# Easy and maintainable test data — The Kotlin way

6 min read · Feb 27, 2020

Anders Sveen Follow




Photo by Matthew Waring on Unsplash

**TLDR;** You can have *tests that are fast, easy to maintain, debug and breaks only when they should.* (not when something entirely different changes).

An important aspect to achieve this is making your test data robust. Keep on reading this article to see how.

I love writing software, and solving problems for users. Some times you stumble upon the right solution that makes your code beautiful and you feel really satisfied with the days work.

> *But those pesky users also want to change stuff. That is when you really* need automated tests to be able to change the system *with minimum risk. But while tests help you know what works and not,* ***they[the tests] can also be a major blocker to change*** *when implementing new things.*

Suddenly you do not just spend time changing and re-factoring your code. You also have to change all the unrelated tests that are breaking. This change would have been done already if it was not for all those pesky tests…

I see lots of people struggle with this, and I understand why so many hate tests. Or do not think it is worth the struggle.

> Writing and maintaining tests can be easier, cheaper and even fun.

It requires a bit of discipline and some common techniques. In this article I will describe how we use a combination of Object Mother and Test Data Builders patterns in Kotlin to tackle changing test data.

Examples are Kotlin specific as it has some neat mechanisms that we can use. But a lot of if should be transferable to other languages as well. Just use your imagination, and I am sure you can find a way to transfer it to your language.

But first, lets see what good tests are. They are:

**Easy to write.** It should be easy to write setup code (test data, system dependencies etc.), and use clear concepts that reduces the distance between your mental model

and code. It should be almost like typing out a sentence. It will never be completely like that, but you should try. And don't build a framework! ;)

**Fast to run.** As short as possible. Less than a minute for the whole project (even in a monolith)! I think there is an important distinction around 20–30 seconds for running your entire suite. We currently need about 40 seconds to run all our tests, and live with it for now. Slower and you will use your tests less, and in a different way.

**Readable.** Keep the gap between your mental model and code as small as possible. Write clear code, have concise set up, use some libraries that makes things clear, and consider doing a DSL.

**Maintainable.** In its simplest form this means that tests should break when something is wrong. And not break when something unrelated changes. That is kind of utopia. I have never done a re-factoring without stupid tests breaking for the wrong reasons. But you try to minimize the effects of unrelated changes.

Test data is an important part of achieving the above, so let me try to explain my favorite techniques. I try to follow the rules below. If they sound bombastic to you, it is because they are.

But I am going to show you ways that make it easy to adhere to them. There is always some exceptions, but these are the broad strokes:

- **Test data should be in code**. Do not make SQL files or JSON files. They are hard to re-factor and tend to grow large. In Kotlin the multiline string literals really help when you need (should be the exception) to test JSON or any other text format in code. Prefer domain objects though.
- **Each test sets up the data it needs**. If your test needs an Order to test modifications on, create that in the set up. You make code to make that easy, and you avoid problems with parallel running of tests and concurrent testing of the same DB.
- **Use common setup of data, that gets modified in the test for specific usage.** So fetch a valid Address, then modify the parts you need to be wrong. If a data

setup pattern occurs in multiple tests, pull it out into a common place for re-use.

- **Make it easy to find the test data.** We use extension methods in Kotlin for this.
- **Make it easy to modify the object for specific usage in the tests.** We do this with a combination of extension methods and data classes in Kotlin.

It helps to have good language support for concepts like map/filter/reduce as well. Preferably in the language or via libraries.



Oh, and *only use the DB when you absolutely have to.* Like when you are testing the actual SQL code. It is slow, hard to debug and not necessary to test business logic. I will try to write about faking and stubs (to avoid using the DB) in a later post.

You want the flow of writing tests to be as close to saying them out loud as you can. You can use frameworks for BDD or more functional style testing for that, but I usually do it in regular Unit/Integration tests. Here is a simple example:

> *"When I have an order that has been registered and is valid, it should be possible to approve it."*

```
1       @Test
2       fun shouldBeAbleToApproveOrderWithNoDeviations() {
3           assertThat(Order.validOrder().canBeApproved(), equalTo(true))
4       }
```

TestDataInitialTest.kt hosted with ❤ by GitHub view raw

A simple test that constructs testdata in a predefined way and validates the business assumption.

