FSM ON ANDROID: HOW TO REACH APP MODULARITY?

INTRODUCTION

In this document, I want to explain how I ended up with a FSM as a solution to make Android application development fully-modular. That's why this document is organized in steps, describing the journey I made, from discovering the EasyFlow FSM to using it for modularity purposes.

When talking about modularity, all developers agree on a standard definition:

- divide a program into separated sub-programs (called modules) according to the features to implement,
- group similar functions in the same unit of code.

Advantages:

- ease incremental builds and deliveries,
- module is unit-testable,
- modules can be added, modified or removed without any impact on one another,
- modules can be reused.

But on some implementations, I didn't agree with some choices. For example, to navigate to the next screen, a module knows the name of this next screen thanks to a constant (from the hosting application or from the next module itself): this introduces a high-coupling that stinks. To my mind, to be fully independant and agnostic, a module must not depend on a constant from anywhere else. At best, it exposes constants to be used by the hosting application. So, how to transit from one screen to the next one? The answer became clear: thanks to events. Indeed, events bring the needed abstraction: the module fires an event, the hosting application receives this event and acts accordingly. This is how I discovered the "event-driven development" paradigm.

According to this paradigm, the flow of the program is determined by events (user actions, network requests, sensors, timer, other threads, etc.). A specific piece of code (often called the "event listener" or "main loop" according to the language) detects that an event has occurred and performs the corresponding event "handler" (a callback method for example). The resulting computation changes the program state.

Finally, keeping in mind my final goal, I remembered the terms used by the Android SDK: when an `Activity` is destroyed, it mentions the fact of saving and restoring the "instance state". And all became clear for me: a "screen" can be considered as a "state" of the application, and navigation is somehow a finite state machine.

A finite state machine (FMS) is defined by:

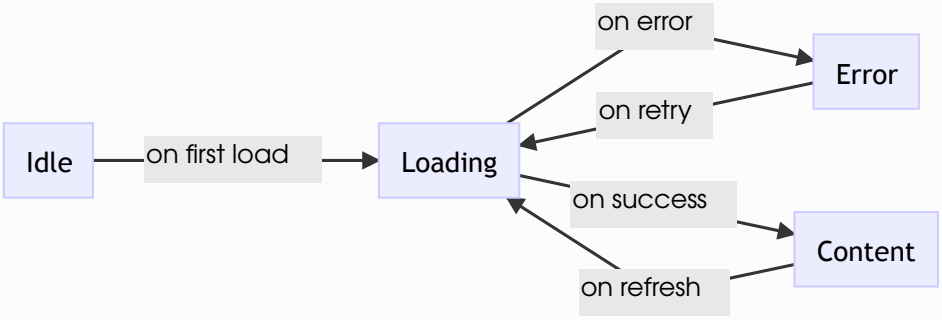
- sequential logic circuits,
- finite number of states,
- one state at a time (the *current state*),
- change from one state to another by triggering an event (a *transition*).

For Java, the implementation I've chosen is EasyFlow, because:

- it's very simple to set up,
- it's possible to define a global context (useful to pass arguments for example),
- states are easily defined by declaring an *enum* implementing `StateEnum`,
- events are defined easily by declaring an *enum* implementing `EventEnum`,
- it provides a fluent API, which allows developers to declare the state machine in a clear and intelligible way (especially when combined with lambda expressions),
- it's possible to declare callbacks to perform specific jobs when entering or leaving a state.

FSM TO MANAGE VIEW STATES

Let's consider a typical example of a simple screen loading data from the network to display the result. Below we can see a state diagram corresponding to this screen.



