**(**BR**.**passwordQuality**);**

**}**

**}**

Just like you can bind a String to an EditText, you can also bind a TextWatcher to an EditText. Just add this method to your view model:

@Bindable

**public** TextWatcher **getPasswordTextWatcher()** **{**

**return** **new** **TextWatcher()** **{**

@Override

**public** **void** **beforeTextChanged(**CharSequence s**,** **int** start**,** **int** count**,** **int** after**)** **{**

*// Do nothing.*

**}**

@Override

**public** **void** **onTextChanged(**CharSequence s**,** **int** start**,** **int** before**,** **int** count**)** **{**

setPassword**(**s**.**toString**());**

**}**

@Override

**public** **void** **afterTextChanged(**Editable s**)** **{**

*// Do nothing.*

**}**

**};**

**}**

And update the EditText in your layout to look like this instead:

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

app:textChangedListener="@{viewModel.passwordTextWatcher}" />

When you’re ready to try it out, you can go ahead and click the run button only to be greeted by a fairly long compiler warning. That’s because the Data Binding compiler plugin doesn’t know how to set the textChangedListener property of an EditText since there isn’t a method on EditText that’s called setTextChangedListener.

To fix it, you’ll have to create a BindingAdapter to tell the compiler how to set a TextWatcher on an EditText. You can create a new class with the following method, or add it to your existing binding adapters:

**public** **class** **EditTextBindingAdapters** **{**

@BindingAdapter**(**"textChangedListener"**)**

**public** **static** **void** **bindTextWatcher(**EditText editText**,** TextWatcher textWatcher**)** **{**

editText**.**addTextChangedWatcher**(**textWatcher**);**

**}**

**}**

Now when your layout is visible, you’ll see the password strength indicator updating to match what you type. This is very similar to how you would normally accomplish this using the MVC pattern—you’re creating a listener that updates the layout and attaching that listener to the EditText. The only difference here is that the generated binding class will attach the TextWatcher to the EditText instead of you doing it manually.

One difference between the MVC approach and the MVVM approach in this example is that you have to be very careful about not binding a second TextWatcher. If your view model accidentally calls notifyPropertyChanged(BR.passwordTextWatcher) or notifyChange(), then you’ll add another TextWatcher. This will waste memory, and if your view model does this too often, you’ll notice your layout’s performance decrease dramatically when the user types into the field because of the unnecessary work it’s doing.

Implementing two-way Data Binding in this way is very tedious and error-prone. It only pushes the problem of setting up this TextWatcher from your Activity or Fragment into your Data Binding variables. On top of that, it introduces another layer of complexity by forcing you to be mindful of all your listeners! Luckily, Data Binding has a built-in solution for this problem.

Two-Way Data Binding with Binding Adapters

Instead of having to create an XML attribute to bind a listener, you can take advantage of Data Binding’s built-in support for two-way bindings. Behind the scenes, the generated Binding class for your layout is still going to create a listener, but it will do all the heavy-lifting to maintain and keep track of this listener.

Go back to the original layout and view model you started out with. Instead of adding a getPasswordTextWatcher method to your view model, add a getPassword method as shown below:

@Bindable

**public** String **getPassword()** **{**

**return** password**;**

**}**

Make sure you also update your setPassword method with a notifyPropertyChangedcall as shown below:

**public** **void** **setPassword(**String password**)** **{**

**this.**password **=** password**;**

notifyPropertyChanged**(**BR**.**password**);**

notifyPropertyChanged**(**BR**.**passwordQuality**);**

**}**

Next, update your EditText like this:

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="@={viewModel.password}" />

Note the = in the Data Binding operator. This will cause the EditText to be populated with the value of password from the ViewModel as you’re used to, but the addition of the = tells the generated Binding class to call password’s setter whenever the text changes. This means that your PasswordViewModel’s password field will always contain the text in the EditText.

The next time you run your app, you’ll see that the password strength label still updates without you having to write a single BindingAdapter and without creating a listener yourself. Behind the scenes, your binding class is creating its own TextWatcher that functions similarly to the one that you defined manually and is binding and managing the TextWatcher itself.

Writing Your Own Inverse Binding Adapter

Just like how you have to write your own binding adapters to tell Data Binding how to call setters on views it doesn’t know about, you’ll sometimes have to the same thing for the views getters when you use two-way Data Binding.

Suppose that after a user has created a password, you want to show them legal information before they can start using the app. Since legal information tends to be very long, you decide to add a button that takes the user back up to the top of the page. When the user is already at the top of the screen, you also want to hide the button.