The above code makes it easy to write a new test. The set up is actually in the assert, in the *Order.validOrder()* call. You always know how to get a valid Order object. And if our definition if *valid* changes, I only have to change it one place: In the *validOrder()* method.

The method is a special Kotlin construct called extension functions. You can do something similar in Java, but in our case the *validOrder()* method is only available in tests, so you don't pollute your main objects. Here is how we define it (in src/test/kotlin so it's only available in test scope):

```kotlin
fun Order.Companion.validOrder(deviations: Int = 0): Order {
    return Order(
            id = UUID.randomUUID().toString(),
            date = LocalDate.now(),
            deviations = (1..deviations).toList().map { Deviation("Deviation number $it
    )
}

fun Order.Companion.invalidOrder(): Order {
    return validOrder(deviations = 1).copy(id = "")
}
```

TestDataExtensionMethods.kt hosted with ❤️ by GitHub view raw

Extension methods applied to the Order Companion object.

The *Companion* here is the static context of *Order*. That's why you can do *Order.validOrder()* and not *myOrderInstance.validOrder()*.

In the code above you see that we have both a *valid* and an *invalid* method. It varies how many of these we have. But usually we have at least one valid method for each domain object. For Addresses we have something like *validMyStreetAddress()* and *validOtherCityStreetAddress()*.

Back to another test that shows how we use the default valid state and modify it specifically for the state we need:

> *"When I have a order that has been registered, but has one deviation, it should be impossible to approve it."*

```
    @Test
    fun shouldNotApproveOrderWithDeviations() {
        val order = Order.validOrder()
            .copy(deviations = listOf(Deviation("Could not find destination")))
        assertThat(order.canBeApproved(), equalTo(false))
    }

    @Test
    fun shouldNotApproveOrderWithDeviationsWithGeneratorMethod() {
        val order = Order.validOrder(deviations = 1)
        assertThat(order.canBeApproved(), equalTo(false))
    }
```

TestDataAllTests.kt hosted with ❤️ by GitHub view raw

How to modify a default structure illustrated in two ways

The above code shows how you can take a “default valid” structure and make it invalid for the sake of the current test. The secret to this (which replaces Builders) is Kotlin Data Classes. Every data class in Kotlin is immutable, but can be copied with named parameters to change values for the new object.

*The second test above shows the same case*, but we have added the option to modify the object as a parameter (with a default value) to validOrder(). We do this when we see that the setup is something we will use in multiple places. Here is the same method again. Notice the deviations parameter to validOrder:

```
fun Order.Companion.validOrder(deviations: Int = 0): Order {
    return Order(
            id = UUID.randomUUID().toString(),
            date = LocalDate.now(),
            deviations = (1..deviations).toList().map { Deviation("Deviation number $it
    )
}

fun Order.Companion.invalidOrder(): Order {
    return validOrder(deviations = 1).copy(id = "")
}
```

TestDataExtensionMethods.kt hosted with ❤️ by GitHub view raw

Notice also that some times it makes sense to use the valid method inside the invalid method. This way there are even fewer places to change when something changes.

That's it really. We use this everywhere, and have been able to do quite extensive re-factorings without everything rippling out out control. You'll never get around the wiring and setup completely. But it is possible to minimize it.

To summarize, these techniques helps us:

**Reuse and find test data** by placing methods on the class as a extension methods. They are also fast because they are just code.

**Minimize unintended consequences when something changes** by having the same data setup in as few methods as possible. If fields are added, renamed or changed there are not thousands of places that needs changing.

**Make it easy to write tests** by having quick and easy to find methods. We know that resistance to writing tests leads to band aids and even not writing tests. We want to make it as easy as possible.

