# Filip Babić

## Skyrocketing Android with Kotlin — Fully embracing advanced Kotlin features in Android development

Filip Babić

*1st Edition*

1

# Intro

The idea of this book is to provide a well documented and thoroughly explained process of creating an advanced Android application using intermediate Kotlin features and language constructs.

This is not a toy project or some code whipped up together just to work and show off. It's a full process, from the debate of why use Kotlin, and not Java, to finer things in Kotlin like using extensions and generics to build your own DSL-like syntax for some Android specific things.

Every single feature, library, plugin and construct used in this book is what I personally favor currently in Android development, and the current Kotlin ecosystem. I won't enforce you to use these tools if you prefer something else, but be advised that I won't provide examples for every single tool there is, I'll stick to some specific tools.

It is my only wish that you fall in love with Kotlin as much as I did, and that you broaden and strengthen your Android development knowledge while reading this book, so you may once refer it to others as a good learning resource.

If on the other hand you have no Android experience whatsoever, but you do have some experience in programming, don't be afraid of reading this book. I'll make sure to cover as much ground as possible, so that even people who don't develop Android apps can learn how to do so.

## Getting Started

You must have heard of all of Kotlin's awesomeness in the past year or two since it's been out, and you might familiar with it's syntax, or you may know a few things but haven't quite started learning it.

Everything we'll cover will be elaborated extensively, but I'd advise you, if you aren't already familiar with Kotlin, to go over Kotlin's documentation[1], and the Kotlin Online Koans[2] examples, which will give you a rough image of what this beautiful language can do.

---

[1] https://kotlinlang.org/docs/reference/ - Kotlin official documentation

[2] https://try.kotlinlang.org/ - Kotlin online console and compiler with practical examples

Since the topics we'll be covering are intermediate to say the very least, if you do get lost, be sure to check a few things before fully delving into this book. Check out Kotlin in Action[3] and Kotlin for Android Developers[4] by Antonio Leiva[5] books

[3] https://www.manning.com/books/kotlin-in-action - Kotlin in Action book

[4] https://antonioleiva.com/kotlin-android-developers-book/ - Kotlin for Android Developers book

[5] https://antonioleiva.com/ - Antonio Leiva's website

## This book's lifecycle

Wow, we haven't even gotten to the App, and I'm already mentioning Lifecycle? Don't worry, this isn't the pain-in-the-ass Android Lifecycle, but just an overview of what I've divided this book into.

Visualising whatever you're trying to learn and combining previously gained knowledge makes you understand and remember it far better. This is great when you're building an application, because you can visualize whatever you build and reuse a lot of things learned along the way. This is why this book will be followed by a full working project rather than plain simple examples.

You'll go through the tough task of creating a Social network, something you could really be proud of and something you can easily add to your portfolio.

You'll see every single detail of what it takes to build such an app, from the concept and organizational process to the code implementation. The process, or rather the methodology you'll be using is Agile development[6].

## Being agile

Agile is an iterative form of building products, in which you work in time periods of *sprints* which last about two to four weeks. Each sprint you take a certain set of epics and stories which you want to complete. An epic represents one solid feature in the application, like *user authentication,* where each story is a jigsaw piece within that epic, like *login*, *registration* and *social login*.

The idea is not to plan everything ahead, but rather be agile! React to feedbacks, developer insights, client wishes and so on. If you develop using this approach, you don't have to think about everything at once and even if you stop somewhere in the middle of development, you'll have a working version of stories up to the point of stopping.

Software companies use this approach because it's easier to bugfix only a sprint at a time, you can test each iteration of the application with a user group and get invaluable feedback and it's cheaper to change or even cancel a project if you're not satisfied with the results up to that point.

Agile however is not only a way to optimize the development process. It also focuses a lot on building applications that your users will love. The goal is not to build what you want to have in the app, but to create a general concept of what you want to achieve, research the market and the personas that would use your application and sculpt the features according to their needs or suggestions.

After all, we are researching *user* stories, not *developer* stories. :]

---

[6] https://www.qasymphony.com/blog/agile-methodology-guide-agile-testing/ - Agile manifesto

# Audience level

On one hand I've taken up writing this book as a challenge to myself. It seems like a great learning experience knowing that I have to think about everything from different perspectives. On the other hand I really do hope many a person choose this book as their next source of information of all things Kotlin, as it would really be a proof that I've done something beneficial to the community.