The associated XML layout is not very relevant at this point. But just consider the following declarations in the `Activity` are valid:

```
@BindView(R.id.ActivityMain_TextView_Content)
TextView mTextViewContent;
@BindView(R.id.ActivityMain_ProgressBar)
ProgressBar mProgressBar;
@BindView(R.id.ActivityMain_ViewGroup_Content)
LinearLayout mViewGroupContent;
@BindView(R.id.ActivityMain_ViewGroup_Error)
LinearLayout mViewGroupError;
```

Now, using the great library called Switcher, we can define some fields as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mUnbinder = ButterKnife.bind(this);

    mSwitcher = new Switcher.Builder(this)
            .addContentView(mViewGroupContent)
            .addErrorView(mViewGroupError)
            .addProgressView(mProgressBar)
            .build();
```

Now, to setup the FSM corresponding to the state diagram, we need to define the available `States`, `Events` and `Context` as follows:

```
enum States implements StateEnum {
    LOADING, ERROR, CONTENT
}

enum Events implements EventEnum {
    onError, onSuccess, onRetry, onRefresh
}

class FlowContext extends StatefulContext {
}
```

And then we can set up our FSM programmatically as exposed below:

```
EasyFlow<FlowContext> flow =
        from(States.LOADING).transit(
                 on(Events.onError).to(States.ERROR).transit(
                        on(Events.onRetry).to(States.LOADING)
                ),
                on(Events.onSuccess).to(States.CONTENT).transit(
                        on(Events.onRefresh).to(States.LOADING)
                )
        );
```

And now we have to define what to perform when entering a given state (with the help of the retrolambda library):

```
flow.whenEnter(States.LOADING, (final StatefulContext context) -> startRequest());
flow.whenEnter(States.CONTENT, (final StatefulContext context) -> mSwitcher.showContentView());
flow.whenEnter(States.ERROR, (final StatefulContext context) -> mSwitcher.showErrorView());
```

We can start the FSM as follows:

```
flow.executor(new UiThreadExecutor());
mFlowContext = new FlowContext();
flow.start(mFlowContext);
```

The `Executor` instance allows developpers to configure the type of thread used to perfom FSM operations (such as `whenEnter`). It looks like:

```
public class UiThreadExecutor implements Executor {
    private Handler mHandler = new
Handler(Looper.getMainLooper());

    @Override
    public void execute(Runnable command) {
        mHandler.post(command);
    }
}
```

To perform the network request in this example, I use the Volley library as follows:

```
private void startRequest() {
    mSwitcher.showProgressViewImmediately();
    StringRequest request = new StringRequest(
            "https://api.github.com/users/RoRoche",
            (final String response) -> {
                mTextViewContent.setText(response);
                try {
                    mFlowContext.trigger(Events.onSuccess);
                } catch (LogicViolationError logicViolationError)
{
                    Log.e(TAG, "startRequest",
logicViolationError);
                }
            },
            (final VolleyError error) -> {
                try {
                    mFlowContext.trigger(Events.onError);
                } catch (LogicViolationError logicViolationError)
{
                    Log.e(TAG, "startRequest",
logicViolationError);
                }
            });
    mQueue.add(request);
}
```

All that remains for me to do, therefore, is to implement methods listening to user interactions:

```java
@OnClick(R.id.ActivityMain_Button_Refresh)
public void onClickRefresh() {
    try {
        mFlowContext.trigger(Events.onRefresh);
    } catch (LogicViolationError logicViolationError) {
        Log.e(TAG, "onClickRefresh", logicViolationError);
    }
}