Please let me know if you find something weird. I will be more than happy to clarify and correct. :)

Comment or reach out to me on Twitter if you have any questions. :)

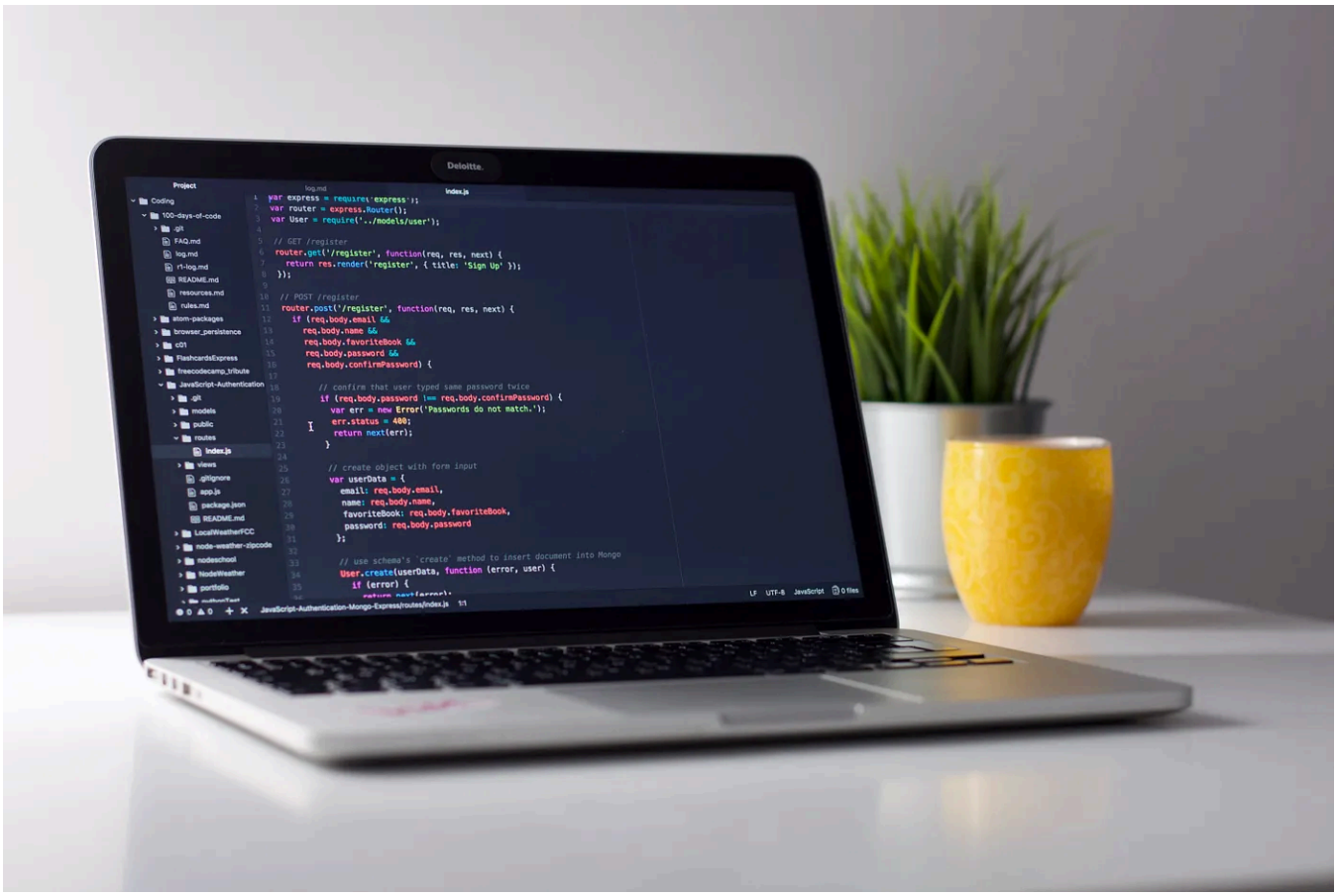


Photo by Clément H on Unsplash

Kotlin Tdd Test Data Automated Testing Refactoring



Follow

## Written by Anders Sveen

418 followers · 716 following

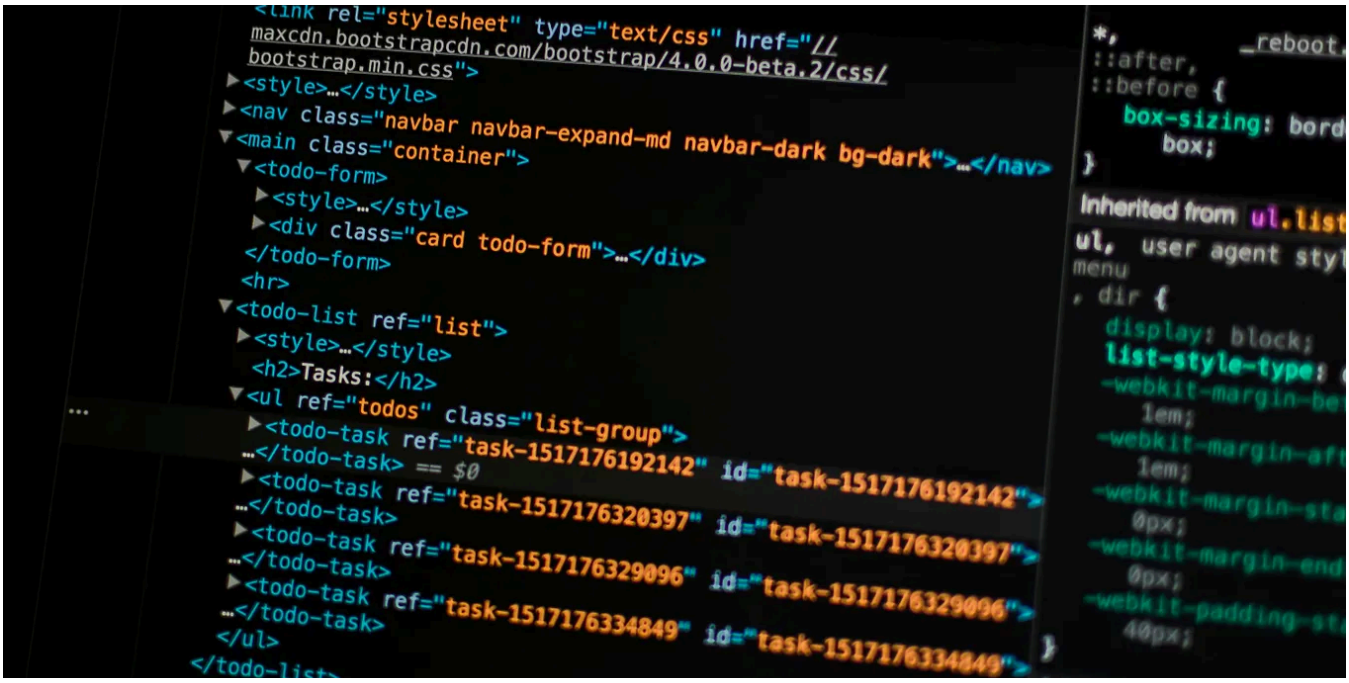
Passionate agile developer and mentor for hire @ Mikill Digital

## No responses yet

Write a response

What are your thoughts?

## More from Anders Sveen




Anders Sveen

### Using Claude Code with GitHub Copilot: A Guide

Because we can’t send company information to just any LLM (in this case Anthropic), I wanted to use Claude Code with our company approved...

Sep 22, 2025 5

In Smidigalliansen by Anders Sveen

## Smidigposten #27

Photo by Jamie Fenn on Unsplash

4d ago 1

Anders Sveen

## Full stack development with KTor and HTMX ❤️

Sometimes it feels like I have been yak shaving for the last 20 years. Well, not entirely. But things like Rules Engines, SOA, App Servers...

Aug 30, 2023 103 1

Anders Sveen

## Living the stream: Reducing memory usage in Java and Kotlin

TLDR; Change how you fetch lists of information in your HTTP endpoints, and your application can often survive on 300–600MB of heap. Ours...