However, given my explanation of what you'll be building you can realise I haven't written a tutorial, or an article on a cute little Kotlin language feature I've just found out about. I've written a book filled with hardcore stuff like generics, mapping operators, coroutines and monads, and I expect of you hold on for dear life.

You *will* be annoyed, frustrated, going crazy, bloodthirsty even when I drop crazy ass functions for networking who have a bunch of these "<T : Mappable<R>,  R : Any>" in their signature. If you don't get annoyed, and figure them out momentarily, I'll be even happier knowing you, yourself, are badass.

The reason I've chosen this type of learning is because I believe that the things that annoy, frustrate or don't make sense to you can channel amazing creativity, even more so if you're thrown in head-first into something that is yet unknown to you.

A good example would be RecyclerView adapters. I've personally been frustrated by the fact that it's pure boilerplate that I had written dozens of times, so frustrated that I'd made my own library[7] that abstracts it away and reduces about 90% of the boilerplate behind.

Another example is legacy projects. Although I agree that they are pure chaos and torture, I still think they are a great opportunity to build your knowledge base. My first project was a 4 year old Android application with roughly 200 thousand monthly active users. It was hell. It still is hell. But it taught me the most important thing there is - refactoring.

Refactoring may be the most tedious thing to do, but with enough time spent doing so it can lead you to use design patterns, to structure your code better, to learn how to understand obfuscated code and make something out of nothing. Trust me, one day you'll work on a legacy project (if you already haven't) and you'll know this to be true.

It might seem weird at first, but it really was the quickest and best way to learn for me. A good developer will find his own way around, without much help, and he'll become even better for it. A great developer will do the same, but he won't stop at that, he'll continue to teach others how to do it (yes, I'm implying I'm great).

---

[7] https://github.com/filbabic/FlexibleAdapter - My RecyclerView adapter library

2

# Brainstorming our App

*Let's socialize*

Most of you have probably used a Social network, but for those of you who haven't it's a platform on which people can communicate in one way or another. Usually there is a place on which you can post your thoughts, memories, life events and questions - either by creating a thread or by posting an image or a textual post.

The concept is pretty straightforward, try to entice your user-base to generate new content daily, in order to gain new followers, likes/hearts, shares and comments on their posts. This way people could always come to the application and see something new in the feed or on their posts.

## Why a social network

Social networks are a really popular thing nowadays. Most developers think each has it's own enormous codebase which is composed of spaghetti code, and that only tough veterans can do anything productive in such an environment.

I'm not really buying it, because even if that were true, each network had to start somewhere. I believe anyone could create their own social network, which you'll see by following this book!

I've personally never built a social network (up until now), so this will be something new for me too. But I wanted it to be this kind of app because it has many components, it's extensible and you can always go back to it to experiment with something new, like a new architecture pattern, new networking paradigm and so on.

## What will it do?

As previously mentioned, you'll be able to create posts shown on a global feed and like other people's posts. You'll have some form of communication with other users, a profile which other users can check out if they want to get in contact with you, and whatever else I think would be a good feature to implement on the way.

Since there are nearly infinite possibilities, I wouldn't plan absolutely everything ahead, let's try to go step by step and see what comes out!

# *The first door is the hardest*

You'll be learning how to use the aforementioned Agile methodology to achieve great UX (User experience). So your first step should be to take a look at what a social network is composed of, and try to derive user stories from that. Each user story will begin with: "As a user I want to... so that I can..."

An example user story would be: "As a user I want to to log in so that I can be recognized as a unique individual." Try to write down a few yourself, before heading to the next section. I'll try to walk you through this process as I would do it, but try to write down a few .

## We have to build for someone

We've already said that each Social network needs users - therefore we need an Authentication component (epic). Now you'd say it's easy, we have Register and Login stories, we've built them for so many projects (if you have some experience).

But what I want you to understand is that most people will want an easy path within your application, so that in only a few clicks they could participate in the full experience. With this in mind, a regular register/login screen would take 3-4 steps for the user to participate, could we reduce this somehow? //TODO add some illustrations here

Most applications have integrated other Social networks' login for this reason only. Which is why we will do too.