All of this can be accomplished with two-way Data Binding. You can start out by creating a basic layout as shown:

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<data>

<variable

name="viewModel"

type="com.example.LegalViewModel" />

</data>

<FrameLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<ScrollView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="@string/lorem\_ipsum" />

</ScrollView>

<android.support.design.widget.FloatingActionButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:src="@drawable/ic\_scroll\_to\_top"

android:layout\_gravity="bottom|end" />

</FrameLayout>

</layout>

And create an empty view model class:

**public** **class** **LegalViewModel** **extends** BaseObservable **{**

**}**

The first thing you’ll need to do is to add a scrollY member variable to your view model. You can do this just like you would for a regular object:

**public** **class** **LegalViewModel** **extends** BaseObservable **{**

**private** **int** scrollY**;**

@Bindable

**public** **int** **getScrollY()** **{**

**return** scrollY**;**

**}**

**public** **void** **setScrollY(int** scrollY**)** **{**

**this.**scrollY **=** scrollY**;**

notifyPropertyChanged**(**BR**.**scrollY**);**

**}**

**}**

While you’re in the view model, you can also define the rest of the behavior for the scroll to top button. You can add the following method to determine the floating action button’s visibility:

@Bindable

**public** **int** **getScrollToTopFabVisibility()** **{**

**if** **(**scrollY **==** 0**)** **{**

**return** View**.**GONE**;**

**}** **else** **{**

**return** View**.**VISIBLE**;**

**}**

**}**

And another method to act as the click listener for the FloatingActionButton:

**public** **void** **scrollToTop()** **{**

setScrollY**(**0**);**

**}**

When you add the getScrollToTopFabVisibility() method, remember to update your setScrollY() method to dispatch the appropriate notifyPropertyChanged()call as shown below:

**public** **void** **setScrollY(int** scrollY**)** **{**

**this.**scrollY **=** scrollY**;**

notifyPropertyChanged**(**BR**.**scrollY**);**

notifyPropertyChanged**(**BR**.**scrollToTopFabVisibility**);**

**}**

Next, it’s time to update the layout XML to tell the Data Binding compiler how to use your newly created methods. You can start out by updating the floating action button as shown:

<android.support.design.widget.FloatingActionButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:src="@drawable/ic\_scroll\_to\_top"

android:layout\_gravity="bottom|end"

android:visibility="@{viewModel.scrollToTopFabVisibility}"

android:onClick="@{() -> scrollToTop()}"/>

Then, it’s time to set up two-way Data Binding on the scroll position of your ScrollView. The syntax for this is the exact same as before, using the @={variable} operator as shown below:

<ScrollView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

app:scrollY="@={viewModel.scrollY}">

If you try to compile the app right now, you’ll get a compiler error since the Data Binding compiler plugin doesn’t know how to get or observe the vertical scroll position—which is to say that the compiler doesn’t know how to get the scrollY value or set up a listener on it. To fix this, you’ll need to make an Inverse Binding Adapter. This inverse binding adapter will tell Data Binding how to get the current attribute of the view.

Start out by creating a new class for your inverse binding adapter (if you already have a binding adapters class you want to reuse, you can do all of these steps in that class instead if you prefer). Then you can go ahead and create the inverse binding adapter as shown:

**public** **class** **ScrollViewBindingAdapters** **{**

@InverseBindingAdapter**(**attribute **=** "scrollY"**)**

**public** **static** **int** **getScrollY(**ScrollView scrollView**)** **{**

**return** scrollView**.**getScrollY**();**

**}**

**}**

This tells the compiler how to get the value from the view, but it still doesn’t know how to observe the value on the view. To fix this, you’ll need another binding adapter with a method signature that looks like the one shown below. Note that this example is using OnScrollChangeListener, which was added in Android Marshmallow (API 23).

@BindingAdapter**(**value **=** **{**"scrollListener"**,** "scrollYAttrChanged"**,** requireAll **=** **false})**

**public** **static** **setScrollListeners(**ScrollView scrollView**,**

ScrollView**.**OnScrollChangeListener scrollListener**,**

InverseBindingListener inverseBindingListener**)**

There’s a lot going on here, so it’s a good idea to take a closer look before trying to implement this method, starting with the annotation. Those two strings labeled as valueare actually the names of XML attributes that this Binding Adapter is responsible for. Having "scrollListener" lets you manually set an OnScrollChangeListener in your XML with the app:scrollListener attribute.

The "scrollYAttrChanged" label is generated by the Data Binding compiler plugin. Every time you use two-way Data Binding, an internal attribute is created with the AttrChanged suffix. This is the attribute that Data Binding is going to look for when it goes to setup its event listener to update your view model. Behind the scenes, the generated binding class is going to use this attribute to bind its observer.

The last thing in the annotation is the requireAll flag. This is simply telling the compiler that this method can be called without all of the parameters. For example, if your layout sets up two-way Data Binding on the vertical scroll position but doesn’t setup a scroll listener (or vice versa), then Data Binding will still call this method but it will pass in null for the scrollListener.