Sep 17, 2020 284 1

See all from Anders Sveen

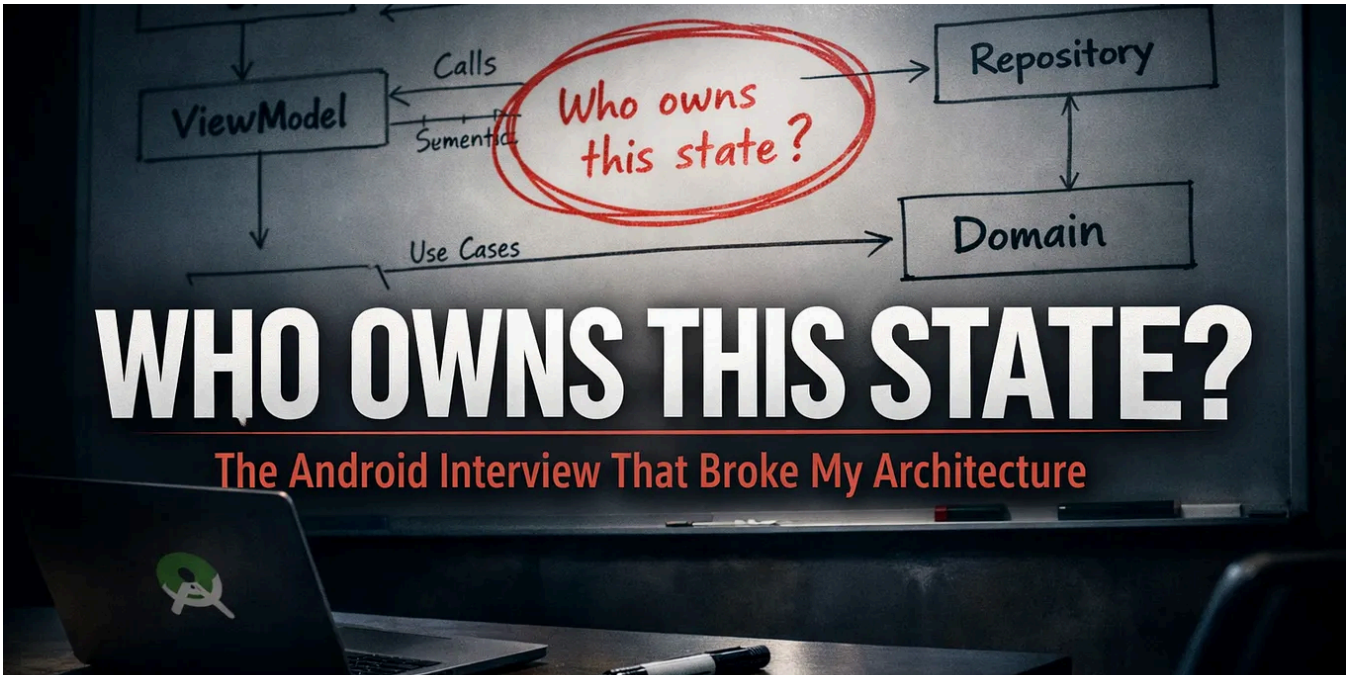
## Recommended from Medium


Sixtin - Android Developer

## What Maintaining an Android App for 5 Years Really Looks Like

The unglamorous truth about Android longevity dependency hell, API migrations, and why your architecture decisions from year one still...