## Connections all around

It's great UX (User experience) to have a quick log in option by using accounts from other Social networks. Be it Google Sign In or Facebook Login, or more often both, it's an amazing way to draw users to use your application.

Think about it, if you had the option to fill in a ton of forms, confirm your e-mail, and only then enter the app, or the option to click a button and choose an account, which one would you take?

For navigation, you'll have social login on a welcome screen, so that the first entry point is available as soon as possible.

## Welcoming our guests

Most applications welcome their users by showing a short splash screen depicting their product's logo and core theme. It's a great way to leave a first impression by using live colors and a logo by which your application will be recognized.

After the splash we have to show something to our Users. A compact welcome screen, with all the entry points into the core application is required. Regular authentication and social authentication will take place here because we want to let the user choose how he'd want to join the community, rather than providing only one type of login.

Another thing you'll have to consider now is the fluidity of the App. If your application has unclear navigational flows or a repetitive process, your users won't be too happy using it. For example if a user has to authenticate every time he opens the app, they will quickly be annoyed and leave negative reviews on the Play Store.

With that said, once the User is authenticated, there is no reason to show a Welcome screen, you should simply show them the feed, so they don't have to log in each time they open your app. A small thing, albeit important to the users.

## Restraining order

Notice how we've gone through one component of the app without mentioning the case when a user doesn't want to sign up. Simply put, it's a huge effort to block entire features of your application when you're not authenticated, which is why I've decided to only let the user use the app if he's signed up.

# *Feeding our users*

From an economic perspective, each application has to have something that'll make users want to use it. People just won't use your app because it's something not so much different from the competition. Each app has to have a "killer feature" to stand out from the vast sea of similar applications. Something that will draw users to use your app. Like Instagram[8] did with its "stories".

But because I'm not creating an app that each of you will be able to sell and make a living off, this Social network won't exactly have a killer feature, but the concept stays the same. Just make sure that if you're building a product, you have that one thing that differentiates you from everyone else!

Since your users will need a reason to use the app, you will build a feed where all the new things people post will be visible and you could explore new things. You can like posts as previously mentioned, and leave comments on them.

## Improving the posts

//TODO add illustrations

As before, you want to improve the experience for the user. If you've been a part of some Social network, you know how tedious it can be to find a post so you can show it to your friends. Having a favorites page can simplify this process a lot.

Furthermore, having a page just for your posts, so you can see their progress (number of likes) and so you could delete them if you want is also something that's crucial. This covers the posts section of feed, so you can move on to writing user stories.

Try to come up with the user stories that best fit these features yourself, before checking what I came up below.

The stories I believe best describe what the users want, are next:

- "As a user I want to be able to preview posts from other people, so I can find interesting topics."
- "As a user I want to be able to like interesting posts so I can browse them later."
- "As a user I want to be able to leave comments on posts, so I can communicate with other people."
- "As a user I want to be able to preview my posts."
- "As a user I want to be able to delete my unwanted posts."

---

[8] https://www.instagram.com/ - Instagram application website

Notice how I split up the my posts page into two stories, one to view your posts and one to delete them. If you come up with a story that seems to have two small tasks, it's best to split them up and work on them separately.

What else do we need to wrap up feed? We will work on lazy loading (loading items page by page rather than all at once), refreshing, updating lists when liking posts and so on, but all of that will be covered deeply in the implementation part, for now we're just conceptualizing.

Now, we keep talking about the users, but we haven't said how they'll interact. I know it seems that it's not a programming book so far, but from my experience, coding is just about 5-15% of a programmer's job, the rest is talking, planning, discussing and brainstorming! Let's move to the profile feature.

# *Profiling*

Each user likes to feel special, which is why they need their own section in an application. By having a user profile we give them an overview of their current status in the community (number of collected likes, number of posts). Furthermore we give them a way to check out other Users as well, by looking at their public profiles.

We've come to a new possibility now, should we add a "Friends" section as well? Why not! Since we will allow users to communicate and look at their profiles, you'll also add a Friends section for them to be able to stalk people they like.

On the profile we will give the user a way to change their names and profile images, passwords (if they have any), and to log out or disable their account. But more on that later when we implement it.