@OnClick(R.id.ActivityMain_Button_Retry)
public void onClickRetry() {
    try {
        mFlowContext.trigger(Events.onRetry);
    } catch (LogicViolationError logicViolationError) {
        Log.e(TAG, "onClickRetry", logicViolationError);
    }
}
```

Putting it all together and it works like a charm! Transitions are logical, fluid and well-defined. The logical code is devoted to the FSM. The rest of the code consists in triggering the suitable event.

FSM TO MANAGE APP NAVIGATION

Following the same logic, I decided to set up 2 screens and manage navigation thanks to a FSM.

Why Conductor?

First of all, it's important to focus on the Conductor library I chose to create a "View-based application".

Here are many advantages of using this library:

- Navigation concerns
- Simple to build an instance of `Controller` and provide its dependencies (no args `Bundle` like when using `Fragment`)
- Bring forward the `ViewController` concept
- Pretty transitions
- Easy to integrate in a MVP or VIPER architecture

"Screens" as "States"

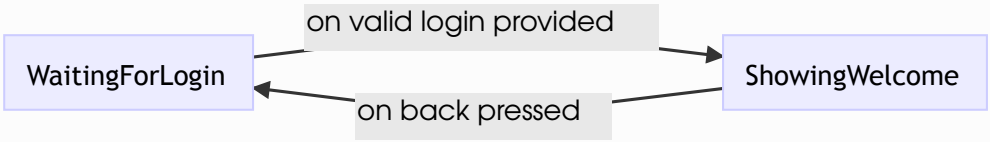
The idea came to me that, in fact, a screen displayed to the user can be considered as a state of the application. I mean: for example, when presenting a screen to log in, it's a state "waiting for login", isn't it? And the event to change the app state is "valid login provided", isn't it?

The idea is to use Conductor to define each screen, and use the `Activity` to monitor the interactions and navigation.

To keep each screen independant, I decided to use the `StatefulContext` implementation to hold arguments as a `Bundle` instance (pretty familiar in the Android world, isn't?). This way, the implementation looks like:

```
1  public class FlowContext extends StatefulContext {
2      private final Bundle mArgs = new Bundle();
3
4      public Bundle args() {
5          return mArgs;
6      }
7  }
```

Now it's time to design the state diagram of the features to implement:



Very simple in fact. So it's time to define our first screen ("waiting for login"):

```
1  public class FirstController extends Controller {
2      //region Constants
3      private static final String ARG_KEY_LOGIN = "LOGIN";
4      //endregion
5
6      //region Fields
7      private FlowContext mFlowContext;
8      private TextInputEditText mEditTextLogin;
9      //endregion
10
11     //region Constructors
12     public FirstController() {
13         this(new FlowContext());
14     }
15
```

```
16      public FirstController(FlowContext flowContext) {
17          super();
18          mFlowContext = flowContext;
19      }
20      //endregion
21
22      //region Controller
23      @NonNull
24      @Override
25      protected View onCreateView(LayoutInflater inflater,
    ViewGroup container) {
26          View view = inflater.inflate(R.layout.first_controller,
    container, false);
27          mEditTextLogin = (TextInputEditText)
    view.findViewById(R.id.FirstController_EditText_Login);
28
    view.findViewById(R.id.FirstController_Button_Start).setOnClickLi
    stener(new View.OnClickListener() {
29              @Override
30              public void onClick(View pView) {
31                  onClickButtonStart();
32              }
33          });
34          return view;
35      }
36      //endregion
37
38      //region User interaction
39      private void onClickButtonStart() {
40          String login = mEditTextLogin.getText().toString();
41          if (TextUtils.isEmpty(login)) {
42
    mEditTextLogin.setError(getApplicationContext().getString(R.strin
    g.login_error));
43          } else {
44              try {
45                  mFlowContext.args().putString(ARG_KEY_LOGIN,
    login);
46                  mFlowContext.trigger(Events.loginProvided);
47              } catch (final LogicViolationError
    poLogicViolationError) {
48              }
49          }
50      }
51      //endregion
52
53      //region FSM
54      public enum States implements StateEnum {
55          WAITING_LOGIN
56      }
57
58      public enum Events implements EventEnum {
```

```
59          loginProvided
60      }
61
62      public static String getLogin(FlowContext flowContext) {
63          return flowContext.args().getString(ARG_KEY_LOGIN);
64      }
65      //endregion
66  }
```

This screen defines its available states and events. It's necessary to pass the `FlowContext` to its constructor. After some logical controls, this screen puts the provided value in the `FlowContext` arguments and triggers the `loginProvided` event. It's its single responsability: display an interface to fill a login value, control it and notify of the sequence success.

In the same way, let's define the second screen (for simplicity, it just displays the provided login):

```
public class SecondController extends Controller {
    //region Args
    private final String mLogin;
    //endregion

    //region Constructors
    public SecondController() {
        this("");
    }

    public SecondController(String login) {
        super();
        mLogin = login;
    }
    //endregion