6d ago 1

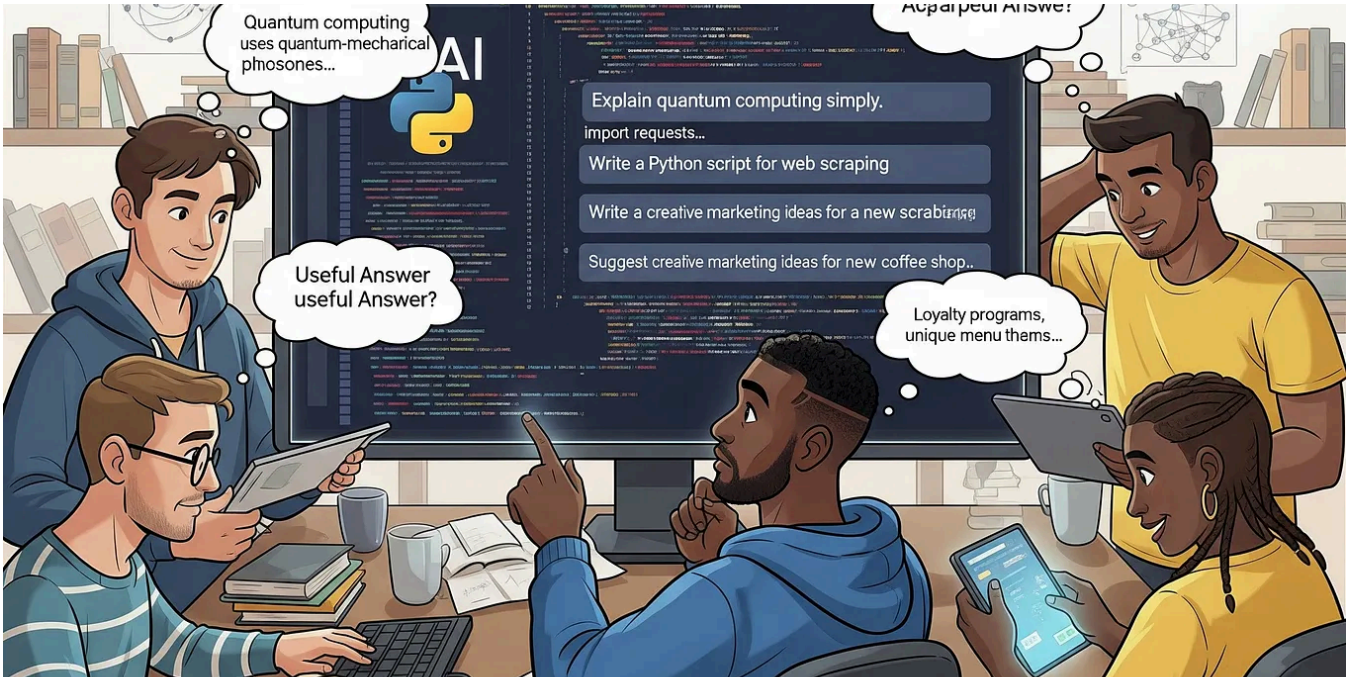



Mobile App Developer

## Android: The Interviewer Asked: “Who Owns This State?” I Froze.

The Android architecture question I thought I knew—until the digging started.

Jan 2 65 6




Freny Christian

## Writing Prompts Like a Pro: How Developers Can Actually Get Useful Answers from AI

I'll be honest—when I first started messing around with AI tools like ChatGPT and Copilot, I thought I could just type something vague...