One last thing left to think about is feed details and the comments section, which we will do in the next section. It's really important to structure your features into logically separate units, so you can finish them one by one, without breaking the rest of the app.

# *Details matter*

Having just posts, without the ability to see them with a detailed description feels incomplete. Even more so because we're adding comments. You don't want to scroll endlessly through a feed of comments, you want to open up a post and then see what it's about, and comment if you like what you see.

When creating a post, you'll add the ability to write a detailed description, if there would be any, and to add a picture for the post. This way the app won't be a huge wall of text, the users will be able to see images describing posts before reading them.

## Any comments?

Comments will be a vital part of each post's details page, because the users can express their opinion for each post. For the sake of simplicity, you'll leave the comments to be textual only, having tons of images in the comments section will make you scroll endlessly to find what you're looking for.

We can make this quite easy, and Facebook-like. Each person can comment, but you won't have nested comments. Often it is best to stick to simplicity to provide the best experience for your users. If you overcomplicate the code, it only affects your team. But if you overcomplicate how the application works, it affects every single user.

## Conclusion

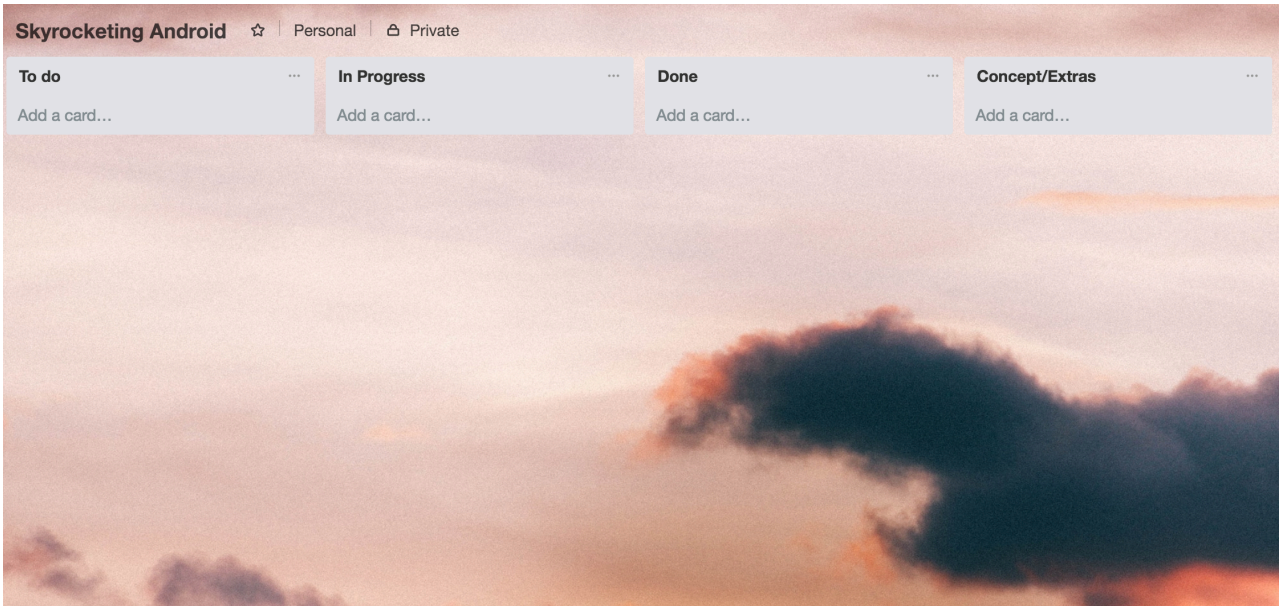
It's very easy to brainstorm all the features and what you want to see, but it's important not to think too much ahead. Overthinking, or rather over-engineering, is a key step to failure and process breakdown. Once you start going too far ahead, you'll overcomplicate even the simplest things, end up with tunnel vision and lose productivity, and your users won't be happy with the app complexity.

# Writing up Stories and Epics

You've already learned what user stories are. It's time you write them down somewhere, so you can organize yourself while coding. Rather than using the ol' paper and pen, try using one of the tools like Jira[9], Trello[10], YouTrack[11]. They are useful when you need to have a visual representation of your tasks.