    //region Controller
    @NonNull
    @Override
    protected View onCreateView(LayoutInflater inflater,
ViewGroup container) {
        View view = inflater.inflate(R.layout.second_controller,
container, false);
        TextView textViewWelcome = (TextView)
view.findViewById(R.id.SecondController_TextView_Welcome);

textViewWelcome.setText(getApplicationContext().getString(R.strin
g.welcome, mLogin));
        return view;
    }
    //endregion

    //region FSM
    public enum States implements StateEnum {
        SHOWING_WELCOME
    }
    //endregion
}
```

Much simpler than the previous one! But now you should ask yourself "wait! where are these screens built? where is the navigation logic? where is the back event?". Here is my point: in the `Activity`. No screen should know how to transit from one state to another. They just have to define the state they represent and the events that can be triggered. Moreover, for me, this second screen must not know the "back pressed" event. It is not its responsability to navigate back. So, now, it's time to have a look at the `Activity`:

Navigation between screens

```
public class MainActivity extends AppCompatActivity {
```

```
2
3       //region Fields
4       private Router mRouter;
5       private EasyFlow<FlowContext> mFlow;
6       private FlowContext mFlowContext;
7       //endregion
8
9       //region Lifecycle
10      @Override
11      protected void onCreate(Bundle savedInstanceState) {
12          super.onCreate(savedInstanceState);
13          setContentView(R.layout.activity_main);
14
15          ViewGroup container = (ViewGroup)
    findViewById(R.id.ViewGroup_Container);
16          mRouter = Conductor.attachRouter(this, container,
    savedInstanceState);
17
18          mFlow =
19
    from(FirstController.States.WAITING_LOGIN).transit(
20
    on(FirstController.Events.loginProvided).to(SecondController.Stat
    es.SHOWING_WELCOME).transit(
21
    on(Events.backPressed).to(FirstController.States.WAITING_LOGIN)
22                          )
23                  );
24
25          mFlow.executor(new UiThreadExecutor());
26
27          mFlow.whenEnter(FirstController.States.WAITING_LOGIN,
    (FlowContext context) -> {
28              if (!mRouter.hasRootController()) {
29                  mRouter.setRoot(RouterTransaction.with(new
    FirstController(context)));
30              }
31          });
32
33          mFlow.whenEnter(SecondController.States.SHOWING_WELCOME,
    (FlowContext context) -> {
34              SecondController loController = new
    SecondController(FirstController.getLogin(context));
35
36
    AndroidModularApplication.getInstance().getComponentSecondControl
    ler().inject(loController);
37
38              RouterTransaction transaction =
    RouterTransaction.with(loController)
39                      .pushChangeHandler(new FadeChangeHandler())
40                      .popChangeHandler(new FadeChangeHandler());
```

```
41
42              mRouter.pushController(transaction);
43          });
44
45          mFlow.whenLeave(SecondController.States.SHOWING_WELCOME,
    (FlowContext context) -> {
46              context.args().clear();
47          });
48
49          mFlowContext = new FlowContext();
50          mFlow.start(mFlowContext);
51      }
52
53      @Override
54      public void onBackPressed() {
55          try {
56              mFlowContext.trigger(Events.backPressed);
57          } catch (LogicViolationError logicViolationError) {
58          }
59
60          if (!mRouter.handleBack()) {
61              super.onBackPressed();
62          }
63      }
64      //endregion
65
66      //region FSM
67      public enum Events implements EventEnum {
68          backPressed
69      }
70      //endregion
71  }
```

All the valuable jobs are done in the "whenEnter" sequences. Indeed, when entering the first state, we create and push the first screen to the Conductor-specific `Router` instance. When entering the second state, we build the second screen, pass the arguments to it and push it to the `Router` using a fading `RouterTransaction`.

We override the `onBackPressed` and trigger the corresponding event to update the FSM current state. Then if the `Router` instance allows "back" at this level, this event is processed.

Finally, it is not that much complicated to do. But we gain a clean design, with a perfect application of the "Single responsability principle". This Android application is now a valuable combination of "states" and "events".

FSM TO MAKE APP MODULAR

Screns as independent modules

You know what? It's the simplest step of this article: you just have to create two Android modules ("first" and "second") thanks to the Android Studio wizard, drag and drop each piece of code in the corresponding module and that's it! Well, not really: you have to create a "common" module to place the `FlowContext` and `UiThreadExecutor`. Both "first" and "second" modules reference this "common" module. Now, these modules can be successfully referenced by the application.

Too simple maybe? So let's go a step further with the dependency injection topic.

DI ready: example with Dagger 2

Now you have an application divided into multiple screens (states) and responding to various events, you may ask yourself how to define and provide the dependencies needed by your screens.
To introduce a solution, I will use the Dagger 2 library.

Suppose we want to display the current date on the second screen of our application. I'll start by defining an interface in the "second" module:

```
1  public interface IDateFormatter {
2      String format(Date date);
3  }
```

In the "app" module, I'm going to implement this one:

```
1  public final class DateFormatter implements IDateFormatter {
2
3      //region Constants
4      private static final String DATE_FORMAT = "dd/MM/yyyy";
5      private static final SimpleDateFormat sSimpleDateFormat = new
   SimpleDateFormat(DATE_FORMAT);
6      //endregion
7
8      //region IDateFormatter
9      @Override
10     public String format(Date date) {
11         return sSimpleDateFormat.format(date);
12     }
13     //endregion
14 }
```

Now, let's create the suitable Dagger 2 `Module` as follows:

```
@Module
public class ModuleSecondController {

    @Provides
    @Singleton
    public IDateFormatter providesDateFormatter() {
        return new DateFormatter();
    }

}
```

And the `Component` to configure the second screen:

```
@Singleton
@Component(modules = {ModuleSecondController.class})
public interface ComponentSecondController {

    void inject(SecondController secondController);

}
```

The next step is to subclass the Android `Application` class and build the `Module` and `Component` as follows:

```
public class AndroidModularApplication extends Application {

    //region Static field
    private static AndroidModularApplication sInstance;
    //endregion

    //region Field
    private ComponentSecondController mComponentSecondController;
    //endregion

    //region Overridden method
    @Override
    public void onCreate() {
        super.onCreate();
        sInstance = this;
        mComponentSecondController =
DaggerComponentSecondController.builder()
                .moduleSecondController(new
ModuleSecondController())
                .build();
    }
    //endregion

    //region Static getter
    public static AndroidModularApplication getInstance() {
        return sInstance;
    }
    //endregion

    //region Getter
    public ComponentSecondController
getComponentSecondController() {
        return mComponentSecondController;
    }
    //endregion
}
```

And now, the final step takes place in the `MainActivity`. It's time to inject dependencies in the second screen. So we're back to the "whenEnter" sequence when building the second screen:

```
SecondController controller = new
SecondController(FirstController.getLogin(context));

AndroidModularApplication.getInstance().getComponentSecondControll
er().inject(controller);

RouterTransaction transaction = RouterTransaction.with(controller)
                    .pushChangeHandler(new FadeChangeHandler())
                    .popChangeHandler(new FadeChangeHandler());

mRouter.pushController(transaction);
```

Eventually, we can add a preconditions to the second screen (defensive programming!) as follows:

```
protected View onCreateView(LayoutInflater inflater, ViewGroup
container) {
    Preconditions.checkNotNull(mDateFormatter, "Field
mDateFormatter is null, did you miss to inject it with your
dependency injection mechanism?");
```

This way, you can inject any element you want (data store, user preferences, etc.). But the targetted screen remains agnostic, independent and highly configurable.

CONCLUSION

- An easy way to deal with the states of a screen
- A new approach to set up the navigation flow of an application
- A solution to build trully reusable modules
- High level of configuration with DI
- Perspectives: what about a library of existing modules, used by a "drag & drop" Web interface to design an entire application in a user-friendly way?

BIBLIOGRAPHY

- Applications as State Machines