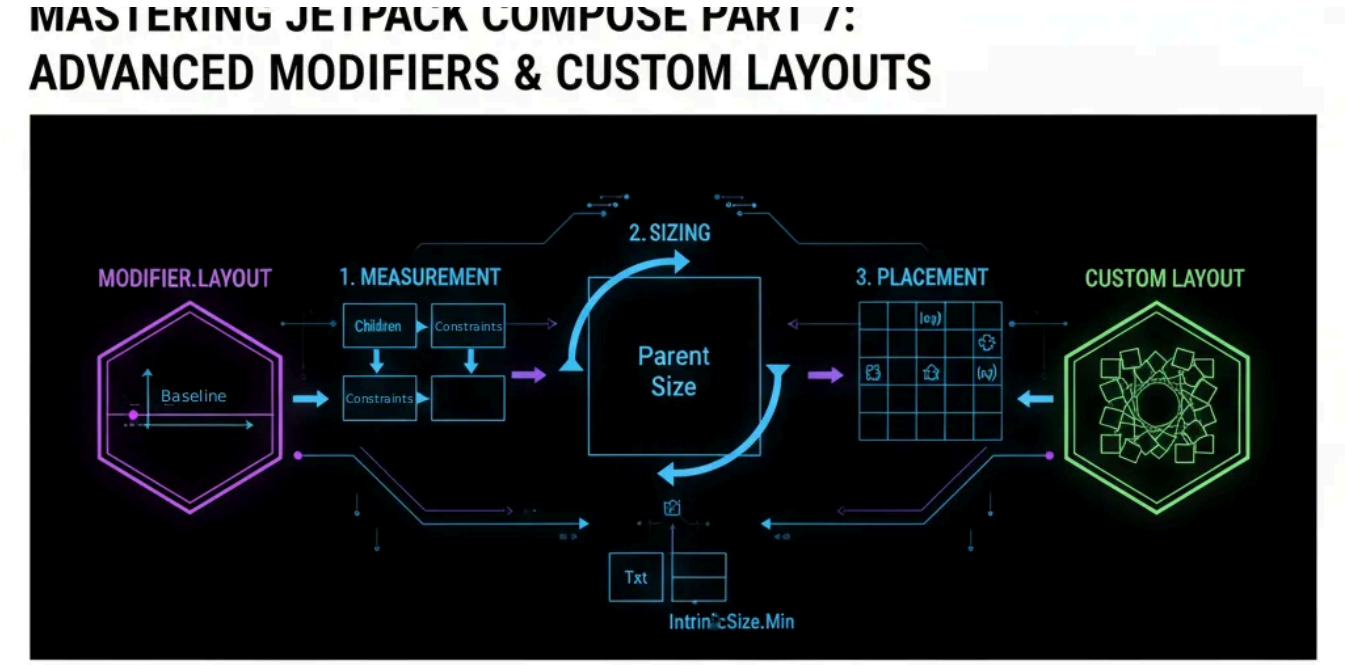
Jul 30, 2025 1


AndroidLab by Andre

## Why Kotlin 2.3 Is a Bigger Deal Than You Think

The compiler speedups, AI integration, and Compose enhancements that will reshape Android development in 2025.

Jan 5 10

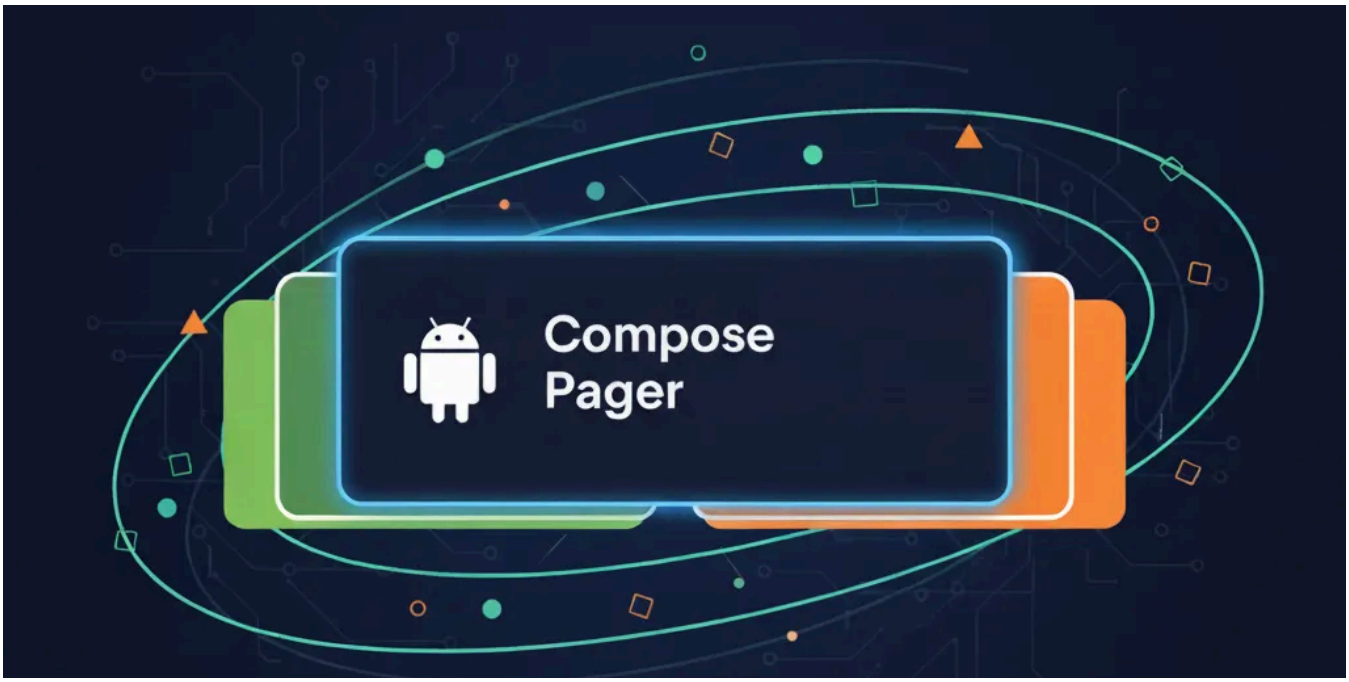



Sivavishnu

## Part 7: Mastering Jetpack Compose: Advanced Modifiers & Custom Layouts

Unlocking the Layout Phase: Building bespoke components with custom measurement logic, intrinsic sizes, and high-performance modifiers.

Jan 2 2




In ProAndroidDev by Oğuzhan Aslan

## Mastering Pagers in Android Jetpack Compose

For years, the ViewPager and its successor, ViewPager2, were the standard-bearers for creating swippable layouts in Android development...

5d ago 35