Most companies use these tools to better organize the whole project process. You can create boards for each project, create epics and stories for agile-based projects, tasks and bugs for each task and much more.

I'll use Trello while writing this book, since it's the simplest solution. You can head over there now to create your own board as well! Once you create a board add the following lists:




*To do* holds all the stories/tasks you are yet to finish. *In progress* are the ones that you started, but haven't yet finished, like those in *Done*. Concept/Extras are just wishes like: "Add more animations to splash."

---

[9] https://www.atlassian.com/software/jira - Jira software tracking tool

[10] https://trello.com/ - Trello board software tracking tool

[11] https://www.jetbrains.com/youtrack/ - YouTrack software tracking tool by Jetbrains

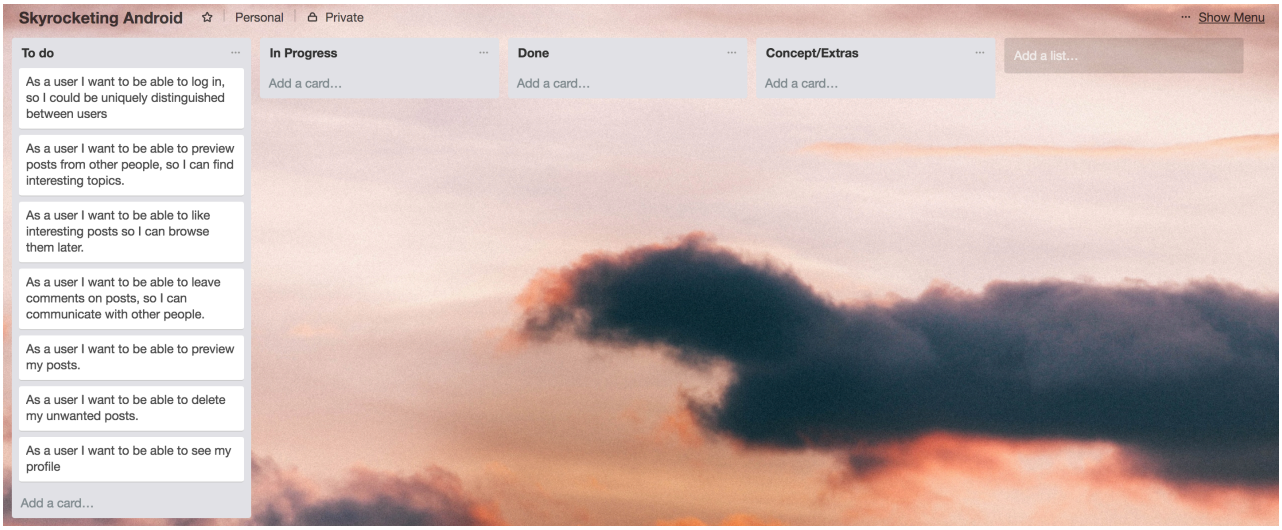
Now, since you have a board for all your tasks, it's vital that you write them down, once and for all. Remember all the user stories you've come up with. Here is an overview of what I thought would describe what this book wants to achieve:

1. "As a user I want to be able to log in, so I could be uniquely distinguished between users"
2. "As a user I want to be able to preview posts from other people, so I can find interesting topics."
3. "As a user I want to be able to like interesting posts so I can browse them later."
4. "As a user I want to be able to leave comments on posts, so I can communicate with other people."
5. "As a user I want to be able to preview my posts."
6. "As a user I want to be able to delete my unwanted posts."
7. "As a user I want to be able to see my profile"

These are the core things you will try to implement in this book. I'll write up some more things that will maybe improve the UI/UX. These may include some nice animations or design tweaks, fun stuff!

## Interesting story my friend

Now that you've written down all the stories, your Trello board should look like this:




Nothing is yet in progress, but you'll get to that. You still have a few more things to cover.

# Diving into the technicalities

In the next chapter you'll dive into coding (I promise). Before you do that, I want to prepare you for what's to come. The set of tools and practices you'll be presented with is chosen to make everything as easy as possible. I'll explain why I prefer using them and what the benefits are. Personally, over the last two and a half years, I've met with a lot of variations for the same common issues in Android, and these work the best for me.