The first two parameters on this method should look pretty familiar. The ScrollView is the view whose scroll position will be observed, and the scrollListener is the optional listener that you can manually attach. The final parameter is a listener that Data Binding expects a callback on when a value changes. In a nutshell, this method’s responsibility will be to create a new OnScrollChangeListener that both wraps the scrollListenerand calls the inverse binding listener. This implementation can be achieved like this:

@BindingAdapter**(**value **=** **{**"scrollListener"**,** "scrollYAttrChanged"**,** requireAll **=** **false}**

**public** **static** **setScrollListeners(**ScrollView scrollView**,**

ScrollView**.**OnScrollChangeListener scrollListener**,**

InverseBindingListener inverseBindingListener**)** **{**

ScrollView**.**OnScrollChangeListener newListener**;**

**if** **(**inverseBindingListener **==** **null)** **{**

newListener **=** scrollListener**;**

**}** **else** **{**

newListener **=** **new** ScrollView**.**OnScrollChangeListener**()** **{**

@Override

**public** **void** **onScrollChange(**View v**,** **int** scrollX**,** **int** scrollY**,** **int** oldX**,** **int** oldY**)** **{**

**if** **(**scrollListener **!=** **null)** **{**

scrollListener**.**onScrollChange**(**v**,** scrollX**,** scrollY**,** oldX**,** oldY**);**

**}**

inverseBindingListener**.**onChange**();**

**}**

**};**

**}**

scrollView**.**setOnScrollChangeListener**(**newListener**);**

**}**

Every time that inverseBindingListener.onChange() is called, the Binding class for your layout will use the inverse binding adapter from before to get the current vertical scroll position of the view and will pass that value to the setScrollY method in the view model.

If you run the app now and open this layout, you’ll see that everything is behaving exactly as intended—the FloatingActionButton is hidden by default and will appear once the user starts to scroll. This is great, but there’s one really important optimization you should make. Depending on which view you’re using and which attribute your view model is subscribing to, setting up two-way Data Binding like this can cause an infinite loop. What’s worse is that this infinite loop won’t cause your app to crash or even freeze—it will silently consume CPU resources and drain the battery without any obvious problems.

The example with ScrollView doesn’t exhibit this behavior, but suppose that ScrollView implemented a setScrollY method like this:

**public** **class** **ScrollView** **extends** View **{**

**public** **void** **setScrollY(int** y**)** **{**

mScrollY **=** y**;**

**if** **(**mScrollListener **!=** **null)** **{**

mScrollListener**.**onScrollChanged**(this,** mScrollX**,** mScrollY**,** mScrollX**,** mOldScrollY**);**

**}**

**}**

**}**

Every time that this ScrollView.setScrollY method is called, your callback is also triggered. Your callback is triggering the setScrollY method in your LegalViewModelclass from before. For reference, the setScrollY method is implemented like this:

**public** **class** **LegalViewModel** **extends** BaseObservable **{**

*// Other methods omitted*

**public** **void** **setScrollY(int** scrollY**)** **{**

**this.**scrollY **=** scrollY**;**

notifyPropertyChanged**(**BR**.**scrollY**);**

notifyPropertyChanged**(**BR**.**scrollToTopFabVisibility**);**

**}**

**}**

Because of the call to notifyPropertyChanged(BR.scrollY), your layout’s binding class is going to call ScrollView.setScrollY again, which causes this entire cycle to start all over. Because of the way Data Binding implements binding classes, though, it will wait until the next frame before calling setScrollY on the ScrollView.

The solution to this is very simple. In your LegalViewModel, simply update the setScrollY method to do nothing if the value didn’t change:

**public** **class** **LegalViewModel** **extends** BaseObservable **{**

*// Other methods omitted*

**public** **void** **setScrollY(int** scrollY**)** **{**

**if** **(this.**scrollY **!=** scrollY**)** **{**

**this.**scrollY **=** scrollY**;**

notifyPropertyChanged**(**BR**.**scrollY**);**

notifyPropertyChanged**(**BR**.**scrollToTopFabVisibility**);**

**}**

**}**

**}**

It’s a good idea to do this on all the setters that you use with two-way Data Binding. View implementations can change over time so updating one of your libraries or using a different version of Android may cause this infinite loop behavior without any warnings. Adding this check is a simple way to guarantee that your two-way Data Binding won’t cause an infinite loop.

When to Use It

If you’re using MVVM or another similar architecture that takes advantage of Data Binding, then two-way Data Binding is a great way of getting information about your view into your view models. If you’ve manually been binding listeners, then you should consider to the built-in technique for two-way Data Binding provided by the @={variable} syntax. It allows you to remove all references of view listeners from your view models, which in turn makes your view models more simple and easier to test.

If you’re not using MVVM, then two-way Data Binding may not be right for your app. Data Binding on its own works best when you have a model object that can directly be used by the view. If the layout XML has any logic, then suddenly you may find yourself debugging your layouts in addition to Java and Kotlin code. Adding two-way Data Binding without the right architecture can make debugging much more likely and difficult.

When you’re ready to set up two-way Data Binding on one of your layout attributes, there are just a couple of steps. To summarize:

1. Setup one-way Data Binding for the attribute
2. Add a setter to your model—make sure that this setter ignores the value if it hasn’t changed
3. Add an = in your layout to use the @={variable} syntax
4. Create an inverse binding adapter for the attribute if necessary.