Dataflow async for Android™ reference

FEMA Studios S.r.l.s.

Version 1, revision 1

Table of Contents

Dataflow async for Android™

This extension adapts the async library to be used effortlessly on Android™. It contains few methods because most of the things with attributes must be completely customizable. Following this guide you'll learn what this library does, and what is needed on your part to take advantage of the async library on Android™.

Getting started

Setup

To add this extension all you have to do is add the following dependency:

Gradle

```
implementation 'com.femastudios:dataflow-async-android:1.0.0'
```

Further instructions on how to configure you Android™ project can be found here.

View binding

Listening attributes on views

Like what we do for fields, we'll also want to listen for changes in attributes. Recalling that an attribute can be seen as a field of `AttributeData<T>`, we can now see the listen() extension function available on `Views`. Like its field counterpart, the callback is called on the UI thread.

Example:

Kotlin

```kotlin
val tv = TextView(context)
val attr = attributeOf("hello")
tv.listen(attr) { //Will be called on the UI thread
    tv.setText(it)
}
```

Java

```java
TextView tv = new TextView(getContext());
Attribute<String> attr = Attribute.of("hello");
AttributeListenUtils.listen(tv, attr, str -> { //Will be called on the UI thread
    tv.setText(str);
});
```

Async `LifecycleOwner`

If you want your listener to be called asynchronously, you can use the asAsyncLifecycleOwner() function:

Kotlin

```kotlin
val tv = TextView(context)
val attr = attributeOf("hello")
tv.asAsyncLifecycleOwner().listen(attr) { //Will be called on a background thread
    tv.setText(it)
}
```

Java

```java
TextView tv = new TextView(getContext());
Attribute<String> attr = Attribute.of("hello");
AttributeListenUtils.asAsyncLifecycleOwner(tv).listen(attr, str -> { //Will be called on a
background thread
    tv.setText(str);
});
```

View state

View state work exactly the same as they work for fields. Take a look at this page to learn more.

The problem with attributes is that the library cannot know what to do when there is an *Error* or *Loading* state, because it highly depends on the needs of the programmer. (For example: should we make the background red on error? Which red? Should we write something? In what language?)

For this reason we decided **not** to include a useless default behavior, but to create some functions that will help you to create your custom design.

Creating your own extension functions

The suggested way to proceed in order to accomplish this task is to create one or more extension functions for each view type you're interested in. Inside this functions we'll need to call the `toViewState()` function on the passed attribute. This method will accept one or more lambdas to create our desired view state.

For instance, suppose we want to have the following behavior when we set an attribute in a `TextView`:

- When the attribute status is *Loaded*, we want a blue text representing the attribute content and no background;
- When the attribute status is *Loading*, we want a light grey background and no text;
- When the attribute status is *Error*, we want a red text saying "Error" and no background.

In any case, the text gravity must be center.

This is how it's declared:

Kotlin

```
fun TextView.setText(text: Attribute<CharSequence?>) {
    setTextViewState(text.toViewState(
        fixedStates = textGravity(Gravity.CENTER),
        onLoading   = { backgroundColor(Color.LIGHT_GREY) },
        onError     = { textColor(Color.RED) + text("Error") },
        onLoaded    = { text(it) }
))}
```

Let's analyze it:

- We create an extension function for the view type we're interested in (in this case `TextView`), accepting an attribute as input;
- We call the function to set our state, in this case `setTextViewState()`
- We pass to this function the result of the `toViewState()` call on the attribute
- To this function we pass:
    - `fixedStates`: one or more states that will always be active, regardless of the attribute status
    - `onLoading`: function that, given the loading status, returns the view states to show when the attribute is *Loading*
    - `onError`: function that, given the error status, returns the view states to show when the attribute is *Error*
    - `onLoaded`: function that, given the attribute loaded value, returns the view states to show when the attribute is *Loaded*

To use it we can simply call this method, like so:

Kotlin

```kotlin
val textView = TextView(context)
val attr = attributeOf("hello")

textView.setText(attr)
```

Java

```java
//In java we need to call this method statically. Here we assume we told the compiler to
put it in a file `TextViewUtils`

TextView textView = TextView(getContext());
Attribute<String> attr = Attribute.of("hello");

TextViewUtils.setText(textView, attr);
```

Note that there are multiple versions of the `toViewState()` function, so you can choose the one that better suits your needs.

Using this technique you can implement all the functions you need with your custom UI behavior and then use them when you need to set an attribute to a view.