For example, I've used five different ways of binding the XML views to the UI(User interface)[12]. And while each had it's own set of ups and downs, they all served the same purpose. Currently what I'm using is what I believe is the best solution for this problem, even more so because it relies on pure Kotlin language constructs! So let's start with KotlinAndroidExtensions[13].

## *UI has never been so easy*

One of the craziest things you can face when starting to learn Android is the UI(User interface). Sure, the XML[14] is simple enough when you remember a few tags, but the real "WTF moment" comes when you start binding your XML views to the code. You write some XML, the view gets *inflated*, and suddenly you can *find them by id*?

But okay, you understand that it's just how it works. You bind some views to local variables in each Activity[15], and use them whenever you need them. Nope. Sometimes they get deallocated when they are not supposed to. Other times you use the wrong id, or view type and the app just crashes.

That's where KotlinAndroidExtensions come in!

---

[12] https://en.wikipedia.org/wiki/User_interface - UI(User interface) explanation

[13] https://kotlinlang.org/docs/tutorials/android-plugin.html - Official Jetbrains documentation for KotlinAndroidExtensions

[14] https://developer.android.com/guide/topics/ui/declaring-layout.html - Layouts in Android

[15] https://developer.android.com/reference/android/app/Activity.html - Activities in Android

## What is it?

KAE (KotlinAndroidExtensions) are an automized wrapper around the view initialization. Simply put, all your *findViewById* code is removed, all your local variables are removed, but the functionality stays the same.

It works by using something called *synthetic properties* (basically generated fields), and lazy-delegation, allowing you to use all of the views inside your XML in a clean way, without damaging performance. Behind the scenes it does call *findViewById*, but in a cleaner way.

The views are cached, so that the *findViewById* doesn't get called too often. And, as I've said, the best part is you don't have to create your own variables, you just call the wanted synthetic property.

## The usage

The *synthetic properties* are named after the views' ID's, so if you have an *ImageView* with the ID userAvatar like so:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/userAvatar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</android.support.constraint.ConstraintLayout>
```

you can use this ImageView from inside an Android component (Activity/Fragment) like this:

It's as simple as that. This way you can get rid of every single local variable, as you don't have to create them on your own, they get created automatically.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    userAvatar.setImageDrawable(someDrawable)
}
```

There really isn't anything more to the KAE. You now realize why I prefer using them. Because they are simple and concise and make sense (you add a view with an ID, and you just use it) I believe this to be currently the best way to bind UI components to your code.

## Is there a downside?

There is only one issue with KAE for me. It's when you use a the same IDs multiple times (like userAvatar on different screens), there will be a few synthetic properties with the same name. However, the import statement is what gives you the idea to which layout it's referring.

```
import kotlinx.android.synthetic.main.activity_main.*
```

You can clearly see here that the *activity_main* layout is being used. In case you do import from a wrong layout you can check the import statement to be sure.

Apart from this, I cannot think of any other problem with KAE, it's awesome as it is!

# *Make error handling great again*

One of the things I really want to show to you is a neat way of handling errors when validating input fields or processing data and requests. If you're unfamiliar with error handling, it's a procedure which, when you receive an error, makes sure you provide the user with feedback that something went wrong.

An error could be an exception being thrown at runtime or wrong data being sent from point A to point B. It can be caused by  an external source (there is no Internet connection) or it could be caused by your local validation mechanism (email/password checks).

Either way, feedback is necessary for the user to understand what's going on. If you don't show messages when things go bad, your users will have an awful experience. They would probably leave bad reviews on the Play Store saying that things break and they don't know what to do.

Showing a message (even a Toast) gives them the idea of what happened, making them less frustrated. You'll see how I've cooked up a way of showing errors that handles issues like *localization, reusability* and *testability*.

## *Localization*

When strings are localized it means that we can easily swap the message language if we have a multi-language app. For example, the default language for errors might be English, whereas our second language could be German. You want each user to get their native language when you display an input field or a dialog error.

## *Reusability*

Reusability means that we can, simply put, copy and paste the same error handling and error states from one screen to another. Without much change it should work on both screens, meaning we can save time and focus on other things.

## *Testability*

What do we mean when we say testability. Generally when a piece of code is testable, it means it can be covered with Unit tests[16]. This allows us to put our code into various states and situations to see how it performs.

---

[16] https://en.wikipedia.org/wiki/Unit_testing - Unit testing

If, for example, we have an email validation method, we can test it against different inputs and see which emails pass the validation and which don't. This way we can be certain our code works only when it's supposed to — for correct inputs.

Later on, when we have some working business logic code, you'll see how testing is done is Android.

But for now you'll go over the simplest case of error handling - input field validation. You'll also learn how networking errors could be prettified in a way that you don't have to manually check the response data.

Let's start with the input field validation first!

## Hello Enum my old friend

Enum[17] classes are a concept which exists in most programming languages. It's a special constant which is represented by a class. Each constant (also called a type) is a single class instance, and they are statically compiled, meaning they cannot be created at the runtime.

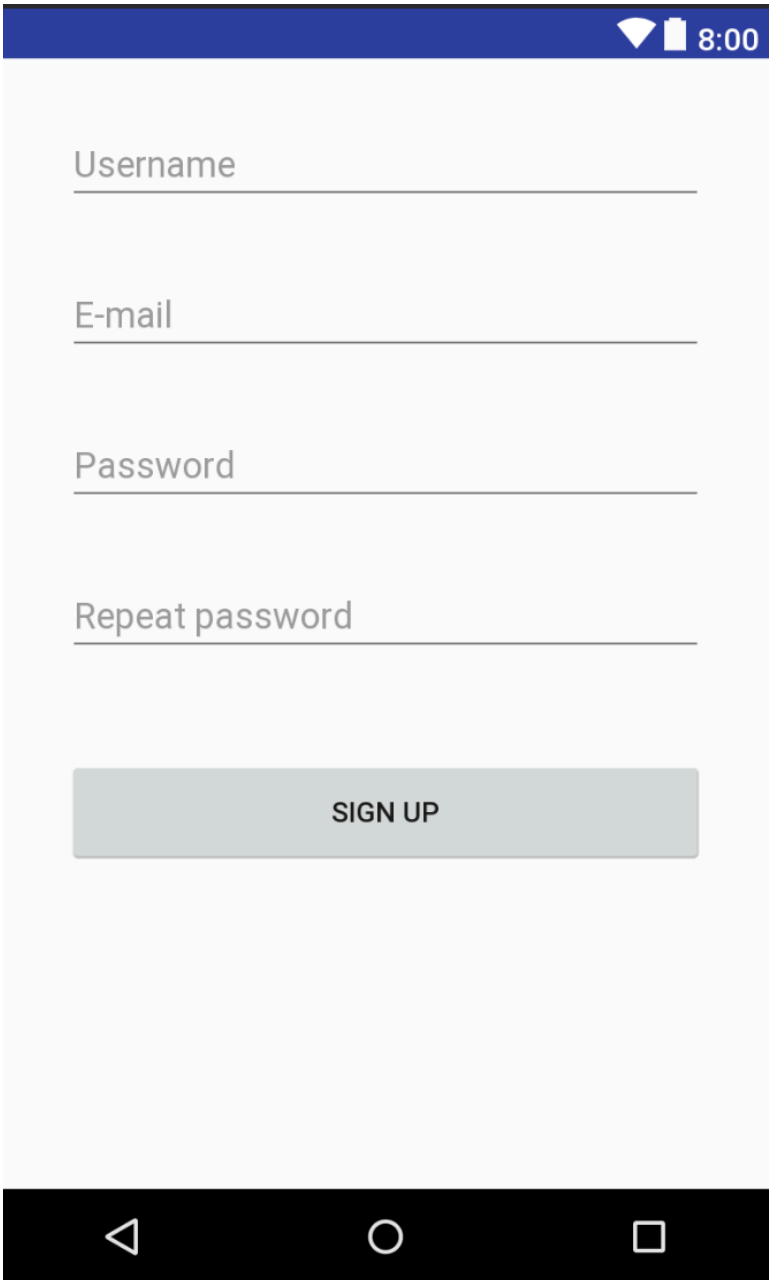
Think of it like this, you can have one enum named Coca-Cola. It represents a refreshing beverage. However, there are different *types* of Coke. There's a regular Coca-cola, Coke Zero, Cherry Coke, Vanilla Coke and so on. And each Coke Zero instance is the same, no matter where you buy it.

Each of the aforementioned types would be an *enum case*. So when you're serving Coke in your application, you know exactly which types you can expect! Now, serving Coke makes sense, but why would you use special constants for errors?

Well if you think about it, errors are constants as well. Every "*invalid email*" error is the same, no matter on which screen it's being shown. By using enums we can encapsulate each error into one specific constant. and just display it.

[17] https://en.wikipedia.org/wiki/Enumerated_type - Enumerated types

Before building our error enum, let's take a look at your usual email validation process using nothing but different methods to display each of the error. Say you're presented with the following form:



Your task is to take the email input and validate it as the users type their email. The email *EditText*'s identifier is "emailInput", which you'll reference and use with KotlinAndroidExtensions.

First attach a *TextWatcher* in order to receive the input every time it changes:

```
private fun initUi() {
  emailInput.onTextChanged { onEmailChanged(it) }
}
private fun onEmailChanged(email: String) {
}
```

Now, every time the email is changed, you can react to it and call your validation mechanism. You have three cases—you either don't have anything to work with, whatever you have to work with is wrong or everything is fine. You'd probably write your validation like so:

```
private fun onEmailChanged(email: String) = when {
  email.isEmpty() -> showEmptyEmailError()
  !Pattern.matches(EMAIL_REGEX, email) -> showEmailInvalidError()
  else -> removeEmailError()
}
```

This is pretty clean, but we have three specific methods, of which each handles it's own error state. Wouldn't it be nice if you had just one method— setEmailErrorState(state)?

## Time for the long waited enum

Start off by defining an enum class *EmailErrorType* and define the three types mentioned above.

```
enum class EmailErrorType {
  NONE, EMPTY, INVALID
}
```

The *NONE* case serves as a way to tell that the error has to be removed. *EMPTY* shows an "Email is empty" message and the *INVALID* shows an "Email format is invalid" message.

Change your previous validation method to this:

```
private fun onEmailChanged(email: String):EmailErrorType = when {
  email.isEmpty() -> EmailErrorType.EMPTY
  !Pattern.matches(EMAIL_REGEX, email) -> EmailErrorType.INVALID
  else -> EmailErrorType.NONE
}
```

Rather than calling different functions to handle each of the input states, you return an error type and show the error message using the returned type.

But how to map these error types into actual error messages? Well enums can have class properties as well! You can add a property *errorResource* of type *Int* to the constructor this way:

```
enum class EmailErrorType(@StringRes val errorResource: Int) {
  NONE(0),
  EMPTY(R.string.email_empty_error),
  INVALID(R.string.email_error)
}
```

Now if you want to display an error, you have the required resource out of an enum shaped box.

But the *NONE* case is a bit troubling. If you try to call *context.getString(resource)*, when the resource is a zero, your application will crash. Which is why you can add the following method:

```
fun getError(context: Context, resource: Int) =
if (resource == 0) null else context.getString(resource)
```

Not only do you handle the possibility of the resource to be a zero, but you also handle the removal of an error. If you call *editText.setError(null)* it removes the previous error!

In order to show (or hide) an error now, simply call:

```
emailInput.error = getError(this, errorType.errorResource)
```

Just what you need, a uniform way to display and hide errors. If you don't want to call the getError function as a global function, you can add this function to the enum class and use it on the enum type instead.

## Is this really a better approach?

You may wonder now: "Is adding a class for each input field, coming up with error types and writing the code to return the proper type easier than the regular solution?"

From my experience, the answer is: **yes**. Even though it might seem like overhead, you're actually saving time. By writing the *EmailErrorType* class once, you can reuse it across all forms in both your current and other applications.

You can even place the validation step into the enum class, so that when you copy the code to another project, you have the mapping to a type as well. And if you don't need a case, just delete it. If you need a new case, add it to the existing enum. It's that simple!

Just remember the three things you covered by adding an enum case for errors: localization, reusability and testability.

1. Each error is displayed with string resources so you have multi-language support.

2. The enums and their validation are completely reusable, not just across the forms in a single application, but also on different projects, as you can copy and paste the code which will work without changing anything

3. The code is simple and testable. In order to verify that the correct error is displayed, you need only check the enum type used after each validation.